Dreidimensionale Visualisierung von Simulationsdaten Mobiler Ad-Hoc-Netzwerke

Diplomarbeit
von
Björn Scheuermann
aus
Mannheim

vorgelegt am
Lehrstuhl für Praktische Informatik IV
Prof. Dr. W. Effelsberg
Fakultät für Mathematik und Informatik
Universität Mannheim

Dezember 2004

Betreuer:

Dipl.-Wirtsch.-Inf. Holger Füßler
Dipl.-Inf. Matthias Transier
Dipl.-Wirtsch.-Inf. Marcel Busse

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne die Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mannheim, den 3. Dezember 2004

Björn Scheuermann

Danksagung

Ich bin einer Reihe von Leuten zu großem Dank verpflichtet, da sie durch ihre Unterstützung und ihren Einsatz nicht ganz unschuldig an dieser Arbeit in ihrer vorliegenden Form sind. Diesem Dank soll hier Ausdruck verliehen werden.

Zunächst wären hier meine drei Betreuer Holger Füßler, Matthias Transier und Marcel Busse zu nennen. Ihre Anregungen und Kommentare waren stets eine Quelle frischer Motivation, und ihre Ideen hinderten meine To-Do-Liste immer zuverlässig am kleiner werden.

Ebenso gilt mein Dank Daniela und Andreas Lindenblatt von der Firma Solution – The Computer People e. K. in Mannheim. Sie waren maßgeblich an der Entwicklung der Idee zu dieser Arbeit beteiligt. Auch ihre Hinweise und Anregungen genau wie ihre unerschütterliche Begeisterung für die Sache waren ein stetiger Ansporn, alles noch ein bisschen besser und schöner zu machen. Das Arbeitsmaterial und der Arbeitsplatz, den sie mir zur Verfügung stellten, waren eine große Erleichterung.

Benjamin Guthier hat großen Anteil an Konzeption und Umsetzung der Software aus dem V-IDS-Projekt, die mir hier als unverzichtbare Grundlage diente. Hierfür sei ihm ebenso mein Dank ausgesprochen wie für die schonungslose Aufdeckung meiner Fehler.

Meinen Eltern Christa und Bernhard Scheuermann möchte ich ebenso wie meiner Freundin Michaela Metzger für ihr ständiges uneingeschränktes Einstehen für mich und meine Arbeit danken, ebenso wie für alle – oft nicht ganz unwillkommenen – Ablenkungen davon.

Severin Heim und Frederic Dahl verdanke ich einige wertvolle Tipps.

Allen hier genannten sowie all jenen, die ich vergessen habe: Danke!

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

2PQ	2-Phasen-Queue
3D	dreidimensional
ACK	Acknowledgement, Empfangsbestätigung
AGT	Agent(-Layer), Anwendungsschicht
AODV	Ad-hoc On-demand Distance Vector routing
ARP	Address Resolution Protocol
CAD	Computer Aided Design
CMU	Carnegie Mellon University
CTS	Clear To Send, Sendefreigabe
DCF	Distributed Coordination Function
DES	Discrete Event Simulation, diskrete Ereignissimulation
DFWMAC	Distributed Foundation Wireless Medium Access Control
DSDV	Destination-Sequenced Distance Vector routing
DSR	Dynamic Source Routing
FIFO	First-In-First-Out
FLTK	Fast Light ToolKit
GNU	GNU's Not Unix
GPL	GNU Public License
GUI	Graphical User Interface, grafische Benutzeroberfläche
Huginn	Handy Utility for Graphical Interpretation of Ns-2 wireless Network traces
ID	Identifier, Identifikation
IEEE	Institute of Electrical and Electronics Engineers
IFQ	InterFace Queue
iNSpect	Interactive NS-2 Protocol and Environment Confirmation Tool
MAC	Medium Access Control, Medienzugriffskontrolle
MANET	Mobile Ad-Hoc Network, mobiles Ad-Hoc-Netz
nam	Network AniMator

ns	Network Simulator
PC	Personal Computer
PDA	Personal Digital Assistant
RGB	Rot-Grün-Blau
RTR	Routing(-Layer)
RTS	Request To Send, Sendewunsch
STDIN	Standard In, Standardeingabe
STDOUT	Standard Out, Standardausgabe
Tcl	Tool Command Language
Tk	ToolKit
TORA	Temporally-Ordered Routing Algorithm
V-IDS	Visual Intrusion Detection System
V-QL	Visual Query Language
Yacc	Yet Another Compiler Compiler

Hinweise zur mathematischen Notation

Logarithmen

In dieser Arbeit wird die folgende Schreibweise für Logarithmen verwendet:

ln	Logarithmus zur Basis e (natürlicher Logarithmus)
lg	Logarithmus zur Basis 10
$\log_\beta$	Logarithmus zur Basis β

Integration

Bei der Integration wird die gebräuchliche Notation für die Differenz des Wertes der Stammfunktion zwischen oberer und unterer Integrationsgrenze mittels eckiger Klammern in der folgenden Weise verwendet:

$$\Big[F(x)\Big]_{x=a}^{x=b} := F(b) - F(a)$$

Kapitel 1

Einführung

1.1 Problemstellung

Diese Arbeit beschreibt die Software Huginn, ein Werkzeug zur dreidimensionalen Darstellung von Netzwerksimulationsdaten.

Die Entwicklung neuer Netzwerktechniken, die Untersuchung ihrer Eigenschaften und der Vergleich unterschiedlicher Ansätze bilden ein breit gefächertes Teilgebiet der Informatik. Eine besondere Herausforderung dieses Gebietes ist die praxisnahe Erprobung von Ansätzen unter wechselnden Rahmenbedingungen. Bei der Untersuchung der Effekte, die durch das Zusammenwirken vieler autonomer Geräte (*Knoten*) in einem Netzwerk entstehen, sind reale Experimente sehr aufwändig und deswegen wenig verbreitet. Die Anzahl der bereitzustellenden und zu betreibenden Netzwerkknoten wäre bei vielen durchaus praxisrelevanten Untersuchungen unrealistisch hoch.

Aus diesem Grund spielen Simulationen auf dem Gebiet der Entwicklung und Untersuchung von Netzwerkprotokollen eine zentrale Rolle. Viele wesentliche Eigenschaften der zu erprobenden Verfahren lassen sich aus Simulationen ableiten.

Fast noch mehr als auf kabelgebundene Netzwerke trifft das oben Gesagte auf Mobile Ad-Hoc-Netze (MANETs) zu. In diesen Netzen kommunizieren unabhängige, mobile Knoten kooperativ miteinander, indem sie auch Daten anderer Knoten aktiv weiterleiten. Hier haben die räumliche Anordnung und Bewegung der Netzwerkknoten und veränderliche physikalische Eigenschaften des Übertragungskanals oft großen Einfluss auch auf die höheren Protokollschichten.

Das meistverwandte Werkzeug zur Simulation kabelgebundener Netze ist der Netzwerksimulator ns-2 [ns2, VIN03]. Zur Simulation von MANETs ist ebenfalls ns-2 mit einer speziellen Erweiterung – CMU Monarch [Mon99] – gebräuchlich. Diese Erweiterung gehört mittlerweile zum Standardlieferumfang des Simulators.

Neben dem weitaus geringeren Aufwand einer Simulation im Vergleich zu einem realen Test zeigt sich ein weiterer Vorteil dieser Technik: Simulationen liefern weitaus genauere Daten. Die Möglichkeiten, die Zusammenhänge der auftretenden Ereignisse zu analysieren, sind aufgrund der in der Simulation quasi beliebig genauen Zeitauflösung

sehr viel besser. In realen Tests hätte man zum Beispiel das Problem einer fehlenden globalen Uhr zu lösen [TS01], diese Schwierigkeit entfällt in der Simulation.

Simulationen können vollständig nachvollziehbare, weil absolut deterministische Ergebnisse liefern. Gewöhnlich geschieht das in Form eines detaillierten Ereignisprotokolls, einem sog. Tracefile. In diesem werden alle wesentlichen Ereignisse, die in Netzwerkknoten auftreten, erfasst.

Für ein derart ausführliches Protokoll fallen schon bei kleineren Szenarien große Datenmengen an. Der hohe Detailgrad kann einerseits äußerst hilfreich sein, wenn Zusammenhänge auf kleiner zeitlicher und räumlicher Ebene analysiert werden sollen. Andererseits gleicht das Auffinden der tatsächlich relevanten Stellen in der großen Menge an Information oftmals der Suche nach der sprichwörtlichen „Nadel im Heuhaufen“. Größere Zusammenhänge intuitiv zu sehen oder auch aktiv nach ihnen zu suchen wird gerade durch die Detailliertheit erschwert.

Selbstverständlich können mit gängigen Werkzeugen leicht Durchschnittswerte, Summen, Maxima o. ä. über die gesamte Simulationsdauer oder auch über gezielt ausgewählte Zeitintervalle berechnet werden. Ebenso können zeit- und ortsabhängige Kurven solcher Werte leicht erzeugt werden. Diese aber mit konkreten Ereignissen oder Ereignisfolgen in Zusammenhang zu bringen, die bestimmte signifikante Veränderungen verursachen, erfordert einen verhältnismäßig hohen Arbeitsaufwand und eine gewisse Intuition bezüglich des Ziels der Suche.

Der hier verfolgte Ansatz zielt darauf, aus den Ergebnisdaten des ns-2-Simulators für MANETs eine intuitive, dreidimensionale grafische Darstellung zu erzeugen. Einige wesentliche Eckpunkte des simulierten Geschehens sollen grafisch so aufbereitet werden, dass ein schneller Überblick leichter zu gewinnen ist. So soll die gezielte Suche nach interessanten Stellen in den Simulationsdaten erleichtert und das Gewinnen eines ersten Eindrucks von den simulierten Abläufen beschleunigt werden. Eine direkt in die Visualisierung des Simulationsgeschehens eingebettete Darstellung statistischer Werte soll die Interpretation in Bezug auf die für Veränderungen verantwortlichen Ursachen erleichtern.

Dass grafische Darstellungen das Verständnis komplexer Zusammenhänge wie auch die Interpretation von statistischen Werten erleichtern können, ist vielfach belegt [Jac94, Jac90]. Auch für dreidimensionale, animierte Darstellungen existieren entsprechende Erfahrungswerte [RMC91].

1.2 Angestrebte Ziele

Durch eine dreidimensionale Darstellung wird ein zusätzlicher Freiheitsgrad gewonnen. Dieser ermöglicht gegenüber einer zur Darstellung von Knotenpositionen, -bewegungen und des Paketflusses zunächst ausreichenden zweidimensionalen Draufsicht-Darstellung die Veränderung der Umgebung „über“ und „unter“ den einzelnen Knoten. Es können so zugleich die Netzwerkknoten und deren Aktivitäten angezeigt und außerdem verschiede-

ne Parameter zu jedem einzelnen Knoten verfolgt werden. Die Möglichkeiten moderner Arbeitsplatz-PCs mit ihren leistungsfähigen Grafikadaptern erlauben eine solche Darstellung ohne weitere Schwierigkeiten.

Intuitive und doch flexible Einsetzbarkeit sind bei der Gestaltung eines solchen die Datenanalyse durch den Menschen und das Gewinnen von Verständnis unterstützenden Werkzeugs fundamentale Anforderungen. Ein Werkzeug, bei dem der Aufwand zu seiner Anwendung den dadurch gewonnenen Produktivitätsvorteil übersteigt, ist nicht hilfreich.

Bei den Anwendern eines Werkzeugs zur Tracefileanalyse kann ein gewisses Grundverständnis der Materie vorausgesetzt werden. Die Form der zu verarbeitenden Daten, deren wesentlichen Informationsgehalt und die Möglichkeiten, diese sinnvoll weiterzuverarbeiten, sollten jedem Benutzer geläufig sein. Die Gestaltung der Software sollte auf einen Anwender mit diesen Voraussetzungen zugeschnitten sein.

Die prinzipielle Vertrautheit mit textbasierten Konfigurationsdateien und Programmquellcodes kann von einem ns-2-Anwender ebenfalls erwartet werden. Dennoch ist eine grafische Oberfläche gerade bei einem Programm sinnvoll, das nur unterstützend bei der Datenauswertung eingesetzt wird. Die Lernkurve ist bei einer richtig gestalteten grafischen Anwendung sehr viel steiler als bei einem textbasierten Werkzeug [Nie94]. Und gerade letzteres kann sich stark auf die Akzeptanz eines solchen Hilfsmittels im täglichen Einsatz auswirken. Dies bedeutet, dass die Konfiguration der zu berechnenden und in die Darstellung einzubettenden Werte grafisch erfolgen sollte.

Gleichzeitig sollte jedoch auch die Berechnung komplexerer, auf den Anwendungszweck zugeschnittener Auswertungen möglich sein. Eine integrierte Programmierschnittstelle für fortgeschrittene, eigene Auswertungsroutinen ist deshalb vorgesehen.

Selbstverständlich sind weitere Anforderungen ebenfalls wesentlich. So ist eine ausreichend hohe Performance für flüssiges Arbeiten wichtig, ebenso wie die leichte Navigation in der dargestellten 3D-Szene und auf der Zeitachse.

1.3 Weiterer Aufbau dieser Arbeit

In Kapitel 2 sollen zunächst einige grundlegende Technologien – allen voran der Netzwerksimulator ns-2 selbst – angesprochen werden, auf denen diese Arbeit aufbaut. Nachdem im Anschluss daran in Kapitel 3 drei andere Werkzeuge mit einer ähnlichen Zielsetzung kurz vorgestellt wurden, widmet sich Kapitel 4 der Erläuterung des hier gewählten Weges, eine Visualisierung mit den gewünschten Eigenschaften zu erzeugen. Insbesondere wird auf die Möglichkeiten des Anwenders zur Individualisierung der Darstellung eingegangen.

Kapitel 5 und 6 konzentrieren sich dann auf die eher technischen Aspekte der Software: Kapitel 5 beschreibt die einzelnen Softwarekomponenten, ihre Aufgaben und ihr Zusammenspiel ebenso wie einige fundamentale Designentscheidungen, die in allen Teilen des Programmes ihren Niederschlag gefunden haben. Im Gegensatz dazu greift sich Kapitel 6 gezielt einige ausgewählte Algorithmen heraus, die bestimmte auftretende Pro-

bleme lösen, beschreibt ihre Funktion und analysiert ihre Eigenschaften bezogen auf den vorliegenden Einsatzzweck.

Zum Abschluss nennt Kapitel 7 einige Möglichkeiten, wie Weiterentwicklungen der Software aussehen könnten, bevor Kapitel 8 das Erreichte nochmals kurz zusammenfasst.

Kapitel 2

Grundlagen

2.1 Mobile Netze

Mobile Ad-Hoc-Netze (MANETs) sind Netzwerke von mobilen Knoten, in denen die Kommunikation auf drahtlosen Übertragungstechniken beruht. Die Knoten kommunizieren direkt miteinander, soweit dies mit der zur Verfügung stehenden Sendereichweite möglich ist. Anderenfalls leiten andere Knoten die Nachrichten kooperativ zum Empfänger weiter. Die Kommunikation geschieht also nicht über eine fest installierte Infrastruktur, sondern alle notwendigen Mittel werden von den Netzwerkknoten selbst zur Verfügung gestellt.

Mobile Netze sind zwischen nahezu beliebigen Klassen von mobilen Geräten vorstellbar. Die denkbaren Szenarien reichen von eher klassischen Anwendungsfällen wie der Kommunikation zwischen Notebooks oder Personal Digital Assistants (PDAs) über die spontane Vernetzung von Mobiltelefonen bis hin zur Vernetzung von fahrenden Automobilen oder zu Gruppen von Sensoren.

Als vorherrschende Übertragungstechnik haben sich die verschiedenen Varianten des Standards IEEE 802.11 [IEE97] etabliert, die die Datenübertragung per Funk in den lizenzfreien Frequenzbändern bei 2,4 bzw. 5 GHz spezifizieren.

Durch Bewegungen der mobilen Knoten verändert sich die Topologie des Netzes potentiell sehr häufig. Damit die Weiterleitung von Datenpaketen in einem solchen Netzwerk effizient möglich ist, sind entsprechend angepasste Routingprotokolle notwendig. Es existiert eine ganze Reihe solcher Protokolle, die sich durch die ihnen zugrunde liegenden Annahmen, die zu ihrem Einsatz notwendigen Voraussetzungen und ihre Arbeitsweise unterscheiden.

Genannt seien hier exemplarisch die Protokolle AODV (Ad hoc On-Demand Distance Vector routing) [PR99], DSDV (Destination-Sequenced Distance Vector routing) [PB94], DSR (Dynamic Source Routing) [JM96, Mal01] und TORA (Temporally-Ordered Routing Algorithm) [PC97]. Implementationen dieser Protokolle für ns-2 existieren seit längerer Zeit und sind hinreichend erprobt. Eine vollständigere Aufstellung von Routingprotokollen für MANETs findet sich in [Wik].

2.2 Diskrete Ereignissimulation

Vor der Beschäftigung mit den Ergebnisdaten einer Netzwerksimulation und deren möglicher Weiterverarbeitung bzw. Darstellung ist es sinnvoll, einen Blick auf den Prozess zu werfen, der diese Daten produziert. Der Netzwerksimulator ns-2 ist ein Werkzeug zur diskreten Ereignissimulation (DES). Dies ist eine Technik, um das zeitabhängige Verhalten von Systemen zu modellieren, in denen zeitdiskrete Zustandsübergänge bzw. Ereignisse auftreten [Hei90].

Bei der diskreten Ereignissimulation wird das zu simulierende System – also beispielsweise das Netzwerk – als Gesamtheit von endlich vielen Zuständen betrachtet, die durch zeitdiskrete Ereignisse verändert werden können. Aus einem etwas weniger abstrakten Blickwinkel betrachtet, besteht das simulierte System aus Funktionseinheiten, die Zustände und Logik von realen Gegenstücken repräsentieren. Diese Funktionseinheiten interagieren, indem sie Ereignisse auslösen, die ihrerseits Wirkungen auf eine andere (oder dieselbe) Funktionseinheit haben können. Ereignisse haben in der DES stets einen exakt spezifizierten Zeitpunkt und keine zeitliche Ausdehnung.

Die Funktionseinheiten können hierarchisch strukturiert sein: Als höhere Hierarchieebene sind etwa die einzelnen Netzwerkknoten denkbar, die dann für sich jeweils weiter unterteilt sind.

Die Zeit im Simulationsgeschehen ist unabhängig von der real während des Simulationslaufes verstreichenden Zeit. Stattdessen werden die zukünftig eintretenden Ereignisse in den gängigen Implementierungen in einer Warteschlange vorgehalten. Lösen Vorkommnisse in einer Funktionseinheit neue Ereignisse aus, werden diese in die Ereigniswarteschlange eingefügt. Die simulierte Zeit springt stets zum zeitlich nächstgelegenen Ereignis in dieser Warteschlange vor. Die Zeit zwischen zwei aufeinanderfolgenden Ereignissen ist in gewisser Weise in der Simulation nicht existent. Auf diese Art überspringt die Simulation einerseits inaktive Phasen, andererseits ist so auch die serielle Verarbeitung mehrerer zeitgleich auftretender Ereignisse möglich. Dieses Verfahren wird in der Literatur häufig als *Next-Event Time Advance* bezeichnet.

2.3 Der Netzwerksimulator ns-2

2.3.1 Überblick

Ns-2 [VIN03, ns2] ist der am weitesten verbreitete Simulator sowohl für drahtgebundene als auch für drahtlose Netzwerke. Trotz der Existenz einiger anderer entsprechender Werkzeuge kann er mit einiger Berechtigung als das Standard-Werkzeug in diesem Bereich bezeichnet werden. Er bietet Unterstützung für die Simulation von Transport-, Routing- und Multicast-Protokollen.

Die Entwicklung von ns-2 wurde und wird durch mehrere Forschungsprojekte an unterschiedlichen Institutionen vorangetrieben. Ursprünglich entstand der Netzwerksimulator ns als Entwicklungszweig des REAL network simulators [Kes88, REA]. Der

Versionssprung von ns zu ns-2 fand 1996 statt.

Ns-2 verwendet eine Kombination von C++ [Str91] und einer objektorientierten Variante der Skriptsprache Tcl [Tcl] namens OTcl [OTc].

In Zusammenhang mit dieser Arbeit speziell interessant sind die ns-2-Erweiterungen für Funknetze. Diese entstanden zunächst im Projekt CMU Monarch als Erweiterung des damals noch rein auf drahtgebundene Netze ausgerichteten ns-2, bis sie dann 1998 als fester Bestandteil in ns-2 aufgenommen wurden [Mon99].

2.3.2 Wireless-Simulationen und -Traces in ns-2

Ein Simulator muss gewisse Abstraktionen vornehmen, er kann nicht alle Aspekte und Eigenschaften der realen Welt abbilden. Der Aufbau eines simulierten drahtlosen Netzwerkknotens in ns-2 unterscheidet sich daher auch deutlich von dem seines realen Gegenstücks.

Abbildung 2.1 zeigt den schematischen Aufbau der drahtlosen Knoten in den CMU-Monarch-Erweiterungen. Für die Betrachtungen hier sind speziell drei Teile der Knoten interessant. Diese entsprechen der Medienzugriffskontrolle (Medium Access Control, MAC), dem zum Einsatz kommenden Routingprotokoll sowie den in ns-2 stark vereinfacht repräsentierten, zusammengefassten höheren Schichten.

Die Tracingfunktionen von ns-2 erlauben das Erfassen von Ereignissen in diesen Bereichen der Knoten durch Mitschreiben der auftretenden Ereignisse in eine Datei, dem *Tracefile*. Das Tracing an einer der genannten Stellen entspricht der Beobachtung von Protokolldateneinheiten an der Schnittstelle der repräsentierten Protokollschicht zur darunterliegenden in einer realen Implementation des Protokollstapels. Tracefileeinträge, die paketbezogene Ereignisse beschreiben, lassen sich jeweils genau einem dieser Traces zuordnen. Die möglichen Ereignisse sind Versand, Empfang, Weiterleitung und Verwerfen von Paketen.

Die betreffenden Stellen im Aufbau der Knoten und die zugehörigen Traces sind folgende:

- Die Simulation der tatsächlichen Übertragung von Daten zwischen Knoten lässt sich am **MAC**-Trace beobachten. Hier können also alle Paketübertragungsversuche – ob erfolgreich oder nicht – protokolliert werden. Auch reine MAC-Layer-Pakete (RTS, CTS, ACK; s. u.) werden erfasst.

- Das **RTR**-Trace protokolliert ausschließlich Datenpakete. Erfasst werden sowohl vom Routingprotokoll selbst erzeugte und empfangene Pakete als auch die Datenpakete, die verschickt, empfangen oder weitergeleitet werden sollen.

- Die höheren Protokollschichten erfasst das **AGT**-Trace. Es beschreibt das Verhalten der simulierten Anwendungs- und Transportschicht-Protokolle, also Empfang und Versand der entsprechenden Pakete.

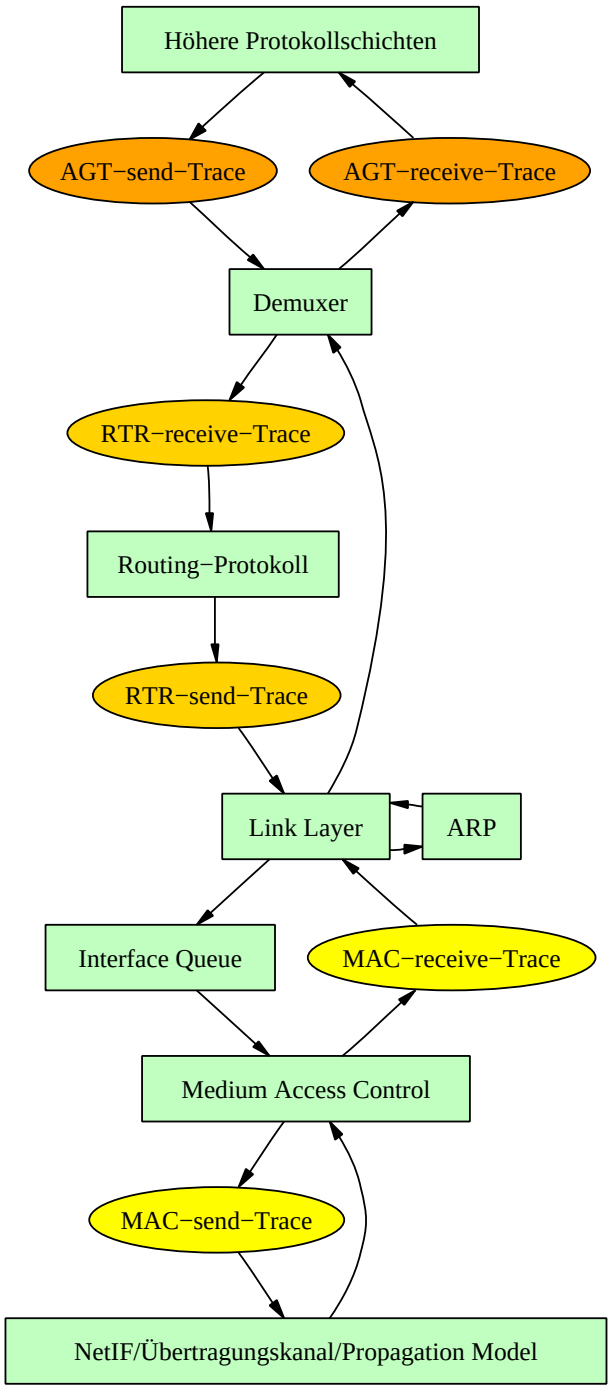


Abbildung 2.1: Schematischer Aufbau eines drahtlosen Knotens in ns-2 (vereinfacht)

Wenn im Folgenden von MAC-, RTR- oder AGT-Ereignissen im Tracefile die Rede ist, dann sind damit Einträge gemeint, die das Auftreten von Ereignissen an den entsprechenden Stellen in einem Knoten ausdrücken.

Eine gewisse Sonderrolle nimmt die Interface Queue der simulierten Knoten ein. Sie repräsentiert die Warteschlange der zum Versand anstehenden Pakete zwischen Routing und Medienzugriffskontrolle. Verwirft sie ein Paket, weil die Länge der simulierten Warteschlange für ein neu ankommendes Paket nicht mehr ausreicht, so kann auch die Interface Queue einen Tracefile-Eintrag erzeugen. Dieser wird dann einem eigenen Trace – bezeichnet mit IFQ – zugeordnet.

Der Ablauf einer simulierten MAC-Layer-Datenübertragung in ns-2 orientiert sich an IEEE 802.11 in der Variante DFWMAC-DCF mit RTS/CTS (Distributed Foundation Wireless Medium Access Control, Distributed Coordination Function mit Request To Send/Clear To Send) [IEE97, DEB93]. Bei einer Übertragung eines Datenpaketes an einen einzelnen Knoten innerhalb der Reichweite (*Unicast*) läuft sie nach dem folgenden Schema ab:

RTS – CTS – Daten – ACK

Zunächst fordert der Sender des Unicast-Paketes eine Sendefreigabe an (Request to Send, RTS). Dies wird vom Empfänger des Datenpaketes mit einer Sendefreigabe (Clear to Send, CTS) bestätigt. Daraufhin findet die eigentliche Datenübertragung statt, die nach ihrem ungestörten Abschluss vom Empfänger bestätigt wird (Acknowledgement, ACK). Dieser Mechanismus vermeidet das sog. *Hidden-Station-Problem*, bei dem zwei Sender sich nicht gegenseitig hören und so nicht feststellen können, dass ihre gleichzeitigen Datenübertragungen sich beim Empfänger überlagern und gegenseitig stören. Das Verfahren trägt also zur Vermeidung von Paketverlusten bei.

Eine Übertragung auf dem MAC-Layer an alle Knoten innerhalb der Sendereichweite (*Broadcast*) besteht nur aus dem Versand des Datenpaketes, ohne vorherige Anforderung einer Sendefreigabe und ohne nachfolgende Bestätigung.

Zwei Radien bestimmen, innerhalb welcher Entfernung andere Knoten eine Übertragung noch korrekt empfangen können, und innerhalb welcher eine Übertragung mit anderen *kollidieren*, d. h. diese stören kann.

2.3.3 ns-2-Tracefiles

Ns-2 kann Tracefiles in mehreren unterschiedlichen Formaten schreiben. Für MANETs sind drei davon interessant:

- das alte Wireless-Format, das ursprünglich mit den CMU-Monarch-Erweiterungen eingeführt wurde,
- ein neues Format, das eine Vereinheitlichung der Trace-Darstellungen der Simulationen von drahtgebundenen und drahtlosen Netzwerken anstrebt und

- ein spezielles Format für die Darstellung mit dem Visualisierer nam.

Eine genaue Beschreibung der Syntax der Tracefile-Formate ist hier nicht notwendig, die Details finden sich in Anhang A. Ein Blick auf die in den Tracefiles enthaltenen Informationen – und damit auf die Informationen, die für eine Visualisierung zur Verfügung stehen – lohnt aber. Er beschränkt sich auf die beiden erstgenannten Formate; das nam-Format enthält zusätzliche Informationen, die speziell die Darstellung in diesem Werkzeug betreffen. In Abschnitt 3.1 wird hierauf bei der Vorstellung von nam nochmals kurz eingegangen werden.

Der Fokus liegt hier auf denjenigen Einträgen, die unabhängig von den simulierten Routing- und Anwendungsschichtprotokollen vorhanden und sinnvoll verwertbar sind. Ein allgemein angelegtes Visualisierungstool sollte zwar die Möglichkeit bieten, auch protokollspezifische Parameter zu verarbeiten, eine sinnvolle Darstellung muss aber auch ohne zusätzliches Wissen über deren Existenz und Eigenschaften möglich sein. Viele Protokolle, deren Darstellung möglich sein soll, sind wahrscheinlich zum Zeitpunkt des Entwicklung des Werkzeugs noch nicht einmal spezifiziert.

Alle Einträge im Tracefile enthalten den Zeitpunkt des zugehörigen Ereigniseintritts sowie die eindeutige ID des betroffenen Netzknotens. Die Einträge für alle MAC-, RTR-AGT- und IFQ-Ereignisse enthalten außerdem die folgenden für eine Visualisierung relevanten Informationen:

- Größe des Pakets
- Den MAC-Header des Pakets, der sich wiederum aus folgenden Informationen zusammensetzt:
  - Allocation
  - MAC-Zieladresse (= ID des Zielknoten)
  - MAC-Quelladresse (= ID des Quellknotens)
  - Art des Paketes (unterschieden werden können IP, ARP und MAC-Layer)

Das Allocation-Feld im MAC-Header dient zur Reservierung zukünftiger Übertragungszeit auf dem Medium.

Im MAC-Header sind die einzelnen Felder abhängig vom Typ des Paketes unterschiedlich – zum Teil auch gar nicht – belegt. Der Typ des Datenpakets ist im alten Traceformat stets aufgeführt, im neuen muss er bei reinen MAC-Layer-Paketen aus der Belegung der Felder im MAC-Header rekonstruiert werden. Dies ist fast immer eindeutig möglich. Die einzige Ausnahme bilden RTS und CTS, die nicht unterschieden werden können, wenn sie von dem Knoten mit der ID 0 verschickt werden. Die Details werden in Abschnitt 6.2 erläutert.

Für reine MAC-Layer-Pakete sind nur die genannten Informationen vorhanden. Pakete des Address Resolution Protocol (ARP) [Plu82] führen noch einen zusätzlichen Header für eben dieses Protokoll mit.

Alle sonstigen Pakete – also Routingnachrichten ebenso wie Datenpakete – enthalten noch weitere Felder. Die meisten Felder sind protokollspezifisch, die folgenden jedoch sind in allen Paketen im *IP-Header* vorhanden und für die vorliegende Problemstellung bedeutend:

- Quelladresse und -port
- Zieladresse und -port
- Größe des Paketes (Datenteil)

Quell- und Zieladresse stehen hierbei – im Gegensatz zu den Angaben für MAC-Quell- und -Zieladresse im MAC-Header – für den ursprünglichen Absender und endgültigen Empfänger des Paketes. Die Angaben im MAC-Header bezeichnen Quelle und Ziel eines einzelnen *Hops*, also einer direkten Übertragung zwischen zwei Knoten; diese beiden Knoten können auch nur Zwischenstationen sein.

Bei allen Ereignissen gibt es außerdem ein Feld für eine Paket-ID. Dieses ist jedoch nur mit sinnvollen Werten belegt, wenn es sich um ein Paket handelt, das vom AGT-Layer erzeugt wurde. Für RTR- und MAC-Ereignisse kann dieses Feld deshalb nicht gewinnbringend ausgewertet werden.

Die genannte Tatsache führt auf ein Problem, mit dem alle Werkzeuge zur Visualisierung von ns-2-Tracefiles zu kämpfen haben: Die Zuordnung von versendeten zu empfangenen Paketen ist in einigen Situationen sehr schwierig. Versendete Pakete können sich ggf. allen im Tracefile auftauchenden Feldern vollständig gleichen. Beim Empfang eines solchen Paketes ist dann nicht erkennbar, welcher Versand diesen Empfang ausgelöst hat. Die existierenden Visualisierungstools fordern deshalb entweder ein speziell angepasstes Traceformat, das die notwendigen Zusammenhänge direkt enthält, oder sie verzichten auf jegliche Ereigniskorrelation.

Die hier beschriebene Visualisierungssoftware geht einen anderen Weg und rekonstruiert aus den Standard-Tracefiles direkt diejenigen Zusammenhänge, bei denen dies möglich ist. Bei Übertragungen auf dem MAC-Layer können die entsprechenden Informationen aufgrund des Wissens, dass Pakete sich auf dem simulierten physikalischen Medium nicht überholen können, gewonnen werden. Abschnitt 6.1 wird sich hiermit noch ausführlicher beschäftigen. Pakete der hohen Schichten können durch die genannte Paket-ID zugeordnet werden. Bei Routing-Ereignissen, die weder eine solche ID bieten noch die gleichen einschränkenden Annahmen wie auf dem MAC-Layer erlauben, ist die Zuordnung jedoch nicht eindeutig möglich. Auch keines der existierenden Werkzeuge mit speziellen Traceformaten bietet derartige Funktionalität.

Neben den Simulationsereignissen in den Traces AGT, RTR, MAC und IFQ können auch noch Statusparameter des Simulators oder einzelner Knoten ebenso wie protokollspezifische Nachrichten im Tracefile festgehalten sein. Von Interesse für eine Visualisierung sind die Informationen über die Positionen und Bewegungen der Knoten. Ns-2 unterstützt stückweise lineare Knotenbewegungen auf einer zweidimensionalen Ebene.

Ein entsprechender Eintrag enthält neben den stets vorhandenen Angaben zu Zeitpunkt und ID des betroffenen Knotens folgende Informationen:

- momentane Position des Knotens bzw. Startpunkt der Bewegung
- Zielposition der Bewegung
- Geschwindigkeit der Bewegung

Sind diese Informationen für alle Knoten und alle linearen Teilstücke der Bewegungen im Tracefile erfasst, so kann daraus das Bewegungsmuster und die Knotenposition zu jedem beliebigen Zeitpunkt rekonstruiert werden.

Beim neuen Traceformat stets, beim alten optional, können außerdem in jeder Zeile Positionsinformationen für den jeweils betroffenen Knoten aufgeführt sein. Dies kann bei unbeweglichen Knoten als Alternative zur Rekonstruktion der Knotenpositionen aus eigenen Einträgen dienen. Deshalb nur bei unbeweglichen, weil Zielpunkte oder Geschwindigkeitsangaben fehlen.

Die anderen zusätzlichen Tracefileeinträge hängen stark von den eingesetzten Protokollen ab und tragen deshalb nicht zur Erzeugung einer protokollunabhängig angelegten Visualisierung bei.

2.3.4 Tracefile-Erweiterungen

Das in dieser Arbeit diskutierte Auswertungswerkzeug unterstützt eine Reihe von Erweiterungen der normalen Tracefile-Formate. Diese Erweiterungen ermöglichen es, im Standard-Tracefile nicht enthaltene Parameter zu übergeben und so in die Visualisierung einfließen zu lassen.

Für ein Funktionieren der Software ist die Verwendung dieser Erweiterungen nicht notwendig, ihr Gebrauch ist rein fakultativ. Die Syntax der Erweiterungen ist in Anhang A.4 beschrieben.

Kapitel 3

Verwandte Ansätze

Die Vorteile einer visuellen Darstellung realen wie simulierten Netzwerkgeschehens wurden schon früher erkannt. Entsprechend gibt es auch eine Reihe von Werkzeugen, die sich dieser Aufgabe annehmen. So existieren z. B. Ansätze, die die Darstellung von realen Netzwerkperformancedaten fokussieren [BMB00, BEW95], oder solche, die sicherheitsrelevante Daten grafisch aufbereiten [NCSLO02, Axe03, KKB04, SLLG04].

Auch für Simulationen von MANETs mit ns-2 existieren bereits Visualisierungswerkzeuge, die nachfolgend kurz vorgestellt werden sollen.

3.1 nam

Bereits im ns-2-Paket enthalten ist der Network AniMator nam [EHH+99]. Dieses Werkzeug beherrschte lange Zeit nur die Visualisierung von Simulationsdaten drahtgebundener Netzwerke. Unterstützung für MANETs ist erst seit vergleichsweise kurzer Zeit vorhanden. Der Schwerpunkt von nam liegt nach wie vor deutlich auf der Darstellung von drahtgebundenen Netzen.

Nam erfordert für die Darstellung die Bereitstellung des Tracefiles in einem speziellen Format. Mechanismen zur Erzeugung dieser nam-Traces sind in ns-2 eingebaut. Beim Durchführen eines Simulationslaufes muss wegen dieser Voraussetzung allerdings bereits zu Beginn die spätere Analyse mit nam geplant sein, damit ein passendes Tracefile erzeugt wird. Da die nam-Traces mehr Informationen enthalten als die normalen ns-2-Tracefiles, ist eine spätere Umwandlung nicht ohne Probleme möglich. Entsprechende Werkzeuge stehen nicht zur Verfügung.

Die nam-Traces enthalten nicht nur das Simulationsgeschehen, sondern auch Zusatzinformationen, die direkt die gewünschte Darstellung betreffen. So können Angaben im Tracefile etwa Farbe und Form der Knoten-Darstellung verändern oder Kommentare zu definierten Zeitpunkten in die Darstellung einblenden.

Nam erzeugt eine zweidimensionale Darstellung des Simulationsgeschehens. Eine freie Navigation auf der Zeitachse ist ebenso möglich wie das Abrufen nicht unmittelbar visuell dargestellter Information durch Klicken mit der Maus auf Darstellungselemente. Das

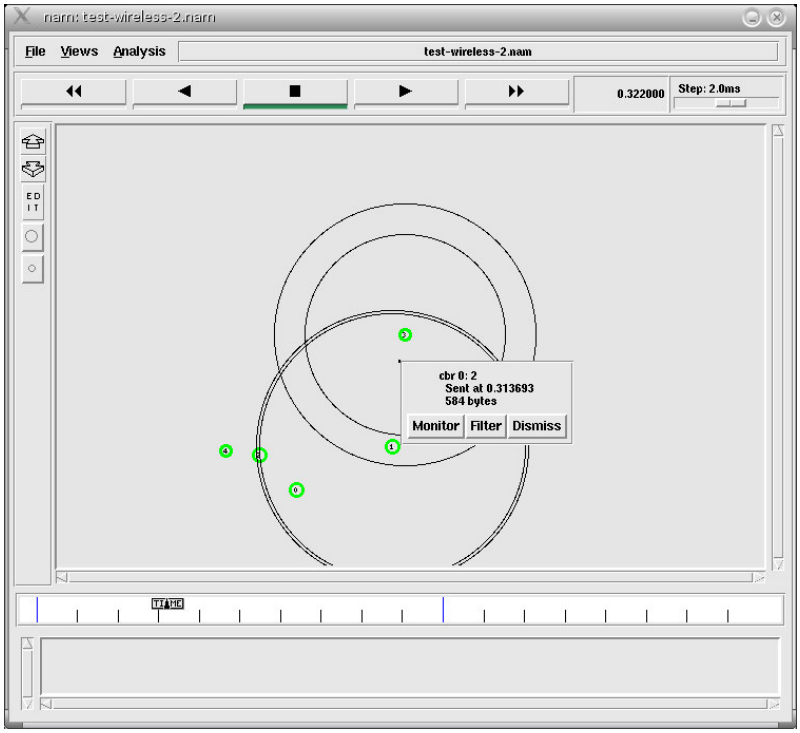


Abbildung 3.1: Der „Network AniMator“ nam

Anzeigen von einfachen Graphen etwa für die Bandbreite oder für Paketverluste ist ebenfalls möglich.

Berechnungen komplexerer oder benutzerdefinierter Auswertungen während der Darstellung sind mit nam – ebenso wie mit den anderen beiden hier vorgestellten Tools – nicht möglich. Solche Werte müssen im Voraus getrennt berechnet und dann in das nam-Tracefile an den betreffenden Stellen eingebettet werden. Erst dann können sie in der dabei definierten Form dargestellt werden. Daher ist eine Auswertung desselben Tracefiles nach verschiedenen Kriterien, indem etwa unterschiedliche Parameter auf verschiedene Darstellungselemente abgebildet werden, nur mit größerem Aufwand möglich.

Nam ist in C++ [Str91] implementiert, als GUI-Bibliothek kommt Tk [Tcl] zum Einsatz.

Neben der Visualisierung von drahtgebundenen und drahtlosen Simulationen enthält nam auch einen Editor, mittels dem sich Simulationsszenarien erstellen lassen.

3.2 Ad-Hockey

Gemeinsam mit den CMU-Monarch-Erweiterungen entstand Ad-Hockey [Mon98, Mal01, Mon99]. Ad-Hockey war lange Zeit das einzige Werkzeug zur Visualisierung von Wireless-Simulationsdaten, bevor nam um die entsprechenden Fähigkeiten erweitert wurde.

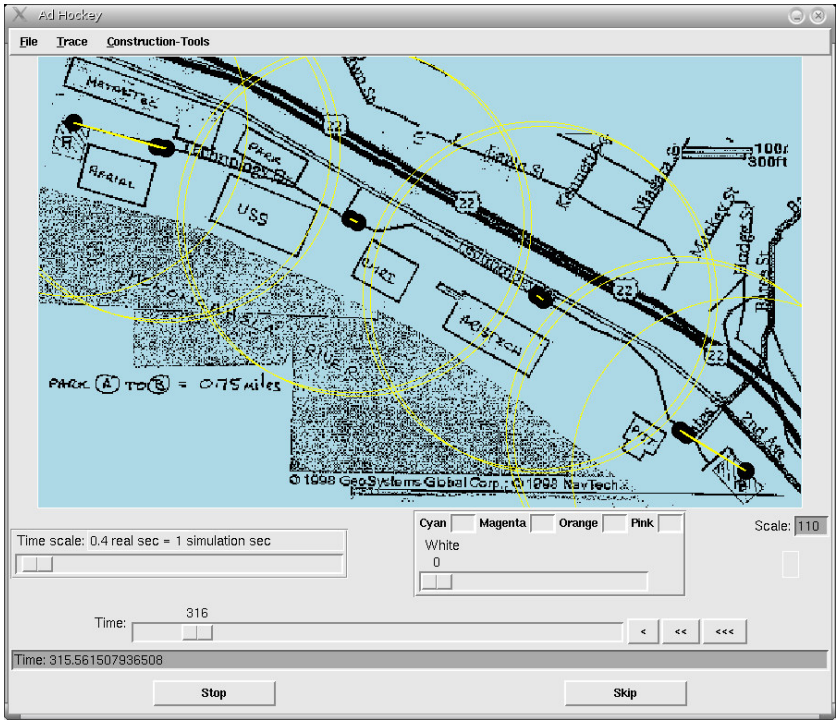


Abbildung 3.2: Ad-Hockey

Das in Perl [CPA] – auch hier in Verbindung mit Tk für die Benutzerschnittstelle – implementierte Werkzeug bietet Möglichkeiten nicht nur zur Visualisierung von Simulationsgeschehen, sondern auch zur Erzeugung von Simulationsszenarien. Ein Editor ermöglicht das Platzieren von Knoten und Hindernissen sowie das Planen von Ereignissen.

Für die Darstellung von Tracefiles benötigt Ad-Hockey kein eigenes Format, allerdings muss das Tracefile zuvor mittels eines speziellen Werkzeugs für die Darstellung aufbereitet werden. Die Positionen und Bewegungen der Netzknoten entnimmt Ad-Hockey aus den Eingabedaten, die auch ns-2 einliest, und nicht aus dem Tracefile. Auf diese Weise ist eine Darstellung bereits vor dem eigentlichen Simulationslauf zur Kontrolle des Szenarios möglich.

Ad-Hockey zeigt von den im Tracefile festgehaltenen Ereignissen ausschließlich die des Routing- und des Agent-Layers an. Die Darstellung des MAC-Layers ist nicht möglich. Entsprechend können auch nur einzelne Ereignisse – Senden und Empfangen – dargestellt werden. Ein Zusammenhang zwischen diesen Ereignissen, etwa, welcher Sende- zu welchem Empfangsvorgang gehört, kann nicht direkt hergestellt werden.

Ad-Hockey bietet eine zweidimensionale Darstellung, in die keine zusätzlichen Daten eingebettet werden können. Ist die Visualisierung angehalten, dann kann auf der Zeitachse gesprungen werden.

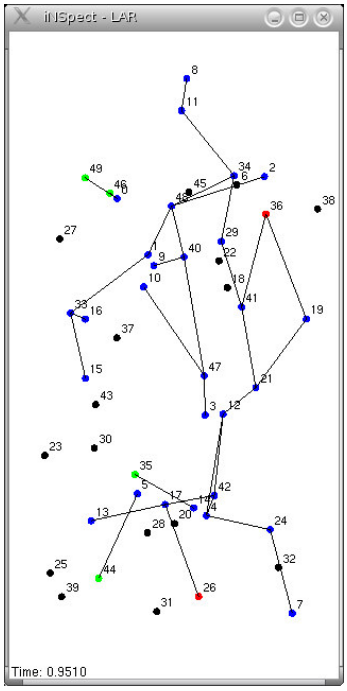


Abbildung 3.3: iNSpect, das „Interactive NS-2 Protocol and Environment Confirmation Tool“

3.3 iNSpect

Erst seit kurzer Zeit verfügbar ist das an der Colorado School of Mines entwickelte Werkzeug iNSpect (Interactive NS-2 Protocol and Environment Confirmation Tool) [KCC04]. Es unterstützt ähnlich wie Ad-Hockey die Analyse von Knotenbewegungs-Szenarien bereits vor dem Simulationslauf durch Einlesen des ns-2 steuernden OTcl-Szenario-Skriptes.

Außerdem bietet es – wie Ad-Hockey – die Kombination dieser Daten mit einem Tracefile. Dies macht eine Anzeige der auftretenden Ereignisse und der gewählten Routen möglich: Zwischen zwei Knoten, zwischen denen eine Datenübertragung stattfindet, kann eine Verbindungslinie eingezeichnet werden, die über eine gewisse Zeitspanne in der Darstellung erhalten bleibt. So zeichnet sich ein Bild der gesamten Route vom ursprünglichen Sender bis zum endgültigen Zielknoten ab.

Leider kann iNSpect keine Standard-ns-2-Tracefiles lesen, sondern verlangt ein eigenes Trace-Format mit einer Zeile für jede erfolgende Übertragung. So ist erst eine Vorverarbeitung der ns-2-Ergebnisdaten nötig, für die keine Werkzeuge bereitgestellt werden.

Eine freie Navigation auf der Zeitachse ist in iNSpect nicht möglich. Es kann nur zu Beginn der Darstellung der Einstiegszeitpunkt festgelegt und während der Darstellung die Ablaufgeschwindigkeit variiert werden. Ein Anspringen beliebiger Zeitpunkte bei bereits laufender Darstellung wird jedoch nicht unterstützt.

Die Konfiguration weiterer Darstellungsparameter – beispielsweise der Gesamtgröße des Gebietes, in dem sich die Knoten in der Simulation bewegen können oder die Anzeigedauer der genannten Verbindungslinien – erfolgt über eine Konfigurationsdatei oder über die Kommandozeile.

Als Implementationssprache wurde für iNSpect C++ gewählt. Die Darstellung des Simulationsgeschehens erfolgt auch hier zweidimensional, in iNSpect allerdings mittels der OpenGL-Bibliotheken. Dies erleichtert das Erzeugen von Videodateien aus der Darstellung, die dann z. B. in Präsentationen eingebunden werden können. Die grafische Oberfläche von iNSpect ist sehr rudimentär, das Programm bietet keine grafischen Steuerelemente sondern nur fensterfüllend die eigentliche Darstellung.

Kapitel 4

Darstellungsform in der Visualisierung

Ein Visualisierungswerkzeug das den Anspruch stellt, allgemein anwendbar zu sein, sieht sich zwangsläufig mit einer großen Bandbreite von unterschiedlichen Anforderungen konfrontiert. Unterschiedliche Benutzer werden unterschiedliche Probleme mit Hilfe des Werkzeugs lösen wollen. Und die überwältigende Mehrheit dieser Probleme wird beim Entwurf ebenso wie bei der Implementation der Software niemand vorhersehen können.

Um tatsächlich auf breiter Basis nutzbar sein zu können, ist deshalb eine gewisse Generalität des gewählten Ansatzes notwendig. Die Darstellung jeglicher Information aus der überwältigenden Detailfülle, die die Simulations-Traces bieten, würde jede Darstellungsform überfrachten und so nur kontraproduktiv wirken. Die Beschränkung auf nur wenige, fest gewählte Informationen schränkt aber den Nutzen in vielen Bereichen ebenfalls stark ein. Als Lösung bleibt ein konfigurierbares System, das sich vom Benutzer für den jeweiligen Einsatzzweck anpassen lässt.

Der Anwender soll durch die Software in die Lage versetzt werden, seine Auswertungsziele stets durch geeignet gewählte Darstellungen unterstützen zu können. Bestimmte Ereignisse, Werte oder Veränderungen sollen gezielt hervorgehoben werden können, ohne dass zu viel Zeit durch die Konfiguration der Darstellungsform verloren geht.

Dieses Kapitel diskutiert, wie die gewünschte Generalität und Flexibilität mit der hier beschriebenen Software zu erreichen versucht wurde. Dazu wird darauf eingegangen, wie die erzeugte Darstellung aufgebaut ist und welche Freiheitsgrade der Anwender hat, sie seinen individuellen Bedürfnissen anzupassen.

Der Name des besprochenen Programms – Huginn – entstammt übrigens der nordischen Mythologie: Der Gott Odin sendet jeden Morgen zwei Raben aus, die über die Welt fliegen und ihm von den Geschehnissen des Tages berichten. Der Name eines dieser Raben ist Huginn („Gedanke“) [Sim78]. Außerdem kann Huginn als Akronym interpretiert werden: Handy Utility for Graphical Interpretation of Ns-2 wireless Network traces.

4.1 Vorhandene Darstellungselemente

Die gewählte Darstellungsform orientiert sich an den gleichen zwei grundlegenden Bestandteilen, die die Arbeit mit ns-2 in jeglicher Hinsicht dominieren: Knoten und Ereignisse bzw Datenübertragungen. So ist auch jedes veränderliche Darstellungselement einem dieser beiden Primitive zugeordnet. Es gehört also zu einem bestimmten Knoten oder zu einem konkreten Ereignis.

Wie einführend bereits angesprochen, ist die erzeugte Darstellung dreidimensional. Der Betrachter kann frei im Raum navigieren. Die Knoten selbst werden durch Kegel dargestellt, die in einer Raumebene angeordnet sind. Die Position des Kegels auf der Ebene spiegelt die Position des Knotens in der Simulation wieder. Kegel haben sich als besonders geeignete Darstellungsform erwiesen, weil sie trotz ihrer Ausdehnung in drei Raumdimensionen einen einzelnen, deutlich hervortretenden Punkt besitzen: Die Spitze hilft dem Betrachter, die Position des Knotens genau zu lokalisieren und die relative Lage zu anderen Knoten besser erfassen zu können.

Die Knotenkegel lassen sich in zwei Freiheitsgraden variieren: Der Benutzer kann Radius und Farbe des Kegels festlegen. Dies ist in Abhängigkeit von beliebigen Parametern möglich, von denen einige vordefiniert sind. Zusätzliche können aber auch vom Benutzer selbst anwendungsspezifisch definiert werden. Wie die Abbildung dieser Parameter auf die Freiheitsgrade erfolgen kann, erläutern die nachfolgenden Abschnitte näher.

Um zusätzliche Freiheit beim Festlegen der Darstellung der Knoten zu bekommen, lassen sich weitere Elemente nach Bedarf zu den Knoten hinzufügen: Ein oder mehrere Diagrammbalken, die über den Knotenkegeln schweben, können durch Höhe, Radius und/oder Farbe frei wählbare Daten wiedergeben. Außerdem können Textdisplays oberhalb der Knoten eingeblendet werden, die numerische Werte oder Zeichenketten anzeigen.

Die Darstellung von Paketübertragungen auf den einzelnen Layern erfolgt in anderer Form. RTR- und AGT-Ereignisse werden durch kleine Kegel angezeigt, die entweder – für Empfangsvorgänge – in den Knotenkegel hinein oder – für Sendevorgänge – aus diesem heraus zeigen. Diese erscheinen bei Eintritt eines entsprechenden Ereignisses für eine (konfigurierbare) Zeitspanne. Eine grafische Assoziation zusammengehöriger Sende- und Empfangsvorgänge findet nicht statt. Dass dies aufgrund der im Tracefile enthaltenen Information für RTR-Ereignisse nicht eindeutig möglich ist, wurde bereits in Abschnitt 2.3.3 diskutiert.

Da AGT-Sende- und -Empfangsvorgänge zeitlich sehr weit auseinander liegen können, wird auch hierbei auf eine grafische Assoziation verzichtet. Dies hat vor allem Übersichtlichkeitsgründe, denn viele AGT-Übertragungen können gleichzeitig und über weite Strecken unterwegs sein. Außerdem existieren aber auch technische Argumente, die gegen eine Darstellung von grafischen Verbindungselementen für AGT-Übertragungen sprechen, wenn wie beim hier gewählten Ansatz das dargestellte Tracefile ohne weitere Vorverarbeitungsschritte in den Standardformaten eingelesen werden können soll. Abschnitt 6.1.4 wird sich mit dieser Problematik noch genauer auseinandersetzen.

MAC-Layer-Übertragungen können und werden dagegen sehr wohl durch grafische Verbindungen zwischen den Knoten angedeutet. Ihre Darstellung erfolgt gleich durch eine ganze Reihe grafischer Primitive:

- Drei nach oben hin größer werdende Tori über dem sendenden Knoten zeigen den Sendevorgang als solchen an. Ihre Farbe sowie Radius und Höhe des durch sie angedeuteten *Sendekegels* lassen sich beeinflussen.
- Zwei konzentrische Tori um den sendenden Knoten bezeichnen den Radius, innerhalb dem die Übertragung korrekt empfangen werden kann, und den Radius, innerhalb dem die Übertragung andere stört. Hier kann die Farbe datenabhängig festgelegt werden.
- Zwischen den Spitzen des Knotenkegels des Senders und denen aller Empfänger wird je ein horizontal liegender *Übertragungskegel* angezeigt; seine Spitze zeigt zum Sender. Für ihn sind der Radius und wiederum die Farbe variabel.
- Bei nicht erfolgreicher Übertragung, wenn der Empfänger das Paket verwirft, erscheint das Ende des jeweiligen Übertragungskegels verdickt und rot, um den Übertragungsfehler augenfällig zu machen.
- Nach Bedarf können sender- wie empfängerseitige Textdisplays Informationen zur Übertragung im Klartext anzeigen.

Abbildung 4.1 zeigt einen Knoten in der dreidimensionalen Darstellung mit allen verfügbaren Darstellungselementen während einer Datenübertragung.

4.2 Festlegung der Darstellungseigenschaften

Um die beschriebenen Freiheitsgrade der Darstellung mit Inhalt belegen zu können, ist ein Schema notwendig, das die Abbildung von Datenquellen auf Darstellungseigenschaften abstrahiert. Hierfür wurde eine Darstellung des Datenflusses und der Verarbeitungsschritte in Form eines Flussdiagramms entwickelt. Der Benutzer wählt die Funktionen aus, die das gewünschte Ergebnis erzeugen, fügt diese in das Diagramm ein und stellt Verbindungen gemäß des gewünschten Datenflusses her.

Die Werte aus den Datenquellen können in einheitlicher Form durch unterschiedliche Aggregatfunktionen wie Summe, Durchschnitt oder Maximalwert vorverarbeitet und auf Elemente der dreidimensionalen Darstellung – z. B. Farbe oder Form von Objekten – abgebildet werden. Die Erstellung dieses Flussdiagramms durch den Benutzer geht dem Start der tatsächlichen Visualisierung voraus.

Die Flussrichtung im Diagramm ist stets die gleiche: Während von links die unverarbeiteten, direkt im Tracefile vorhandenen Daten hereinkommen, werden sie in mehreren teils optionalen, teils obligatorischen Verarbeitungsschritten manipuliert und aggregiert, bis sie schließlich an der rechten Seite des Diagramms in ein Darstellungselement

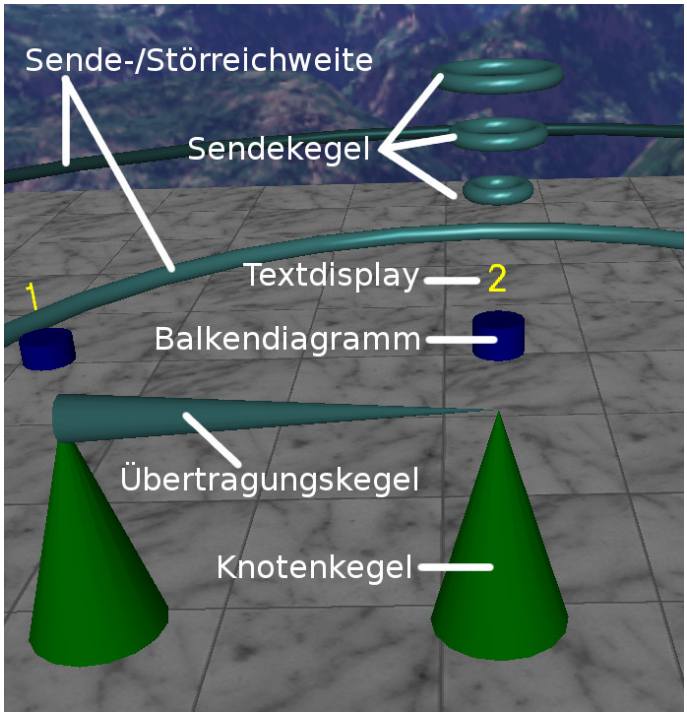


Abbildung 4.1: Darstellungselemente in der dreidimensionalen Darstellung

einmünden. Eine Verzweigung des Datenflusses – dasselbe Zwischenergebnis wird in verschiedenen Formen weiterverarbeitet – ist dabei immer möglich.

Bei der Verarbeitung der Daten können an bestimmten Stellen auch eigene Codefragmente in das Flussdiagramm integriert werden. Diese müssen in der Skriptsprache Ruby [TH00, Rub] formuliert sein. Sie werden beim Start der Visualisierung direkt in den für die Auswertungsrechnungen verantwortlichen Programmteil eingebunden.

4.2.1 Datenquellen

Die Eingangsdaten haben zum Teil sehr unterschiedlichen Charakter. Es gibt Werte, die über die gesamte Simulationsdauer existieren und sich dabei unter Umständen stetig verändern, beispielsweise die Position eines Knotens. Andere Werte existieren nur über bestimmten Zeitintervallen, hier sei beispielhaft die Größe eines momentan übertragenen Paketes genannt. Wiederum in eine andere Kategorie fallen Werte, die mit Ereignissen assoziiert sind, die ohne zeitliche Ausdehnung zu einem bestimmten Zeitpunkt auftreten. Zu letzteren gehören etwa Informationen über das Verwerfen eines Paketes durch einen Knoten.

Für so verschieden geartete Datenquellen sind unterschiedliche Aggregatfunktionen sinnvoll. Während für gelegentlich eintretende Ereignisse etwa ein Zähler für die Anzahl der bisher aufgetretenen Ereignisinstanzen sinnvoll sein kann, kann es einen solcher Zähler für kontinuierliche Werte wie z. B. die Knotenposition nicht geben.

Bei der Festlegung der Darstellung werden aus diesem Grund drei Formen von Datenquellen unterschieden, mit denen sich die auftretenden Werte klassifizieren lassen:

- **Knotenzustände** (oder kurz *Zustände*) beschreiben Eigenschaften von Knoten, die zu jedem einzelnen Knoten über dessen gesamte Existenzdauer kontinuierlich vorhanden sind: Knoten-ID, momentane Position, usw. Ein Zustand hat zu jedem Zeitpunkt einen eindeutig definierten Wert. Dieser Wert kann verwendet werden, um das Aussehen der Knoten in der Visualisierung zu bestimmen.

- **Ereignisse** treten zu einem klar definierten Zeitpunkt in einzelnen Knoten auf. Ein Ereignis hat aus Sicht der Visualisierungssoftware einen *Typ* sowie eine Reihe von *Parametern*. Der Typ bezeichnet, was in dem betreffenden Knoten passiert ist – etwa der Start eines MAC-Sendevorganges –, die Parameter liefern zusätzliche Informationen zu dem Ereignis. Parameter haben einen Namen und einen Wert. Parameter für einen Sendestart sind z. B. die Eigenschaften wie Größe, Sender, Empfänger etc. des Paketes. Ereignisse im Sinne der Visualisierungssoftware haben in der Regel analoge Gegenstücke im Simulator. Auch das Auftreten von Ereignissen kann verwendet werden, um Darstellungseigenschaften der Knoten selbst zu beeinflussen.

- **Übertragungsparameter** beziehen sich dagegen nicht direkt auf einen Knoten: Da Übertragungen auf dem MAC-Layer explizit dargestellt werden, können Parameter einer laufenden Übertragung – z. B. die Paketgröße oder die Anzahl der Empfänger – verwendet werden, um ebendiese Darstellung zu beeinflussen. Dabei existieren einige Parameter wie Paketgröße und -typ für die gesamte Übertragung. Andere, wie die Angabe, ob der Empfang erfolgreich war, sind jeweils für alle einzelnen Empfänger definiert.

Einige Zustände, Ereignisse und Ereignisparameter, darunter alle oben genannten, sind vordefiniert und werden automatisch von der Software zur Verfügung gestellt. Eine Übersicht geben die Tabellen 4.1 bis 4.3. Der Benutzer hat aber auch die Möglichkeit, selbst weitere durch speziell formatierte Tracefileeinträge zu definieren und in der Darstellung zu verwenden.

4.2.2 Filter und Aggregate

Oft soll nicht nur ein einzelner Wert als Momentaufnahme auf ein Darstellungselement abgebildet werden, sondern mehrere Werte oder die zeitliche Entwicklung eines Zustandes sollen zusammengefasst werden. Wie bereits angedeutet, geschieht das über Aggregatfunktionen, von denen je nach Art der Datenquelle unterschiedliche zur Verfügung stehen. Einen Überblick geben die Tabellen 4.4 und 4.5. Zustandsaggregate verarbeiten dabei den Wert eines Zustandes. Ereignisaggregate betrachten hingegen meist einen speziellen Parameter aller eintretenden Ereignisse; dieser Parameter muss dann noch spezifiziert werden.

Ereignistyp	**Tritt auf bei...**	**Parameter**
MACSendStart	Start eines MAC-Sendevorgangs	size (Paketgröße) type (Pakettyp) duration (Übertragungsdauer) lost (kein Empfänger?)
MACSendEnd	Ende eines MAC-Sendevorgangs	*wie MACSendStart*
MACReceiveStart	Start eines MAC-Empfanges	size (Paketgröße) type (Pakettyp) duration (Übertragungsdauer) sender (sendender Knoten) dropped (Paket verworfen?) sendline (Trace-Eintrag Senden)
MACReceiveEnd	Ende eines MAC-Empfanges	*wie MACReceiveStart*
IFQDrop	Verwerfen eines Paketes durch IFQ	*keine weiteren Parameter*
RTRSend	RTR-Sendeereignis	destination (Zieladresse) size (Paketgröße) type (Pakettyp)
RTRReceive	RTR-Empfangsereignis	source (Quelladresse) size (Paketgröße) type (Pakettyp)
AGTSend	AGT-Sendeereignis	packetid (Paket-ID) destination (Zieladresse) size (Paketgröße) type (Pakettyp)
AGTReceive	AGT-Empfangsereignis	packetid (Paket-ID) source (Quelladresse) size (Paketgröße) type (Pakettyp) sendtime (Sendezeitpunkt) duration (Zeit seit Versand) sendline (Trace-Eintrag Senden)
Alle Ereignisse haben folgende Parameter:		time (Zeitpunkt) node (Knoten-ID) traceline (Trace-Eintrag)

Tabelle 4.1: Die vordefinierten Ereignistypen und ihre Parameter

Zustand	Beschreibung
NodeID	Knoten-ID
PositionX	x-Koordinate der Knotenposition
PositionY	y-Koordinate der Knotenposition
DestinationX	x-Koordinate des Zielpunktes der momentanen Bewegung
DestinationY	y-Koordinate des Zielpunktes der momentanen Bewegung
Speed	momentane Bewegungsgeschwindigkeit

Tabelle 4.2: Die vordefinierten Zustände

Parameter	**senderseitig**	**empfängerseitig**	**Beschreibung**
node	+	+	Knoten-ID
size	+	+	Paketgröße
type	+	+	Pakettyp
duration	+	+	Übertragungsdauer
lost	+	−	Keine Empfänger?
sender	−	+	Knoten-ID des Senders
dropped	−	−	Wurde das Paket verworfen?

Tabelle 4.3: Unterstützte Übertragungsparameter; bei den Parametern *node* und ggf. auch *duration* weicht der Inhalt sender- und empfängerseitig voneinander ab

Die grafische Darstellung von Übertragungen bezieht sich stets nur auf die einzelne, momentan laufende Übertragung. Die Werte der Übertragungsparameter ändern sich während der Übertragung nicht. Deshalb kann hier nicht aggregiert werden, stattdessen werden Übertragungsparameter direkt auf grafische Elemente abgebildet.

Bei Ereignissen sowie bei den Übertragungsparametern besteht die Möglichkeit, zusätzliche Parameter durch benutzergeschriebenen Quellcode zu berechnen. Die Ereignisse können hierfür durch einen *Filter* vorverarbeitet werden. Alle standardmäßig oder vom Benutzer via Tracefileeintrag definierten Parameter stehen für einen solchen Quellcodeabschnitt in einer Ruby-Hashtabelle namens *params* zur Verfügung. Diese kann im Quellcode des Filters verändert und um zusätzliche Einträge erweitert werden. Außerdem besteht für Ereignisse die Möglichkeit, sie abhängig von ihren Parametern vollständig zu verwerfen.

Unter anderem werden dem Rubycode in den Filtern auch die Zeilen des Tracefiles als Zeichenkette übergeben, die mit dem jeweiligen Ereignis zu tun haben – bei einem MAC-Empfangsereignis etwa die Zeile, die den Empfang beschreibt, ebenso wie die des zugehörigen Sendevorgangs. So können Filter auch verwendet werden, um z. B. protokollspezifische Felder auszuwerten, die die Visualisierungssoftware nicht kennt und unterstützt.

Bezeichnung	**Funktion**	**Konfigurations-Parameter**
activity	Häufigkeit des Eintretens des Ereignisses in letzter Zeit; erhöht sich um 1 mit jedem Eintritt des Ereignisses und halbiert sich immer über den Verlauf einer spezifizierten Halbwertszeit	Halbwertszeit (für das Abfallen des Wertes, in Simulations-Sekunden)
average	Arithmetisches Mittel eines Parameters über alle Eintritte eines Ereignistyps	Zu mittelnder Parameter
count	Anzahl der bisherigen Ereignis-Eintritte	
last	Wert eines Parameters beim zuletzt aufgetretenen Ereignis	Beobachteter Parameter
max / min	Höchster / niedrigster bisher aufgetretener Wert eines Parameters	Beobachteter Parameter
events per second	Durchschnittliche Anzahl der Ereignis-Eintritte pro Sekunde vergangener Simulationszeit	
sum	Summe aller Werte eines Parameters über die bisherigen Ereigniseintritte	Summierter Parameter

Tabelle 4.4: Verfügbare Ereignisaggregate und deren Funktion

Bezeichnung	**Funktion**
average	Mittel des Zustandswertes über den bisherigen Simulationsverlauf
current	Gegenwärtiger Wert des Zustandes
integral	Integral der Wertefunktion des Zustandes über den bisherigen Simulationsverlauf
max / min	Höchster / niedrigster bisher aufgetretener Wert des Zustandes

Tabelle 4.5: Verfügbare Zustandsaggregate und deren Funktion

Es ergibt sich nun also folgendes Bild:

- Eintretende Ereignisse können optional mittels in Ruby verfasstem Filtercode zunächst vorverarbeitet, verändert oder ggf. auch verworfen werden. Danach werden Parameter dieser Ereignisse knotenweise in Aggregatfunktionen zusammengefasst. Diese Aggregate können dann kontinuierlich für jeden Knoten einen Wert zur Verfügung stellen, der in die Art der Darstellung der Knoten einfließen kann.
- Knotenzustände werden direkt in eine Zustandsaggregatfunktion geleitet, die ebenfalls kontinuierlich für jeden Knoten einen Wert zur Verfügung stellt. Auch diese Werte können das Aussehen eines Knotens bestimmen.
- Parameter einer Übertragung können direkt verwendet werden, um für die Dauer der Darstellung der jeweiligen Übertragung deren Erscheinungsbild festzulegen. Dabei sind aber Parameter zu unterscheiden, die die Übertragung insgesamt betreffen, und solche, die empfängerspezifisch sind. Bei mehreren Empfängern können letztere sich zwischen diesen unterscheiden:
  - Für die Wahl der Farbe der Reichweitenanzeige, die Freiheitsgrade des Sendekegels und den Inhalt von über dem Sender angezeigten Textdisplays sind nur Parameter verfügbar, die für die gesamte Übertragung definiert sind – etwa die Paketgröße oder der sendende Knoten.
  - Für die zu jedem Empfänger eingeblendeten Übertragungskegel oder für empfängerseitige Textdisplays können auch diesen speziellen Empfang betreffende Parameter ausgewertet werden. Zu diesen zählen etwa die Knoten-ID des Empfängers oder der boolesche Wert, ob die Übertragung erfolgreich war oder ob das Paket verworfen wurde.

4.2.3 Skalierung

Für verschiedene Darstellungselemente werden unterschiedliche Datentypen als Eingangswerte erwartet werden. Um beispielsweise die Größe eines Paketes durch eine Farbe darstellen zu können, ist die Abbildung vom Wertebereich der Paketgrößen in den Wertebereich der RGB-Farben notwendig. Ebenso kann in Balkendiagrammen nicht stets dieselbe Größenskala verwendet werden: Wenn Werte so unterschiedlicher Größenordnung wie Übertragungsdauern von Paketen (ausgedrückt in Sekunden) oder Paketgrößen in Byte dargestellt werden sollen, ist eine jeweils angepasste Skalierung unerlässlich.

Deshalb können und müssen in die Verbindung zwischen dem darzustellenden Wert, also der Aggregatfunktion bzw. dem Übertragungsparameter, und dem darstellenden grafischen Element Skalierungsfunktionen geschaltet werden, die die notwendige Anpassung des Wertebereiches vornehmen. Die verfügbaren Skalierungsfunktionen unterscheiden sich bezüglich des Datentyps, den sie erzeugen. Auf Textdisplays werden ohne Skalierung direkt die jeweiligen Werte dargestellt.

Es stehen vier unterschiedliche Skalierungsfunktionen zur Verfügung. Zwei von ihnen erzeugen Fließkommazahlen, die beispielsweise für Knotenradien oder Höhen von Balkendiagrammen verwendet werden können. Die anderen beiden bilden auf den RGB-Farbraum ab. Diese Skalierungsfunktionen sind:

- Die **lineare Skalierung** bildet ein vorgegebenes Intervall von Ausgangswerten $[a, b]$ auf ein Zielintervall $[x, y]$ ab. Der Benutzer gibt zwei Kontrollpunkte (a, x) und (b, y) vor, aus denen sich die lineare Funktion eindeutig ergibt. Links und rechts von $[a, b]$ wird wahlweise entweder jeder Wert auf den entsprechenden Randwert von $[x, y]$ abgebildet – zu große oder zu kleine Werte werden „abgeschnitten“ –, oder die Abbildung wird linear fortgesetzt.

- Ähnlich verhält sich die **logarithmische Skalierung**. Hier wird ebenfalls ein Intervall $[a, b]$ auf ein Zielintervall $[x, y]$ abgebildet, so dass a zu x und b zu y wird – wiederum gibt der Benutzer einfach für zwei Punkte die expliziten Werte an. Diese definieren jedoch diesmal eine Logarithmusfunktion mit einer variablen Basis β, die um s parallel zur y-Achse verschoben ist. Diese hat also die folgende allgemeine Form:

  $$l(\alpha) = \log_\beta(\alpha) + s \tag{4.1}$$

  Die Behandlung von Werten außerhalb $[a, b]$ ist wiederum wählbar. Abbildung 4.2 zeigt das Verhalten einer linearen und einer logarithmischen Skalierung.

- Die **ColorMap-Skalierung** bildet Zahlenwerte auf Farbwerte ab. Sie tut das, indem eine oder mehrere Kontrollstellen definiert werden, bei denen einem Zahlenwert explizit eine bestimmte Farbe zugewiesen wird. Für Zahlenwerte, die nicht genau auf einer Kontrollstelle liegen, werden die RGB-Farbvektoren der beiden benachbarten Kontrollstellen linear interpoliert. Zahlenwerte kleiner der kleinsten oder größer der größten Kontrollstelle werden auf den Farbwert der kleinsten respektive größten abgebildet. Beispielhaft illustriert das Abbildung 4.3.

- Auch die **ColorList-Skalierung** bildet auf Farben ab. Hierbei wird jedoch jedem Eingangswert ohne Interpolation ein eigener Farbwert zugeordnet. Die ColorList ist also auch für nicht-numerische Parameter – etwa Pakettypen – verwendbar. Eine Zuordnungstabelle kann vom Benutzer vorgegeben werden. Für in dieser Tabelle nicht definierte Werte kann automatisch eine neue Farbe beim ersten Auftreten vergeben werden; alternativ wird eine fest gewählte Farbe für unbekannte Werte verwendet. Zur Laufzeit wird die Interpretation durch eine wahlweise einblendbare Legende unterstützt.

Es hat sich gezeigt, dass die Spezifikation einer linearen oder logarithmischen Skalierung wie beschrieben mittels der Intervalle $[a, b]$ und $[x, y]$ sehr intuitiv ist und leichter von der Hand geht, als Parameter wie Steigung und y-Achsenabschnitt direkt anzugeben.

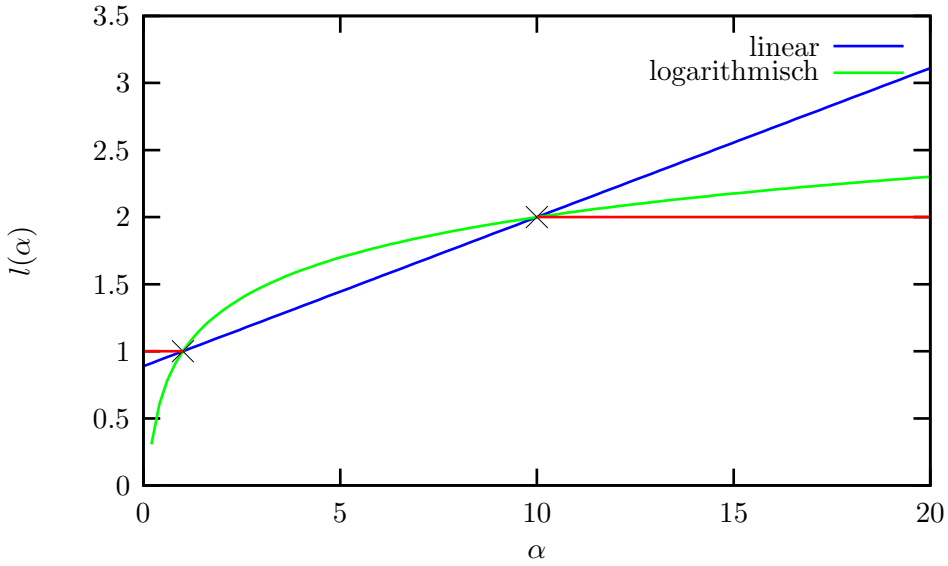


Abbildung 4.2: Kurven für eine lineare und eine logarithmische Skalierung. Beide Skalierungen werden durch die beiden Kontrollpunkte $(1, 1)$ und $(10, 2)$ definiert. Links und rechts der Kontrollpunkte folgen die Skalierungen – getrennt wählbar – entweder der roten Linie oder setzen ihr lineares bzw. logarithmisches Verhalten fort.

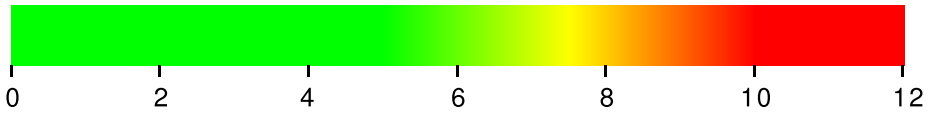


Abbildung 4.3: Beispiel für eine ColorMap-Skalierung mit den folgenden vier Kontrollstellen: $0 \rightarrow [0.0, 1.0, 0.0]$, $5 \rightarrow [0.0, 1.0, 0.0]$, $7.5 \rightarrow [1.0, 1.0, 0.0]$ und $10 \rightarrow [1.0, 0.0, 0.0]$

Die Berechnung der jeweiligen Skalierungsfunktion ist aus den beiden vorgegebenen Kontrollpunkten leicht möglich, Details hierzu finden sich in Abschnitt 6.4.

Für Darstellungsparameter, die für die gewünschte Darstellung konstant bleiben sollen, können feste Vorgabewerte vergeben werden. Diese Darstellungseigenschaften werden im Flussdiagramm für die Berechnung der Darstellung nicht mit einer Datenquelle verbunden. So kann zum Beispiel eine feste Knotenfarbe oder ein stets gleicher Radius von Balkendiagrammen verwendet werden.

4.2.4 Zusammenfassung

Zusammenfassend soll nochmals der gesamte Fluss der Werte für die Knotendarstellungs- und die Übertragungsdarstellungs-Definition betrachtet werden. Aus den hier geschilderten Kombinationsmöglichkeiten ergeben sich die Möglichkeiten, das Auswertungsflussdiagramm festzulegen. Einige abschließende Beispiele sollen diese Möglichkeiten veranschaulichen.

Das Knotenerscheinungsbild kann durch eine Kombination von Ereignis- und Zustandswerten, zusammengefasst durch unterschiedliche Aggregatfunktionen und ggf. vorverarbeitet durch Filter, festgelegt werden. Abgebildet werden können die Ergebnisse der Aggregatfunktionen auf unterschiedliche Parameter des Kegels, der den Knoten darstellt, auf Balkendiagramme sowie auf Textdisplays über den Knoten. Abhängig vom Datentyp, den das gewählte Darstellungselement benötigt, können unterschiedliche Skalierungsfunktionen eingefügt werden. Abbildung 4.4 zeigt den gesamten Fluss der Daten für die Knoten-Darstellung.

Das Aussehen von Übertragungsvorgängen lässt sich durch Parameter der jeweiligen Übertragung beeinflussen. Aggregatfunktionen werden hier nicht verwendet, ein Filter kann aber – wiederum per Rubycode – zuvor weitere Parameter berechnen oder vorhandene verändern. Dieser Filter taucht nicht im Flussdiagramm auf, da nur ein einzelner benötigt wird und dessen Position ohnehin eindeutig festgelegt ist. Den Datenfluss für die Darstellungsdefinition von Übertragungen veranschaulicht Abbildung 4.5.

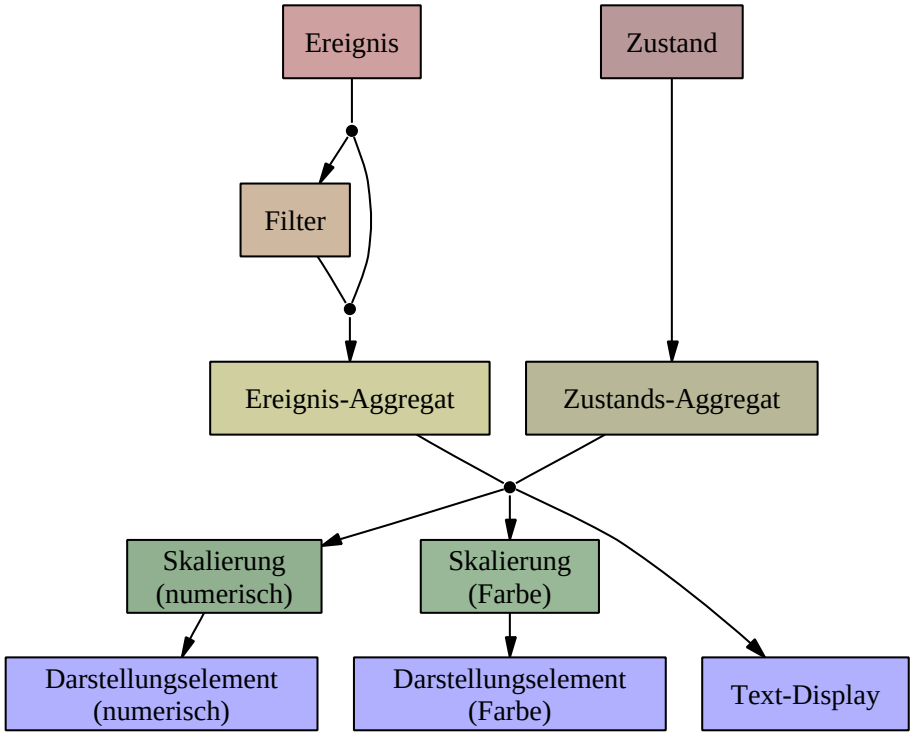


Abbildung 4.4: Datenfluss durch die einzelnen Verarbeitungsstufen bei der Festlegung der Knotendarstellung

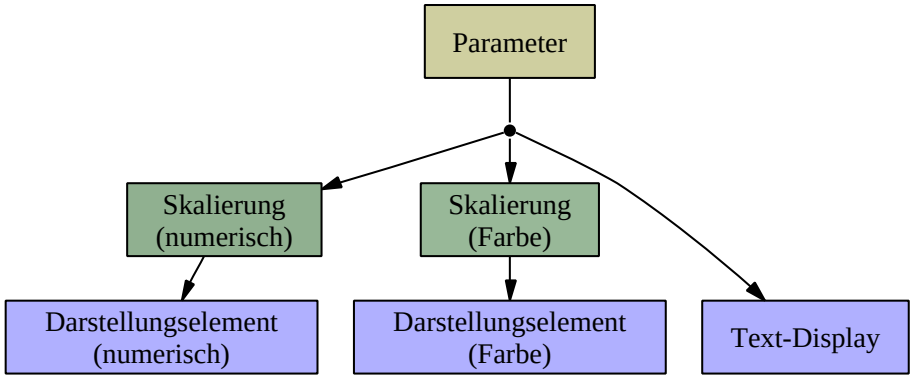
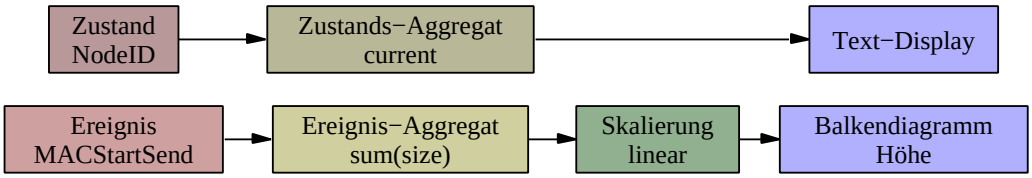


Abbildung 4.5: Datenfluss durch die einzelnen Verarbeitungsstufen bei der Festlegung der Übertragungsdarstellung

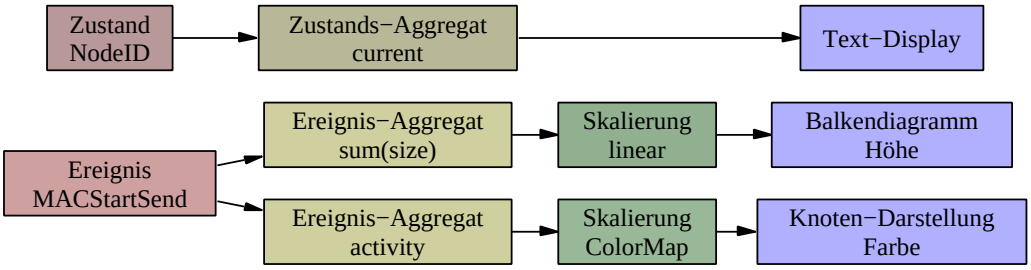
Beispiele

Angenommen, in einer Darstellung soll die ID jedes Knotens über diesem auf einem Textdisplay erscheinen. Außerdem soll zu jedem Knoten ein Diagrammbalken angezeigt werden, der die Gesamtgröße aller bisher von dem jeweiligen Knoten gesendeten Daten mit linearer Skala darstellt.

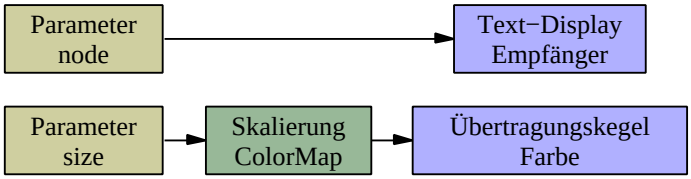
Das Flussdiagramm für die Knotendarstellung hätte dann die folgende Form:



Wenn nun zusätzlich die Farbe des den Knoten repräsentierenden Kegels mit der momentanen Sendeaktivität des Kegels variieren soll, kann hierfür ein zusätzliches Ereignisaggregat eingefügt und über eine ColorMap-Skalierung mit der Knotenfarbe gekoppelt werden:



Sollen bei Übertragungsvorgängen die Übertragungskegel abhängig von der Größe des Paketes eingefärbt und zusätzlich auf einem Textdisplay über jedem Empfänger dessen ID angezeigt werden, würde das entsprechende Flussdiagramm folgende Struktur haben:



In allen Beispielen müssten selbstverständlich die Aggregate und Skalierungen noch parametrisiert werden, beispielsweise müssten die Kontrollpunkte für die lineare oder die Kontrollstellen für die ColorMap-Skalierung passend gewählt werden.

Kapitel 5

Architektur

Im vorangegangenen Teil der Arbeit wurde die entwickelte Abstraktion der möglichen Darstellungsformen durch Flussdiagramme erläutert. Nun soll betrachtet werden, wie das Programm aufgebaut ist, das mit diesen Flussdiagrammen arbeitet und daraus Darstellungen erzeugt.

Die Architektur der Software sieht eine Aufteilung der anfallenden Aufgaben auf drei Programme vor. Diese kooperieren zur Laufzeit, um die Darstellungskonfiguration und die Umsetzung des Konfigurierten in eine Tracefile-Darstellung zu ermöglichen. In diesem Kapitel soll eine Beschreibung der Aufgaben, der zugrunde liegenden Entwurfsideen, der Arbeitsweise und der Kommunikation dieser Einzelkomponenten und des durch sie gebildeten Gesamtsystems gegeben werden.

5.1 Überblick

Die erste der drei Komponenten, aus denen sich die Visualisierungssoftware zusammensetzt, erlaubt die Bearbeitung der Flussdiagramme, die Auswahl eines darzustellenden Tracefiles, das Setzen von Anfangsparametern usw. in einer grafischen Oberfläche. Ihre Zuständigkeit erstreckt sich damit auf alles, was vor dem Starten der eigentlichen Darstellung konfiguriert werden kann und muss. Dieses Programm soll im Folgenden grafische Konfigurationsoberfläche oder kurz GUI genannt werden.

Die zweite, unsichtbar im Hintergrund arbeitende Komponente übernimmt das Einlesen des Tracefiles, dessen Vorverarbeitung, stellt Mechanismen bereit, um schnell im Tracefile springen zu können, und wertet statistische Parameter aus. Dieses Programm soll hier Vorverarbeitungsmodul heißen. Das Ergebnis des Vorverarbeitungsmoduls ist eine Beschreibung der sichtbaren 3D-Szene und ein kontinuierlicher Fluss von Anweisungen zu deren Veränderung auf einem abstrakten Niveau.

Diese Anweisungen werden von der Grafikengine interpretiert, die während der 3D-Darstellung hierzu ständig mit dem Vorverarbeitungsmodul kommuniziert. Die Grafikengine setzt die abstrakten Darstellungsanweisungen in konkrete geometrische Objekte um, erlaubt es dem Benutzer in der 3D-Szene zu navigieren und mit dieser zu interagieren. Sie

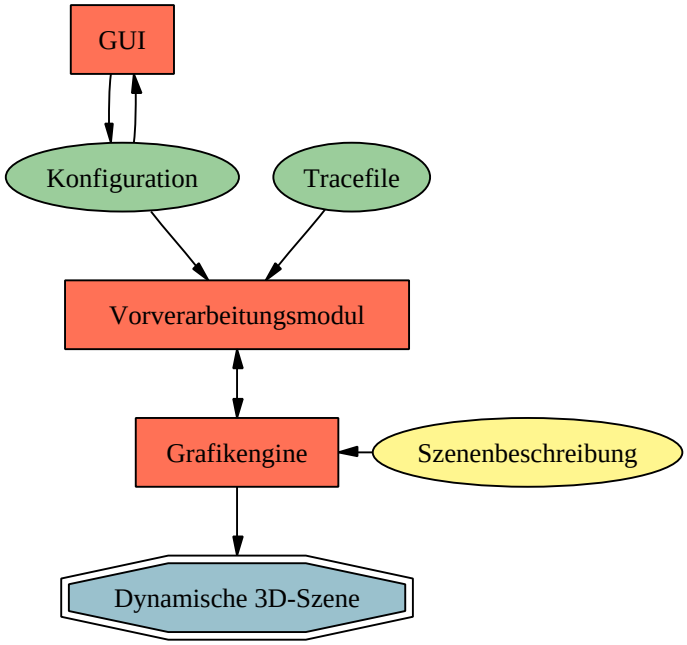


Abbildung 5.1: Architektur der Software

ist die Schnittstelle zum Anwender, während die Visualisierung läuft. Die Form der Umsetzung der Darstellungsanweisung und die verfügbaren Interaktionsmöglichkeiten sind in einer speziellen Szenenbeschreibungsdatei codiert, die die Grafikengine interpretiert.

Eine Übersicht der beteiligten Softwarekomponenten und der jeweils verarbeiteten und erzeugten Daten gibt Abbildung 5.1.

5.2 Gemeinsame Entwurfsprinzipien

Beim Design aller Komponenten der Software standen hohe Flexibilität und Anpassbarkeit bei gleichzeitig einfacher Einsetzbarkeit im Vordergrund. Eine verständliche Benutzerführung ist ebenso wesentlich wie die Suche nach Mechanismen, die das Einsatzgebiet nicht zu weit einschränken. Es wurde deshalb überall versucht, einen Mittelweg zu finden zwischen Generalität einerseits und intuitiver Anwendbarkeit andererseits.

Primäre Zielplattform der Entwicklung war Linux. Grund dafür ist, dass auch der Simulator ns-2 als Datenquelle häufig auf diesem Betriebssystem eingesetzt wird. Bei der Auswahl der verwendeten Bibliotheken war dennoch die Verfügbarkeit auch auf anderen Plattformen – allen voran aktueller Versionen von Microsoft Windows – stets ein wichtiges Kriterium. So ist sichergestellt, dass die entstandene Software ohne die Notwendigkeit allzu grundlegender Veränderungen auch auf andere Plattformen portiert werden kann.

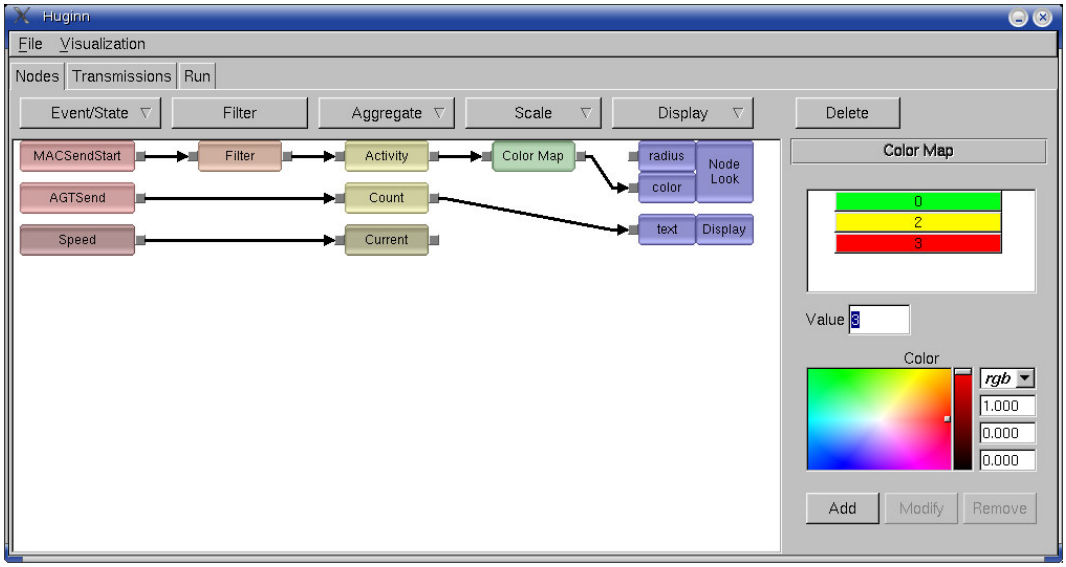


Abbildung 5.2: Screenshot der GUI

5.3 Grafische Konfigurationsoberfläche (GUI)

Die GUI ist der Teil des Programmes, den der Benutzer unmittelbar nach dem Start zu sehen bekommt. Sie ist in C++ [Str91] implementiert. Die Benutzerführung orientiert sich mit einer Menüleiste, Knöpfen, Eingabefeldern usw. an gängigen Standards.

Dem Anwender bietet die GUI die Mittel, festzulegen, welche Auswertungen vorgenommen werden sollen und welche Parameter der Simulation wie in der 3D-Szene dargestellt werden sollen. Diese Festlegung erfolgt durch das Erstellen des Flussdiagramms, das die Auswertungen und die Darstellungsform definiert. Außerdem kann das anzuzeigende Tracefile ausgewählt sowie eine Reihe von Anfangsparametern gesetzt werden.

Mithilfe des Toolkits FLTK [FLT] war eine plattformunabhängige Implementierung der notwendigen eigenen grafischen Steuerelemente möglich. Einige zusätzliche Steuerelemente stammen aus der Sammlung FLTK Utility Widgets (FLU) [FLU].

Eine wichtige Eigenschaft der GUI und ein elementarer Bestandteil des Gesamtentwurfs ist, dass hier das Tracefile nicht eingelesen wird. Die Konfiguration der Darstellung erfolgt vollkommen unabhängig von dem Tracefile, auf das sie später angewendet wird. Ein passendes Tracefile muss zum Zeitpunkt der Konfiguration noch nicht einmal existieren.

Eine in der GUI konfigurierte Darstellungsform kann in eine Konfigurationsdatei gespeichert und natürlich auch später aus dieser wieder geladen werden. So bleiben einmal erstellte Darstellungsformen zur späteren Verwendung erhalten. Sie können dann jederzeit mit demselben oder auch anderen Tracefiles wieder verwendet werden. Der Anwender kann sich so leicht einen individuellen Katalog an mehr oder weniger speziellen Darstellungsformen anlegen und bei Bedarf schnell darauf zurückgreifen.

5.4 Vorverarbeitungsmodul

Das Vorverarbeitungsmodul hat die Aufgabe, nach dem Start der Simulation das Tracefile einzulesen und in der durch den Anwender per GUI vorgegebenen Weise auszuwerten. Es kommuniziert nicht direkt mit dem Anwender.

Für die Implementation wurde die Skriptsprache Ruby [TH00, Rub] gewählt. Diese bietet Mechanismen zur reflexiven Programmierung [Mae87], Objekte und Klassen lassen sich zur Laufzeit in ihrer Struktur verändern. Deshalb ist sie gut geeignet zur Integration von benutzergeschriebenem Code aus den Filtern: Filtercode kann zur Laufzeit eingelesen und in ein vorhandenes Objekt als zusätzliche Methode eingefügt werden. Außerdem bietet Ruby als typische Skriptsprache sehr gute in die Sprache integrierte Möglichkeiten, die Tracefiles zu parsen.

Das Vorverarbeitungsmodul steuert das Einlesen des Tracefiles, legt also fest, wo, wann und wieviel gelesen wird. Es erhält Informationen von der Grafikengine über die Navigationswünsche des Benutzers auf der Zeitachse. Im Gegenzug liefert das Vorverarbeitungsmodul Steuerkommandos an die Grafikengine, die die sichtbare Szene aufbauen und verändern.

Der Großteil der zentralen Verarbeitungsschritte für das Umsetzen der Simulationsdaten in die grafische Darstellung geschieht im Vorverarbeitungsmodul. Daher finden sich hier auch jene Algorithmen, die die in der GUI wählbaren Funktionen realisieren. Einen Überblick über einige davon gibt Kapitel 6.

5.5 Grafikengine

Die Grafikengine der hier besprochenen ns-2-Visualisierungs-Software Huginn wurde nicht speziell und nicht alleine zum Zweck der Visualisierung von ns2-Simulationsdaten entwickelt. Stattdessen ist sie aus dem Projekt V-IDS [VID, Gut04] hervorgegangen, in dem die Entwicklung allgemein einsetzbarer Werkzeuge zur Visualisierung von Abläufen und Zusammenhängen diskreter Ereignisse angestrebt wurde. In diesem Zusammenhang wurde die verwendete Grafikengine insbesondere schon zur Visualisierung realer Daten aus den Protokollen von Firewall- und Intrusion-Detection-Systemen verwendet [SLLG04].

Wie die anderen Programmteile auch, ist die Grafikengine ein selbständig lauffähiges Programm. Sie ist wie die GUI in C++ implementiert und verwendet die OpenGL-Bibliotheken [OGL, Shr04], um plattformunabhängig auf die Grafikhardware zugreifen zu können. Außerdem sind eine Reihe von weiteren Bibliotheken eingebunden, die spezielle Teilaufgaben abdecken:

- *OpenSG* [OSG, RVB02] als Szenegraphen-Bibliothek und damit als weitere Abstraktionsstufe für die Erzeugung der dreidimensionalen Grafik

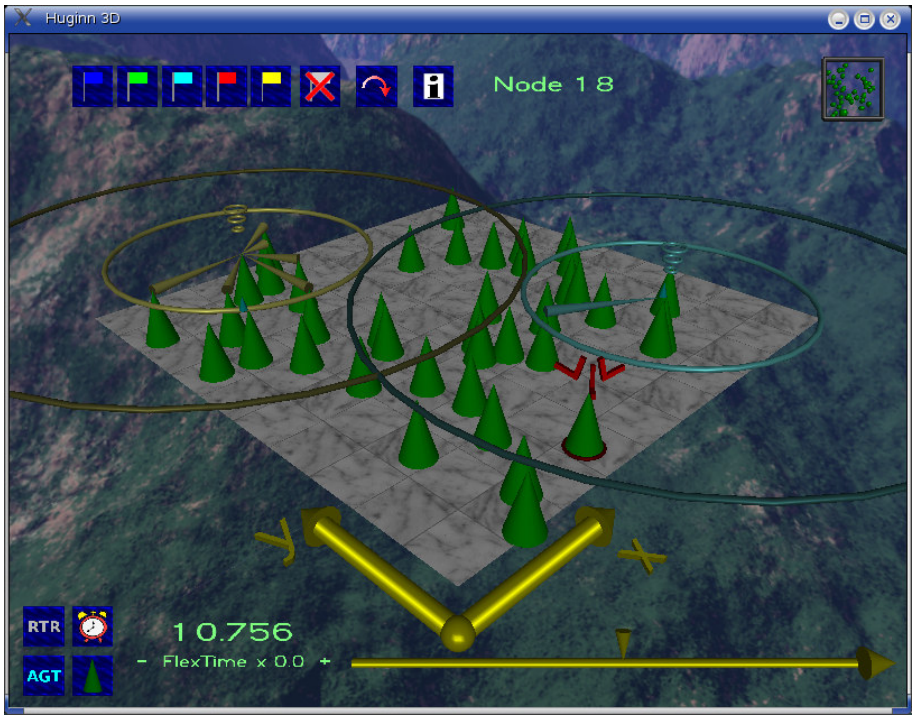


Abbildung 5.3: Screenshot des Hauptfensters der Grafikengine während einer laufenden Visualisierung

- *OpenAL* [OAL, Lok00] und *OpenAL++* [OAP] zum Zugriff auf Mehrkanal-Audiohardware, um die Darstellung durch Raumklang ergänzen zu können
- *libjsw* [JSW] als Schnittstelle zum Joysticktreiber, um einen Joystick oder ein Gamepad als Eingabegerät verwenden zu können
- *FreeType* [FT], um Textanzeigen direkt als Szeneelemente einbetten zu können
- *FLTK* [FLT], das schon für die GUI Verwendung fand, wird in der Grafikengine für das Anzeigen zusätzlicher Fenster verwendet

Während der Visualisierung ist das von der Grafikengine geöffnete Fenster die hauptsächliche Interaktionsschnittstelle zwischen Anwender und Programm. Über dieses kann der Benutzer in der Szene und auf der Zeitachse navigieren und detailliertere Informationen in Form zusätzlicher Anzeigen anfordern. Die Grafikengine kommuniziert dabei mit dem Vorverarbeitungsmodul.

Während der Darstellung führt die Grafikengine Optimierungen durch, die der Beschleunigung der Anzeige dienen. Berechnungen, die mit hohem Aufwand verbunden sind, werden nur so selten wie möglich durchgeführt. Etwa komplex zusammengesetzte Transformationsmatrizen oder Objektgeometrien werden nur dann aktualisiert, wenn die Möglichkeit besteht, dass sie sich tatsächlich geändert haben könnten. Hierzu kom-

men Heuristiken zum Einsatz, die eine untere Schranke für die Zeit bis zur nächsten Veränderung eines Datensatzes in Abhängigkeit der Eingangsdaten abschätzen können.

Weiterhin erfolgen hardwarenahe Optimierungen, etwa eine Sortierung der anzuzeigenden Objekte nach ihren Oberflächeneigenschaften und den auf sie anzuwendenden Transformationen, um die Anzahl der OpenGL-Zustandswechsel zu minimieren.

Eine flexible Grafikengine, die unterschiedlichste Arten von Daten darstellen kann, solche Optimierungen vornimmt und dennoch auf einem hohen Abstraktionsgrad arbeitet, benötigt einen Mechanismus, um sie leicht an den geplanten Anwendungszweck anpassen zu können. Im V-IDS-Projekt fiel die Entscheidung auf eine speziell entwickelte Steuerungssprache mit der Bezeichnung *V-QL*, mittels derer die verfügbaren Objektarten, deren Verhalten (für Bewegungsinterpolation, Benutzerinteraktion etc.), die Zusammenhänge zwischen ihnen und die Form ihrer Darstellung spezifiziert werden.

Die durch eine solche Szenenbeschreibung in V-QL konfigurierte Grafikengine erwartet dann Eingangsdaten in einem speziellen Format über ihre Standardeingabe (STDIN). Ablauf- und Steuerungsdaten, die sich aus Benutzereingaben ergeben, gibt sie über ihre Standardausgabe (STDOUT) zurück. Für die hier vorgestellte Anwendung wurde eine V-QL-Szenenbeschreibung entwickelt, die das Verhalten der Grafikengine für die Visualisierung der betrachteten Datentypen passend festlegt.

Außerdem flossen eine Reihe speziell entwickelter Erweiterungen in die V-IDS-Engine ein, die für die ns-2-Visualisierung notwendig oder wünschenswert waren. Beispielsweise wurden Möglichkeiten zur Einblendung von Zusatzfenstern geschaffen, die zur Anzeige von Detailinformationen zu einzelnen Knoten oder von ColorList-Farbzuordnungstabellen eingesetzt werden.

Für die Beschreibung der Arbeitsweise der V-IDS-Engine fehlt hier leider ebenso wie für eine nähere Erläuterung der Szenenbeschreibungssprache V-QL der Platz. Deshalb sei an dieser Stelle auf die betreffende Dokumentation aus dem V-IDS-Projekt [Gut04, VID] verwiesen.

5.6 Kommunikation der Komponenten

Die vorgestellte Architektur mit ihrer Struktur aus mehreren unabhängigen Prozessen erfordert effiziente Kommunikationskanäle zwischen den Teilkomponenten, um die erwartete Leistung zeigen zu können. Die Menge der zwischen den Komponenten zu übertragenden Daten soll deswegen von möglichst geringem Umfang sein. Wichtig ist außerdem, dass die verwendeten Codierungen einfach und schnell sowohl vom Sender erzeugt als auch vom Empfänger decodiert werden können.

Als weiterer Aspekt kommt der Aufwand für die Fehlersuche hinzu: Verständlichkeit des Inhaltes einer Übertragung durch einen menschlichen „Mitleser“ ohne aufwändige zusätzliche Hilfsmittel kann die Suche nach den Ursachen von nicht wunschgemäßem Verhalten erheblich vereinfachen.

Vor allem aus letzterem Grund fiel die Wahl bei den Kommunikationskanälen auf

rein textbasierte Kommunikationsprotokolle. Obwohl mit optimierten Binärformaten eine wesentlich höhere Informationsdichte im Datenstrom erreichbar ist, zeigen textbasierte Formate in Bezug auf die Möglichkeiten zur Fehlersuche ebenso wie bei den zur Verfügung stehenden Möglichkeiten, sie zu erzeugen und empfängerseitig zu dekodieren, große Vorteile.

5.6.1 Kommunikation der GUI

Der Teil des Gesamtsystems, für den der Kommunikationsumfang mit Abstand am geringsten ist, ist die GUI. Sie muss lediglich beim Start einer Visualisierung dem Vorverarbeitungs-Modul mitteilen, was wie darzustellen ist. Der Inhalt des vom Benutzer erstellten Flussdiagramms muss also hierfür übergeben werden, samt möglicherweise angegebener Codefragmente und globaler Einstellungen.

Diese Informationen sind jenen sehr ähnlich, die die GUI in eine Datei ablegen muss, wenn eine Darstellungsweise gespeichert werden soll, und die sie auch daraus wieder beim Laden einer Konfiguration liest. Es liegt daher nahe, das gleiche Format zu verwenden, um eine Darstellungs-Konfiguration in einer Datei abzulegen und um sie an das Vorverarbeitungs-Modul zur Ausführung weiterzugeben.

In dieser Form wird die Kommunikation auch gehandhabt. Die GUI ist in der Lage, die konfigurierte Darstellungsweise in einem textbasierten Format auszugeben. Auf diese Weise kann eine Konfiguration gespeichert werden. Die GUI enthält außerdem einen Parser für dieses Format. Der Parser wurde mittels der Compilerbau-Tools Flex [Pax95] für die lexikalische Analyse und Bison [DS02] als Parser-Generator erzeugt. Dies sind die kompatiblen, freien Alternativen der Free Software Foundation [FSF] zu den Tools Lex [LS79] und Yacc [Joh79].

Das Vorverarbeitungs-Modul enthält einen zweiten Parser für das gleiche Datenformat, hier allerdings mittels regulärer Ausdrücke in Ruby umgesetzt. Deshalb war ein wesentlicher Aspekt beim Entwurf des Dateiformats die Möglichkeit, dieses sowohl mit einem grammatikbasierten Flex/Bison-Parser in einer C++-Anwendung als auch mit regulären Ausdrücken in Ruby gut verarbeiten zu können.

Abgesehen von dieser Möglichkeit, das Flussdiagramm zu enkodieren und zu dekodieren, ist eine Kommunikation der GUI mit den anderen Programmteilen nicht notwendig.

5.6.2 Kommunikation zwischen dem Vorverarbeitungsmodul und der Grafikengine

Die Kommunikation zwischen dem Vorverarbeitungsmodul, das die anzuzeigenden Daten aufbereitet, und der Grafikengine, die sie in die dreidimensionale Repräsentation umsetzt, ist aus zwei Gründen wesentlich kritischer als die Informationsübermittlung der GUI an das Vorverarbeitungsmodul. Zunächst einmal ist der Umfang der notwendigen Kommunikation um ein Vielfaches größer: Alle dargestellten Vorkommnisse und

Veränderungen müssen übermittelt werden. Speziell in der Richtung vom Vorverarbeitungsmodul zur Grafikengine ist die Datenmenge daher groß.

Noch bedeutender ist aber, dass hier kontinuierlich während der gesamten Visualisierung – und nicht nur einmalig zu Beginn – Daten ausgetauscht werden müssen. Für jedes dargestellte Einzelbild (*Frame*) ist zumindest die Frage, ob es datenbedingte Veränderungen der dargestellten Szene gibt oder nicht – und, falls ja, welche – zu klären.

Das Vorverarbeitungsmodul übermittelt nur solche Veränderungen, und nicht für jeden Frame die gesamte Szene; es tauchen im übertragenen Datenstrom nur die Szenenelemente auf, deren Darstellung sich tatsächlich verändert. Dieses Prinzip der *differenziellen Szenebeschreibung* verringert die zwischen den Programmteilen zu übertragende Datenmenge erheblich.

Die Datenübertragung geschieht mit einem hohen Abstraktionsgrad. Der Grafikengine wird vom Vorverarbeitungsmodul nicht mitgeteilt, welche 3D-Objekte wo angezeigt werden sollen, sondern sie erhält nur Daten wie etwa die Position der sichtbaren Knoten, die momentanen Höhen, Radien und Farben von Balkendiagrammen oder die Sender-Empfänger-Paare einer Datenübertragung. Die Umsetzung in die geometrischen Objekte der 3D-Grafik geschieht dann in der Grafikengine selbst. Sogar die Ausführung von Aufgaben wie die Interpolation von Knotenbewegungen zwischen zwei Wegpunkten kann rein in der Grafikengine erfolgen und erfordert daher keinen Kommunikationsaufwand, solange keine Änderung der Bewegungsrichtung oder -geschwindigkeit erfolgt.

Das eingesetzte Protokoll arbeitet mit von der Grafikengine angestoßenem Polling: Nach der erfolgten Anzeige eines Frames bittet die Grafikengine um die Übermittlung der Veränderungen zum nächsten Frame. Sie übergibt dazu den Zeitpunkt in Simulationszeit, der als nächstes angezeigt werden soll. Daraufhin übermittelt das Vorverarbeitungsmodul die entsprechenden Änderungen mit einem abschließenden Commit-Statement. Bereits während der Übermittlung nimmt die Grafikengine die entsprechenden Veränderungen an der Szene vor und berechnet nach dem Commit das anzuzeigende neue Bild.

Die im Protokoll vorgesehenen Möglichkeiten erlauben es dem Vorverarbeitungsmodul, Szenenelemente (Knoten, Übertragungen,. . .) hinzuzufügen, sie zu verändern und sie zu löschen. Welche Arten von Szenenelementen und welche Parameter für diese verfügbar sind, ist für die Grafikengine gemeinsam mit den Angaben, wie diese in geometrische Objekte umzusetzen sind, in der V-QL-Szenenbeschreibung festgelegt.

Das Kommunikationsprotokoll entstammt ebenfalls dem V-IDS-Projekt. Es wird von der Grafikengine mit einem von Flex und Bison erzeugten Parser interpretiert. Das Format der Nachrichten wurde bewusst einfach entworfen, so dass eine schnelle Verarbeitung mit einem Bison-Parser möglich ist [DS02, GJ90].

Die beschriebene Form der Arbeitsteilung ermöglicht es dem Vorverarbeitungsmodul, sich auf seine eigentliche Aufgabe zu konzentrieren: das Analysieren des Tracefiles und das Vornehmen von Auswertungsberechnungen. Die differenzielle Informationsübermittlung lässt sich neben ihrer Funktion zur Verringerung des Kommunikationsaufwandes außerdem sehr gut mit der Form der vorliegenden Daten vereinbaren: Das Tracefile

enthält – einmal aus einem anderen Blickwinkel betrachtet – ja auch Daten über auftretende Veränderungen.

5.7 Gestaltung der Benutzerinteraktion

Produktive Werkzeuge orientieren sich an den Bedürfnissen ihrer Anwender. Eine steile Lernkurve erhöht die Motivation, und damit auch die Wahrscheinlichkeit des häufigen Einsatzes. Häufiger Einsatz fördert wiederum das Gewinnen von Erfahrung im Umgang mit einem Programm. Erst durch eine breite Benutzerbasis mit dieser Erfahrung können auch fortgeschrittene Funktionen einem breiten Anwenderkreis nutzbar gemacht werden.

Huginn wurde deshalb so entworfen, dass ein sichtbares Ergebnis bereits mit wenig Aufwand und Vorwissen schnell erreicht werden kann. Für eine erste Visualisierung – und damit ein erstes Erfolgserlebnis – genügt es, ein vorhandenes Tracefile im Dateidialog auszuwählen und den Startknopf zu drücken.

Die weitergehenden Möglichkeiten, die die GUI bietet, um Auswertungen vorzunehmen und die Darstellung zu individualisieren, sind in der grafischen Oberfläche deutlich sichtbar. Ein Anwender, der das Grundprinzip der Flussdiagramme verstanden hat, sollte keine Probleme haben, die verfügbaren Funktionen in der GUI zu finden und einzufügen.

Die Gestaltung orientiert sich an einigen allgemeingültigen Entwurfsleitlinien, wie sie etwa von Nielsen in [Nie94] genannt werden. So war z. B. Konsistenz über alle Teile der GUI stets ein wichtiges Gestaltungskriterium. Alle Elemente der Flussdiagramme zeigen sich daher dem Benutzer in einheitlicher Form und bieten dieselben grundlegenden Interaktionsmöglichkeiten, unabhängig von ihrer Funktion; es wurde versucht, möglichst viel Funktionalität mit möglichst wenigen unterschiedlichen Konzepten zu bieten. Das Verbinden von Diagrammelementen mit Pfeilen geschieht an jeder Stelle in der gleichen Weise, ebenso werden die Dialogelemente zum Bearbeiten der Eigenschaften eines Elements immer in der gleichen Weise und an der gleichen Stelle des Programmfensters eingeblendet.

Die grafische Gestaltung versucht einerseits, optisch ansprechend zu wirken, orientiert sich aber streng an gängigen Standards der Benutzerführung; der Anwender soll möglichst viel Wissen über das Verhalten der angezeigten Dialogelemente aus anderen Anwendungen übertragen können. Die Farbwahl soll die Zusammengehörigkeit gleichartiger Elemente hervorheben. Farben werden gezielt eingesetzt, um die Orientierung auf dem Bildschirm zu erleichtern: Beim Verbinden zweier Flussdiagrammelemente z. B. leuchten nach der Auswahl des ersten Endpunktes der Verbindung alle möglichen Partner grün auf, um die Möglichkeit ihrer Selektion als Verbindungspartner anzudeuten.

Orientiert an entsprechenden Erkenntnissen aus der Gestaltpsychologie [Nie94] wurden zusammengehörige Bedienelemente nahe beieinander gruppiert und ggf. auch durch zusätzliche Rahmen abgesetzt. Dies macht die Zusammengehörigkeit für den Benutzer intuitiv erfassbar.

Bei der Auswahl der Operationen, die dem Benutzer zur Verfügung stehen, war

ebenfalls die intuitive Erfassbarkeit aller ihrer Auswirkungen ein Kriterium; aller Auswirkungen bedeutet insbesondere, dass keine unerwarteten Nebenwirkungen oder Seiteneffekte auftreten, die so zunächst nicht zu erwarten waren.

Besonders kritisch ist die Benutzerführung während der tatsächlichen Visualisierung. Es existieren bislang nur verhältnismäßig wenige produktiv eingesetzte Anwendungen, die in vergleichbarer Weise dreidimensionale Grafik einsetzen. Deshalb sind auch die Erfahrungen im Umgang mit einer solchen Oberfläche noch vergleichsweise beschränkt – sowohl, was allgemeine Leitlinien zur Gestaltung solcher Oberflächen betrifft, als auch bezüglich des Vorwissens und des Erfahrungsschatzes des einzelnen Anwenders. Einige wichtige Grundregeln wurden in der Literatur jedoch erarbeitet [RMC91, TFC+89].

Viele dreidimensionale Anwendungen aus dem Architektur- und CAD-Bereich oder aus dem Umfeld von Visualisierungen großer Datenmengen (z. B. aus physikalischen Simulationen) verwenden zur Navigation in und Interaktion mit der Szene besondere Hardware wie z. B. Spaceballs [3Dc] oder Datenhandschuhe [ZLB+87], oft in Verbindung mit speziellen Anzeigegeräten. Schon aufgrund des Preises solcher Spezialhardware kann sie für den Einsatz von Huginn nicht vorausgesetzt werden.

Glücklicherweise existiert aber doch ein sehr großer Bereich von Anwendungen, in dem die Navigation in dreidimensionalen Szenen mit standardmäßig vorhandener oder preiswert zu erwerbender Hardware seit Jahren gängige und erprobte Praxis ist: Spiele. Oft versuchen Spiele allerdings, die Bewegung im Raum physikalisch (mehr oder weniger) korrekt zu gestalten; dies kann Gravitationskräfte, realistisches Beschleunigungs- und Bremsverhalten oder sogar die Nachbildung von Luftströmungen beinhalten. Eine solches Verhalten wäre für die Navigation in einer Simulationsvisualisierung eher hinderlich.

Die Navigation in der 3D-Darstellung von Huginn orientiert sich an dem in Spielen oft verwendeten Schema, verzichtet aber auf eine korrekte Physiknachbildung. Die Navigation ist sowohl mit der Maus als auch mit einem Joystick oder Gamepad möglich. Eine Maus ist an jedem Rechner vorhanden, sodass ohne zusätzliche Investitionen eine Verwendung der Software stets möglich ist. Die Bedienung speziell mit einem Gamepad hat sich als etwas besser und leichter zu handhaben erwiesen; auch ein solches Eingabegerät ist für wenig Geld erhältlich. Mit allen Eingabegeräten kann der Benutzer sich entlang dreier Raumachsen frei bewegen, sowie die Kamera um jede der Achsen schwenken.

Mit der Maus wurde dies durch Klicken und Ziehen mit den drei Maustasten realisiert – je nach Taste wird um unterschiedliche Raumachsen gedreht bzw. entlang unterschiedlicher Achsen verschoben; das Mausrad erlaubt, sofern vorhanden, zusätzlich ebenfalls eine Vor-/Zurück-Bewegung. Bei der Steuerung mit Joystick oder Gamepad werden sechs Achsen für die drei Verschiebungsrichtungen und Drehachsen verwendet; eine weitere Achse erlaubt die Variation der Ablaufgeschwindigkeit der Visualisierung.

Das Auswählen von Knoten ist durch Klicken mit der Maus möglich. Durch in den Vordergrund der 3D-Darstellung eingeblendete Bedienelemente ist das Anzeigen und Verstecken von Zusatzinformationen, die Suche nach und das Anfliegen von bestimmten Knoten und die Navigation auf der Zeitachse möglich. Eine stets eingeblendete klei-

ne Übersichtskarte zeigt immer die aktuellen Knotenpositionen und erleichtert so die Orientierung. Zur leichteren Verfolgung einzelner Knoten können diese mittels verschiedenfarbiger „Fahnen“ markiert werden.

5.8 Lizensierung

Die GUI und das Vorverarbeitungsmodul – also die Teile der Software, die ausschließlich zum Zweck der ns-2-Visualisierung entwickelt wurden – können gemäß den Bestimmungen der GNU General Public Licence (GPL) [SM91] frei verwendet, weitergegeben und verändert werden.

Die verwendete V-IDS-Grafikengine unterliegt anderen Lizenzbestimmungen. Auch sie ist für den privaten, nichtkommerziellen Gebrauch ebenso wie für die nichtkommerzielle Forschung und Lehre einschließlich ihrer Quelltexte frei verfügbar. Für den kommerziellen Gebrauch sowie die Integration in eigene kommerzielle Produkte gilt dies jedoch nicht, hier muss eine entsprechende Lizenz erworben werden.

Kapitel 6

Verwendete Algorithmen

6.1 Rekonstruktion der Ereigniswarteschlange

Die Visualisierung einer Simulation soll die vom Simulator berechnete Ereignisfolge reproduzieren. Die einzelnen simulierten Ereignisse sind chronologisch im erzeugten Tracefile festgehalten. Um die Folge der Ereignisse in der Simulation zu rekonstruieren reicht es grundsätzlich also aus, das Tracefile sequenziell einzulesen. Leider genügt dies aber alleine nicht, um eine sinnvolle Darstellung zu erzeugen.

Im Simulator sind während des Simulationslaufes die Zusammenhänge zwischen den auftretenden Ereignissen vollständig bekannt. Ein Ereignis, das ein anderes auslöst, übergibt diesem alle von ihm benötigten Werte. Die gesamte zur Verarbeitung des Folgeereignisses benötigte Information liegt also stets vor.

Im Tracefile hingegen werden nur die einzelnen Ereignisse aufgeführt; beispielsweise geht aus dem Eintrag, dass ein Paket empfangen wurde, nicht direkt hervor, welcher Sendevorgang diesen Empfang verursacht hat. Die Information über den unmittelbaren Ursache-Wirkung-Zusammenhang geht verloren.

Um eine Übertragung aber durch eine grafische Verbindung zwischen Sender und Empfänger darstellen zu können, muss bekannt sein, welcher Sende- und welcher Empfangsvorgang zusammengehören. Für die Berechnung mancher statistischer Werte ist solches Wissen ebenfalls unabdingbar.

Der Ursache-Wirkung-Zusammenhang aus dem Simulator muss deshalb teilweise aus den Tracefiledaten rekonstruiert werden. Hier kann Hintergrundwissen ausgenutzt werden, um die nicht unmittelbar vorhandene Information wiederzugewinnen.

Ein entsprechendes Verfahren sollte möglichst wenige Annahmen über die Arbeitsweise der zum Einsatz kommenden Protokolle machen; immerhin ist eine zentrale Zielsetzung die Analyse von neuen Protokollen, die möglicherweise erst nach der Implementierung des hier diskutierten Werkzeugs spezifiziert werden. Die Kriterien sollen sich deshalb nur auf das einzelne Paket und nicht auf vorangegangene oder nachfolgende andere Einträge stützen.

6.1.1 Ereigniskorrelation durch MAC-Header-Lookup

Informationen, die zunächst geeignet erscheinen, den gewünschten Zusammenhang wiederherzustellen, erfüllen bei genauerer Betrachtung diese Anforderung leider nicht. Dass Empfangsvorgänge nicht einfach dem unmittelbar im Tracefile vorangegangenen Sendevorgang zugeordnet werden können, ist offensichtlich: Es können durchaus – z. B. in unterschiedlichen Gebieten – zeitlich verschränkte Übertragungsvorgänge stattfinden.

Aber auch die vom Simulator vergebene Paket-ID taugt nicht, Pakete der unteren Schichten zu korrellieren: Reine MAC-Layer-Pakete (RTS, CTS,...) und viele Routing-Layer-Pakete haben keine eigene ID. Stattdessen übernehmen sie die ID des Agent-Layer-Pakets, das sie ursprünglich auslöste. Eine eindeutige Zuordnung ist so nicht möglich.

Ein einfach zu handhabendes Kriterium, das einen Paketempfang dem zugehörigen Sendevorgang zuordnen kann, könnte ausnutzen, dass die Quelladresse im MAC-Header des empfangenen Paketes der Knoten-ID des Senders entspricht. Da die höheren Protokollschichten den MAC-Header nicht beeinflussen, sollte dies stets gegeben sein. Zusätzlich gilt auf dem MAC-Layer: Empfängt Knoten N ein Paket P von Knoten M, so ist P das zuletzt von M gesendete Paket.

Man könnte also annehmen, dass das Speichern des jeweils zuletzt gesendeten Paketes eines jeden Knotens ausreichen sollte, um einen Empfang stets dem zugehörigen Sendevorgang zuordnen zu können. Leider genügt dieses Kriterium alleine jedoch immer noch nicht, um auch alle MAC-Layer-Pakete korrekt zu erfassen: Bei diesen wird das Feld für die Quelle zum Teil nicht belegt. Eine Übersicht über die Belegung der MAC-Header-Felder in den unterschiedlichen Fällen gibt Tabelle 6.1.

Problematisch ist der Fall der ACK- und CTS-Pakete, da hier die ID des Senders nirgendwo auftaucht. Das vom MAC-Header getrennte Feld für den sendenden Knoten in der Tracefilezeile ist nur im neuen Traceformat vorhanden und dort nicht korrekt belegt, so dass auch hieraus diese Information nicht entnommen werden kann. Beim Empfang eines ACK oder CTS ist der entsprechenden Tracefilezeile tatsächlich an keiner Stelle zu entnehmen, welcher Knoten der Sender ist.

Dies ist jedoch nicht unbedingt notwendig, sofern man von einer weiteren Annahme ausgeht: ACKs können nur bei Unicast-Übertragungen von dem einen Empfänger des Datenpaketes nach dem Empfang gesendet werden und der Sender des Datenpakets kann keine Übertragung eines anderen Paketes vor dem korrekten Empfang des ACK beginnen. Analoges wie für ACK- und Datenpakete gilt für CTS und RTS. Die Annahme ist gerechtfertigt, da andernfalls die Regeln des MAC-Layer-Protokolls verletzt würden.

Unter der genannten Annahme kann ein empfangenes ACK eindeutig mit dem zuletzt an die Zieladresse abgesendeten ACK identifiziert werden, ein CTS mit dem letzten CTS. Mehrere ACK- oder CTS-Pakete können nicht gleichzeitig an denselben Knoten unterwegs sein, denn dazu müsste dieser eine zweite Datenübertragung (keine Übertragungswiederholung, diese hätte denselben Zielknoten!) begonnen haben, bevor er das erste empfangen hat. Sender- und Empfängeradresse gemeinsam genügen so – fast – immer, um ein Paket eindeutig wiederzuerkennen.

Pakettyp	**Allocation**	**Ziel**	**Quelle**	**Typfeld**
RTS	$\neq 0$	Ziel-ID	Quell-ID	0
CTS	$\neq 0$	Ziel-ID	0	0
höhere Schicht	(unterschiedlich)	Ziel-ID	Quell-ID	$\neq 0$
ACK	0	Ziel-ID	0	0

Tabelle 6.1: Belegung der MAC-Header-Felder bei unterschiedlichen Pakettypen in ns-2; Quell- und Ziel-ID beziehen sich hierbei jeweils auf den Knoten, der das jeweilige Paket tatsächlich sendet, bzw. an den es gerichtet ist. Insbes. ist also bei CTS und ACK derjenige Knoten Quelle, an den das eigentliche Datenpaket gerichtet ist.

Ein anderer – wenn auch weit weniger bedeutender – Problemfall entsteht dadurch, dass ein nicht belegtes Feld und ein Eintrag, der den Knoten mit der ID 0 (Null) bezeichnet, nicht zu unterscheiden sind. Er zeigt sich, wenn Knoten 0 ein Paket an einen Knoten M verschickt und ein anderer Knoten zeitgleich ein an M gerichtetes ACK oder CTS überträgt. In beiden Paketen ist das Feld für die Quelle im MAC-Header mit 0 und das Zielfeld mit M belegt. Deswegen werden nicht nur Quelle und Ziel, sondern der gesamte MAC-Header mit allen seinen Feldern zur Wiedererkennung verwendet.

Selbst diese Lösung lässt aber noch geringen Spielraum für eventuelle Probleme. In einem sehr speziellen Fall ist es möglich, dass die Zuordnungsregel versagt: Sendet ein Knoten K ein CTS an Knoten M, überträgt gleichzeitig der Knoten mit der ID 0 ein RTS an M und sind außerdem – aufgrund von Paketgrößen exakt der richtigen Differenz – die Allocation-Felder im MAC-Header bei beiden Paketen identisch, dann sind die Header gleich und die Pakete können nicht unterschieden werden. In dieser Situation helfen auch keine anderen der im Tracefile vorhandenen Informationen weiter, die Korrelation ist mit den vorhandenen Mitteln nicht machbar. Der geschilderte Fall ist allerdings so speziell, dass er in der Praxis kaum auftreten wird.

In allen anderen Fällen ist die Zuordnung eindeutig. Im Laufe einer Simulation können natürlich durchaus mehrere Pakete mit demselben MAC-Header auftreten. Sind jedoch die MAC-Header zweier Pakete gleich, so muss sich der Empfang eines solchen Paketes aus den genannten Gründen auf den zuletzt davor aufgetretenen Sendevorgang mit gleichem MAC-Header beziehen.

Es genügt also, eine Lookup-Tabelle anzulegen, die zu jedem MAC-Header auf den zugehörigen Sendevorgang verweist. Wird eine einen Sendevorgang betreffende Tracefilezeile verarbeitet, so wird das Paar aus MAC-Header und Sendevorgang in die Tabelle eingetragen. Ein eventuell vorhandener anderer Eintrag mit demselben MAC-Header kann dabei überschrieben werden. Beim Verarbeiten eines Empfangsvorgangs verweist der hierzu passende Tabelleneintrag auf das zugehörige Senden.

Nachdem die Darstellung einer Übertragung begonnen wurde, kann der zugehörige Tabelleneintrag entfernt werden. Später eingelesene weitere Empfangsvorgänge desselben Paketes können ohnehin nicht mehr korrekt dargestellt werden; dieser Fall kann aber – korrekte Parameter vorausgesetzt – ausgeschlossen werden (siehe hierzu Abschnitt 6.1.2).

6.1.2 Lookahead

Oft müssen während der Visualisierung einer Simulation Ereignisse, die erst zu einem späteren Zeitpunkt eintreten werden, bereits im Voraus bekannt sein. Startet beispielsweise ein Sendevorgang, so sollen sofort Übertragungskegel zu den empfangenden Knoten dargestellt werden. Die Information, welche Knoten das gesendete Paket überhaupt empfangen haben, findet sich im Tracefile jedoch erst später, nämlich in den Einträgen zum Paketempfang, also erst zum Endzeitpunkt der Übertragung. Es ist also im Gegensatz zur Durchführung der eigentlichen Simulationsberechnung nicht nur nötig, zu einer Wirkung die hervorrufende Ursache zu kennen, sondern umgekehrt müssen zur Darstellung einer Ursache auch die später eintretenden Wirkungen bekannt sein!

Um die Darstellung erzeugen zu können, muss darum ein gewisser Teil des Tracefiles vorausschauend eingelesen werden. Leider ist es nicht möglich, aus den Daten zu bestimmen, wie groß dieser sein muss, ohne hierfür alles einzulesen. Prinzipiell könnten stets noch weitere Empfangszeilen zu einem Sendevorgang folgen[1]. Da MAC-Layer-Übertragungen jedoch zeitlich beschränkt sind, reicht es aus, eine beschränkte (Simulations-) Zeitspanne im Voraus einzulesen und zu verarbeiten.

Um dem Benutzer keine unnötigen Einschränkungen durch eine feste Wahl dieser Zeitspanne (des *Lookahead*) aufzuzwingen, lässt sie sich in der GUI konfigurieren. Die Größe sollte dabei mit Bedacht gewählt werden, denn ein größerer Lookahead führt zu einem größeren Speicherbedarf zur Laufzeit (mehr Daten müssen eingelesen werden, bevor sie dargestellt und danach aus dem Speicher entfernt werden können, auch Lookup-Tabellen werden größer) und geringerer Performance (ebenfalls wegen größerer Lookup-Tabellen).

Ein zu kleiner Lookahead hingegen kann dazu führen, dass ein Sendevorgang bereits dargestellt wird, bevor der letzte zugehörige Empfang eingelesen ist. In diesem Fall ist eine korrekte Darstellung – und gegebenenfalls auch eine korrekte Berechnung von Statistiken – nicht möglich. Diese Fehlersituation kann von der Software erkannt werden, eine entsprechende Warnmeldung wird dann ausgegeben.

6.1.3 Reproduzieren der Ereigniswarteschlange

Um die Visualisierung einer Simulation zu erzeugen, wird also das zugehörige Tracefile mit einem festen Lookahead eingelesen. Bezeichne L die Länge des Lookahead in Sekunden Simulationszeit. Steht die Visualisierung beim Zeitpunkt t, so ist das Tracefile bis zum Zeitpunkt $t + L$ eingelesen.

In Arbeitsspeicher gehalten werden muss deshalb die Information des Teils des Tracefiles zwischen dem momentan dargestellten Zeitpunkt t und der Stelle $t + L$. Hierzu wird eine Struktur verwendet, die der Ereigniswarteschlange im Simulator vom Zweck

[1] Zumindest, solange nicht ein weiterer Sendevorgang mit gleichem MAC-Header folgt, vgl. Abschnitt 6.1.1. Da ein solches Paket aber nicht immer folgen muss, wäre beim Verlass nur auf dieses Kriterium das vollständige Einlesen des Tracefile im Voraus nötig, was aus Speicherplatz- wie aus Performance-Gründen nicht akzeptabel ist

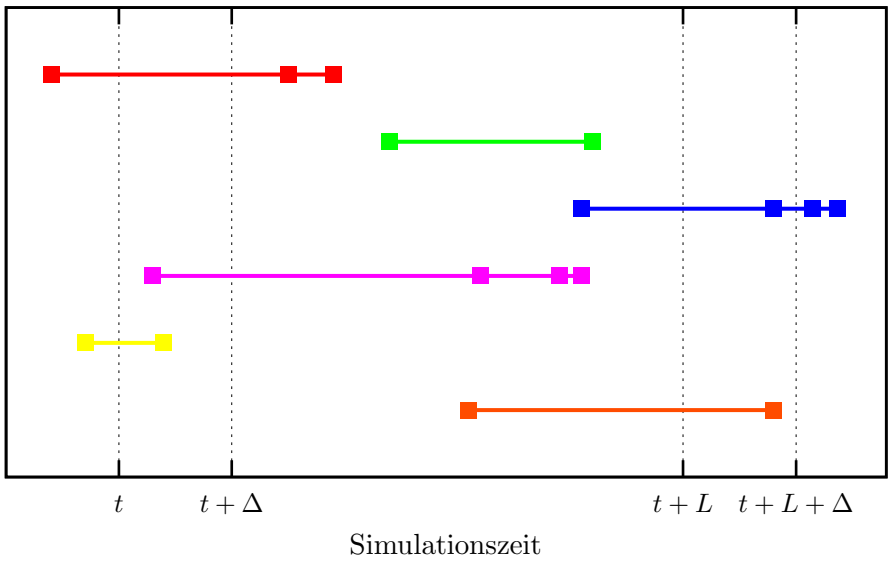


Abbildung 6.1: Stand des Einlesens des Tracefiles beim Übergang von einem Frame zum Zeitpunkt t zu einem Frame zum Zeitpunkt $t+\Delta$. Aus dem Tracefile werden die Ereignisse zwischen $t+L$ und $t+L+\Delta$ eingelesen. Die Punkte markieren die auftretenden Ereignisse, horizontale Linien verbinden zusammengehörige Ereignisse im Sinne von Ursache und Wirkung, beispielsweise also ein Sende- und die zugehörigen Empfangsereignisse.

wie auch von der Implementierung her sehr ähnlich ist. Die Visualisierungssoftware baut sozusagen die Ereigniswarteschlange des Simulators nach.

Wird der nächste Frame der Darstellung berechnet und soll dieser den Zeitpunkt $t+\Delta$ darstellen, so werden zunächst aus dem Tracefile so lange weitere Zeilen eingelesen, bis eine von ihnen ein Ereignis zu einem Zeitpunkt t_E mit $t_E > t+L+\Delta$ bezeichnet. Eingelesene Ereignisse werden mit ihrem Zeitpunkt als Schlüssel in die Warteschlange eingetragen. Werden dann vom Kopf der Warteschlange diejenigen Ereignisse ausgelesen und entfernt, die zwischen t und $t+\Delta$ aufgetreten sind, so lassen sich daraus die zwischen den Frames aufgetretenen Veränderungen der dargestellten Szene herleiten. Dieser Zusammenhang wird in Abbildung 6.1 nochmals veranschaulicht.

Es müssen auch Ereignisse in die Warteschlange eingetragen werden, die nicht zu dem Zeitpunkt auftreten, der dem momentanen Lesestand im Tracefile entspricht. Hierzu zählt z. B. das Ende einer Knotenbewegung beim Erreichen des Zielpunktes, das gleichzeitig mit deren Start eingefügt werden muss. Aus diesem Grund genügt eine einfache FIFO-Queue nicht; Ereignisse müssen effizient an beliebigen Zeitpunkten eingefügt werden können. Eine effiziente Implementierung einer Warteschlange, die den hier gegebenen Voraussetzungen entspricht, basiert auf Heaps [Knu98]. Da die Warteschlange der Visualisierung jedoch nur vergleichsweise wenige Ereignisse enthält und die Ereignisse relativ schnell wieder entnommen werden, ist eine andere Datenstruktur besser geeignet. Diese Struktur – die 2-Phasen-Queue – wird in Abschnitt 6.7 vorgestellt.

Um die Zahl der Ereignisse in der Warteschlange zusätzlich gering zu halten, werden spätere Veränderungen desselben Darstellungselements nicht in dieser globalen Warteschlange eingetragen: Nach der Verarbeitung einer Darstellungsänderung durch das Objekt, das das zugehörige Ereignis repräsentiert, wird der Zeitpunkt der nächsten Veränderung mit einem Zeitpunkt größer $t + \Delta$ bestimmt. Das Ereignisobjekt wird dann mit diesem Zeitpunkt neu in die Warteschlange eingefügt. Innerhalb eines einzelnen Ereignisobjektes ist die Darstellung zukünftig notwendiger Änderungen effizienter möglich als mit einer Ereigniswarteschlange.

Außerdem ist so nur maximal ein Verarbeitungsaufruf pro Frame und Darstellungselement nötig, der alle Veränderungen auf einmal berechnen und u. U. zusammenfassen kann. Beispielsweise kann ein Darstellungselement, das durch zwei Ereignisse zwischen t und $t + \Delta$ zuerst eingeblendet und dann im selben Frame wieder entfernt wird, gleich vollständig ausgelassen werden. Es belastet den Kommunikationskanal so nicht unnötig. Diese Technik wird zum Beispiel für die interne Repräsentation von MAC-Layer-Übertragungen verwendet. Sie findet auch bei der Beschreibung der Knotenbewegungen Anwendung: Bereits eingelesene zukünftige Bewegungen eines Knotens werden nicht in der Warteschlange verwaltet, sondern vom Knotenobjekt selbst. Dieses wiederum ist mit dem Zeitpunkt der nächsten Änderung in der Warteschlange eingetragen.

Mit diesen Veränderungen können auch Situationen, in denen viele Zeilen des Tracefiles im Voraus eingelesen werden müssen – wegen eines langen Lookahead oder vieler Ereignisse kurz nacheinander –, mit einer akzeptablen Warteschlangenlänge und damit kurzen Verarbeitungszeiten für die Darstellung einzelner Frames behandelt werden.

6.1.4 Korrelation von Agent-Layer-Paketen

Agent-Layer-Pakete werden zwischen dem Agent-Layer-Sende- und dem Agent-Layer-Empfangsereignis in der Regel von mehreren Zwischenstationen weitergeleitet. In diesen können sie auch – für eine prinzipiell nicht beschränkte Zeitdauer – vor der Weiterleitung zwischengespeichert werden. Eine Schranke für die maximale Zeit zwischen Senden und Empfangen auf dem Agent-Layer lässt sich deshalb nicht so leicht abschätzen wie auf dem MAC-Layer. Hinzu kommt, dass die Lösung, den Lookahead entsprechend weit auszudehnen, bei den üblichen Übertragungsdauern ohnehin unpraktikabel ist.

In der Folge sind bei der Darstellung eines Agent-Layer-Sendens die zugehörigen Empfangsvorgänge in der Regel nicht bekannt. Auch die Antwort auf die Frage, ob das Paket verloren gehen wird, oder vielleicht sogar mehrfach den Empfänger erreicht, ist noch offen. Deshalb ist die unmittelbare Darstellung einer Verbindung auf Agent-Ebene in der Weise, wie sie auf MAC-Ebene erfolgt, nicht möglich.

Die Korrelation von Sende- und Empfangsvorgängen für die Berechnung von Statistiken ist jedoch unproblematisch: Auf dem Agent-Layer gibt es eindeutige Paket-IDs, so dass eine darauf basierende Lookup-Tabelle genügt. Diese Tabelle kann im Gegensatz zu ihrem MAC-Layer-Pendant nicht mehr verkleinert werden, denn ein weiterer Empfang eines einmal gesendeten Agent-Layer-Paketes ist immer möglich.

Die Datenmenge ist aber vergleichsweise klein, üblicherweise zieht eine einzelne Agent-Layer-Übertragung eine Vielzahl von Aktivitäten auf den tieferen Schichten nach sich. Daher erweist sich die Lösung einer ständig wachsenden Tabelle mit einem Eintrag für jede bisher aufgetretene Paket-ID durchaus als praktikabel.

6.2 Typerkennung bei MAC-Layer-Paketen

Im neuen Traceformat ist – im Gegensatz zum alten – für reine MAC-Layer-Pakete deren Typ nicht explizit aufgeführt. Er muss stattdessen aus anderen, vorhandenen Feldern indirekt ermittelt werden. Die nachfolgend beschriebene Heuristik bewährt sich in fast allen Fällen; Schwierigkeiten ergeben sich nur beim Knoten mit der ID 0.

Tabelle 6.1, die die Belegung der MAC-Header-Felder zeigt, wurde bereits in Abschnitt 6.1.1 verwendet. Dort wurde das Wiedererkennen von MAC-Layer-Paketen für die Ereigniskorrelation diskutiert. Die Belegung kann aber auch zum Ermitteln des Pakettyps herangezogen werden.

Es genügt, den Pakettyp bei Sendevorgängen zu bestimmen. Da Empfangsereignisse mit dem zugehörigen Sendeereignis korreliert werden können, steht die dort gewonnene Information auch beim Empfang zur Verfügung. Deshalb ist die ID des Senders immer bekannt und kann für die Ermittlung verwendet werden, auch wenn sie im MAC-Header selbst nicht auftaucht. Sie steht in anderen Feldern des Sendeereignisses im Tracefile.

ACKs lassen sich stets leicht daran erkennen, dass die drei Felder für Allocation, Quelle und Typ alle 0 sind. Diese Konstellation ist bei keinem anderen Pakettyp möglich. Ebenso können Pakete aus höheren Schichten – abgesehen davon, dass die betreffenden Einträge zusätzliche Felder beinhalten und so ohnehin erkennbar sind – leicht identifiziert werden: Nur bei ihnen ist das Typfeld ungleich 0.

Für die Unterscheidung von RTS und CTS soll zunächst davon ausgegangen werden, dass der Sender des Paketes eine Knoten-ID ungleich 0 hat. Dann genügt das Betrachten des Feldes für die Quelle, um zwischen den beiden Pakettypen unterscheiden zu können. Ist jedoch Knoten 0 der Sender, dann kann nicht ermittelt werden, ob ein RTS oder ein CTS verschickt wurde, denn in beiden Fällen ist als Quelle für das Paket 0 eingetragen. Solche Pakete, für die eine Entscheidung nicht möglich ist, erhalten in Huginn den Pakettyp „RTS or CTS“.

Abbildung 6.2 veranschaulicht das Vorgehen des beschriebenen Algorithmus grafisch.

6.3 Statistik-Berechnungen

Zu den einzelnen Knoten werden statistische Werte während der Visualisierung berechnet und mitgeführt. Hierzu sind Berechnungsverfahren nötig, die bei Eintreffen eines neuen Ereignisses oder bei Änderungen eines Zustandswertes (siehe Abschnitt 5.3) die Aktualisierung des Wertes der Aggregatfunktion effizient ermöglichen.

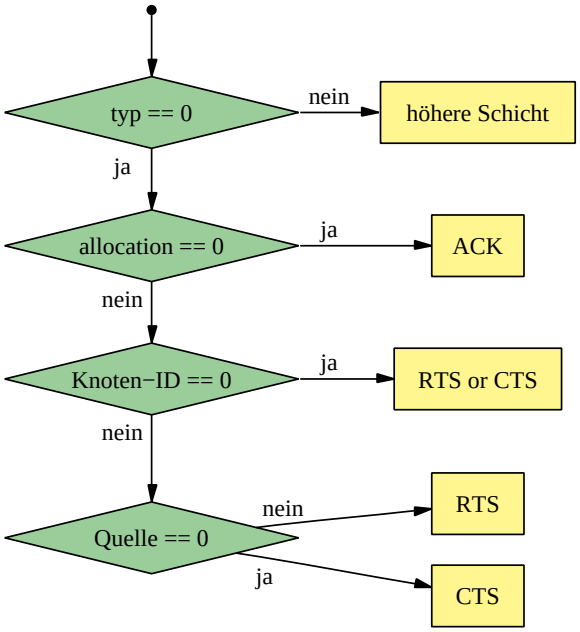


Abbildung 6.2: Vorgehensweise zur Bestimmung des Typs von MAC-Layer-Paketen

Die Module zur Berechnung unterschiedlicher Aggregate können in der GUI frei zusammengestellt werden; die tatsächliche Berechnung während der Visualisierung findet im Vorverarbeitungsmodul statt. Hier wird für jedes zu berechnende Aggregat in jedem Knoten eine Instanz einer Klasse erzeugt, die die Berechnungsvorschrift für das betreffende Aggregat implementiert. Es gibt also beispielsweise Klassen für „Summe“, „letzter aufgetretener Wert“ oder „globales Minimum“. Diese Objekte enthalten für ein bestimmtes Aggregat in einem bestimmten Knoten die jeweils notwendigen Informationen, um bei Änderungen der Ausgangswerte und beim Anzeigen eines neuen Frames den neuen Wert des Aggregates zur Verfügung stellen zu können.

6.3.1 Zustands- und Ereignisaggregate

Ereignisaggregate verarbeiten Parameter, die von zeitlich nicht ausgedehnten, einzelnen Ereignissen stammen. Dies könnte z. B. die Summe der Größen aller von einem Knoten gesendeten Pakete sein: Bei jedem Eintritt eines Ereignisses vom Typ MACSendStart soll die zugehörige Paketgröße zur bisherigen Summe addiert werden.

Zustandsaggregate hingegen verarbeiten die zeitliche Entwicklung von Knotenzuständen, die kontinuierlich definiert sind. Ein einfaches Beispiel wäre hier die maximal bisher erreichte x-Position eines Knotens.

Knotenzustände können sich ändern. Dies kann entweder – für fest in die Software integrierte Zustandswerte – durch Eintreten bestimmter Bedingungen im Knoten geschehen, oder es kann durch explizite Einträge im Tracefile durch den Benutzer eine Zustandsänderung angestoßen werden.

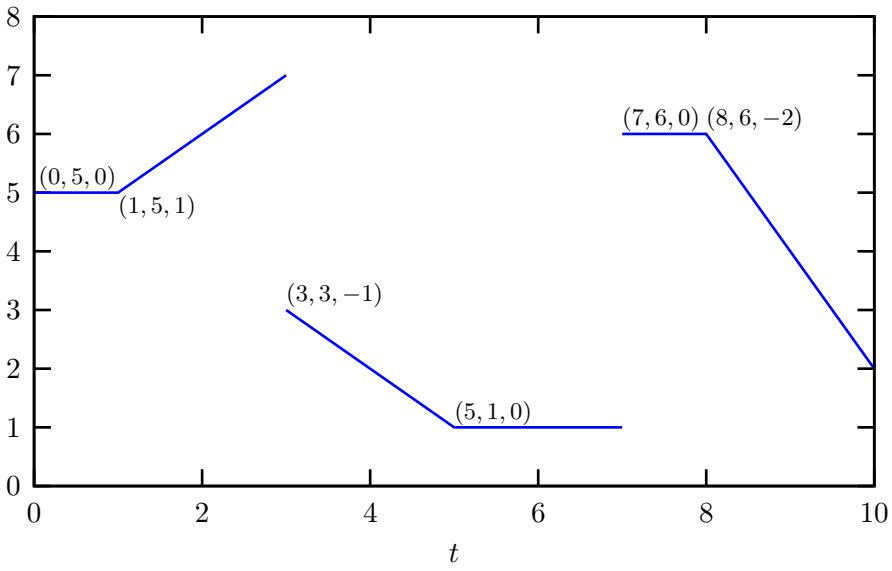


Abbildung 6.3: Exemplarischer Verlauf eines Zustandswertes über die Simulationszeit. An den Stellen einer Zustandsänderung sind jeweils die diesen Verlauf erzeugenden Kontrolltupel der Form (Zeitpunkt, Wert, Änderungsrate) angegeben.

Könnte nur zu bestimmten Zeitpunkten der ab diesem Moment gültige Wert eines Zustands gesetzt werden, so wäre die Beschreibung der Zustandsentwicklung auf Treppenfunktionen eingeschränkt. Viele sinnvolle Anwendungen sind damit nicht umsetzbar. Für das oben genannte Beispiel der maximalen x-Position etwa reicht diese Vorgehensweise – wegen der kontinuierlichen Bewegung der Knoten – nicht aus.

Um das Einsatzgebiet zu erweitern ist es bei Zustandsänderungen möglich, nicht nur den momentan gültigen Wert beliebig zu setzen, sondern auch eine ab diesem Moment anzuwendende Änderungsrate. Hiermit kann also auch eine Art „erste Ableitung“ der Zustandsfunktion übergeben werden. Damit ist dann nicht nur die Beschreibung von Treppenfunktionen, sondern auch die von beliebigen, stückweise linearen Entwicklungen des Wertes eines Zustands – auch mit Sprüngen – möglich. Hierzu müssen nur an den Knick- und Sprungstellen der Funktion die jeweiligen Werte und ab diesem Punkt gültigen Änderungsraten bekannt sein.

Abbildung 6.3 zeigt einen exemplarischen Verlauf eines Zustandswertes und die diesen Verlauf erzeugenden 3-Tupel (Zeitpunkt, Wert, Änderungsrate).

Für Zustände, deren Werte keine reellen Zahlen sind, ist selbstverständlich auch ein Setzen des Wertes ohne Angabe einer Änderungsrate möglich.

6.3.2 Objektorientierte Umsetzung

Die Schnittstelle der Klassen für Aggregatfunktionen und deren Kommunikationsschema sind äußerst minimalistisch gehalten. Jede der Aggregatklassen stellt nach außen hin

nur zwei Methoden zur Verfügung. Diese unterscheiden sich für Ereignisaggregate und Zustandsaggregate.

Für Ereignisaggregate sind die folgenden Methoden definiert:

- Eine Methode *process*, die beim Auftreten von Ereignissen aufgerufen wird; dieser Methode werden der entsprechende Wert, der in das Aggregat einfließen soll, sowie der Zeitpunkt des Ereigniseintritts übergeben
- Eine Methode *getValue*, die bei der Berechnung jedes neuen Frames aufgerufen wird. Dieser Methode wird der Zeitpunkt des anzuzeigenden Frames übergeben.

Das leicht veränderte Interface für Zustandsaggregate sieht folgendermaßen aus:

- Eine Methode *stateChange* verarbeitet eine eintretende Veränderung eines Zustandes. Gleichbleibende Zustände müssen nicht speziell gemeldet werden. Der Methode werden der Zeitpunkt der Änderung, der neue Wert und die ab diesem Moment gültige Änderungsrate übergeben.
- *getValue* ist analog zu den Ereignisaggregaten definiert und liefert zu einem übergebenen Zeitpunkt den aktuellen Wert des Aggregates als Rückgabewert.

Die Aggregat-Klassen werden durch Aufrufe dieser beiden Methoden von außen angesprochen; selbst senden sie keine Signale an andere Objekte. Die Implementation wird vereinfacht, indem eine chronologische Ordnung der Aufrufe gefordert wird: Nachdem für ein Aggregat-Objekt *process*, *stateChange* oder *getValue* für einen bestimmten Zeitpunkt aufgerufen wurde, darf kein anderer Aufruf einer dieser Methoden mit einem früheren Zeitpunkt mehr folgen. Außerdem darf auf einen Aufruf von *getValue* kein Aufruf einer der anderen Methoden mit gleichem Zeitpunkt mehr folgen.

6.3.3 Berechnungsalgorithmen

Auf Basis der geforderten chronologischen Ordnung lassen sich für die meisten Aggregate sehr einfache Berechnungsvorschriften finden. Im Folgenden sollen einige konkrete Beispiele implementierter Aggregatfunktionen betrachtet werden.

Ereignisaggregate

Beim wahrscheinlich einfachsten Ereignisaggregat – dem Zähler für die Anzahl eingetretener Ereignisse – genügt es, beim Eintreten eines Ereignisses einen im Objekt gehaltenen Zähler um 1 zu erhöhen und beim Abfragen des Wertes mittels *getValue* den momentanen Zählerstand zurückzugeben. Nur unwesentlich aufwändiger wird das Vorgehen für das Summen-Aggregat – hier wird nicht ein Zähler um 1 erhöht, sondern zur bisherigen Summe der neue Wert addiert.

Das arithmetische Mittel über alle eingetretenen Ereignisse lässt sich durch Kombination beider Verfahren bestimmen: Werden die Anzahl der Ereignisse und die Summe mitgeführt, kann beim Aufruf von *getValue* jederzeit der Quotient $\frac{\text{Summe}}{\text{Anzahl}}$ berechnet und zurückgegeben werden.

Ein Beispiel für ein etwas komplexeres Ereignisaggregat ist der Aktivitätsmesser. Dieser wird durch den Benutzer mit einer Halbwertszeit h parametrisiert. Beim Eintreffen eines Ereignisses soll sein Wert um 1 erhöht werden, außerdem soll aber der vorhandene Wert kontinuierlich abfallen, so dass er sich stets nach einer Halbwertszeit h halbiert hat.

In einem Aktivitätsmesser-Objekt werden zwei Werte mitgeführt: Der Zeitpunkt t der letzten Aktualisierung des Wertes und der zu diesem Zeitpunkt gültige Wert v. Trifft ein neues Ereignis zum Zeitpunkt t' ein, so wird der neue Wert nach folgender Formel berechnet:

$$v' = \frac{v}{2^{\frac{t'-t}{h}}} + 1 \tag{6.1}$$

Das neue Tupel (t', v') ersetzt das vorhandene (t, v). Durch die Division vollzieht die Berechnung zunächst das Abfallen des Wertes zwischen t und t' nach. Danach wird für das neu eingetroffene Ereignis der Wert um 1 erhöht.

Beim Abfragen des Wertes zum Zeitpunkt t' mittels *getValue* wird der aktuelle Wert in folgender Weise berechnet:

$$\frac{v}{2^{\frac{t'-t}{h}}} \tag{6.2}$$

Auch hier wird durch die Division das Abfallen seit dem Zeitpunkt des letzten eingetroffenen Ereignisses berücksichtigt.

Zustandsaggregate

Wegen der notwendigen Verarbeitung der Änderungsrate sind Zustandsaggregate etwas komplexer als die entsprechenden Ereignisaggregate. Das einfachste Zustandsaggregat berechnet den momentanen Wert eines Zustands. Hierzu werden die folgenden drei Werte mitgeführt:

- Der Zeitpunkt t des letzten expliziten Setzens des Wertes
- Der zum Zeitpunkt t gültige Wert v
- Die ab dem Zeitpunkt t gültige Änderungsrate r

Beim Setzen des Zustandes mittels *stateChange* werden die bisher gespeicherten Werte durch die neuen ersetzt.

Bei der Abfrage des Wertes zum Zeitpunkt t' kann der Rückgabewert nach folgender Formel bestimmt werden:

$$v + r(t' - t) \tag{6.3}$$

Etwas umfangreichere Berechnungen sind für das Bilden des Integrals über den Wert eines Zustandes nötig. Mitgeführt werden hier vier Werte. Die ersten drei entsprechen denen beim Aggregat für den momentanen Wert. Neu hinzu kommt hier noch das Integral über das Intervall $[0, t]$, hier bezeichnet mit l.

Beim Eintreffen einer Zustandsänderung zum Zeitpunkt t' mit Wert v' und Änderungsrate r' wird das Integral l' bis zum Zeitpunkt t' berechnet:

$$l' = l + (t' - t) \cdot \left(v + \frac{r(t' - t)}{2} \right) \tag{6.4}$$

Danach werden die gespeicherten Werte t, v, r und l durch ihre jeweiligen neuen Pendants ersetzt.

Die Berechnungsformel (6.4) ergibt sich aus folgendem Zusammenhang, wobei $f(t)$ für den Wert des Zustands zum Zeitpunkt t stehe:

$$\begin{aligned}
l' = \int_0^{t'} f(x)\,dx &= \int_0^{t} f(x)\,dx + \int_t^{t'} f(x)\,dx \\
&= l + \int_t^{t'} \left(v + r(x - t)\right)\,dx \\
&= l + \left[vx + \frac{rx^2}{2} - rtx \right]_{x=t}^{x=t'} \\
&= l + vt' + \frac{rt'^2}{2} - rtt' - vt - \frac{rt^2}{2} + rt^2 \\
&= l + v(t' - t) + \frac{r}{2}\left(t'^2 - 2tt' + t^2\right) \\
&= l + v(t' - t) + \frac{r}{2}(t' - t)^2 \\
&= l + (t' - t) \cdot \left(v + \frac{r(t' - t)}{2} \right)
\end{aligned} \tag{6.5}$$

Bei der Abfrage des aktuellen Wertes kann analog vorgegangen werden. Der Wert zum Zeitpunkt t' ergibt sich ebenfalls wie in (6.4).

6.4 Bestimmung der Skalierungsfunktionen

Der Benutzer gibt in der GUI für lineare wie auch für logarithmische Skalierungen nur je zwei Kontrollpunkte vor. Dies ist ein für den Anwender intuitiver und leicht verständlicher Weg, die gewünschte Skalierung zu erhalten. Aus den vorgegebenen Kontrollpunkten (a, x) und (b, y) können die entsprechenden Skalierungsfunktionen bestimmt werden.

6.4.1 Lineare Skalierung

Für eine lineare Skalierung müssen die gegebenen Kontrollpunkte die Voraussetzung $a \neq b$ erfüllen. Die Funktion hat die allgemeine Form

$$l(\alpha) = m\alpha + c \tag{6.6}$$

Die Parameter m und c müssen aus den Kontrollpunkten berechnet werden. Dies muss nicht bei jeder Auswertung der Skalierung geschehen, es genügt eine einmalige Berechnung zu Beginn der Visualisierung. Die Steigung m kann bestimmt werden zu:

$$m = \frac{y - x}{b - a} \tag{6.7}$$

Duch einfaches Einsetzen und Auflösen ergibt sich c dann zu:

$$c = x - ma \tag{6.8}$$

6.4.2 Logarithmische Skalierung

Analog kann man für die logarithmische Skalierung vorgehen. Es müssen ebenfalls a, b, x und y vorgegeben sein, hier allerdings mit den schärferen Nebenbedingungen $a, b > 0$, $a \neq b$ und $x \neq y$.

Die logarithmische Skalierungsfunktion hat die allgemeine Form

$$l(\alpha) = \log_\beta(\alpha) + s \tag{6.9}$$

Es können zu Beginn der Visualisierung folgende Werte vorausberechnet werden:

$$k := \frac{\ln\left(\frac{b}{a}\right)}{y - x} \tag{6.10}$$

$$s := x - \frac{\ln a}{k} \tag{6.11}$$

Die logarithmische Skalierungsfunktion hat dann folgende Form:

$$l(\alpha) := \frac{\ln \alpha}{k} + s \tag{6.12}$$

Diese Funktion geht durch die Kontrollpunkte (a, x) und (b, y):

$$\begin{aligned} l(a) &= \frac{\ln a}{k} + x - \frac{\ln a}{k} = x \\ l(b) &= \frac{\ln b}{k} + x - \frac{\ln a}{k} = \frac{\ln \frac{b}{a}}{k} + x = \frac{\ln \frac{b}{a}}{\frac{\ln \frac{b}{a}}{y-x}} + x = (y - x) + x = y \end{aligned} \tag{6.13}$$

Durch Einsetzen von k und s lässt sich (6.12) auch auf die in Formel (6.9) verwendete Form bringen und die Basis β bestimmen:

$$\begin{aligned} l(\alpha) &= \frac{\ln \alpha}{k} + s \\ &= \frac{(\ln \alpha)(y-x)}{\ln\left(\frac{b}{a}\right)} + s \\ &= (y-x)\log_{\frac{b}{a}}(\alpha) + s \\ &= \log_{\sqrt[(y-x)]{\frac{b}{a}}}(\alpha) + s \end{aligned} \tag{6.14}$$

Die Basis β ist also:

$$\beta = \sqrt[(y-x)]{\frac{b}{a}} \tag{6.15}$$

6.5 Checkpoint-Index

Das Betrachten der Visualisierung einer Simulation wird in der Regel nicht darin bestehen, einmalig den gesamten Verlauf vom Anfang bis zum Ende zu verfolgen. Stattdessen sollen in der Praxis oft Teile übersprungen, andere vielleicht mehrfach betrachtet oder nochmals langsamer wiederholt werden – kurz: Eine freie Navigation auf der Zeitachse ist notwendig.

Um das Springen auf der Zeitachse – gleichgültig ob vorwärts oder rückwärts – zu ermöglichen, müssen einige Datenstrukturen in die Form gebracht werden können, die dem angesprungenen Zeitpunkt entspricht. Darunter sind die 3D-Szene selbst, die internen Parameter und momentanen Werte aller Aggregatobjekte für die Statistikberechnungen und die Position im Tracefile.

All das lässt sich durch den *Zustand* des in Ruby implementierten Teils der Software erfassen bzw. – im Falle der momentan sichtbaren Szene – aus diesem rekonstruieren. Der Zustand sei in diesem Zusammenhang die Gesamtheit aller im Programm vorhandenen Objekte, deren Eigenschaften und Beziehungen zueinander. Ausgehend von einem identischen Zustand läuft das Programm also bei gleichen Eingaben stets in exakt der gleichen Form ab. Um einen Zeitpunkt in der Visualisierung gezielt anspringen zu können genügt es, den zugehörigen Zustand zu erzeugen.

6.5.1 Checkpoints als gespeicherte Zustände

Die reflexiven Sprachkonstrukte in Ruby erleichterten bereits das Einbinden benutzergeschriebenen Filtercodes in das Vorverarbeitungsmodul (siehe Abschnitt 5.4). Dieses Sprachfeature kann auch genutzt werden, um Kopien von vollständigen Objekthierarchien samt aller Instanzvariablen und Verweisen auf andere Objekte zu erstellen, ohne

hierfür die vorkommenden Klassen kennen zu müssen. Die Objekte in der Kopie haben dann die gleichen Eigenschaften und identische Beziehungen zueinander wie die Objekte der ursprünglichen Objekthierarchie. Es ist damit möglich, den Zustand zu einem beliebigen Zeitpunkt zu sichern, wenn ein Objekt existiert, das als Einstiegspunkt für eine solche Kopie geeignet ist. Diese Bedingung erfüllt ein Objekt O, wenn

1. von O aus alle anderen Objekte, die Teil des zu sichernden Zustandes sind (deren Eigenschaften also mitgesichert werden sollen), direkt oder indirekt erreichbar sind und
2. kein Objekt, das nicht Teil des Zustandes sein soll, von O aus direkt oder indirekt erreichbar ist

Zu den Objekten, die nicht in den Zustand einfließen sollen, gehören einige sehr zentrale Strukturen, die beispielsweise den Gesamtablauf der Visualisierung oder die Kameraposition verwalten und deshalb nicht wiederhergestellt werden sollen.

Ein *Checkpoint* sei eine solche vollständige Kopie der zustandsrelevanten Objekthierarchie. Durch Wiederherstellen eines Checkpoints – erneutes Kopieren der darin gesicherten Objekthierarchie und Verwenden dieser Kopie als aktuellen Zustand – kann ein einmal festgehaltener Zeitpunkt jederzeit wieder angesprungen werden.

Ein Vorteil dieser Lösung ist der hohe Abstraktionsgrad, der Anpassungen des Mechanismus zum Erstellen von Kopien an neue Programmfunktionalität, neue Klassen und neue Zusammenhänge weitgehend unnötig macht. Das Verfahren integriert sich auch nahtlos mit vom Benutzer erstellten Codefragmenten, ohne hierfür Anpassungen des Algorithmus zu erfordern.

6.5.2 Zeitpunkt und Häufigkeit der Checkpointerstellung

Ein Checkpoint stellt je nach dem Zeitpunkt, den er repräsentiert, eine nicht unerhebliche Datenmenge dar. Praktische Versuche haben Größen im Bereich zwischen 20 und 300 Kilobyte als typische Werte ergeben. Der Großteil hiervon besteht aus den im Lookahead vorausschauend eingelesenen Tracefiledaten. Die Größe der Checkpoints steigt daher auch mit größerem Lookahead stark an.

Bei dieser Größe kann nicht für jeden Zeitpunkt, der möglicherweise angesprungen werden soll, ein Checkpoint vorgehalten werden. Dies ist aber auch nicht notwendig: Zum Anspringen eines Zeitpunktes kann auch der nächstgelegene Checkpoint vor dem Zeitpunkt wiederhergestellt und dann der fehlende Abschnitt aus dem Tracefile eingelesen werden.

Je weiter der Checkpoint entfernt ist, desto größer ist dann der Aufwand zum Anspringen des Zeitpunktes. Der Aufwand hängt dabei jedoch nicht vom zeitlichen Abstand ab, sondern von der Anzahl der Tracefilezeilen, die verarbeitet werden müssen. Um jeden Zeitpunkt mit vertretbarem Aufwand anspringen zu können, sollten die Checkpoints also möglichst gleichverteilt über die Tracefilezeilen sein.

Das gesamte Tracefile zu Beginn einzulesen und zu verarbeiten, um Checkpoints anzulegen, würde den Start der Visualisierung zu sehr verzögern und so die Benutzbarkeit des Programmes stark beeinträchtigen. Deshalb werden Checkpoints parallel zum Lauf der Visualisierung angelegt, wenn ein Tracefileabschnitt zum ersten Mal verarbeitet wird. Das bedeutet insbesondere auch, dass beim Springen nach vorne in einen bislang nicht verarbeiteten Abschnitt des Tracefiles dieser neue Bereich eingelesen und verarbeitet werden muss. Dabei werden dann Checkpoints angelegt, so dass danach schnell navigiert werden kann.

Angelegte Checkpoints sollen so wenig Speicher wie möglich belegen. Deswegen soll die enthaltene Information auf das nötige Minimum beschränkt werden. Das bedeutet vor allem, dass der im Checkpoint enthaltene vorausschauend eingelesene Teil des Tracefiles so klein wie möglich – also nicht größer als der gewählte Lookahead – gehalten werden soll.

Tatsächlich ist beim Anlegen eines Checkpoints aber oft ein größerer Abschnitt im Voraus eingelesen: Befinden wir uns am Zeitpunkt t und möchten den nächsten Frame zum Zeitpunkt $t + \Delta$ berechnen, so wird zunächst das Tracefile bis zum Punkt $t + L + \Delta$ eingelesen (s. Abschnitt 6.1.2). Wird dabei für den Zeitpunkt $t + \alpha\Delta$, $0 < \alpha \leq 1$ ein Checkpoint angelegt, so ist in diesem Moment der Zeitraum $[t, t + L + \alpha\Delta]$ eingelesen – wegen $t + L + \alpha\Delta > t + L$ ist das mehr als notwendig!

Der erstellte Checkpoint kann jedoch verkleinert werden, indem in der angelegten Kopie ein (nicht angezeigter) Frame für den Zeitpunkt $t + \alpha\Delta$ berechnet wird. Nach dessen Berechnung können die Informationen für den Zeitraum $[t, t + \alpha\Delta]$ entfernt werden, so dass nur noch der Bereich $[t + \alpha\Delta, t + L + \alpha\Delta]$ im Checkpoint verbleibt. Dieser hat minimale Größe.

6.5.3 Anpassen der Checkpointdichte

Um schnell im Tracefile springen zu können, sollten so viele Checkpoints wie möglich verfügbar sein. Der hierfür verfügbare Speicher beschränkt jedoch ihre Anzahl. Deshalb wurde ein Verfahren entworfen, das während der Visualisierung Checkpoints mit zunächst hoher Frequenz anlegt. Wenn der eingelesene Tracefileabschnitt und damit die Menge an Checkpoints wächst, werden vorhandene Checkpoints entfernt und so die Dichte der Checkpoints verringert. Außerdem wird die Häufigkeit des Anlegens neuer Checkpoints gesenkt.

Im Folgenden wird dieses Verfahren zum Anpassen der Checkpointdichte genauer beschrieben.

Ob nach dem Einlesen der n-ten Tracefilezeile ein Checkpoint c_n erstellt wird, bestimmt der momentan gültige Checkpointabstand d. Der Checkpoint c_n wird angelegt, wenn $n \mod d = 0$. Zu Beginn einer Visualisierung wird d mit dem Wert 1000 initialisiert, es wird also immer nach 1000 eingelesenen Zeilen ein Checkpoint angelegt.

Die Gesamtzahl der Checkpoints wird durch eine Maximalzahl von Checkpoints m begrenzt. Erreicht die Gesamtzahl der gespeicherten Checkpoints diese Grenze – dies ge-

schieht erstmals nach dem Einlesen der $m \cdot 1000$-sten Zeile –, so können keine zusätzlichen Checkpoints mehr angelegt werden. Deshalb wird Platz geschaffen, indem vorhandene Checkpoints durch neue ersetzt werden. Dabei wird eine Gleichverteilung der Checkpoints über den gesamten verarbeiteten Tracefile-Bereich angestrebt.

Steigt die Zahl der Checkpoints auf m an, so wird der Checkpointabstand d verdoppelt; beim ersten Erreichen dieser Grenze wird also ab sofort nur noch alle 2000 statt zuvor alle 1000 Zeilen ein Checkpoint angelegt. Kommen nun neue Checkpoints hinzu, so wird für jeden neu eingefügten Checkpoint ein früher angelegter entfernt, dessen Zeilenposition nicht durch das neue, verdoppelte d teilbar ist. Existieren irgendwann keine solchen Checkpoints mehr, wird erneut d verdoppelt und so wiederum die Hälfte der vorhandenen Checkpoints zum Ersetzen freigegeben. Der Abstand zweier benachbarter Checkpoints beträgt also zu jedem Zeitpunkt mindestens $\frac{d}{2}$ und maximal d.

Da die nach einer Abstandsanpassung nicht mehr zum neuen Abstand d passenden Checkpoints nicht sofort entfernt, sondern nur quasi zum Überschreiben freigegeben werden, wird kein Checkpoint früher als unbedingt notwendig gelöscht. Jeder Checkpoint bleibt stattdessen so lange wie möglich erhalten, denn auch nach dem Vergrößern von d ist die in ihm enthaltene Information genauso wertvoll wie zuvor.

Ein Beispiel soll die Vorgehensweise nochmals verdeutlichen. Zu Beginn sei noch nichts eingelesen, der anfängliche Abstand der Checkpoints d betrage 1000 Zeilen. Die Maximalzahl der gespeicherten Checkpoints m sei in diesem Beispiel 10.

Nach 10 000 eingelesenen Zeilen existieren die folgenden Checkpoints:

$$1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000$$

Die Maximalzahl an Checkpoints ist jetzt erreicht, d wird deshalb auf 2000 verdoppelt. Nach 11000 Zeilen wird also kein weiterer Checkpoint angelegt. Nach 12000 Zeilen ist jedoch wieder $n \mod d = 0$, also wird ein neuer Checkpoint erstellt. Er ersetzt den ersten vorhandenen Checkpoint, dessen Zeilenposition nicht durch 2000 teilbar ist. Die neue Liste der Checkpoints ist damit:

$$2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 12000$$

In gleicher Weise werden nach und nach auch die Checkpoints bei 3000, 5000, 7000 und 9000 ersetzt, so dass sich nach 20000 eingelesenen Zeilen folgendes Bild ergibt:

$$2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000$$

Nun haben alle Checkpoints wieder den Abstand d zu ihren Nachbarn, es sind keine austauschbaren Checkpoints mehr vorhanden. Darum wird d wieder verdoppelt, ab sofort werden Checkpoints im Abstand von 4000 Zeilen erzeugt. Hierzu werden jetzt nacheinander die Checkpoints bei 2000, 6000, 10000, 14000 und 18000 ausgetauscht. Das Verfahren setzt sich bis zum Ende des Tracefiles so fort.

Abbildung 6.4 zeigt die Position der Checkpoints nochmals grafisch über den Verlauf des Einlesens des Tracefiles.

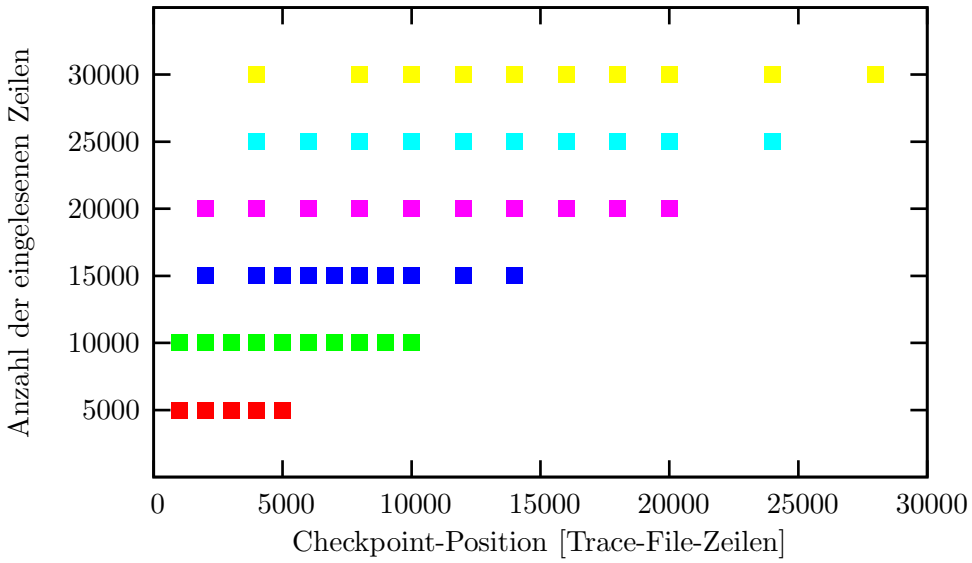


Abbildung 6.4: Position der Checkpoints nach unterschiedlichen Anzahlen von eingelesenen Zeilen; hier wurden 1000 als Startwert für d und $m = 10$ gewählt

6.5.4 Analyse des Algorithmus

Nun soll noch etwas genauer untersucht werden, wie gut der vorgestellte Checkpoint-Index-Algorithmus die zur Verfügung stehenden Kapazitäten nutzt. Angenommen, es können maximal m Checkpoints im Speicher gehalten werden. Sei $r_n(k)$ die Anzahl der Zeilen, die bei einem Sprung zu Zeile k bei insgesamt n eingelesenen und gemäß dem Checkpoint-Index-Algorithmus indizierten Zeilen noch verarbeitet werden müssen ($k \leq n$). Dann lässt sich der Erwartungswert $E[r_n]$ berechnen, wie viele Zeilen bei einem beliebigen, bezogen auf die Zeilennummer gleichverteilten Sprung in den indizierten Bereich des Tracefiles noch eingelesen werden müssen.

Die Annahme der Gleichverteilung bezüglich des Sprungziels ist hier gerechtfertigt, da über die Verteilung der Ereignisse und damit der Zeilenzahl über die Zeitachse in der visualisierten Simulation nichts bekannt ist.

Es seien m Checkpoints gesetzt bei insgesamt n eingelesenen Zeilen. Es existieren dann $n+1$ mögliche Einsprungpunkte, einer vor der ersten Zeile (hier bezeichnet mit 0) sowie je einer hinter jeder eingelesenen Zeile (bezeichnet durch die jw. Zeilennummer). Die m Checkpoints unterteilen die Menge $\{0, \ldots, n\}$ aller möglichen Einsprungpunkte in $m+1$ disjunkte Teilbereiche $I_0, \ldots, I_m$. Um eine Zeile im Bereich $I_i, i > 0$ anzuspringen würde der i-te Checkpoint verwendet werden, für I_0 würde ab dem Anfang des Tracefiles neu eingelesen.

Bei – ohnehin nicht sinnvollen und nicht vorkommenden – Checkpoints am Rand oder mehrfachen Checkpoints an der gleichen Stelle kann für diese Analyse von leeren zugehörigen Bereichen ausgegangen werden, um Spezialfälle zu vermeiden.

Die Größe $|I_i|$ eines Teilbereiches sei die Anzahl der möglichen Einsprungpunkte, die in diesen Teilbereich fallen. Dann gilt:

$$\begin{aligned} &\forall i \in \{0, \ldots, m\} : |I_i| \geq 0 \\ &\sum_{i=0}^{m} |I_i| = n + 1 \end{aligned} \tag{6.16}$$

Auch innerhalb eines Teilbereiches kann für die Verteilung der Zeilen eine Gleichverteilung angenommen werden. Der Erwartungswert für die Anzahl der bei einem Sprung in den i-ten Bereich noch einzulesenden Zeilen ergibt sich so zu:

$$E[r_n(k)|k \in I_i] = \begin{cases} 0 & \text{wenn } I_i = \emptyset \\ \frac{1}{|I_i|} \sum_{j \in I_i} (j - \min(I_i)) & \text{sonst} \end{cases} \tag{6.17}$$

Dies lässt sich vereinfachen zu:

$$E[r_n(k)|k \in I_i] = \frac{|I_i| - 1}{2} \tag{6.18}$$

Wegen der angenommenen Gleichverteilung lässt sich der gesamte Erwartungswert für r_n über alle Intervalle als gewichtetes arithmetisches Mittel der Erwartungswerte der Teilbereiche ausdrücken:

$$\begin{aligned} E[r_n] &= \frac{1}{n+1} \sum_{i=0}^{m} |I_i| \cdot E[r_n(k)|k \in I_i] \\ &= \frac{1}{n+1} \sum_{i=0}^{m} |I_i| \cdot \frac{|I_i| - 1}{2} \\ &= \frac{1}{2(n+1)} \sum_{i=0}^{m} \left(|I_i|^2 - |I_i| \right) \\ &= \frac{1}{2(n+1)} \left(\sum_{i=0}^{m} |I_i|^2 - \sum_{i=0}^{m} |I_i| \right) \\ &= \frac{1}{2(n+1)} \left(\sum_{i=0}^{m} |I_i|^2 - (n+1) \right) \\ &= \frac{1}{2(n+1)} \sum_{i=0}^{m} |I_i|^2 - \frac{1}{2} \end{aligned} \tag{6.19}$$

Die Positionen der Checkpoints und damit die Teilbereiche I_i sind für gegebenes n für den Checkpoint-Index-Algorithmus bekannt. $E[r_n]$ kann also bestimmt werden.

Für eine Bewertung der so berechneten Werte bietet sich ein Vergleich mit dem theoretischen Optimum für $E[r_n]$ an. Diese Schranke würde erreicht, wenn die m insgesamt

zur Verfügung stehenden Checkpoints für jedes n optimal verteilt wären. Die optimale Verteilung liegt vor, wenn die Summe $\sum_{i=0}^{m} |I_i|^2$ aus Formel (6.19) minimal wird.

Es zeigt sich, dass das Minimum unter Berücksichtigung der Nebenbedingungen aus (6.16) für eine Aufteilung in $m+1$ gleichgroße Bereiche erreicht wird, also für

$$\forall i, j \in \{0, \ldots, m\} : |I_i| = |I_j| = \frac{n+1}{m+1} \tag{6.20}$$

Der Beweis hierfür findet sich in Anhang B.

Ein Verfahren, welches diese optimale Verteilung stets gewährleisten wollte, wäre jedoch zwangsläufig aus anderen Gründen äußerst ineffizient: Die Positionen der Checkpoints müssten sich hierfür ständig verändern, was ein Verschieben von Checkpoints in bereits indizierten Bereichen nötig macht. Die Notwendigkeit zu einem solchem Nacharbeiten würde die Geschwindigkeitsgewinne durch das Indizieren ad absurdum führen.

Da im interessanten Wertebereich $n \gg m$ gilt, kann davon ausgegangen werden, dass näherungsweise $n+1$ durch $m+1$ teilbar ist bzw. dass die Abweichungen durch Rundungseffekte minimal sind.

Der ideale Erwartungswert $E[\bar{r}_n]$ errechnet sich dann zu:

$$\begin{aligned} E[\bar{r}_n] &= \frac{1}{2(n+1)} \sum_{i=0}^{m} \left(\frac{n+1}{m+1} \right)^2 - \frac{1}{2} \\ &= \frac{1}{2(n+1)} \cdot (m+1) \cdot \frac{(n+1)^2}{(m+1)^2} - \frac{1}{2} \\ &= \frac{n+1}{2(m+1)} - \frac{1}{2} \end{aligned} \tag{6.21}$$

Abbildung 6.5 vergleicht diesen Idealwert für $m = 10$ und einen Startwert für d von 1000 mit dem Verhalten des vorgestellten Checkpoint-Index-Algorithmus und der Zahl der Zeilen, die ohne Verwendung von Checkpoints im Mittel eingelesen werden müsste. Es wird deutlich, dass die erreichten Werte sich stets in der Nähe der Ideallinie bewegen. Dennoch muss bei Checkpoint-Index jeder Teil des Tracefiles nur einmalig bei dessen erstem Einlesen indiziert werden, auch ein aufwändiges späteres Verschieben von Checkpoints findet nicht statt.

6.6 FlexTime

Eine lineare Abbildung der Simulationszeit auf reale Zeit während der Darstellung ist in den meisten Anwendungsfällen nur schlecht geeignet. In typischen Simulationsszenarien wechseln sich Phasen heftiger Aktivität mit langen Ruhephasen mit wenigen oder gar keinen Ereignissen ab. Ein fester Verlangsamungsfaktor würde so entweder viel zu lange Wartephasen während der aktivitätsfreien Abschnitte oder aber eine viel zu schnelle Darstellung in aktiven Phasen bedeuten. Ein variabler, automatisch an die momentane Aktivität angepasster Verlangsamungsfaktor wurde deshalb entwickelt.

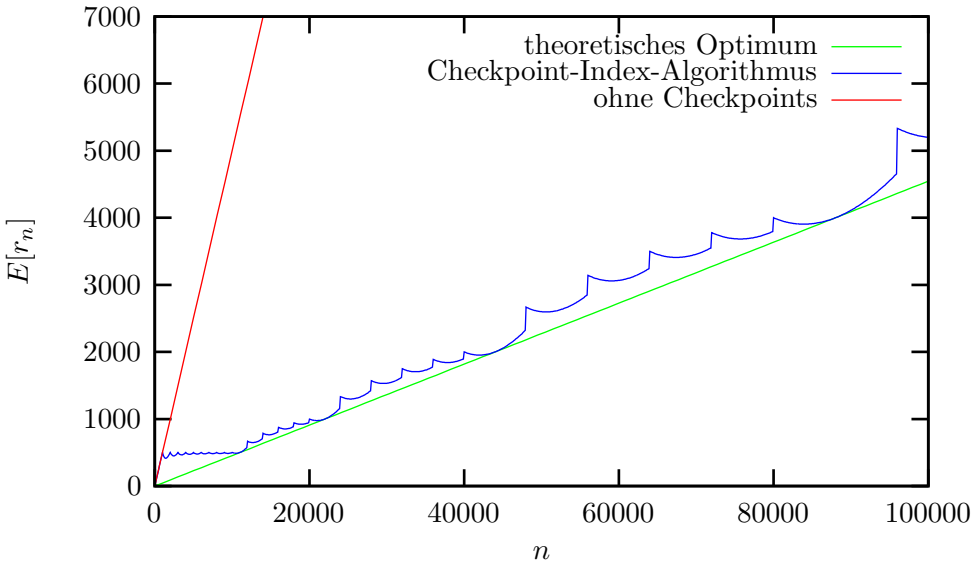


Abbildung 6.5: Verhalten des Checkpoint-Index-Algorithmus in Bezug auf die durchschnittlich bei einem Sprung in den indizierten Bereich noch einzulesende Anzahl von Zeilen; es wird hier von 10 maximal gleichzeitig vorhandenen Checkpoints ausgegangen, der Startwert für d liegt bei 1000 Zeilen

Um eine anschauliche Darstellung zu erhalten ist es wichtig, dass trotz der mitunter raschen Wechsel der Ablaufgeschwindigkeit der Eindruck eines kontinuierlichen Zeitflusses erhalten bleibt. Deshalb sollte die Anpassung der Geschwindigkeit nicht in harten Sprüngen, sondern gleitend erfolgen.

Sei i der zuletzt berechnete Frame. $t_s(i)$ bezeichne den von i dargestellten Zeitpunkt in Simulationszeit, $t_r(i)$ den Zeitpunkt in realer Zeit, zu dem i eingeblendet wird. Soll der nächste Frame $i+1$ angezeigt werden, muss hierfür zunächst $t_s(i+1)$ bestimmt werden. Damit können dann das Tracefile bis zum Zeitpunkt $t_s(i+1)+L$ eingelesen (L bezeichne den Lookahead, s. Abschnitt 6.1.2) und die in der Szene vorzunehmenden Änderungen bestimmt werden.

Im Voraus ist nicht bekannt, wie lange das Einlesen des Tracefile-Abschnittes und die Berechnung des nächsten Frames dauern wird. Daher ist auch nicht bekannt, wie viel reale Zeit zwischen Frame i und Frame $i+1$ verstreichen wird. Diese Zeitspanne $\Delta_r(i+1) := t_r(i+1) - t_r(i)$ kann im Voraus nur geschätzt werden.

Hier wird von einer in etwa konstanten Framerate ausgegangen; auf dieser Grundlage kann man annehmen, dass $t_r(i+1) - t_r(i) \approx t_r(i) - t_r(i-1)$, und so folgende Schätzwerte erhalten:

$$\hat{\Delta}_r(i+1) = t_r(i) - t_r(i-1) \tag{6.22}$$

$$\hat{t}_r(i+1) = t_r(i) + \hat{\Delta}_r(i+1) = 2t_r(i) - t_r(i-1) \tag{6.23}$$

Um bei einem zu starken Abfallen der Framerate zu vermeiden, dass für jeden Frame ein großer Sprung in der Simulationszeit stattfindet und zu große Tracefileabschnitte eingelesen und verarbeitet werden müssen, wird in der praktischen Implementation Δ_r auf maximal 0,1 s, also das Intervall $[0, \frac{1}{10}]$, eingeschränkt.

Für das weitere Einlesen des Tracefiles und die Berechnung der Änderungen der Darstellung ist aber nicht direkt $t_r(i+1)$ relevant, sondern der Zeitpunkt in Simulationszeit $t_s(i+1)$, den Frame $i+1$ wiedergeben soll. Dieser Zeitpunkt wird berechnet, indem der reale Zeitschritt $\Delta_r(i+1)$ – bzw. dessen Schätzung $\hat{\Delta}_r(i+1)$ – auf einen Simulations-Zeitschritt $\Delta_s(i+1)$ abgebildet wird.

Für die Abbildung der Zeitschritte von realer Zeit auf Simulationszeit wird eine Abbildung w definiert. Da wie oben angesprochen ein aktivitätsabhängiges Strecken oder Stauchen der Zeitachse angestrebt wird, ist w abhängig von der Aktivität im betrachteten Zeitraum und damit vom Simulations-Zeitintervall $[t_s(i), t_s(i+1)]$.

Um die Berechnung zu vereinfachen wird statt der Aktivität im genannten Intervall nur die Aktivität zum Zeitpunkt $t_s(i)$ betrachtet. w bildet also ein Tupel aus der Aktivität zum Simulationszeitpunkt $t_s(i)$, ausgedrückt durch eine Aktivitätsfunktion a, der verstreichenden realen Zeitspanne $\Delta_r(i+1)$ und einem vom Benutzer festgelegten relativen Geschwindigkeitsfaktor s auf den in der Simulationszeit zu vollziehenden Zeitschritt $\Delta_s(i+1)$ ab. In der vorliegenden Implementation ist w folgendermaßen definiert:

$$w : (a(t_s(i)), \Delta_r(i+1), s) \longmapsto \Delta_s(i+1) = s\frac{\Delta_r(i+1)}{1 + a(t_s(i))} \tag{6.24}$$

$$\begin{aligned} t_s(i+1) &= t_s(i) + \Delta_s(i+1) \\ &= t_s(i) + w(a(t_s(i)), \Delta_r(i+1), s) \\ &= t_s(i) + s\frac{\Delta_r(i+1)}{1 + a(t_s(i))} \end{aligned} \tag{6.25}$$

Durch Variation von s kann der Benutzer die Ablaufgeschwindigkeit beeinflussen.

Zur endgültigen Bestimmung von $t_s(i+1)$ fehlt nun nur noch eine sinnvolle Definition der Aktivitätsfunktion a. Diese soll in Phasen hoher Aktivität ansteigen, in der Pause zwischen Aktivitäten abfallen. Wegen der gewählten Definition für w kommt als Wertebereich für a nur das Intervall $]-1, \infty[$ in Frage

Sei zunächst a_0 eine Funktion, die an jedem Punkt der Simulationszeit, an dem ein Ereignis auftritt, einen Impuls mit Amplitude 1 hat, sonst aber überall gleich Null ist. Treten mehrere Ereignisse zum selben Zeitpunkt auf, so habe der Impuls entsprechend die mehrfache Höhe.

a_0 ist jedoch als Aktivitätsfunktion noch nicht geeignet. Hierfür müssen zunächst die Impulse geglättet werden, so dass sich ein Ereignis auch in einer beschränkten Umgebung um den Zeitpunkt seines Auftretens auswirkt.

Eine solche Glättung kann durch Faltung mit einer Funktion k erzielt werden, die nur in einer beschränkten Umgebung des Nullpunktes ungleich Null ist und die selbst die gewünschten „glatten“ Eigenschaften aufweist:

$$a_1(t) := (a_0 * k)(t) = \int a_0(\tau)k(t-\tau)\,d\tau \tag{6.26}$$

Für den Eindruck eines glatten Zeitflusses ist Stetigkeit von w ausreichend. Wird a so gewählt, dass $\forall t : a(t) > -1$, dann genügt die Stetigkeit von a, um Stetigkeit von w zu garantieren. Ist k stetig, so wird hierdurch auch a_1 stetig.

Die Funktion k wurde in der Implementation folgendermaßen gewählt; das gewählte k ist stetig:

$$k(t) := \begin{cases} 1 - 400 \cdot |t| & \text{wenn } |t| \leq 0.0025 \\ 0 & \text{sonst} \end{cases} \tag{6.27}$$

Während aktiver Phasen zeigt dieser Ansatz zufriedenstellende Ergebnisse. In längeren Ruhephasen allerdings wäre eine Beschleunigung über das bei $a_1(t_s(i)) = 0$ gegebene Maß hinaus wünschenswert. Deshalb wird die endgültige Aktivitätsfunktion a so definiert, dass sie überall dort, wo $a_1 > 0$, gleich a_1 ist. Überall sonst fällt a jedoch zunächst bis auf höchstens $-\frac{99}{100}$ ab, um dann bis zum Beginn des nächsten Bereiches mit $a_1 > 0$ wieder bis auf 0 anzusteigen. $l(t)$ und $r(t)$ bezeichnen den linken respektive rechten Rand des Intervalls um t, auf dem $a_1 = 0$ gilt.

$$l(t) := \min\left(\{\vartheta \geq 0 | a_1(\vartheta) = 0 \wedge (\forall \tau, \vartheta \leq \tau \leq t : a_1(\tau) = 0)\} \cup \{t\}\right) \tag{6.28}$$

$$r(t) := \max\left(\{\vartheta \geq 0 | a_1(\vartheta) = 0 \wedge (\forall \tau, t \leq \tau \leq \vartheta : a_1(\tau) = 0)\} \cup \{t\}\right) \tag{6.29}$$

$$a(t) := \begin{cases} a_1(t) & \text{wenn } a_1(t) > 0 \\ \max\left\{-\frac{t-l(t)}{(t-l(t))+\frac{1}{60}}, -\frac{r(t)-t}{(r(t)-t)+\frac{1}{60}}, -\frac{99}{100}\right\} & \text{sonst} \end{cases} \tag{6.30}$$

Den Zusammenhang zwischen a_0, a_1 und a veranschaulicht Abbildung 6.6.

Ein Problem für die Bestimmung von a könnte sich ergeben, wenn aus dem Tracefile kein geschwindigkeitsrelevantes Ereignis mit einem Zeipunkt größer $t_s(i+1)$ eingelesen wurde. Dann ist $r(t)$ nicht bestimmbar. Dieser Fall stellt jedoch kein wirkliches Problem dar: Solange das nächste Ereignis ausreichend früh eingelesen wird, um das Ansteigen gegen 0 rechtzeitig einzuleiten, kann zuvor problemlos $r(t) = \infty$ gesetzt werden, ohne dass sich dadurch a ändert. Durch die oben erwähnte Beschränkung von $\Delta_r(i)$ auf Werte im Bereich $[0, \frac{1}{10}]$ wird das ausreichende frühe Einlesen gewährleistet.

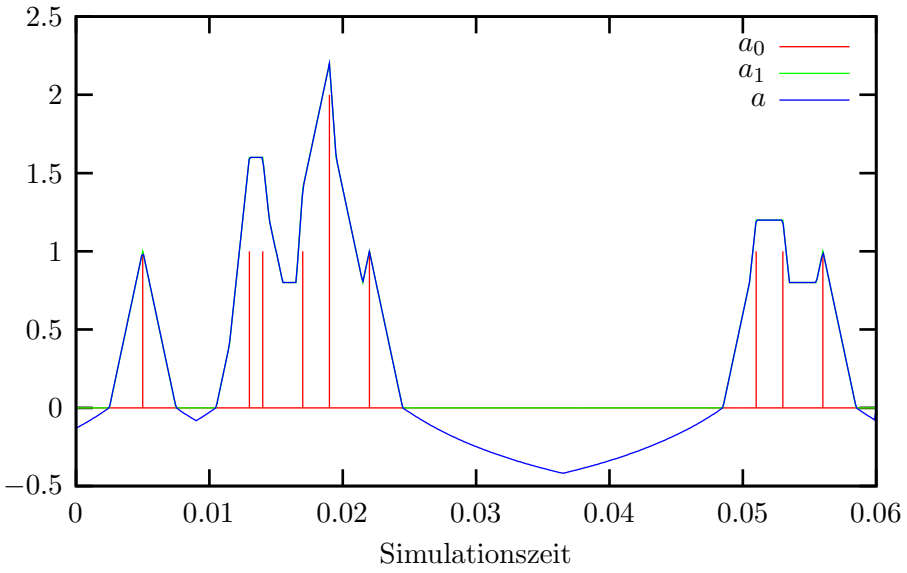


Abbildung 6.6: Zusammenhang zwischen den Aktivitätsfunktionen a_0, a_1 und a für ein exemplarisches Tracefile mit geschwindigkeitsrelevanten Ereignissen zu den Zeitpunkten 0.005, 0.013, 0.014, 0.017, 0.019, 0.019, 0.022, 0.051, 0.053 und 0.056

6.7 2-Phasen-Queues

6.7.1 Problembeschreibung

Es existieren eine Reihe effizienter Algorithmen, die Ereignis- oder Prioritätswarteschlangen[2] implementieren. Ein sehr bekannter basiert auf Heaps, d. h. auf binären Bäumen, die die Heap-Eigenschaft erfüllen: Ein Knoten ist stets kleiner als seine Kinder, oder er ist gleich groß.

In eine durch einen Heap implementierte Queue lassen sich Elemente in $O(\log n)$ Verarbeitungsschritten einfügen, wobei n die Gesamtzahl der Einträge in der Queue bezeichnet. Das Auslesen und Entfernen des vordersten Elements der Queue benötigt ebenfalls $O(\log n)$ Schritte [Knu98].

Es existieren andere Algorithmen mit für solche Anwendungen interessantem Laufzeitverhalten, beispielsweise Calendar Queues [Bro88], Fibonacci Heaps [FT87] oder auch eine von Brodal vorgeschlagene Datenstruktur [Bro96]. Erstere basieren auf einer Hash-Tabelle und erreichen vor allem bei relativ gleichverteilten Ereignissen sehr gute Performancewerte. Die anderen beiden sind vor allem theoretisch interessant, da sie auch im Worst Case $O(1)$ Schritte für die Einfügeoperation erreichen. Allerdings sind beide Algorithmen sehr komplex und der Aufwand pro Verarbeitungsschritt so hoch, dass sich bei einer realistisch angesetzten Ereignisanzahl für die meisten Probleme kein Vorteil ergibt [RA97, CSR93].

[2]Die beiden Probleme sind äquivalent, eine höhere Priorität in einer Prioritätswarteschlange kann als früherer Zeitpunkt in einer Ereigniswarteschlange interpretiert werden und umgekehrt

6.7.2 Lösungsansatz

Hier soll nun ein Algorithmus, die 2-Phasen-Queue (2PQ), beschrieben werden, der in einem hier besonders interessanten Spezialfall die Laufzeit verbessern kann. Es sollen Warteschlangen unterschiedlicher Länge untersucht werden, bei denen sich die Verarbeitung in zwei wechselweise auftretende Phasen unterteilen lässt:

1. (ggf. wiederholtes) Entfernen des Kopfelementes der Queue (*Abbauphase*)
2. Einfügen von neuen Elementen in die Queue (*Einfügephase*)

Während der Abbauphase können – in einer anderen Datenstruktur, z. B. einer Liste – schon neue Elemente für das Einfügen in der folgenden Einfügephase eingeplant werden.

Mit einer einfachen Erweiterung ist in beiden Phasen das Entfernen von beliebigen Ereignissen aus der Queue in $O(1)$ Schritten möglich. Damit können in der Einfügephase durch Löschen und erneutes Einfügen auch Elemente in ihrer Priorität verändert werden.

Man trifft den geschilderten 2-Phasen-Fall häufig in grafischen Anwendungen an, die periodisch neue Einzelbilder berechnen und dabei eine Teilmenge der angezeigten Objekte in ihrer Darstellung verändern müssen. So auch in der in Abschnitt 6.1 beschriebenen rekonstruierten Ereignis-Queue: Während der Berechnung eines Frames werden aus der Queue die vorzunehmenden Veränderungen entnommen, während des folgenden weiteren Einlesens des Tracefiles für den nächsten Frame wird die Queue mit neuen Ereignissen gefüllt.

Für diese Anwendung sind Queue-Längen bis zu mehreren tausend Elementen realistisch: Müssen bei einem Sprung nach vorne größere Tracefileabschnitte indiziert werden, so werden relativ viele Zeilen eingelesen und damit viele Ereignisse erzeugt, bevor erstmals wieder Ereignisse aus der Queue entnommen werden (siehe auch Abschnitt 6.5).

6.7.3 Algorithmus und asymptotisches Laufzeitverhalten

Die zentrale Datenstruktur einer 2-Phasen-Queue ist eine einfach verkettete Liste L_1, in der die Elemente nach ihrer Priorität geordnet vorliegen. Während der Abbauphase werden Elemente vom Kopf der Liste entfernt. Eine solche Operation ist in $O(1)$ Schritten möglich.

Sollen beliebige Elemente, auf die ein Zeiger vorhanden ist, in $O(1)$ Schritten entfernt werden können, so wird statt der einfachen eine doppelt verkettete Liste verwendet. Dann kann einfach der Listeneintrag gelöscht werden. Dies ist in einer doppelt verketteten Liste ebenfalls in $O(1)$ Schritten machbar.

Neu hinzukommende Elemente werden in einer getrennten Liste L_2 zunächst unsortiert gesammelt. Hier können wie erwähnt auch schon während der Abbauphase neue Elemente zum Einfügen vorgesehen werden. Auch dieses Einplanen eines Elementes ist durch Anhängen an die Liste in $O(1)$ Schritten möglich.

Am Ende der Einfügephase, wenn alle neu einzufügenden Elemente bekannt sind, wird die Queue reorganisiert. Dazu wird zunächst die Liste L_2 sortiert. Dies ist mit gängigen Sortierverfahren bei n Elementen in L_2 in $O(n \log n)$ möglich. In einigen Spezialfällen betreffend den Datentyp des Sortierschlüssels sind noch bessere Verfahren bekannt [Knu98]. Die Betrachtung soll hier aber auf den allgemeinen Fall $O(n \log n)$ beschränkt bleiben.

Die in der vorangegangenen Abbauphase nicht entfernten Elemente liegen weiterhin in L_1 in sortierter Form vor, die neuen Elemente finden sich sortiert in L_2. Das Zusammenführen dieser beiden Listen geschieht nun durch *merging*, wie es bei externen Sortierverfahren [Knu98] häufig zum Einsatz kommt. Dabei werden jeweils die vordersten Elemente der beiden Listen verglichen, das kleinere von beiden aus seiner Liste entnommen und in eine neue Liste L_3 am Ende eingefügt. Dieser Schritt wird wiederholt, bis eine der Listen L_1, L_2 leer ist. Danach wird der verbliebene Rest der anderen Liste unverändert an L_3 angehängt.

Beim *merging* wird jedes Element aus L_1, L_2 höchstens einmal entfernt und in L_3 eingefügt. Für jeden notwendigen Vergleichsvorgang zweier Kopfelemente wird ein Element nach L_3 verschoben. Wenn n_i die Anzahl der Elemente in L_i bezeichnet, so ist die Anzahl der Verarbeitungsschritte während des *merging* also durch $O(n_1 + n_2)$ beschränkt.

Nach dem *merging* enthält das entstandene L_3 alle Elemente der Queue in sortierter Form. Es wird als neues L_1 für die folgende Abbauphase verwendet.

Werden während eines Zyklus – also einer Abbau- und einer Einfüge-Phase samt folgender Reorganisation – von n ursprünglich vorhandenen Elementen k durch Dequeueing (oder auch gezieltes Löschen anderer Elemente) entfernt und werden m Elemente neu eingefügt, so ergibt sich für 2PQ folgende Schranke für die gesamte Verarbeitungszeit in dem Zyklus:

$$O(\underbrace{k}_{\text{Löschen}} + \underbrace{m}_{\text{Einplanen}} + \underbrace{m \log m}_{\text{Vorsortieren}} + \underbrace{(n-k)+m)}_{\text{merging}} = O(n + k + m \log m) \tag{6.31}$$

Bei einer auf einem Heap basierenden Queue werden Einfügeoperationen sofort ausgeführt. Dann ist über die Anzahl der zum Zeitpunkt von Lösch- und Einfügeoperationen im Heap vorhandenen Elemente nur bekannt, dass diese innerhalb des Zyklus stets durch $n+m$ nach oben beschränkt ist. Unter dieser Prämisse ergibt sich folgende Schranke für die Laufzeit:

$$O(\underbrace{k \log(n+m)}_{\text{Entfernen}} + \underbrace{m \log(n+m))}_{\text{Einfügen}} = O((k+m)\log(n+m)) \tag{6.32}$$

Führt man bei einem Heap im 2-Phasen-Fall wie bei 2PQ zunächst alle Entfernungs- und danach die Einfüge-Operationen eines Zyklus aus, erhält man eine geringfügig bessere Schranke:

$$O(\underbrace{k \log n}_{\text{Entfernen}} + \underbrace{m \log(n-k+m)}_{\text{Einfügen}}) \tag{6.33}$$

Für lange Queues, aus denen in jedem Zyklus nur relativ wenige Elemente entnommen werden (n groß, k und m klein), ist wegen der nur logarithmisch von n abhängigen Laufzeit die Heap-Implementation im Vorteil. Ist hingegen n eher klein oder die Queue sehr dynamisch (k, m groß), so fällt die lineare Abhängigkeit von n bei 2PQ weniger ins Gewicht. Dann kann sich ein Vorteil für 2PQ ergeben. Diese Erwartungen werden durch die in 6.7.4 geschilderten Messungen bestätigt.

Im Gegensatz zu einem Heap ist 2PQ bei geeigneter Implementierung stabil in dem Sinne, dass neue Elemente hinter früher eingefügten mit gleicher Priorität eingeordnet werden. Bei Heaps sind hier zusätzliche Vorkehrungen wie etwa eine Nummerierung der Elemente entsprechend ihrer Einfügereihenfolge notwendig.

6.7.4 Performance-Messung

Zum Vergleich der tatsächlichen Performance von Hash-Queues und 2PQ wurden beide Algorithmen implementiert und die sich ergebende Geschwindigkeit gemessen. Beide Implementationen erfolgten in C++ [Str91]. Kompiliert wurden die Programme mit gcc 3.3.2 [GCC] mit ausgeschalteter Compileroptimierung (-O0).

Zur Messung der Geschwindigkeit wurde für beide Algorithmen zunächst eine korrekt initialisierte Queue mit n pseudozufälligen 32-Bit-Integer-Werten als Elementen erzeugt. Die hierfür notwendige Zeit floss nicht in die Messung ein. Im Anschluss wurden Zyklen simuliert. In jedem Zyklus wurden zunächst die k kleinsten Elemente entfernt, danach wieder k neue pseudozufällige Elemente eingefügt. Es wurde gemessen, wie viele solcher Zyklen innerhalb von 60 Sekunden bearbeitet werden konnten.

Für die Messung wurde ein Ein-CPU-Rechner, ein AMD Athlon XP 2600+, unter Mandrakelinux 10 [MDK] verwendet.

Der Mersenne-Twister-Algorithmus [MN98] in der Implementation aus der Boost C++ Random Number Library [BOO] in Version 1.17 kam zur Erzeugung der Pseudozufallszahlen zum Einsatz. Die Parametrisierung erfolgte gemäß mt19937 (definiert in [MN98]). Der Random Seed und damit die erzeugte Folge von Zahlen waren bei allen Experimenten identisch.

Bei der 2-Phasen-Queue wurde die Variante mit einer einfach verketteten Liste L_1 implementiert. Als Sortieralgorithmus kam QuickSort [Hoa62, Knu98] zum Einsatz.

Während die 2PQ-Implementation mit C++-Objekten und Zeigern arbeitet, verwendet die Implementation des Hash-Algorithmus einen C-Array von Integer-Werten. Auf

dieser Datenstruktur ist eine besonders effiziente direkte Abbildung der Baum- auf die Arraypositionen möglich, bei der keine Zeigeroperationen notwendig sind.

Die Quelltexte der verwendeten Implementationen finden sich in Anhang C.

Abbildung 6.7 zeigt die Messwerte für unterschiedliche n und k. Der Vorteil von 2PQ für kurze Queues sowie für solche mit hoher Dynamik (k nahe n) ist deutlich sichtbar. Obwohl bei 2PQ zur Bearbeitung der Listen viele Zeigeroperationen notwendig sind, ergibt sich in solchen Fällen eine insgesamt höhere Performance.

Der Vorteil von 2PQ speziell bei sehr dynamischen Queues mit vielen Veränderungen in jedem Zyklus ist für grafische Anwendungen besonders relevant: Entspricht hier ein Frame einem Zyklus, so ist die 2PQ vor allem dann schneller, wenn viele Veränderungen vorgenommen werden müssen, wenn also auch die sonstigen Berechnungen umfangreich sind. 2PQ verhält sich also während kritischer Phasen mit ohnehin geringen Frameraten besonders vorteilhaft.

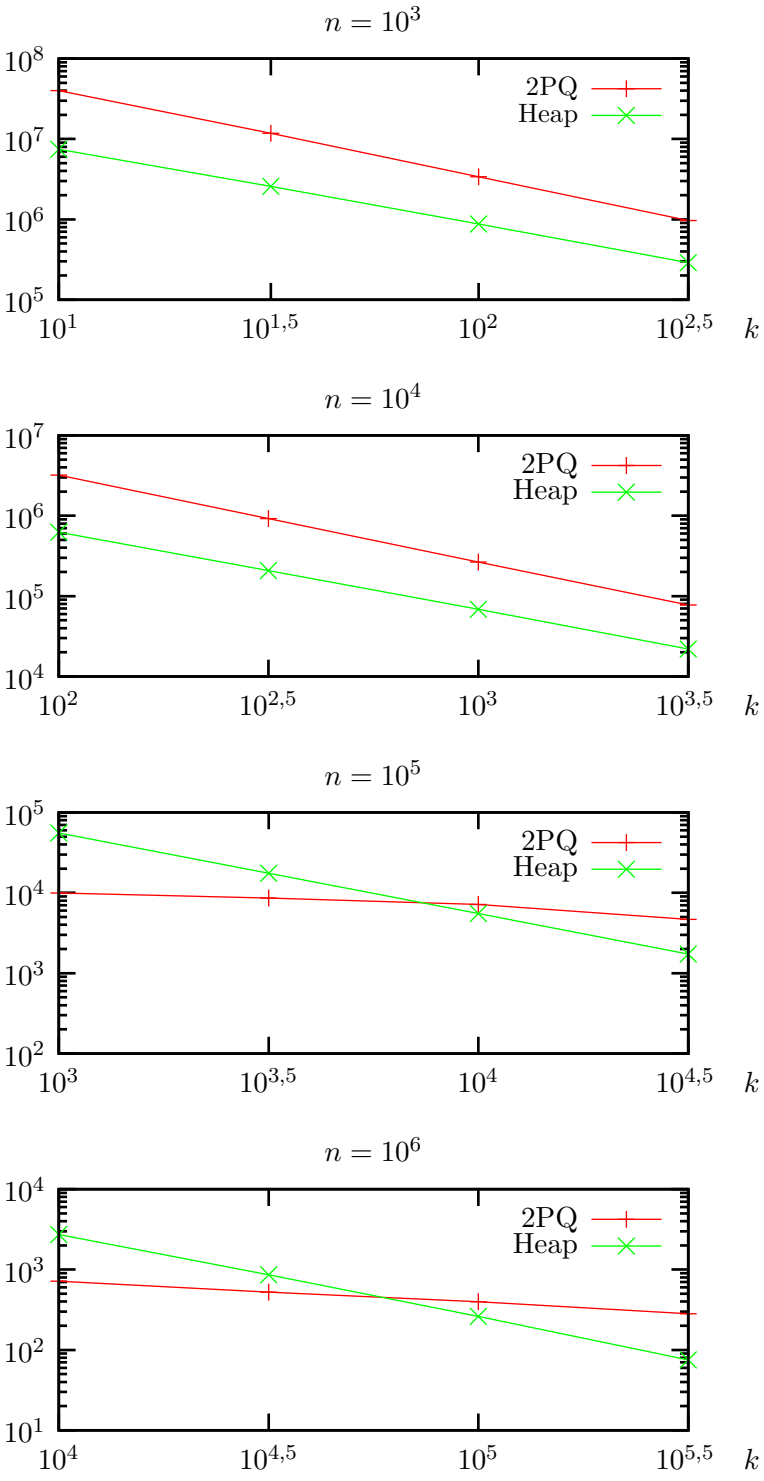


Abbildung 6.7: Performancevergleich von 2PQ und einer arraybasierten Heap-Implementation. n bezeichnet die Anzahl der Elemente in der Queue, k die Anzahl der in jedem Zyklus ausgetauschten Queue-Elemente. Aufgetragen ist die Anzahl der in einem Zeitraum von 60 Sekunden bearbeiteten Zyklen.

Kapitel 7

Erweiterungsmöglichkeiten

Ns-2 ist zweifellos ein höchst komplexes System, das noch wesentlich mehr Daten zur Verfügung stellen kann, als momentan mit Huginn ausgewertet und angezeigt werden können. Es liegt nahe, weitere verfügbare Arten von Simulationen und weitere interessante Datenquellen für die Darstellung zu erschließen. Daraus ergeben sich viele Ideen, wie die Fähigkeiten des Programms in Zukunft noch ausgebaut werden könnten.

Naheliegend wäre mit Sicherheit die Erweiterung der Darstellung auf Simulationen kabelgebundener Netzwerke. Damit einhergehen könnte die Unterstützung von Szenarien, die teilweise drahtlos und teilweise kabelgebunden arbeiten. Solche *Wired-cum-Wireless*-Szenarien werden zunehmend wichtig.

Auch die Integration weiterer Aggregat- oder Skalierungsfunktionen oder zusätzlicher optionaler Szeneelemente könnte in einigen Anwendungsbereichen nutzbringend sein. Denkbar sind zum Beispiel Aggregatfunktionen, die nur über bestimmte Zeitintervalle anstatt über die gesamte bisherige Simulationszeit aggregieren, oder einblendbare Pfeile, die Richtung oder Zielpunkt einer Knotenbewegung anzeigen könnten.

Ns-2 unterstützt die Entwicklung von Protokollen für MANETs unter Berücksichtigung ihres Energieverbrauchs durch Mitführen eines „Energiestatus“ für die einzelnen Knoten. Außerdem können unterschiedliche Sendereichweiten durch verschiedene Antennenhöhen modelliert werden. Eine Unterstützung für beide Funktionen ließe sich mit vertretbarem Aufwand implementieren.

Ebenfalls vorstellbar ist die Einbringung von Unterstützung für einzelne Routing- und Anwendungsprotokolle. Der momentane Ansatz versucht ja, in dieser Hinsicht vollkommen protokollunabhängig und ohne Annahmen über die Arbeitsweise etwa des Routingprotokolls zu arbeiten. Es sind allerdings auch Anwendungsgebiete denkbar, in denen ein implementiertes (und ausreichend verifiziertes) Protokoll auf sein Verhalten in unterschiedlichen Situationen analysiert werden soll. Um hier noch bessere Unterstützung als durch die bereits vorhandenen Mechanismen zu bieten, könnten zusätzlich neue, protokollspezifische Darstellungsformen jeweils passend entwickelt werden.

Bei etwas weiterer Verbreitung des Tools und regelmäßigem Einsatz werden sich voraussichtlich einige Auswertungen (d. h. einige bestimmte Flussdiagramme) als für be-

stimmte Aufgaben besonders geeignet herauskristallisieren. Da sich solche Auswertungen als Dateien speichern lassen und unabhängig vom Tracefile angewendet werden können, lassen sie sich leicht austauschen und weitergeben. Es wäre deshalb möglich, eine Bibliothek aufzubauen, die die erfolgreich eingesetzten Auswertungen vieler Benutzer enthält; so könnten alle Anwender von den Erfahrungen der anderen profitieren.

Nicht zuletzt sind aber auch Veränderungen der Datenquelle ns-2 selbst überlegenswert, wenn sie auch nicht im zentralen Fokus dieser Arbeit stehen. Wie die hier vorgestellten Ergebnisse bezüglich der Möglichkeiten zur Zuordnung von Sende- und Empfangsvorgängen deutlich gezeigt haben, sind die Tracingfunktionen von ns-2 als solche durchaus noch zu verbessern. Schon die Einführung von Paket-IDs, die für jeden Sendevorgang auch auf RTR- und MAC-Ebene eindeutig sind, würde vieles erleichtern. Auch eine Lösung für die Schwierigkeiten bei der Unterscheidung von nicht belegten und mit null belegten Feldern wäre wünschenswert.

Aber es existieren auch Algorithmen, die weit darüber hinausgehende Möglichkeiten eröffnen könnten, um die Vorgänge bei der Simulation im Nachhinein noch besser nachvollziehen zu können. Würde beispielsweise eine Unterstützung für Vektoruhren [TS01, Mat89, Fid88] in ns-2 konsequent umgesetzt, so wäre für alle auftretenden Ereignisse eine eindeutige Vorher-Nachher-Relation im Sinne klarer Kausalitäten verfügbar: Es wäre immer möglich, beim Auftreten zweier Ereignisse gleich welcher Art zu bestimmen, ob eines davon ursächlich mit dem anderen zusammenhängen kann oder ob ein solcher Zusammenhang mit Sicherheit ausgeschlossen ist. Das würde es ermöglichen, bei der Identifikation eines speziellen Problems an einer Stelle im Tracefile alle Vorgänge auszublenden, bei denen ein unmittelbarer ursächlicher Zusammenhang zu dieser Stelle unmöglich ist. Es würde alles das weggelassen werden können, was unter keinen Umständen zur Entstehung des Problems beigetragen hat. Die Folgen wären eine starke Reduktion des „Rauschens“ und damit noch klarere Darstellungen.

Der Umfang der für solche Fähigkeiten am Simulator ns-2 notwendigen Änderungen soll und kann hier nicht abgeschätzt werden. Fest steht jedoch, dass eine Unterstützung zum Beispiel von Vektoruhren mit einem nicht unerheblichen zusätzlichen Overhead bezüglich der verarbeiteten Datenmengen verbunden ist – der Speicherplatz für einen Zeitvektor wächst linear mit der Anzahl der Funktionseinheiten in der Simulation. Auch die Größe eines Tracefile-Eintrages, der einen solchen Zeitvektor enthält, wäre dementsprechend groß. In jedem Falle müsste also eine solche Implementierung als optionale, abschaltbare Funktion erfolgen.

Kapitel 8

Zusammenfassung

Mit Huginn ist ein Werkzeug verfügbar, das zur Untersuchung von ns-2-Simulationsergebnissen für MANETs beitragen kann. Mittels grafischer Darstellungen kann ein besseres Verständnis für die Vorgänge auf kleiner zeitlicher und räumlicher Ebene erzielt werden. Ebenso wird ein schnellerer Überblick über größere Szenarien, die zeitliche Abfolge und den Zusammenhang von Ereignissen ermöglicht. Huginn ermöglicht solche Darstellungen in dreidimensionaler Form.

Die Möglichkeiten zur Individualisierung der Darstellungsform erlauben eine Anpassung auf den jeweiligen konkreten Einsatzzweck, bestimmte Simulationsparameter können gezielt betont oder weggelassen werden. Diese Individualisierung geschieht mittels spezieller Flussdiagramme, die den Datenfluss von unterschiedlichen Datenquellen bis zu den Darstellungselementen beschreiben. Das erlaubt die Umsetzung vieler höchst unterschiedlicher Darstellungsformen, die dennoch alle auf denselben Grundgedanken des Datenflusses von Simulationsparametern durch Auswertungsfunktionen zu den Darstellungselementen zurückgeführt werden.

In die Szene eingebettete statistische Auswertungen ermöglichen ein leichteres Verständnis des Zusammenhangs zwischen dem Simulationsgeschehen und dessen Auswirkungen auf Parameter wie etwa Paketlaufzeiten. Speziellen Auswertungsanforderungen wird durch die Möglichkeit zur Integration von benutzergeschriebenem Code Rechnung getragen.

Die Visualisierung ist unabhängig von den verwendeten Routing- und Agent-Layer-Protokollen, und kann daher auch nutzbringend zur Untersuchung neuer Protokolle dieser Schichten eingesetzt werden, ohne dass hierfür eine Anpassung der Software notwendig wird.

Um den Einsatz des Werkzeugs zu erleichtern und seine Akzeptanz zu fördern, wurde auf eine intuitive Benutzerschnittstelle und ein möglichst einfaches Modell zur Abstraktion der Abbildung der Simulationsdaten auf die Eigenschaften von Szeneparametern Wert gelegt.

Um es zu ermöglichen, die gewünschte Darstellung zu erzeugen, ohne Annahmen über die Funktionsweisen der verwendeten Protokolle machen zu müssen und ohne ein spezi-

elles Format und zusätzliche Informationen im Tracefile zu erfordern, wurden eine Reihe von Algorithmen entwickelt. Diese machen die Zuordnung zusammengehöriger Vorgänge in den Standard-Tracefile-Formaten erst möglich, und sie ermöglichen die schnelle Navigation im Tracefile trotz der vorgenommenen, unter Umständen recht umfangreichen Auswertungsrechnungen. Das Problem der vorliegenden, sehr unterschiedlichen zeitlichen Auflösungen – lange Pausen und kurze Phasen mit vielen Ereignissen – wurde mittels einer aktivitätsabhängigen automatischen Variation der Ablaufgeschwindigkeit gelöst.

Anhang A

Syntax der ns-2-Tracefiles

Hier soll eine Beschreibung der von ns-2 verwendeten Traceformate für Simulationen drahtloser Netze gegeben werden. Die hier wiedergegebenen Informationen entstammen dem „ns Manual“ [VIN03], den ns-2-Quelltexten (speziell der Datei trace/cmu-trace.cc) [ns2] sowie einer Erläuterung der Traceformate von Richard Griswold [Gri03].

Auf spezielle Traceformate einzelner Protokolle wird hier nicht näher eingegangen. Weitere Informationen hierzu finden sich in den oben angegebenen Quellen.

A.1 Altes Traceformat

Eine Zeile im alten Traceformat beginnt mit einem der folgenden vier Buchstaben, die jeweils für einen Ereignistyp stehen:

s	Paketversand
f	Weiterleiten eines Paketes
r	Paketemfang
D	Verwerfen eines Paketes

Danach folgen, jeweils durch Leerzeichen getrennt, die folgenden Felder:

Feld	**Datentyp, Repräsentation**
Zeitpunkt	double, dezimal
Knoten-ID	int, dezimal, umschlossen von Unterstrichen
Tracename/Layer	string
Begründung (bei Paketverlust)	string
Paket-ID	int, dezimal
Pakettyp	string
Paketgröße	int, dezimal

Alternativ ist für einen Traceeintrag folgendes Format möglich, wenn Positionsdaten der Knoten mitprotokolliert werden:

Feld	**Datentyp, Repräsentation**
Zeitpunkt	double, dezimal
Knoten-ID	int, dezimal
Knotenposition	2 x double, dezimal, zwischen runden Klammern
Tracename/Layer	string
Begründung (bei Paketverlust)	string
Paket-ID	int, dezimal
Pakettyp	string
Paketgröße	int, dezimal

Der sich anschließende MAC-Header ist von eckigen Klammern umschlossen und hat folgenden Aufbau:

Feld	**Datentyp, Repräsentation**
Dauer der Datenübertragung	int, hexadezimal
MAC-Zieladresse (= Knoten-ID)	int, hexadezimal
MAC-Quelladresse (= Knoten-ID)	int, hexadezimal
Pakettyp	0 = MAC-Layer, 800 = IP, 806 = ARP

Die Einträge für reine MAC-Layer-Pakete enden an dieser Stelle. Für die anderen Pakettypen folgen zunächst sieben Striche als Trennzeichen (- - - - - - -).

Für ein ARP-Paket setzt sich die Zeile – von eckigen Klammern umschlossen – folgendermaßen fort:

Feld	**Datentyp, Repräsentation**
ARP-Pakettyp	string, „REQUEST“ oder „REPLY“
MAC- und IP-Quelladresse	2 x int, dezimal, getrennt durch Schrägstrich
MAC- und IP-Zieladresse	2 x int, dezimal, getrennt durch Schrägstrich

Für IP-Datenpakete folgen stattdessen die folgenden Felder, wiederum mit eckigen Klammern zusammengefasst:

Feld	**Datentyp, Repräsentation**
IP-Quelladresse und -port	2 x int, getrennt durch Doppelpunkt
IP-Zieladresse und -port	2 x int, getrennt durch Doppelpunkt
TTL	int
Adresse des nächsten Hops	int

Daran können sich optional noch protokollspezifische Einträge anschließen.

Nachfolgend ein beispielhafter Ausschnitt aus einem Tracefile im alten Format; ein Backslash am Zeilenende markiert einen darstellungsbedingten Zeilenumbruch:

```
f 2.619243624 _1_ RTR  --- 0 AODV 44 [13a 1 2 800] ------- [8:255 0:255 23 0] [0x4 8\
  [8 4] 10.000000] (REPLY)
r 2.619533207 _2_ MAC  --- 0 ACK 38 [0 2 0 0]
s 2.619742624 _1_ MAC  --- 0 ARP 80 [0 ffffffff 1 806] ------- [REQUEST 1/1 0/0]
r 2.620383207 _0_ MAC  --- 0 ARP 28 [0 ffffffff 1 806] ------- [REQUEST 1/1 0/0]
r 2.620383207 _2_ MAC  --- 0 ARP 28 [0 ffffffff 1 806] ------- [REQUEST 1/1 0/0]
s 2.620738207 _0_ MAC  --- 0 RTS 44 [4fe 1 0 0]
r 2.621090791 _1_ MAC  --- 0 RTS 44 [4fe 1 0 0]
s 2.621100791 _1_ MAC  --- 0 CTS 38 [3c4 0 0 0]
r 2.621405374 _0_ MAC  --- 0 CTS 38 [3c4 0 0 0]
s 2.621415374 _0_ MAC  --- 0 ARP 80 [13a 1 0 806] ------- [REPLY 0/0 1/1]
r 2.622055957 _1_ MAC  --- 0 ARP 28 [13a 1 0 806] ------- [REPLY 0/0 1/1]
s 2.622065957 _1_ MAC  --- 0 ACK 38 [0 0 0 0]
r 2.622370541 _0_ MAC  --- 0 ACK 38 [0 0 0 0]
s 2.622699957 _1_ MAC  --- 0 RTS 44 [57e 0 1 0]
r 2.623052541 _0_ MAC  --- 0 RTS 44 [57e 0 1 0]
s 2.623062541 _0_ MAC  --- 0 CTS 38 [444 1 0 0]
r 2.623367124 _1_ MAC  --- 0 CTS 38 [444 1 0 0]
s 2.623377124 _1_ MAC  --- 0 AODV 96 [13a 0 1 800] ------- [8:255 0:255 23 0] [0x4 8\
  [8 4] 10.000000] (REPLY)
r 2.624145707 _0_ MAC  --- 0 AODV 44 [13a 0 1 800] ------- [8:255 0:255 23 0] [0x4 8\
  [8 4] 10.000000] (REPLY)
s 2.624155707 _0_ MAC  --- 0 ACK 38 [0 1 0 0]
r 2.624170707 _0_ RTR  --- 0 AODV 44 [13a 0 1 800] ------- [8:255 0:255 23 0] [0x4 8\
  [8 4] 10.000000] (REPLY)
s 2.624170707 _0_ RTR  --- 0 cbr 532 [0 0 0 0] ------- [0:0 8:0 30 1] [0] 0 0
r 2.624460291 _1_ MAC  --- 0 ACK 38 [0 1 0 0]
s 2.624989707 _0_ MAC  --- 0 RTS 44 [14be 1 0 0]
r 2.625342291 _1_ MAC  --- 0 RTS 44 [14be 1 0 0]
s 2.625352291 _1_ MAC  --- 0 CTS 38 [1384 0 0 0]
r 2.625656874 _0_ MAC  --- 0 CTS 38 [1384 0 0 0]
s 2.625666874 _0_ MAC  --- 0 cbr 584 [13a 1 0 800] ------- [0:0 8:0 30 1] [0] 0 0
r 2.630339457 _1_ MAC  --- 0 cbr 532 [13a 1 0 800] ------- [0:0 8:0 30 1] [0] 1 0
s 2.630349457 _1_ MAC  --- 0 ACK 38 [0 0 0 0]
r 2.630364457 _1_ RTR  --- 0 cbr 532 [13a 1 0 800] ------- [0:0 8:0 30 1] [0] 1 0
f 2.630364457 _1_ RTR  --- 0 cbr 532 [13a 1 0 800] ------- [0:0 8:0 29 2] [0] 1 0
r 2.630654041 _0_ MAC  --- 0 ACK 38 [0 0 0 0]
```

A.2 Neues Traceformat

Im neuen Traceformat beginnen ebenfalls alle Zeilen zunächst mit einem Buchstaben, der den Ereignistyp anzeigt:

- s Paketversand
- f Weiterleiten eines Paketes
- r Paketemfang
- d Verwerfen eines Paketes

Darauf folgen Bezeichner/Wert-Paare, wobei folgende Bezeichner verwendet werden:

Bezeichner	**Bedeutung**	**Datentyp, Repräsentation**
-t	Zeitpunkt	double, dezimal
-Ni	Knoten-ID	int, dezimal
-Nx	x-Koordinate der Knotenposition	double, dezimal
-Ny	y-Koordinate der Knotenposition	double, dezimal
-Nz	z-Koordinate der Knotenposition	double, dezimal
-Ne	Energiestand des Knotens	double, dezimal
-Nl	Tracename/Layer	string
-Nw	Begründung für verworfenes Paket	string
-Hs	Hop-Quellknoten-ID	int, dezimal
-Hd	Hop-Zielknoten-ID	int, dezimal
-Ma	Dauer der Datenübertragung	int, hexadezimal
-Md	MAC-Zieladresse (= Knoten-ID)	int, hexadezimal
-Ms	MAC-Quelladresse (= Knoten-ID)	int, hexadezimal
-Mt	Pakettyp	0 = MAC-Layer
		800 = IP
		806 = ARP

Für ARP-Pakete schließt sich daran „-P arp“ an, gefolgt von folgenden Bezeichnern:

Bezeichner	**Bedeutung**	**Datentyp, Repräsentation**
-Po	ARP-Pakettyp	string, „REQUEST“ oder „REPLY“
-Pms	MAC-Quelladresse	int, dezimal
-Ps	IP-Quelladresse	int, dezimal
-Pmd	MAC-Zieladresse	int, dezimal
-Pd	IP-Zieladresse	int, dezimal

Für IP-Datenpakete folgen stattdessen:

Bezeichner	**Bedeutung**	**Datentyp, Repräsentation**
-Is	IP-Quelladresse und -port	2 x int, dezimal, durch Punkt getrennt
-Id	IP-Zieladresse und -port	2 x int, dezimal, durch Punkt getrennt
-It	Pakettyp	string
-Il	Paketgröße	int, dezimal
-If	Paketfluss-ID	int, dezimal
-Ii	Paket-ID	int, dezimal
-Iv	TTL	int, dezimal

Darauf können dann noch protokollspezifische Angaben folgen, eingeleitet durch „-P *Protokollbezeichner*“. Die Bezeichner protokollspezifischer Werte beginnen stets mit „-P“, gefolgt von einem oder mehreren Kleinbuchstaben.

Hier ein Ausschnitt aus einem Tracefile im neuen Format; ein Backslash am Zeilenende steht wiederum für einen darstellungsbedingten Zeilenumbruch:

```
f -t 4.616909523 -Hs 24 -Hd 12 -Ni 24 -Nx 525.00 -Ny 525.00 -Nz 0.00 -Ne -1.000000\
  -Nl RTR -Nw --- -Ma 13a -Md 18 -Ms 19 -Mt 800 -Is 108.255 -Id 12.255 -It AODV -Il 44\
  -If 0 -Ii 0 -Iv 19 -P aodv -Pt 0x4 -Ph 12 -Pd 108 -Pds 6 -Pl 10.000000 -Pc REPLY
s -t 4.617628523 -Hs 24 -Hd -2 -Ni 24 -Nx 525.00 -Ny 525.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 0 -Md ffffffff -Ms 18 -Mt 806 -P arp -Po REQUEST -Pms 24 -Ps 24\
  -Pmd 0 -Pd 12
r -t 4.618269106 -Hs 23 -Hd -2 -Ni 23 -Nx 525.00 -Ny 350.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 0 -Md ffffffff -Ms 18 -Mt 806 -P arp -Po REQUEST -Pms 24 -Ps 24\
  -Pmd 0 -Pd 12
r -t 4.618269106 -Hs 13 -Hd -2 -Ni 13 -Nx 350.00 -Ny 525.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 0 -Md ffffffff -Ms 18 -Mt 806 -P arp -Po REQUEST -Pms 24 -Ps 24\
  -Pmd 0 -Pd 12
r -t 4.618269106 -Hs 25 -Hd -2 -Ni 25 -Nx 525.00 -Ny 700.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 0 -Md ffffffff -Ms 18 -Mt 806 -P arp -Po REQUEST -Pms 24 -Ps 24\
  -Pmd 0 -Pd 12
r -t 4.618269106 -Hs 35 -Hd -2 -Ni 35 -Nx 700.00 -Ny 525.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 0 -Md ffffffff -Ms 18 -Mt 806 -P arp -Po REQUEST -Pms 24 -Ps 24\
  -Pmd 0 -Pd 12
r -t 4.618269348 -Hs 14 -Hd -2 -Ni 14 -Nx 350.00 -Ny 700.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 0 -Md ffffffff -Ms 18 -Mt 806 -P arp -Po REQUEST -Pms 24 -Ps 24\
  -Pmd 0 -Pd 12
r -t 4.618269348 -Hs 12 -Hd -2 -Ni 12 -Nx 350.00 -Ny 350.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 0 -Md ffffffff -Ms 18 -Mt 806 -P arp -Po REQUEST -Pms 24 -Ps 24\
  -Pmd 0 -Pd 12
r -t 4.618269348 -Hs 34 -Hd -2 -Ni 34 -Nx 700.00 -Ny 350.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 0 -Md ffffffff -Ms 18 -Mt 806 -P arp -Po REQUEST -Pms 24 -Ps 24\
  -Pmd 0 -Pd 12
r -t 4.618269348 -Hs 36 -Hd -2 -Ni 36 -Nx 700.00 -Ny 700.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 0 -Md ffffffff -Ms 18 -Mt 806 -P arp -Po REQUEST -Pms 24 -Ps 24\
  -Pmd 0 -Pd 12
s -t 4.618624348 -Hs 12 -Hd -2 -Ni 12 -Nx 350.00 -Ny 350.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 4fe -Md 18 -Ms c -Mt 0
r -t 4.618977173 -Hs 24 -Hd -2 -Ni 24 -Nx 525.00 -Ny 525.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 4fe -Md 18 -Ms c -Mt 0
s -t 4.618987173 -Hs 24 -Hd -2 -Ni 24 -Nx 525.00 -Ny 525.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 3c4 -Md c -Ms 0 -Mt 0
r -t 4.619291998 -Hs 12 -Hd -2 -Ni 12 -Nx 350.00 -Ny 350.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 3c4 -Md c -Ms 0 -Mt 0
s -t 4.619301998 -Hs 12 -Hd -2 -Ni 12 -Nx 350.00 -Ny 350.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 13a -Md 18 -Ms c -Mt 806 -P arp -Po REPLY -Pms 12 -Ps 12 -Pmd 24\
  -Pd 24
r -t 4.619942823 -Hs 24 -Hd -2 -Ni 24 -Nx 525.00 -Ny 525.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 13a -Md 18 -Ms c -Mt 806 -P arp -Po REPLY -Pms 12 -Ps 12 -Pmd 24\
  -Pd 24
s -t 4.619952823 -Hs 24 -Hd -2 -Ni 24 -Nx 525.00 -Ny 525.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 0 -Md c -Ms 0 -Mt 0
r -t 4.620257648 -Hs 12 -Hd -2 -Ni 12 -Nx 350.00 -Ny 350.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 0 -Md c -Ms 0 -Mt 0
s -t 4.620886823 -Hs 24 -Hd -2 -Ni 24 -Nx 525.00 -Ny 525.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 57e -Md c -Ms 18 -Mt 0
r -t 4.621239648 -Hs 12 -Hd -2 -Ni 12 -Nx 350.00 -Ny 350.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 57e -Md c -Ms 18 -Mt 0
s -t 4.621249648 -Hs 12 -Hd -2 -Ni 12 -Nx 350.00 -Ny 350.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 444 -Md 18 -Ms 0 -Mt 0
```

A.3 Knotenbewegungen

Knotenbewegungen werden im alten und neuen Traceformat in gleicher Weise erfasst. Die entsprechenden Zeilen beginnen mit dem Buchstaben „M “, gefolgt von folgenden Feldern:

Feld	**Datentyp, Repräsentation**
Zeitpunkt	double, dezimal
Knoten-ID	int, dezimal
Startpunkt	3 x double, dezimal
Zielpunkt	2 x double, dezimal
Geschwindigkeit	double, dezimal

Die Koordinaten des Start- und Zielpunktes sind jeweils von runden Klammern umschlossen und durch Kommata getrennt. Zwischen den Feldern Startpunkt, Zielpunkt und Geschwindigkeit ist zusätzlich auch je ein Komma eingefügt.

Tracefile-Einträge zur Knotenbewegung sehen z. B. folgendermaßen aus:

```
M 0.00000 38 (348.98, 236.49, 0.00), (787.36, 621.39), 11.82
M 0.00000 39 (904.04, 550.26, 0.00), (641.95, 636.41), 11.33
M 2.45607 29 (456.31, 474.17, 0.00), (344.06, 96.08), 11.99
```

A.4 Huginn-Erweiterungen

Huginn führt einige Erweiterungen der Traceformate ein, die zum Definieren von Ereignissen, Zuständen und Parametern durch den Benutzer dienen können.

A.4.1 Zusätzliche Ereignisparameter

An alle Tracefilezeilen, die Sende-, Empfangs- oder Weiterleitungsereignisse oder das Verwerfen eines Paketes bezeichnen, können Zusatzinformationen angehängt werden, die weitere Parameter der durch diese Zeilen erzeugten Ereignisse in den jw. Knoten angeben. Für jeden so hinzugefügten Parameter muss ein Block in folgendem Format angehängt werden, separiert durch Leerzeichen:

$$\{\textit{Parametername}\text{: }\textit{Parameterwert}\}$$

Der Parametername darf dabei aus den Zeichen a–z, A–Z, 0–9 und _ bestehen, der Wert kann ein String, ein Ganzzahl- oder ein Fließkommawert sein.

Werden zusätzliche Parameter bei einem Sende- oder Weiterleitungsereignis angegeben, das mit zugehörigen Empfangsereignissen korreliert werden kann, so stehen die Parameter auch im Empfangsereignis zur Verfügung.

Bei einem MAC-Sendeereignis angegebene Parameter sind auch bei der erzeugten Übertragung als globale Übertragungsparameter verfügbar, bei einem MAC-Empfangsereignis stehen sie als empfängerspezifische Parameter zur Verfügung.

Tracefilezeilen mit in dieser Weise zusätzlich angehängten Parametern wären zum Beispiel:

```
s 2.622699957 _1_ MAC  --- 0 RTS 44 [57e 0 1 0] {someparameter: 12.345} {anotherone: -33}
r 2.630339457 _1_ MAC  --- 0 cbr 532 [13a 1 0 800] ------- [0:0 8:0 30 1]\
  [0] 1 0 {myparam: hello}
```

In Tracefiles im neuen Format funktioniert das in gleicher Weise:

```
s -t 4.620886823 -Hs 24 -Hd -2 -Ni 24 -Nx 525.00 -Ny 525.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 57e -Md c -Ms 18 -Mt 0  {p: 9.876}
r -t 4.621239648 -Hs 12 -Hd -2 -Ni 12 -Nx 350.00 -Ny 350.00 -Nz 0.00 -Ne -1.000000\
  -Nl MAC -Nw --- -Ma 57e -Md c -Ms 18 -Mt 0  {q1: avalue} {q2: anothervalue}
```

A.4.2 Benutzerdefinierte Ereignisse

Das Auftreten eines beliebigen benutzerdefinierten Ereignisses lässt sich durch eine Tracefilezeile ausdrücken, die mit dem Kleinbuchstaben „e“ beginnt. Darauf folgen durch Leerzeichen getrennt die nachfolgenden Felder:

Feld	**Datentyp, Repräsentation**
Zeitpunkt	double, dezimal
Knoten-ID	int, dezimal
Ereignistyp	string aus den Zeichen a–z, A–Z, 0–9 und _

Danach können als zusätzliche Information weitere Ereignisparameter folgen, die in der oben definierten Form als Name/Wert-Paare in geschweiften Klammern übergeben werden.

Tracefilezeilen für benutzerdefinierte Ereignisse könnten wie folgt aussehen:

```
e 1.045262485 1 anEventType
e 1.534355454 2 anotherEventType {parameter: 17} {anotherparameter: some text}
e 2.424534002 2 anEventType {someparameter: 2.334}
```

A.4.3 Benutzerdefinierte Zustände

In ähnlicher Weise wie benutzerdefinierte Ereignisse lassen sich auch zusätzliche Zustände definieren. Entsprechende Tracefilezeilen beginnen mit dem Kleinbuchstaben „c“, gefolgt von diesen Feldern:

Feld	**Datentyp, Repräsentation**
Zeitpunkt	double, dezimal
Knoten-ID	int, dezimal
Zustandsbezeichner	string aus den Zeichen a–z, A–Z, 0–9 und _
Änderungsrate	double, dezimal, 0 für nicht-numerische Zustände
Zustandswert	int, dezimal oder double, dezimal oder string

Die Änderungsrate bezeichnet die Veränderung des Zustandswertes eines numerischen Zustands pro Sekunde Simulationszeit.

Benutzerdefinierte Knotenzustände könnten etwa durch folgende Tracefilezeilen definiert werden:

```
c 3.063743625 34 someState 0 this state has a string value
c 3.957383111 12 state2 -1 77
c 5.475622442 7 state2 0.35 12.445
```

Anhang B

Beweis der Minimalität für Gleichverteilung

In Abschnitt 6.5.4 blieb der Beweis offen, dass unter den Voraussetzungen

$$\begin{aligned} &\forall i \in \{0, \ldots, m\} : |I_i| \geq 0 \\ &\sum_{i=0}^{m} |I_i| = n + 1 \end{aligned} \tag{B.1}$$

für beliebiges $n \geq 0$ die Summe $\sum_{i=0}^{m} |I_i|^2$ genau dann minimal wird, wenn gilt

$$\forall i, j \in \{0, \ldots, m\} : |I_i| = |I_j| = \frac{n+1}{m+1} \tag{B.2}$$

Dieser Beweis soll hier geführt werden. Angewendet wird hierfür das Prinzip der vollständigen Induktion.

Induktionsanfang: Sei $m = 0$. Für $m = 0$ muss $|I_0| = n + 1$ gelten, damit die Voraussetzungen erfüllt sind. Es folgt unmittelbar die Behauptung.

Induktionsannahme: Für m gilt die Behauptung.

Induktionsschritt: $m \longrightarrow m + 1$

Vorausgesetzt wird nun:

$$\sum_{i=0}^{m+1} |I_i| = n + 1 \tag{B.3}$$

Dies lässt sich umformen zu:

$$\sum_{i=0}^{m} |I_i| = n + 1 - |I_{m+1}| \tag{B.4}$$

Mit der Induktionsannahme folgt daraus, dass $\sum_{i=0}^{m} |I_i|^2$ für gegebenes $|I_{m+1}|$ minimal ist genau dann, wenn

$$\forall i, j \in \{0, \ldots, m\} : |I_i| = |I_j| = \frac{n + 1 - |I_{m+1}|}{m + 1} \tag{B.5}$$

Der minimale Wert dieser Teilsumme errechnet sich damit zu:

$$\sum_{i=0}^{m} |I_i|^2 = (m + 1) \left(\frac{n + 1 - |I_{m+1}|}{m + 1} \right)^2 = \frac{(n + 1 - |I_{m+1}|)^2}{m + 1} \tag{B.6}$$

Für die gesamte Summe $\sum_{i=0}^{m+1} |I_i|^2$ ergibt sich daraus – noch immer abhängig von $|I_{m+1}|$ – der folgende Wert:

$$\sum_{i=0}^{m+1} |I_i|^2 = \sum_{i=0}^{m} |I_i|^2 + |I_{m+1}|^2 = \frac{(n + 1 - |I_{m+1}|)^2}{m + 1} + |I_{m+1}|^2 \tag{B.7}$$

Bestimme nun $|I_{m+1}|$ so, dass dieser Ausdruck insgesamt minimal wird. Betrachte hierfür obige Summe als Funktion von $|I_{m+1}|$ und leite diese nach $|I_{m+1}|$ ab. Dies ist möglich, da die Funktion ein Polynom in $|I_{m+1}|$ und damit beliebig oft nach $|I_{m+1}|$ differenzierbar ist:

$$\frac{d}{d|I_{m+1}|} \left(\frac{(n + 1 - |I_{m+1}|)^2}{m + 1} + |I_{m+1}|^2 \right) = \frac{-2n - 2 + 2|I_{m+1}|}{m + 1} + 2|I_{m+1}| \tag{B.8}$$

Durch Nullsetzen erhält man eine Bedingung für eine Extremstelle:

$$\begin{aligned} & \frac{-2n - 2 + 2|I_{m+1}|}{m + 1} + 2|I_{m+1}| = 0 \\ \iff \quad & \frac{-n - 1 + |I_{m+1}|}{m + 1} + |I_{m+1}| = 0 \\ \iff \quad & |I_{m+1}| = \frac{n + 1 - |I_{m+1}|}{m + 1} \end{aligned} \tag{B.9}$$

Die zweite Ableitung der betrachteten Summe ergibt sich konstant zu:

$$\frac{d}{d|I_{m+1}|^2} \left(\frac{(n + 1 - |I_{m+1}|)^2}{m + 1} + |I_{m+1}|^2 \right) = \frac{2}{m + 1} + 2 > 0 \tag{B.10}$$

Also handelt es sich bei der gefundenen Extremstelle um ein Minimum. Aus (B.5) und (B.9) folgt für dieses Minimum die Behauptung:

$$\forall i, j \in \{0, \ldots, m + 1\} : |I_i| = |I_j| = \frac{n + 1 - |I_{m+1}|}{m + 1} = \frac{n + 1}{m + 2} \tag{B.11}$$

□

Anhang C

Quelltexte der Queue-Implementationen

In Abschnitt 6.7 wurden Messwerte für die Zahl der Zyklen angegeben, die eine Heap-basierte Warteschlange und eine auf Basis des Algorithmus 2PQ in gleicher Zeit bearbeiten konnten. Hier sind die Quelltexte der beiden für die Messungen verwendeten Implementationen wiedergegeben.

Für die Erzeugung der Pseudozufallszahlen sowie für die Zeitmessung verwenden beide Implementationen die Boost-C++-Bibliotheken [BOO].

C.1 Heap-Queue

```
1  #include <iostream>
2  #include <boost/random.hpp>
3  #include <boost/timer.hpp>
4
5  boost::mt19937 rng;
6
7  int num;          // Number of elements in the heap
8  int cnum;         // Number of elements changed during each cycle
9  double duration;  // Measurement duration
10 int cycles;       // Cycle counter
11
12 boost::uint32_t* data;  // Array for the heap
13
14
15 void heapify(int pos, int num) {
16
17   int v = pos;
18   int w=2*v+1;             // left child
19   while (w<num) {
20     if (w+1<num)             // second child present?
21       if (data[w+1]>data[w]) w++;  // use right child
22
23     if (data[v]>=data[w]) return;  // done
```

```
24
25      int t = data[w];
26      data[w] = data[v];
27      data[v] = t;
28
29      v=w;
30      w=2*v+1;
31    }
32
33  }
34
35
36  int main(int argc, char** argv) {
37
38    std::cout << "Queue length: " << argv[1] << "\n";
39    std::cout << "Dynamics:     " << argv[2] << "\n";
40    std::cout << "Duration:     " << argv[3] << "\n";
41
42    num = atoi(argv[1]);
43    cnum = atoi(argv[2]);
44    duration = atof(argv[3]);
45
46    // Generate initial values
47    data = new boost::uint32_t[num];
48
49    for (int i=0; i<num; i++) {
50      data[i] = rng();
51    }
52
53    // Initialize Heap
54    std::cout << "Heapifying\n";
55    for (int i=num/2-1; i>=0; i--) {
56      heapify(i, num);
57    }
58
59    // Start timer
60    boost::timer timer;
61    cycles = 0;
62
63    while (timer.elapsed() < duration) {
64      cycles++;
65
66      int length = num;
67
68      // Dequeue cnum elements
69      for (int i=0; i<cnum; i++) {
70        data[0] = data[length];
71        length--;
72
73        // Restore heap property
74        heapify(0, length);
75      }
76
```

```
77      // Insert cnum new elements
78      for (int i=0; i<cnum; i++) {
79        data[length] = rng();
80        length++;
81
82        // Restore heap property
83        int k = length-1;
84        while (k>0) {
85          k = (k-1) / 2;
86          heapify(k, length);
87        }
88      }
89
90    }
91
92    // Time elapsed, done
93    printf("%i cycles in %g seconds\n", cycles, timer.elapsed());
94
95  }
```

C.2 2PQ

```
1   #include <iostream>
2   #include <boost/random.hpp>
3   #include <boost/timer.hpp>
4
5   boost::mt19937 rng;
6
7   int num;          // Queue length
8   int cnum;         // Number of elements changed in each cycle
9   double duration;  // Measurement duration
10
11  boost::uint32_t* data;
12
13
14  class Element {  // An entry in the 2PQ
15
16    public:
17      Element* next;
18      boost::uint32_t value;
19
20  };
21
22
23  void quicksort(int lo, int hi) {  // Hoare's quicksort algorithm
24
25    int i = lo;
26    int j = hi;
27
28    int pivot = data[(lo+hi)/2];
29
```

```
30    while (i<=j) {
31      while (data[i]<pivot) i++;
32      while (data[j]>pivot) j--;
33      if (i<=j) {
34        int t = data[i];
35        data[i] = data[j];
36        data[j] = t;
37        i++;
38        j--;
39      }
40    }
41
42    if (j>lo) quicksort(lo, j);
43    if (i<hi) quicksort(i, hi);
44  }
45
46
47  int main(int argc, char** argv) {
48
49    std::cout << "Queue length: " << argv[1] << "\n";
50    std::cout << "Dynamics:     " << argv[2] << "\n";
51    std::cout << "Duration:     " << argv[3] << "\n";
52
53    num = atoi(argv[1]);
54    cnum = atoi(argv[2]);
55    duration = atof(argv[3]);
56
57    // Generate initial values
58    data = new uint32_t[num];
59    Element* elem = new Element[num];
60
61    for (int i=0; i<num; i++) {
62      data[i] = rng();
63    }
64
65    std::cout << "Quicksorting\n";
66    quicksort(0, num-1);
67    std::cout << "Building initial queue\n";
68
69    for (int i=0; i<num; i++) {
70      elem[i].value = data[i];
71      elem[i].next = &(elem[i+1]);
72    }
73    elem[num-1].next = 0;
74
75    Element* head = &(elem[0]);
76
77    // Initialize timer
78    boost::timer timer;
79    int cycles = 0;
80
81    while (timer.elapsed() < duration) {
82      cycles++;
```

```
83
84      Element* head2 = head;
85
86      // Dequeue cnum elements
87      for (int i=0; i<cnum; i++) {
88        head = head->next;
89      }
90
91      // Produce cnum new values
92      for (int i=0; i<cnum; i++) {
93        data[i] = rng();
94      }
95
96      // Sort new values
97      quicksort(0, cnum-1);
98
99      // Create list L2; reuse previously allocated memory
100     Element* e = head2;
101     for (int i=0; i<cnum; i++) {
102       e->value = data[i];
103       if (i<cnum-1) e = e->next;
104     }
105     e->next = 0;
106
107     // Merge L1 and L2
108     e = 0;
109     Element* newhead = head->value<=head2->value ? head : head2;
110     while (head && head2) {
111       if (head->value <= head2->value) {
112         if (e) e->next = head;
113         e = head;
114         head = head->next;
115       } else {
116         if (e) e->next = head2;
117         e = head2;
118         head2 = head2->next;
119       }
120     }
121
122     if (!head) {
123       e->next = head2;
124     } else {
125       e->next = head;
126     }
127
128     head = newhead;
129
130   }
131
132   // Time elapsed, done
133   printf("%i cycles in %g seconds\n", cycles, timer.elapsed());
134
135 }
```

Literaturverzeichnis

[3Dc] 3Dconnexion website. http://www.3dconnexion.com/.

[Axe03] Stefan Axelsson. Visualization for intrusion detection hooking the worm. In *Computer Security – ESORICS 2003 – 8th European Symposium on Research in Computer Security, Gjørvik, Norway*, pages 309–325. Springer Verlag, 2003.

[BEW95] Richard A. Becker, Stephen G. Eick, and Allan R. Wilks. Visualizing network data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):16–28, 1995.

[BMB00] J. A. Brown, A. J. McGregor, and H.-W. Braun. Network performance visualization: Insight through animation. In *Proceedings of PAM2000: Passive and Active Measurement Workshop*, pages 33–41, 2000.

[BOO] The boost C++ libraries. http://www.boost.org/.

[Bro88] Randy Brown. Calendar queues: a fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[Bro96] Gerth Stølting Brodal. Worst-case efficient priority queues. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1996.

[CPA] The comprehensive perl archive network. http://www.cpan.org/.

[CSR93] Kehsiung Chung, Janche Sang, and Vernon Rego. A performance comparison of event calendar algorithms: an empirical approach. *Software—Practice & Experience*, 23(10):1107–1138, 1993.

[DEB93] Wim Diepstraten, Greg Ennis, and Phil Belanger. DFWMAC: Distributed foundation wireless medium access control. IEEE Document P802.11-93/190, November 1993.

[DS02] Charles Donnelly and Richard Stallman. Bison, the yacc-compatible parser generator, version 1.35. http://www.gnu.org/software/bison/manual/, February 2002.

[EHH+99] Deborah Estrin, Mark Handley, John Heidemann, Steven McCanne, Ya Xu, and Haobo Yu. Network visualization with the VINT network animator nam. *Technical Report 99-703, University of Southern California*, 1999.

[Fid88] Colin J. Fidge. Timestamps in message-passing systems that preserve partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, February 1988.

[FLT] The fast light toolkit FLTK. http://www.fltk.org.

[FLU] FLU – FLTK utility widgets. http://www.osc.edu/ jbryan/FLU/.

[FSF] Free software foundation. http://www.fsf.org.

[FT] The freetype project. http://www.freetype.org/.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[GCC] The GNU compiler collection. http://gcc.gnu.org/.

[GJ90] Dick Grune and Ceriel J. H. Jacobs. *Parsing techniques – a practical guide*. Ellis Horwood Limited, Chichester, England, 1990. Available online at http://www.cs.vu.nl/∼dick/PTAPG.html.

[Gri03] Richard Griswold. ns-2 traces. http://k-lug.org/∼griswold/NS2/ns2-trace-formats.html, Last changes in 2003.

[Gut04] Benjamin Guthier. Referenz zu V-IDS. http://www.v-ids.org, 2004.

[Hei90] Philip Heidelberger. Introduction—discrete event simulation. *Commununications of the ACM*, 33(10):28–29, 1990.

[Hoa62] Charles Antony Richard Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[IEE97] IEEE Computer Society LAN MAN Standards Committee. *Wireless LAN medium access control (MAC) and physical layer (PHY) specifications*. The Institute of Electrical and Electronic Engineers, New York, NY, 1997.

[Jac90] Bernhard Jacobs. Ein Vergleich der Auswirkungen graphischer und tabellarischer Präsentationsformen auf die Schnelligkeit und Genauigkeit beim Erkennen und Interpretieren statistischer Daten. *Arbeitsberichte des Medienzentrums der Universität des Saarlandes*, 3, 1990.

[Jac94] Bernhard Jacobs. Graphische vs. tabellarische Präsentation von statistischen Daten. *Zeitschrift für Pädagogische Psychologie*, 8:73–84, 1994.

[JM96] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart and Winston, New York, NY, USA, 1979.

[JSW] libjsw joystick wrapper library. http://wolfpack.twu.net/libjsw/.

[KCC04] Stuart Kurkowski, Tracy Camp, and Michael Colagrosso. A visualization and animation tool for ns-2 wireless simulations: iNSpect. Technical Report MCS-04-03, The Colorado School of Mines, June 2004.

[Kes88] Srinivasan Keshav. REAL: A network simulator. Technical report, University of California at Berkeley, 1988.

[KKB04] Hyogon Kim, Inhye Kang, and Saewoong Bahk. Radar real-time visualization of network attacks on high-speed links. *IEEE Network*, 18(5):30–39, September/October 2004.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[Lok00] Loki Software. OpenAL specification and reference, version 1.0 draft edition, 2000.

[LS79] Michael E. Lesk and Eric Schmidt. lex: A lexical analyzer generator. In *UNIX Programmer's Manual*, volume 2, pages 388–400. Holt, Rinehart and Winston, New York, NY, USA, 1979.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155. ACM Press, 1987.

[Mal01] David A. Maltz. On demand routing in multi-hop wireless mobile ad hoc networks. PhD thesis, Carnegie Mellon University. http://www.monarch.cs.cmu.edu/monarchpapers/maltz-thesis.ps.gz, 2001.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989.

[MDK] Mandrakesoft homepage. http://www.mandrakesoft.com/.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[Mon98] CMU Monarch Project. The CMU Monarch project's ad-hockey visualization tool for ns scenario and trace files. Carnegie Mellon University. http://www.monarch.cs.rice.edu/ftp/monarch/wireless-sim/ad-hockey.ps, August 1998.

[Mon99] CMU Monarch Project. The CMU Monarch project's wireless and mobility extensions to ns. Carnegie Mellon University. http://www.monarch.cs.cmu.edu, 1999.

[NCSLO02] Kofi Nyarko, Tanya Capers, Craig Scott, and Kemi Ladeji-Osias. Network intrusion visualization with niva, an intrusion detection visual analyzer with haptic integration. In *Proceedings of the 10th Symposium On Haptic Interfaces For Virtual Environment and Teleoperator Systems*, page 277. IEEE Computer Society, 2002.

[Nie94] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, San Francisco, 1994.

[ns2] The ns-2 network simulator. http://www.isi.edu/nsnam/ns/.

[OAL] OpenAL cross-platform 3D audio library. http://www.openal.org.

[OAP] OpenAL++. http://alpp.sourceforge.net/.

[OGL] OpenGl. http://www.opengl.org/.

[OSG] The OpenSG scene graph library. http://www.opensg.org.

[OTc] OTcl – MIT object tcl extensions. http://otcl-tclcl.sourceforge.net/otcl/.

[Pax95] Vern Paxson. Flex, a fast scanner generator, edition 2.5. http://www.gnu.org/software/flex/manual/html_node/flex_toc.html, March 1995.

[PB94] Charles Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[PC97] Vincent D. Park and M. Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM (3)*, pages 1405–1413, 1997.

[Plu82] David C. Plummer. An ethernet address resolution protocol: Or converting network protocol addresses to 48 bit ethernet addresses for transmission on ethernet hardware. RFC 862, November 1982.

[PR99] Charles Perkins and Elizabeth Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 99–100, February 1999.

[RA97] Robert Rönngren and Rassul Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(2):157–209, 1997.

[REA] The REAL network simulator. http://minnie.tuhs.org/REAL/.

[RMC91] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Information visualization using 3D interactive animation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 461–462. ACM Press, 1991.

[Rub] The object-oriented scripting language ruby. http://www.ruby-lang.org.

[RVB02] Dirk Reiners, Gerrit Voß, and Johannes Behr. OpenSG: Basic concepts. In *First OpenSG Symposium OpenSG 2002*, 2002.

[Shr04] Dave Shreiner. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.4*. Addison-Wesley Longman Publishing Co., Inc., 2004.

[Sim78] Karl Simrock. *Die Edda, die ältere und jüngere, übersetzt und mit Erläuterungen begleitet*, chapter Hrafnagaldr Odhins, Odhins Rabenzauber. Verlag der J. G. Cotta'schen Buchhandlung, Stuttgart, siebente verbeßerte Auflage, 1878.

[SLLG04] Björn Scheuermann, Andreas Lindenblatt, Daniela Lindenblatt, and Benjamin Guthier. Trau, SCHAU, wem? – V-IDS oder eine andere Sicht der Dinge. In *DIMVA Conference on Detection of Intrusions and Malware and Vulnerability Assessment, Kurzbeiträge*, http://www.gi-fb-sicherheit.de/fg/sidar/dimva2004/, 2004.

[SM91] Richard Stallman and Eben Moglen. The GNU general public license, version 2. http://www.gnu.org/licenses/gpl.html, June 1991.

[Str91] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, Massachusetts, USA, 1991.

[Tcl] Tcl and Tk. http://www.tcl.tk/.

[TFC+89] L. A. Treinish, J. D. Foley, W. J. Campbell, R. B. Habor, and R. F. Gurwitz. Effective software systems for scientific data visualization. In *ACM SIGGRAPH 89 Panel Proceedings*, pages 111–136. ACM Press, 1989.

[TH00] Dave Thomas and Andy Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2001.

[VID] V-IDS project. http://www.v-ids.org.

[VIN03] The VINT Project. The ns manual (formerly ns notes and documentation). http://www.isi.edu/nsnam/ns/doc/index.html, December 2003.

[Wik] Wikipedia: Ad hoc protocol list. http://en.wikipedia.org/wiki/Ad_hoc_protocol_list. Artikelrevision vom 24. November 2004.

[ZLB+87] Thomas G. Zimmerman, Jaron Lanier, Chuck Blanchard, Steve Bryson, and Young Harvill. A hand gesture interface device. In *Proceedings of the SIGCHI/GI conference on Human factors in computing systems and graphics interface*, pages 189–192. ACM Press, 1987.

Stichwortverzeichnis

Z