

Konzeption und Implementierung einer Middleware zur Manipulation von Netzwerkdiensten und -verbindungen in einem hochverfügbaren Gateway-Cluster

Diplomarbeit im Fach Informatik

vorgelegt von
Alexander von Gernler
geboren am in

Institut für Informatik, Lehrstuhl für Informatik IV:
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Prof. Dr.-Ing. Wolfgang Schröder-Preikschat¹
Dr. Jürgen Kleinöder¹
Dr. Magnus Harlander²
Anton Röckseisen²
Steffen Ullrich²
Beginn: 01. Februar 2005
Abgabe: 19. September 2005

¹Universität Erlangen-Nürnberg

²GeNUA mbH, Kirchheim bei München

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.

Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Alexander von Gernler

Erlangen, 19. September 2005

Alexander von Gernler hält alle Rechte an der erstellten Diplomarbeit in Form des vorliegenden Skripts. Die GeNUA mbH hält die Rechte an dem im Rahmen der Arbeit zum Produkt GeNUGate hinzugefügten Quellcode.

GeNUGate ist ein eingetragenes Markenzeichen der GeNUA mbH, Kirchheim. UNIX ist ein eingetragenes Markenzeichen der Open Group. Alle anderen verwendeten Markenzeichen werden ebenfalls anerkannt.

Kurzzusammenfassung

Das von der Firma GeNUA mbH in Kirchheim bei München hergestellte Produkt *GeNUGate* ist eine zweistufige Hochsicherheits-Firewall, bestehend aus Paketfilter (PFL) und Application-Level-Gateway (ALG).

In dieser Diplomarbeit wird der Relay-Mechanismus des ALGs um eine vorher nicht vorhandene Schnittstelle zur Rekonfiguration der Relay-Prozesse und Manipulation von laufenden Netzwerkverbindungen erweitert. Besonderes Augenmerk wird hierbei darauf gerichtet, dass die Schnittstelle nicht nur performant und einfach erweiterbar ist, sondern dem speziellen Einsatz des GeNUGates im Cluster und im Betrieb auf Multiprozessormaschinen gerecht wird.

Dafür wird zunächst eine allgemeine Anforderungsanalyse durchgeführt, die die speziellen Gegebenheiten bei der Firma GeNUA berücksichtigt. Aus diesen Erkenntnissen leitet sich dann ein umfangreicher Entwurf ab. Die Implementation der Middleware und einer Textkonsole als Referenzoberfläche (*proof of concept*) bilden einen großen Teil der Arbeit.

Eine kritische Rückschau und ein Blick auf weitere Möglichkeiten zum Ausbau des Projekts – etwa Ideen für eine graphische Oberfläche – runden die Arbeit inhaltlich ab.

Abstract

The company GeNUA mbH in Kirchheim/Munich, Germany, manufactures the *GeNUGate*, a two tier firewall consisting of a packet filter (PFL) and an application level gateway (ALG).

This diploma thesis describes the extension of the ALG's relay mechanism by an interface to reconfigure relay processes and manipulate established network connections. Great care was taken to develop not just a high-performance and easily extendable interface, but also to integrate cluster and multiprocessor capabilities for the specialized use on the GeNUGate.

A general requirement analysis was performed that took the specific needs of GeNUA into account. A detailed draft was compiled from the findings. The middleware implementation, as well as the design of a text console as proof of concept comprised a major part of the development effort.

A critical review and outlook on possible additions to the project – e.g., a graphical interface – complete this thesis.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Themenfeld	1
1.2	Problemstellung	2
1.3	Ziele der Arbeit	3
1.3.1	Systemidee	3
1.3.2	Vorgehen	3
1.3.3	Eigener Anspruch	4
1.4	Konventionen	4
1.4.1	Unix Manual Seiten	4
1.4.2	Autorennamen	5
1.4.3	Fachbegriffe	5
1.4.4	Wissenschaftliche <i>ich/wir</i> -Form	5
1.4.5	Abkürzungen	5
2	Analyse	7
2.1	Terminologie	7
2.1.1	Proxies	7
2.1.2	Netzwerkmodelle	8
2.1.3	Nomenklatur	10
2.2	Abstrakte Beschreibung der Ausgangssituation	10
2.2.1	Sicherheitskonzept des GeNUGate	12
2.2.2	Verteilte Relay-Objekte	15
2.2.3	Spezialisierung der Relays	15
2.2.4	Vater-Kind Aufgabenteilung der Relays	15
2.2.5	Modus der Dienstbereitstellung	17
2.2.6	Überwachung und Neustart von Diensten	17
2.2.7	Hochverfügbarkeitskonzept	18
2.2.8	Zugriffskontrolle	18
2.2.9	Konfigurationsmechanismus	20
2.3	Anforderungsanalyse	20
2.3.1	Identifizierung der Interessenhalter	21
2.3.2	GeNUA-spezifische Anforderungen	23
2.4	Verwandte Problemstellungen	24

2.4.1	Interprozesskommunikation	24
2.4.2	Beispiele für Frontend-Backend Kommunikation . .	27
2.4.3	Verteilte Systeme	29
2.4.4	Netzwerkmanagement	30
2.4.5	Clustercomputing	31
2.4.6	Sicherheitskonzepte in der Softwaretechnik	32
3	Middleware	36
3.1	Konzept	36
3.2	Entwurf	41
3.3	Implementierung	45
3.4	Funktionsumfang	52
4	Benutzeroberfläche	54
4.1	Textuelle Oberfläche	54
4.1.1	Konzept	54
4.1.2	Entwurf	54
4.1.3	Implementierung	55
4.2	Ansätze für eine Graphische Oberfläche	57
4.2.1	Konzept	57
4.2.2	Entwurf	58
4.2.3	Implementierung	58
5	Fazit	59
5.1	Resultate und Diskussion	59
5.1.1	Teststellungen	59
5.1.2	Not “yet another” Middleware	61
5.2	Verwandte Arbeiten	62
5.2.1	Jackal DSM	62
5.2.2	Cactus	63
5.2.3	AspectIX	63
5.3	Erweiterungsmöglichkeiten	63
5.3.1	Wissenschaftliche Perspektive	63
5.3.2	GeNUA	64
5.3.3	Handwerk und guter Stil	65
5.4	Kritischer Rückblick auf die Projektplanung	66
A	Praktische Arbeit	69
A.1	Knotencontroller <code>commd</code>	69
A.1.1	Konfigurationsdatei <code>commd.conf</code>	69
A.1.2	Manualeite <code>commd(8)</code>	69
A.1.3	Verzeichnisansicht des Quellcodes	71
A.1.4	Ausgabe eines Aufrufs von <code>commd -dvv</code>	71
A.1.5	Ansicht des <code>commd</code> in der Prozessliste	72

Abbildungsverzeichnis

1.1	GeNUGate Schema	2
2.1	Graphische Darstellung der Nomenklatur	11
2.2	PAP-Firewall-Modell	13
2.3	Sicherheitskonzept der Relays	14
2.4	Relay Mechanismus status quo	16
2.5	Hochverfügbarkeits-Szenario	19
2.6	Ausgebeutete Subroutine	34
3.1	Schichtmodell Kommunikation	40
3.2	Vollständige Graphen	46
3.3	Kommunikationsmodell der Relays	49
4.1	EBNF der Kommandozeilensprache	56

Tabellenverzeichnis

2.1	ISO/OSI und TCP Stack	9
2.2	Nomenklatur	11
2.3	Auflistung der Interessenhalter	22
2.4	IPC Mechanismen	24
2.5	Herkömmliche vs. Verteilte Systeme	29
2.6	VMI/MPI-Architektur	31
3.1	Kommunikationsschicht als OSI Modell	43
3.2	Wachstum von $ K_n $	44
3.3	Kommunikationsschicht und Implementierung	46
3.4	Methoden von <code>IMSG.pm</code>	48
4.1	Aufbau Kommandozeilenschnittstelle	57

Kapitel 1

Einleitung

Dieses Kapitel beschreibt die Themenstellung und das vorliegende Problem, und stellt eine Strategie zur Lösung desselben im Rahmen der Diplomarbeit auf. Konventionen, die dem strukturierten Vorgehen in der Arbeit zu Grunde liegen, sind in Abschnitt 1.4 aufgeführt.

1.1 Themenfeld

Mit dem Auftreten signifikanter wirtschaftlicher Schäden durch Ausfall von Rechnersystemen ist Computersicherheit verstärkt ins öffentliche Bewußtsein gerückt. Sei es wegen Würmern, Viren, Trojanern und Sicherheitslücken in Betriebssystemen, oder einfach wegen des Datenschutzes – kaum eine Institution kann es sich leisten, unkontrollierten und unbefugten Zugriff von außen auf ihre interne Netze zu geben.

Die seit 1992 existierende Firma GeNUA mbH¹ in Kirchheim bei München stellt BSI²-zertifizierte Sicherheitslösungen für den Einsatz in Behörden- oder Firmennetzen her. Das Produkt GeNUGate, das im Rahmen dieser Diplomarbeit um Funktionalität erweitert werden soll, ist eine zweistufige Firewall, bestehend aus Paketfilter und Application-Level-Gateway (Abbildung 1.1). Beide Systeme laufen jeweils auf eigenen, voneinander unabhängigen physikalischen Rechnern und sind nur per Netzkabel miteinander verbunden.

Der Application-Level-Gateway (ALG) wird durch Userland-Proxies (also Software, die auf Anwendungs- und nicht Kernebene arbeitet und Proxy-Funktionalität bereitstellt, im Weiteren als *Relay* bezeichnet) realisiert. Die Relays bieten jeweils einzelne Internet-Dienste (z. B. HTTP, SMTP) gegenüber dem internen Netz an und reichen die betreffenden Pakete nach Überprüfung des Inhalts nach außen durch. Auch ein transparentes Weiterreichen nicht proxy-fähiger Protokolle ohne spezielle Anpassung

¹<http://www.genua.de/>

²Bundesamt für Sicherheit in der Informationstechnik, <http://www.bsi.bund.de/>

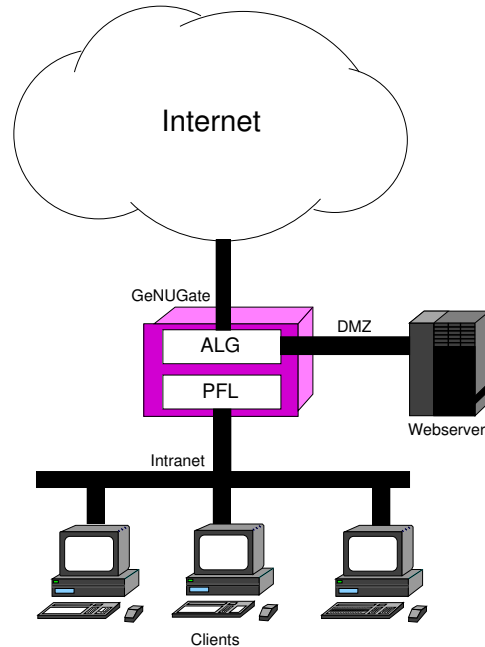


Abbildung 1.1: Schematischer Aufbau und Einsatz des GeNUGates als zweistufige Firewall. Quelle: GeNUA

der Anwendungen ist möglich. Der ALG ist sowohl SMP- als auch clusterfähig, d. h. ein Betrieb auf mehreren Prozessoren einer Maschine, sowie ein gleichzeitiges Anbieten des ALG auf mehreren Maschinen per Round-Robin-DNS oder OSPF³ ist möglich.

GeNUGates werden an prominenten Stellen mit hohen Sicherheits- und Performanceanforderungen eingesetzt, etwa im Deutschen Bundestag⁴.

1.2 Problemstellung

Das Manko der zu Beginn der Diplomarbeit bestehenden Implementierung der Relays ist die unzureichende Kontrolle, die man als Administrator über die einzelnen Prozesse des ALG-Clusters besitzt. So war es bisher nur möglich, den Relays POSIX [27] Signale (SIGUSR1, SIGUSR2, SIGALRM usw.) zu schicken, woraufhin diese ihr Verhalten vordefiniert ändern. Dieser Mechanismus soll im Rahmen der Diplomarbeit durch eine mächtigere Form der Kommunikation ersetzt werden.

³Open Shortest Path First, ein Routing-Protokoll zum Routing innerhalb Autonomer Systeme (AS), sog. *interior gateway routing protocol*. Das heute verwendete OSPFv2 wird beschrieben in RFC 2328 [25].

⁴GeNUA listet prominente Referenzkunden auf ihrer Webseite auf: <http://www.genua.de/genua/kunden/index.html>

Beabsichtigt ist, die Relays in Zukunft nicht nur einzeln, sondern in ihrer Gesamtmenge (die durchaus über einen Cluster bestehend aus Multiprozessormaschinen verteilt sein kann) regelbasiert zu verwalten. Hierfür ist nicht nur die Konzeption einer Nachrichtenschnittstelle zu einzelnen Relays erforderlich. Es muss auch ein Mechanismus entworfen werden, der die Gesamtheit oder Untermengen der Relays regelbasiert und ortstransparent ansprechen und manipulieren kann.

Als Frontend dieser Schnittstelle sind mehrere verschiedene Kombinationen aus Views und Controllern⁵ denkbar, so etwa eine Textkonsole und ein Webfrontend.

1.3 Ziele der Arbeit

Um ein systematisches Vorgehen zu erreichen, verwende ich für die Erstellung dieser Diplomarbeit Teile des sog. *Object Engineering Process* von OESTERREICH [26]. Auch die Aufgabenstellung dieser Diplomarbeit formuliere ich deshalb gleich als Systemidee laut OEP.

1.3.1 Systemidee

Auf dem Produkt GeNUGate der Firma GeNUA mbH, Kirchheim, soll die clusterorientierte Ansteuerung der Relay-Prozesse durch Einführung einer Schnittstelle ermöglicht werden.

Das neu zu entwickelnde Untersystem soll eine Kommunikationsschicht bereitstellen, die entferntes Aufrufen von Methoden auf den Relays ungeachtet ihrer Lage oder Funktion im Cluster gestattet. Dazu gehört die Referenzimplementation einer Basismenge aufrufbarer Funktionen und das Vorsehen zukünftiger einfacher Erweiterbarkeit. Außerdem ist eine textorientierte Bedienoberfläche als *proof of concept* zu entwerfen und zu implementieren, sowie Ideen für die Darstellung des Modells in einer graphischen Oberfläche zu erarbeiten.

Eine Neuimplementation bisher nicht existierender Funktionalität in den Relays, sowie die Produktion einer graphischen Oberfläche sind nicht Bestandteil dieser Diplomarbeit.

1.3.2 Vorgehen

Um die genannten Ziele zu erreichen, setze ich folgende Schritte an:

1. Aufstellung einer Terminologie und abstrakte Beschreibung der bisher vorliegenden Architektur. Dies wird in Abschnitt 2.2 behandelt.

⁵gemäß dem MVC-Paradigma bestehend aus Model, View und Controller

2. Allgemeine Anforderungsanalyse des Projekts. Verallgemeinerung des GeNUGate-Problems auf generische Probleme aus der Informatik. Siehe hierzu Abschnitt 2.3.
3. Spezielle Anforderungsanalyse, bedingt durch die existierende Architektur und die Vorstellungen der Firma GeNUA. Dies wird in Abschnitt 2.3.2 beschrieben.
4. Realisierung der Schnittstelle in Kapitel 3.
5. Implementation einer Benutzeroberfläche in Kapitel 4.

1.3.3 Eigener Anspruch

Ich setze mir zum Ziel, nicht nur die Problemstellung der Firma GeNUA umfassend zu bearbeiten, sondern dabei auch eine möglichst sauber implementierte, fehlerfreie, plattform- und sprachunabhängige (im Bezug auf die Sprache, in der die Relays realisiert sind) sowie gut dokumentierte Lösung zu erstellen. Entstehender Code soll idealerweise übersichtlich strukturiert und gut wiederverwendbar sein.

1.4 Konventionen

In dieser Arbeit verwende ich bestimmte Notationen, die ich im Folgenden kurz erläutere.

1.4.1 Unix Manual Seiten

Begriffe, die in gängigen Unix-Varianten im Manual System nachgeschlagen werden können, werden unter Angabe des jeweiligen Kapitels als Maschinenschrift dargestellt. So gibt z. B. ein `write(2)` an, dass von dem UNIX-Systemaufruf `write` im Kapitel 2 („Systemaufrufe und Fehlercodes“) des Manual Systems die Rede ist.

Als Referenz für die Manualseiten wird aus mehreren Gründen das Handbuchsystem von OpenBSD [9] verwendet: Es handelt sich im Vergleich zu den Manualseiten der gängigen Linux-Distributionen um ein sehr ausführliches und aktuelles Werk. Die Manualseiten sind auch ohne lokale OpenBSD-Installation komplett per WWW⁶ abrufbar. Das OpenBSD-Projekt bemüht sich um eine strenge Einhaltung der für UNIX relevanten Standards (vgl. POSIX, Unix 98, Single Unix Specification [27]). Außerdem ist das komplette GeNUGate ab Version 6.0 auf OpenBSD realisiert und wird auch unter OpenBSD entwickelt.

⁶<http://www.openbsd.org/cgi-bin/man.cgi>

Ohne eine in Klammern folgende Kapitelnummer ist davon auszugehen, dass es sich um ein Programm ohne zugeordnete Handbuchseite handelt, das nicht in der Standardinstallation von OpenBSD enthalten ist.

1.4.2 Autorennamen

Namen von zitierten Autoren oder Bekanntheiten aus dem Informatikfeld werden in Kapitälchen notiert, z. B. W. RICHARD STEVENS.

1.4.3 Fachbegriffe

Bei der Erstellung der Arbeit versuche ich, einen möglichst flüssig lesbaren deutschen Text zu produzieren. Um die diskutierten Begriffe trotzdem auch beim Lesen englischsprachiger Referenzliteratur einordnen zu können, gebe ich (oft in Klammern) den zugehörigen englischen Begriff in *kursiver* Schreibweise an.

So ist in meiner Arbeit z. B. desöfteren von Knoten (*Nodes*) die Rede.

1.4.4 Wissenschaftliche *ich/wir*-Form

Ich benutze in meiner Arbeit ganz bewußt die wissenschaftlichen Formen *ich* und *wir*, um unterschiedliche Umstände (Beschreibung eines eigenen Weges, Einbezug des Lesers bzw. Veranschaulichung) zu verdeutlichen. Die unterschiedlichen Nuancen der wissenschaftlichen Anrede sind bei DEININGER et al. [11] nachzulesen.

1.4.5 Abkürzungen

ALG	Application Level Gateway
API	Application Programming Interface
BSD	Berkeley Software Distribution
DA	Diplomarbeit
EBNF	Erweiterte Backus-Naur-Form
GG	GeNUGate
GUI	Graphical User Interface
HA	High Availability
LOC	Lines of Code
OEP	Object Engineering Process
OSPF	Open Shortest Path First
P2P	Peer to Peer

PFL Paketfilter
RPC Remote Procedure Call
SSL Secure Socket Layer

Kapitel 2

Analyse

In diesem Kapitel werden wir für die Arbeit relevante Begriffe definieren (Abschnitt 2.1). Es folgt eine abstrakte Beschreibung der Ausgangssituation (Abschnitt 2.2) und eine Anforderungsanalyse (Abschnitt 2.3). Das so erhaltene Spektrum an Lösungsansätzen schränken wir in Abschnitt 2.3.2 durch die speziellen Vorgaben und Anforderungen der Firma GeNUA weiter ein.

2.1 Terminologie

2.1.1 Proxies

Obwohl dieser Begriff sehr oft nur in einer Spezialbedeutung im Zusammenhang mit Web-Browsern verwendet wird, bedeutet das Wort auf deutsch zunächst nur „Stellvertreter“. Es handelt sich im allgemeinen Fall nicht um einen Netzwerkdienst, sondern um ein Entwurfsmuster für Software [14].

Definition 2.1 (Proxy) *Ein Objekt, das als vorgelagerter Stellvertreter für das wirklich gemeinte Objekt fungiert. GAMMA et al. [14] unterscheiden vier Varianten der Proxy-Funktion:*

1. Ein **Remote-Proxy** stellt einen lokalen Stellvertreter für ein Objekt in einem anderen Adreßraum dar.
2. Ein **virtueller Proxy** stellt einen Platzhalter für teure Objekte dar, die erst im letzten Moment erzeugt oder angesprochen werden sollen.
3. Ein **Schutzproxy** kontrolliert den Zugriff auf das Originalobjekt.
4. Eine **Smart-Reference** (Smart Proxy) ist ein Ersatz für einen einfachen Zeiger, der zusätzliche Aktionen ausführt, wenn auf das Objekt zugegriffen wird. Typische Verwendungen umfassen:

- das Zählen der Referenzen auf das eigentliche Objekt
- das Laden eines persistenten Objekts in den Speicher bei der ersten Dereferenzierung
- das Testen, ob das eigentliche Objekt gesperrt (locked) ist, bevor darauf zugegriffen wird

Dem allgemeinen Begriff gegenüber steht die gewöhnliche Verwendung des Wortes *Proxy*:

Definition 2.2 (Caching Proxy) *Eine Spezialisierung des Smart Proxy, die häufig gestellte Anfragen und die dazugehörigen Antworten zwischenspeichert und bei Bedarf wieder zur Verfügung stellt.*

Eine bestimmte Ausprägung des Caching Proxy ist den meisten Benutzern eines Web-Browsers bekannt:

Beispiel 2.1 (HTTP-Proxy) *Ein Webproxy zur Beschleunigung von Anfragen, deren Antwort bereits bekannt ist. Dadurch wird die Netzwerklast einer Außenanbindung signifikant verringert. Eine sehr verbreitete Open Source Implementation eines Caching Proxys für HTTP ist das Programm `squid` [42].*

Im Rahmen dieser Diplomarbeit werden wir oft das Wort *Relay* benutzen:

Beispiel 2.2 (Relay) *Ein Remote-Proxy mit Schutzfunktion und zusätzlichen Smartfunktionen (z. B. hier speziell Virenskan, Entfernen von Java und JavaScript) für den Netzwerkverkehr über das in der Arbeit besprochene GeNUGate. Insbesondere wird bei GeNUA als Relay der laufende Prozess bezeichnet, der selbst wieder mehrere Relay-Objekte beherbergt, die dann jeweils eine einzelne Verbindung abwickeln.*

2.1.2 Netzwerkmodelle

ISO/OSI Modell Im Jahre 1982 begann die *International Organization for Standardization (ISO)* mit der Gründung einer neuen Institution zur Normierung von Netzwerktechnik, der *Open Systems Interconnect (OSI)*. Große Bekanntheit hat das 7-Schichten Modell (Tabelle 2.1) erlangt, das verschiedene Ebenen des Netzwerkverkehrs definiert. Da das Modell für viele Anwendungen zu umfangreich ist, wird heute für den praktischen Einsatz das sog. DoD¹-Modell verwendet, das z. B. den TCP-Stack hinreichend genau abbilden kann. Trotzdem ist das OSI-Modell für theoretische Betrachtungen sehr nützlich. Das OSI Projekt selbst wurde 1996 eingestellt.²

Eine Gegenüberstellung des OSI-Modells mit dem TCP/IP-Modell, sowie eine kritische Diskussion über diese beiden Modelle ist bei TANENBAUM [41] zu finden.

¹Department of Defense

²Informationen von Wikipedia [46]

	ISO/OSI		DoD/TCP	Manifestation
7	Application	Anwendung	Application	HTTP SMTP SNMP
6	Presentation	Darstellung	—	ASN.1 XML SMB
5	Session	Sitzung		TLS RPC NetBIOS
4	Transport	Transport	Host-to-Host	TCP UDP
3	Network	Vermittlung	Internet	IP ICMP IGMP
2	Data Link	Sicherung	Link	Ethernet PPP FDDI
1	Physical Link	Bitübertrg.		Laser Funk Elektr.

Tabelle 2.1: Modell des ISO/OSI Stacks (englisch und deutsch) im Vergleich zum DoD-Modell des TCP/IP Stacks, jeweils mit konkreten Beispielen aus der Praxis
Quelle: Wikipedia [46]

Definition 2.3 (Router) Ein Router verrichtet im Netzwerk auf OSI-Schicht 3 die Aufgabe, Netzwerkpakete je nach Quell- und Zieladresse sowie anderen protokollspezifischen Eigenschaften über ausgesuchte Pfade zum Empfänger weiterzuleiten. Hierbei bleibt der Inhalt der Pakete (OSI-Schichten 4 und höher) vollkommen unbeachtet.

Oft auf Routern (aber nicht nur dort) eingesetzt werden Paketfilter.

Definition 2.4 (Paketfilter) Ein Mechanismus, der auf OSI-Schicht 3 und 4 (evtl. auch 2) arbeitet und Netzwerkpakete beim Passieren auf Grund von festgelegten Regeln entfernen, modifizieren oder unverändert weiterleiten kann. Die Menge an Filterregeln stellt den lokalen Zustand des Paketfilters dar.

Obwohl die Filterung meist auf OSI-Schicht 3 und 4 stattfindet, ist es möglich, einen Paketfilter transparent z. B. auf einer sog. *Bridge* (OSI Schicht 2) zu installieren. So werden Pakete auf dem Weg von einem Netzsegment in ein anderes gefiltert, ohne dass das Gerät selbst die Funktion eines Routers wahrnimmt.

Definition 2.5 (Zustandsbehafteter (Stateful) Paketfilter) Paketfilter mit der Eigenschaft, die Semantik durchgeleiteter Pakete auf Schicht 3 zu verstehen und darauf mit Änderung des eigenen Zustands reagieren zu können.

Stateful filtering ist heute bei praktisch allen Paketfilterimplementationen auf freien UNIX-artigen Betriebssystemen vorzufinden.

Beispiel 2.3 (Zustandsbehaftete Paketfilter unter Linux und BSD) Die Paketfilterimplementationen von Linux³ und den freien BSD-Varianten Free-, Net-

³<http://www.netfilter.org/>

und OpenBSD (pf(4)) erlauben durch das dynamische Erstellen von Regeln beispielsweise, den Rückkanal einer bereits aufgebauten TCP-Verbindung passieren zu lassen. Eine Variante ist das Erlauben von beliebigen, zur Sitzung gehörenden FTP-Datenverbindungen, nachdem erfolgreich eine FTP-Kontrollverbindung aufgebaut wurde.

Definition 2.6 (Application Level Gateway) Ein Mechanismus, der ankommende Pakete auf OSI-Schicht 4 und höher analysiert und Netzwerkpakete auf Grund von festgelegten Regeln verwerfen, modifizieren oder unverändert weiterleiten kann.

Die Arbeitsweise des ALGs ist vergleichbar mit der eines Paketfilters, nur findet eine Überprüfung des *Inhalts* statt. Bei der Entscheidung, wie ein Paket behandelt werden soll, können aber auch Informationen aus niedrigeren Protokollschichten, z. B. die Absenderadresse, mit einbezogen werden.

FRANK TRÖGER [43] liefert in seiner Diplomarbeit folgende

Definition 2.7 (Cluster) Ein Cluster stellt einen Parallelrechner mit verteiltem Speicher dar. Darauf laufende Anwendungen werden unterteilt in die Gruppe, die den Adreßraum als verteilt begreift und expliziten Nachrichtenaustausch zur Kommunikation durchführt, und in die Gruppe, die den Adreßraum als gemeinsam wahrnimmt und diese Sicht durch impliziten Nachrichtenaustausch simuliert.

Beispiel 2.4 (GeNUGate HA-Cluster) Im Falle des GeNUGate HA-Clusters liegt ein extrem schwach gekoppelter Cluster vor: Der Speicher wird prinzipiell als verteilt begriffen, und die einzelnen Knoten im Verbund sind praktisch autark. Das Modell des Parallelrechners wird dadurch erfüllt, dass die Knoten gemeinsam und verteilt an der Aufgabe arbeiten, einkommende Pakete nach bestimmten Regeln und Heuristiken zu prüfen und weiterzuleiten.

2.1.3 Nomenklatur

Wir werden des Öfteren eine graphentheoretische Betrachtung der Relay- und P2P-Konstellationen vornehmen. Hierfür lege ich die in Tabelle 2.2 beschriebenen Symbole fest. Die Nomenklatur ist in Abbildung 2.1 veranschaulicht. Zur Bedeutung des Kästchens *OSPF* verweise ich auf Abschnitt 2.2.7, für die Nomenklatur hat es keine weitere Bedeutung. Eine hervorragende Einführung in die wichtigsten Grundbegriffe der Graphentheorie gibt STREHL in "Topics in Algorithms and Complexity" [40].

2.2 Abstrakte Beschreibung der Ausgangssituation

Wir sind bereits in Abschnitt 1.1 auf die konkrete Situation eingegangen und werden diese im Zuge der Implementation in Kapitel 3 noch einmal

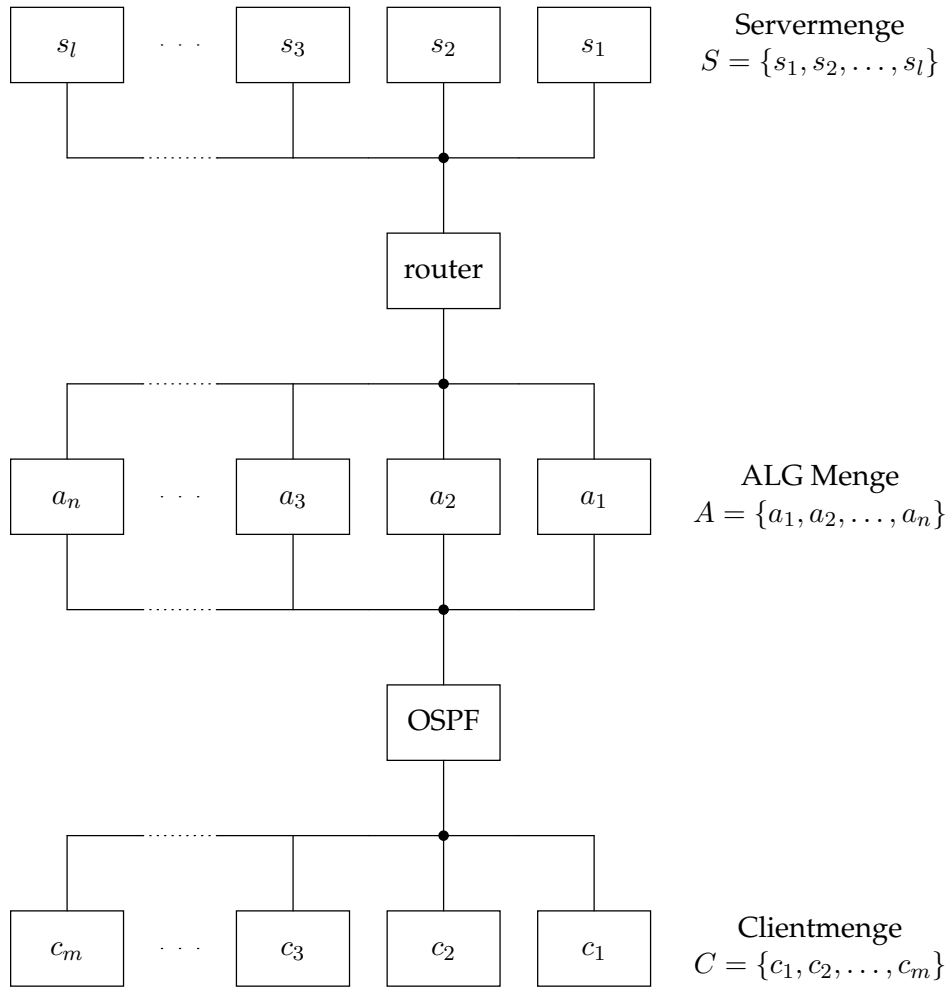


Abbildung 2.1: Darstellung der Nomenklatur für die Komponenten eines HA-Clusters samt innerem und äußerem Netzwerk

Symbole	Bedeutung
$S = \{s_1, \dots, s_l\}$	Menge entfernter Knoten (Server)
$A = \{a_1, \dots, a_n\}$	Menge der ALG HA-Knoten
$C = \{c_1, \dots, c_m\}$	Menge lokaler Knoten (Clients)
$V = C \cup A \cup S$	Gesamtknotenmenge
K_n	Vollverbundener Graph mit n Knoten
$E \subseteq \left(\binom{A}{2} \stackrel{!}{=} K_n \right)$	Menge der Verbindungen im HA-Cluster mit $ E = k$

Tabelle 2.2: Nomenklatur: In der Diplomarbeit durchgängig verwendete Symbole und Bezeichnungen

aufgreifen. Hier folgt dagegen eine abstrakte Darstellung der problemrelevanten Komponenten. Diese ermöglicht im Rahmen eines Software-Engineering Prozesses eine anschließende genaue Anforderungsanalyse.

Eine detaillierte Beschreibung zur Bedienung des GeNUGate kann im *GeNUGate Installations- und Konfigurationshandbuch*⁴ nachgeschlagen werden. Diese ist aber für die Diplomarbeit nicht relevant.

2.2.1 Sicherheitskonzept des GeNUGate

Die Systemidee des GeNUGate ist die konkrete Implementation des vom BSI empfohlenen⁵ und im IT-Grundschutzhandbuch [18] als Maßnahme M 2.77 erläuterten **PAP**-Modells (**P**aketfilter / **A**pplicationlevel-Gateway / **P**aketfilter) für Firewalls. Dem Modell zufolge wird das innere Netz durch eine Firewall-Konstellation bestehend aus den seriell geschalteten drei Komponenten **P**, **A** und **P** geschützt.

Es ist ausserdem möglich, eine DMZ⁶ auf Höhe des Paketfilters (und/oder des ALGs) zu schalten. Da es aus sicherheitstechnischen Gesichtspunkten nicht ratsam ist, die beiden Paketfilter baugleich zu realisieren (d. h. gleiches Betriebssystem, gleiche Software und gleiche Hardware), beschränkt sich GeNUGate auf die Auslieferung eines zweistufigen Systems. Die Empfehlung ist, als äußeres System einen Paketfilter eines Drittherstellers einzusetzen (Abbildung 2.2).

Zur PAP-Sicherheitsidee gehört außerdem, dass der Rechner, auf dem der Application Level Gateway läuft, *keinerlei* Routing auf OSI-Schicht 3 durchführt, sondern alle Pakete ausschließlich auf Anwendungsebene (Schicht 4) behandelt. Diese Aufgabe wird beim GeNUGate je nach durchgereichtem Dienst von einer Gruppe von Relay-Prozessen nach dem besprochenen Entwurfsmuster *Proxy* (Abschnitt 2.1.1) erledigt. Diese verwerfen oder erlauben Pakete auf Grund des Inhalts und dem Zutreffen von Berechtigungen des Absenders (realisiert über sog. *Access Control Lists*). Durchgereichte Pakete können nach unerwünschtem Inhalt gefiltert werden, bei HTTP etwa nach Java, JavaScript etc.

Bestimmte Protokolle lassen sich nicht auf dieser Ebene analysieren, da sie bereits verschlüsselt übertragen werden. Hierzu gehören beispielsweise SSH- oder SSL-Verbindungen. Da aber per Definition alle Datenverbindungen auf dem ALG auf Anwendungsschicht weitergereicht werden müssen, existieren für derartige Verbindungen elementare Relays (`tcprelay`, `udprelay`), die einfach nur Netzwerkpakete weiterreichen.

Das Sicherheitskonzept von GeNUGate sieht vor, dass das ALG *keinerlei* Netzwerkdienste direkt an irgendeiner Netzwerkschnittstelle nach au-

⁴ohne ISBN, erhältlich bei der GeNUGate mbH, Kirchheim bei München

⁵vgl. hierzu das Informations-Faltblatt zum Thema „Sicherheit im Internet“ des BSI unter <http://www.bsi.bund.de/literat/faltbl/F29Internet.htm>

⁶Demilitarisierte Zone

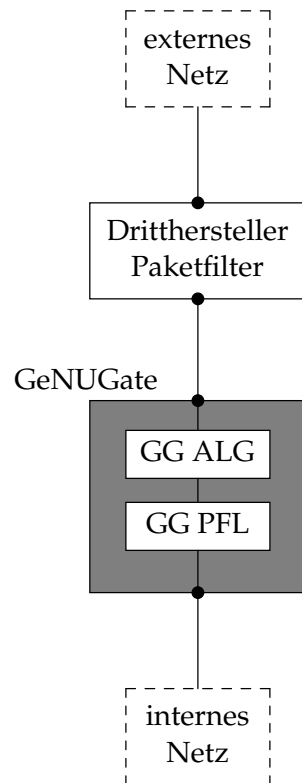


Abbildung 2.2: GeNUAs Realisierung des vom BSI empfohlenen PAP-Modells zur Konstruktion von Firewalls: innerer Paketfilter und Application Level Gateway von GeNUA in Form des GeNUGate, äußere Firewall von einem Dritthersteller

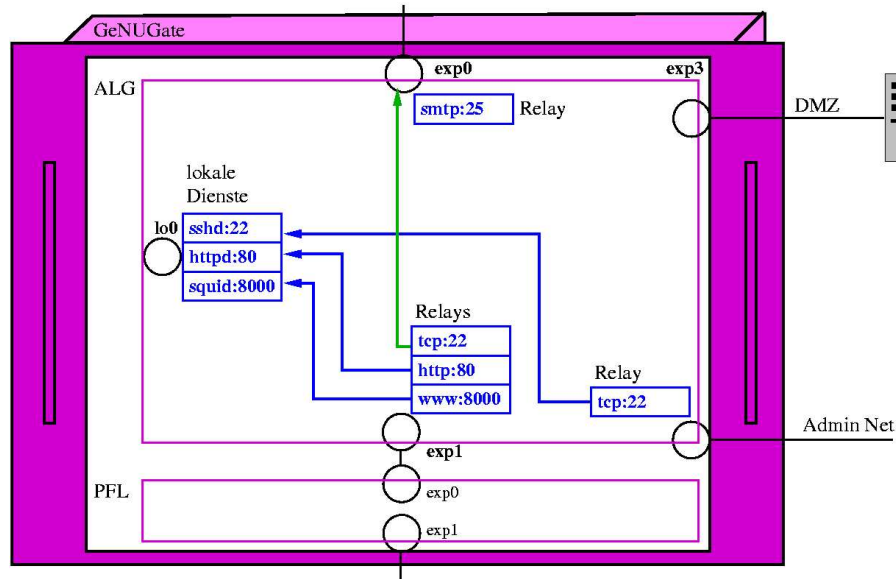


Abbildung 2.3: Sicherheitskonzept: Das ALG stellt Dienste bereit, die nur an der Loopback Schnittstelle gebunden sind, und Relays, die von den (jeweils durch Kreise gekennzeichneten) externen Schnittstellen dorthin vermitteln. Ein TCP-Relay sorgt beispielsweise für die Bereitstellung des nur auf localhost gebundenen sshd auf der Admin-Netzwerkschnittstelle. Ein WWW-Relay vermittelt Benutzer aus dem internen Netz an den nur lokal gebundenen squid WWW-Proxy weiter. *Quelle: GeNUA*

ßen anbietet. Alle auf dem ALG laufenden Dienste binden sich nur an die Loopback-Schnittstelle⁷. Jegliches Bereitstellen von Diensten nach außen muss über ein Relay erfolgen, welches die Daten dann durchreicht (Abbildung 2.3). Auf diesem Weg wird die Kontrolle über Berechtigungen und (soweit möglich) über den Inhalt der Verbindung sichergestellt.

Alle Relays laufen in sog. *Cages*. Hierbei handelt es sich um durch einen GeNUA-eigenen Kernel-Patch noch weitergehend eingeschränkte *chroot*-Umgebungen. Zusätzlich zum bekannten *chroot*-Verhalten werden noch einige Systemaufrufe der eingesperrten Prozesse eingeschränkt. Auch das Senden von Signalen an Prozesse außerhalb eines Cages ist den eingesperrten Prozessen nicht möglich.

⁷Unter IP-fähigen Betriebssystemen ist die Loopback-Schnittstelle nur vom lokalen Rechner aus erreichbar. IP-Adressen, die per Definition nach Loopback geleitet werden, sind z. B. 127.0.0.1 (IPv4, RFC 3330 [17]) und ::1 (IPv6, RFC 3513 [16])

2.2.2 Verteilte Relay-Objekte

Relays können im Cluster auf mehreren Maschinen parallel laufen. Übertragen auf ein abstraktes Modell reden wir also von einer Menge verteilter Objekte mit bestimmten Eigenschaften. Die Relay-Prozesse laden nach dem Starten selbsttätig eine Konfiguration, die ihr weiteres Verhalten bestimmt. Ein Relay-Prozess mit seinen Kindprozessen behandelt jeweils eine Protokollart, und innerhalb eines solchen Prozesses können viele einzelne Relay-Objekte existieren.

Die Konfiguration stellt einen lokalen Zustand dar, der in unserem abstrakten Modell durch Variablen ausgedrückt werden kann. Ein weiterer Teil des lokalen Zustands sind die momentan von einem Relay bedienten Netzverbindungen (vgl. Stateful Paketfilter laut Definition 2.5). Es wäre zwar prinzipiell möglich, diesen Zustand auszugeben und von einem Relay der selben Klasse wieder laden zu lassen (wie im Entwurfsmuster *Memento* in [14] erwähnt). Praktisch ist dies aber überhaupt nicht sinnvoll, da die Netzwerkverbindungen zu eng mit dem Relay-Prozess und vor allem mit dem Protokollstack des Betriebssystems interagieren, als dass sie von einem an das nächste Relay einfach so weitergegeben werden könnten.

Aus der abstrakten Sicht wird deutlich, dass die Relay-Objekte nach außen hin keine echten Methoden anbieten. Die POSIX-Signale, mit denen bisher eine primitive Steuerung des Prozessverhaltens möglich ist, liefern weder eine Garantie, dass der Aufruf angekommen ist, noch eine Rückmeldung über dessen Gelingen. Aus diesem Grund ist die Menge der Methoden eines Relays in unserem Modell leer.

2.2.3 Spezialisierung der Relays

Es existiert eine abstrakte Oberklasse des allgemeinen Relays (Datei `Relay.pm`). Davon werden, je nach Aufgabe, spezialisierte Relays abgeleitet, die konkret instantiiert jeweils einzelne Protokollarten übernehmen (Abbildung 2.4). In der vorliegenden Implementation geschieht die Ableitung durch Einbindung des Moduls `Relay.pm` in der entsprechenden Programmdatei (z. B. `tcprelay.pl`) und Verwendung der zur Verfügung gestellten Methoden in Komposition.

2.2.4 Vater-Kind Aufgabenteilung der Relays

Die Architektur des GeNUGate sieht pro Relay-Art eine Trennung der Aufgaben in Kontrolle und Ausführung vor: Der Vaterprozess ist für das Öffnen und Halten von Netzwerk-Sockets sowie für die Kontrolle und das Starten der Kinder zuständig. Die Kinder übernehmen dann die eigentliche Arbeit, das Durchreichen der Netzwerkpakete. Die Sockets stellen die Kommunikationsendpunkte für akzeptierte Netzwerkverbindungen dar

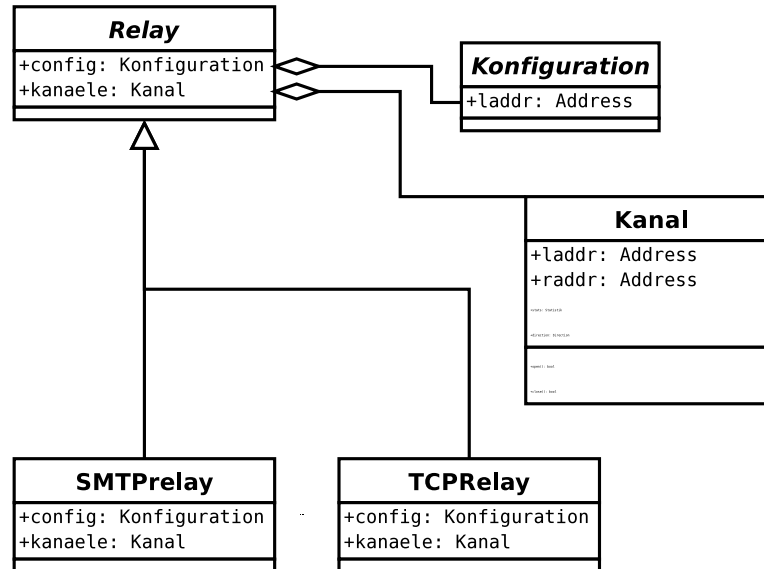


Abbildung 2.4: UML Diagramm des Relay-Mechanismus auf dem GeNU-Gate vor der Diplomarbeit. Auffällig ist das komplette Fehlen von Methoden im Relay-Objekt, weil bisher keine Middleware zur Ansteuerung solcher Methoden existiert.

und werden als Deskriptoren an die Kinder durchgereicht. Die Kinder bearbeiten die konkret eintreffenden Verbindungen. Pro Kind wird eine definierte Zahl von Verbindungen bearbeitet. Treffen mehr Verbindungen ein, so werden je nach Bedarf zusätzliche Kindinstanzen erzeugt. Um etwaige unangenehme Nebeneffekte (z. B. Speicherlecks, Kompromittierung) so klein wie möglich zu halten, wird jedes Kind nach einer bestimmten Anzahl bearbeiteter Verbindungen vorsichtshalber stillgelegt und durch eine neue Instanz ersetzt.

In der bisherigen Architektur ist eine Kommunikation mit dem Kind im Sinne eines entfernten Methodenaufrufs nicht möglich. Beispielsweise wird bei einer Änderung der Konfiguration für die betroffene Relayklasse das Kind mit der alten Konfiguration stillgelegt und eine neue Instanz erzeugt. Ist das Kind allerdings noch beim Bearbeiten von Verbindungen, trennt es sich durch einen zusätzlichen `fork()` vom Vaterprozess ab und terminiert erst, wenn die letzte Verbindung geschlossen wurde. Durch das zusätzliche Abspalten wird eine Kontrolle durch den Vaterprozess, selbst durch Signale, unmöglich gemacht. Mit Einführung der Kommunikationsschicht im Rahmen dieser Diplomarbeit wird dieses Problem gelöst.

2.2.5 Modus der Dienstbereitstellung

STEVENS unterscheidet in [38, Abschnitt 6.2] fünf Modelle, wie mit Unix-Primitiven Ein/Ausgabe in Netzwerkdiensten mit einem einzigen Aktivitätsträger (Prozess) bearbeitet werden kann:

- **blockierend:** Programm wartet im Systemaufruf, bis gelesen oder geschrieben werden konnte.
- **nichtblockierend:** Programm kann beim Systemaufruf entweder Daten lesen/schreiben, oder kehrt sofort mit `EWOULDBLOCK` zurück.
- **Multiplexing:** Systemaufrufe `select(2)`, `poll(2)` oder `kevent(2)` blockieren das Programm, bis ein Ereignis auf einem der zu beobachtenden Deskriptoren eintritt – ideal zur gleichzeitigen Beobachtung mehrerer Ereignisquellen.⁸
- **synchron signalgetrieben (lesend):** Das Programm installiert eine Signalbehandlung und wird bei eintreffenden Daten mittels `SIGIO` verständigt, wenn Daten zur Abholung bereit liegen. Ein anschließend vom Programm ausgeführter Aufruf einer Leseoperation auf den zugehörigen Deskriptor blockiert nur so lange, bis die Daten komplett empfangen sind, aber nie beliebig lange wie bei der blockierenden Semantik.
- **asynchron signalgetrieben (lesend):** Ähnlich wie synchrone signalgetriebene Ein/Ausgabe, nur mit dem Unterschied, dass die Anwendung benachrichtigt wird, wenn die Ein-/Ausgabe Operation *fertig* ist, d. h. ein Aufruf der Leseoperation kehrt sofort erfolgreich zurück.

Die im GeNUGate vorliegenden Relays arbeiten nach dem *Multiplexing-Modell*, d. h. jeder Prozess hält gleichzeitig eine große Zahl an offenen Deskriptoren und wartet per `select(2)` auf Ereignisse an diesen Deskriptoren. Die komplette Kommunikation wird also in einem einzigen Prozess abgewickelt, im Gegensatz zum *forking-* oder *threaded Server*, bei dem einzelne Aktivitätsträger pro Deskriptor eingesetzt werden, die sich an Lese- und Schreibauffrufen blockieren.

2.2.6 Überwachung und Neustart von Diensten

Auf jedem Application Level Gateway existieren mehrere Mechanismen, die periodisch die korrekte Funktion der Relay-Prozesse überwachen. Sollte ein Relay aus irgendeinem Grund seinen Dienst beenden, wird es beim

⁸zur Abstraktion der unter den verschiedenen Betriebssystemen verfügbaren Mechanismen hat NIELS PROVOS eine Bibliothek namens `libevent`[29] geschrieben, die die verschiedenen Schnittstellen unter einer generischen Schnittstelle vereinheitlicht. Wir werden bei der Implementation des Kommunikationsdienstes Gebrauch von `libevent` machen.

nächsten Prüfintervall neu gestartet. Für diese Diplomarbeit relevant ist alleine, dass die einzelnen Relays zeitweise nicht ansprechbar sein können.

2.2.7 Hochverfügbarkeitskonzept

Im GeNUGate-Cluster stehen den Paketen vom internen Netz mehrere Möglichkeiten für den Weg ins externe Netz zur Verfügung (jedes ALG als HA-Knoten stellt eine solche dar). Um auf eine eindeutige Route zu kommen und um Lastverteilung durchzuführen, wird dem GeNUGate-Cluster ein OSPF-Router vorgeschaltet. Dieser hat durch das Beobachten des Routing-Protokolls OSPF und durch die von den momentan funktionierenden ALGs propagierten Routen einen Überblick, über welche Knoten gerade Netzwerkverkehr nach draußen geleitet werden kann.

Alle mit der Hochverfügbarkeit verbundenen Aufgaben im GeNUGate werden von einem eigenen Dienst, dem *high availability daemon* *had*, übernommen, der pro ALG jeweils in einer Instanz läuft. Da es sich um ein über die Jahre gewachsenes System handelt, vereinigt dieser Dienst viele Aufgaben in sich, die prinzipiell voneinander unabhängig sind.

Grundsätzlich gibt es in einer Menge von Rechnern im Cluster zu jeder Zeit einen ausgezeichneten *Master*-Rechner, von dem alle anderen beispielsweise ihre Konfiguration kopieren. Auf allen Knoten des Verbunds läuft eine identische Menge an Relays und lokalen Diensten. Die Kommunikation läuft über ein extra HA-Netz ab (Abbildung 2.5). Per Definition ist der Master in der Menge immer derjenige Rechner mit der niedrigsten IP-Adresse (im Folgenden auch als *logisch links*) bezeichnet. Der *had* prüft über mehrere Wege die Erreichbarkeit seiner Nachbarn. Fällt ein Knoten aus, so übernimmt der nächste noch aktive Knoten logisch rechts vom ausgefallenen alle bedienten Netzwerkadressen des bzw. der ausgefallenen Knoten. Hierbei bleibt die IP-Adresse im HA-Netz unangetastet, lediglich die Adressen auf Innen- und Außenseite der Gateways werden bewegt.

Wir werden in der Diplomarbeit nicht weiter mit dem *had* in Berührung kommen, da er weder für unsere Zwecke geeignet ist, noch mit unserer Kommunikationsschicht in Konflikt steht. Wichtig für die Diplomarbeit ist jedoch, dass der HA-Einsatz auch ortsverteilt möglich ist, d. h. dass GeNUGates in verschiedenen Gebäuden platziert sind und trotzdem als logische HA-Einheit arbeiten. Hierdurch muss das HA-Zwischennetz über längere Strecken gelegt werden, um die Gates zu verbinden. Deshalb muss es in einem Angreiferszenario einer Sicherheitsauswertung als nicht vertrauenswürdig angesehen werden.

2.2.8 Zugriffskontrolle

Alle im GeNUGate konfigurierbaren Zugriffsberechtigungen zum Benutzen von Relays oder der web-basierten Konfigurationsoberfläche (Ab-

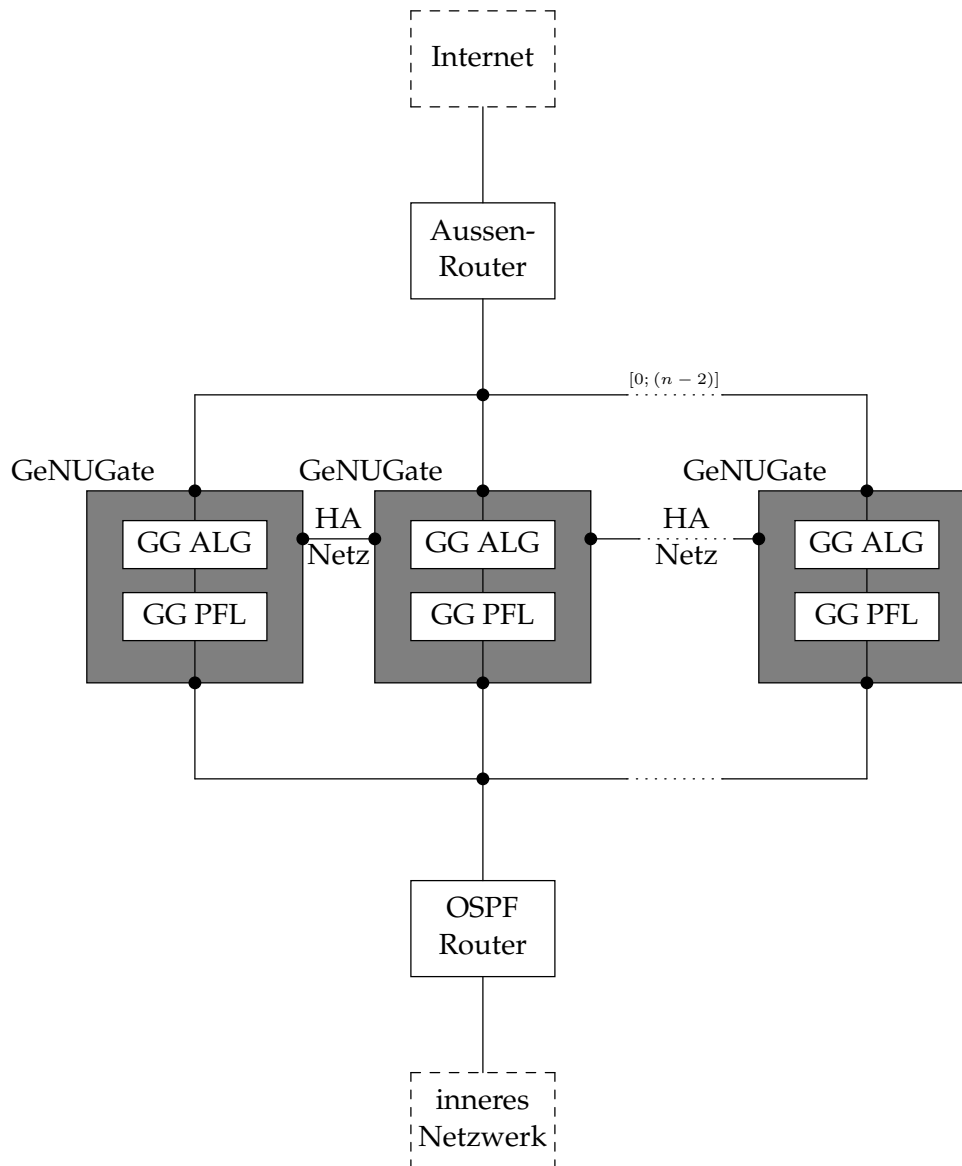


Abbildung 2.5: Momentan unterstütztes Hochverfügbarkeitsszenario: Ein OSPF-Router sorgt für Lastverteilung und leitet um defekte Knoten herum. Die Knoten selbst kommunizieren über ein HA-Zwischennetz. Für die Zukunft ist der vom OpenBSD-Projekt als Alternative zu CISCOs *VRRP* entwickelte Hochverfügbarkeitsmechanismus *CARP* (<http://www.openbsd.org/cgi-bin/man.cgi?query=carp>) technisch sehr interessant, weil direkt im Betriebssystem verfügbar.

schnitt 2.2.9) werden vom Authentisierungs-Dienst `authd` behandelt. Der Dienst ist nur für den maschinenlokalen Gebrauch vorgesehen und lauscht auf UNIX Domain und TCP Sockets. Es wird ein proprietäres Protokoll über die Sockets gesprochen. Der `authd` leistet damit im Prinzip Authentisierungsdienste, wie sie z. B. unter Linux mit dem PAM⁹ Subsystem zur Verfügung gestellt werden.

2.2.9 Konfigurationsmechanismus

Bisher nimmt der Benutzer Einstellungen am Verhalten des GeNUGate über ein Webinterface vor. Diese Oberfläche fungiert in der Sicht des MVC-Paradigmas als *View und Controller*. Das *Modell*, also die interne Darstellung der Daten, ist eine zentrale Konfigurationsdatei namens `fw.cfg`. Da die globale Konfigurationsdatei eine enorme Größe besitzt und jeweils nur Teile für die einzelnen Dienste relevant sind, wurde ein Zerteilungsmechanismus eingeführt, der einzelne, dienstspezifische Unterkonfigurationen erzeugt. Dieser exportiert aus der zentralen Konfiguration die jeweils für einen speziellen Dienst relevanten Teile in eine eigene Datei. Der Vorteil hierbei ist die höhere Geschwindigkeit beim Parsen der Konfiguration durch die konkreten Dienste. Außerdem entspricht dieses Vorgehen gängigen Paradigmen für guten Software-Entwurf (*separation of concerns*, Modularisierung).

Beim Schreiben einer neuen Konfiguration wird geprüft, welche Teile der Einstellungen sich überhaupt verändert haben (vgl. `FwConfig::Configure` im GeNUA Quellcode). Nur für diese wird dann auch ein Neustart des Dienstes bzw. ein Neueinlesen der Konfiguration veranlasst (`Utils::ReadConf`).

2.3 Anforderungsanalyse

In Abschnitt 2.3.1 stellen wir Anforderungen auf, die sich bei einem Problem der vorliegenden Art im allgemeinen Fall stellen; diese Anforderungen werden in Abschnitt 2.3.2 durch die speziellen Bedürfnisse der Firma GeNUA weiter eingeschränkt. Auch für diese Analyse lehnen wir das Vorgehen wieder an den bereits erwähnten OEP [26] an.

Die erwünschte Möglichkeit, mit den Relays clustertransparent kommunizieren und per Kommandozeile oder graphischer Oberfläche auf ihnen beliebige Aktionen ausführen zu können, setzt die Existenz einer Middleware voraus, die den entfernten Funktionsaufruf auf den Relay-Objekten erlaubt. Unsere Hauptaufgabe ist daher, eine für diese Anwendung geeignete Art von Kommunikationsschicht ausfindig zu machen und zu implementieren. Relay-Prozesse müssen in transparenter Weise angesprochen

⁹siehe hierzu <http://www.kernel.org/pub/linux/libs/pam/>

werden können, egal ob sie knotenlokal oder entfernt laufen. Die Schnittstelle muss also implizit die Möglichkeit zur Lokalisierung von Diensten innerhalb des Clusters zur Verfügung stellen.

Neben dem entfernten Methodenaufruf muss auch eine Möglichkeit bereitstehen, sog. *Callbacks* durchzuführen, d. h. eine Übermittlung asynchroner Ereignisse an Kontrollobjekte, die Interesse angemeldet haben. Dies kann nach dem Entwurfsmuster *Beobachter (Observer)* [14] durchgeführt werden, also durch ein bewußtes Anmelden der ereignisgenerierenden Objekte (hier: Relays) an den Relay-Controllern und eine Benachrichtigung im Ereignisfall. Das *Beobachter*-Entwurfsmuster arbeitet eigentlich in die andere Richtung und läßt die Beobachter bei den ereignisgenerierenden Objekten anmelden. Wie wir im weiteren Verlauf aber noch sehen werden, ist dies wegen der eher kurzlebigen Natur der Relays weniger sinnvoll als die umgekehrte Konstellation.

Im Zuge dieser Suche muss auch die Frage beantwortet werden, auf Basis welcher logischen Topologie die Kommunikationsschicht arbeiten soll. Insbesondere ist zwischen einem hierarchischen und einem Peer-to-Peer basierten Ansatz zu unterscheiden. Die Topologie muss den bereits existierenden Hochverfügbarkeitsmechanismus des GeNUGates mit einbeziehen.

Beim Entwurf der Kommunikationsschicht ist besonderer Wert auf die spätere einfache Erweiterbarkeit der Funktionalität im Bezug auf die auf den Relays verfügbaren Methoden zu legen. Eine weitere Aufgabe ist das Aufstellen einer Basismenge von problembezogenen Operationen, die auf den Relays sinnvollerweise ausgeführt werden können. Die Implementation derselben gehört nicht mehr zum wissenschaftlichen Kerngebiet der Diplomarbeit, sondern ist als praktische Programmieraufgabe zu begreifen.

2.3.1 Identifizierung der Interessenhalter

Wie bei jedem Projekt stehen auch bei der Diplomarbeit nur begrenzte Ressourcen für die Bearbeitung eines praktisch beliebig großen Aufwands zur Verfügung. Wir müssen herausfinden, welche Teilprojekte mit welchem Anteil der endlichen Arbeitszeit bedacht werden sollen. Daher ist eine Art der Ressourcenzuteilung (*Scheduling*) genauso wie bei Vorgängen auf einem Computer (Zuteilung von Rechenzeit, Festplattenzugriffen oder Plätzen in der Druckerwarteschlange) auch bei dieser Diplomarbeit erforderlich.

Um die Verteilung herauszufinden, weisen wir jedem zu bearbeitenden Teilgebiet einen korrespondierenden Aufwand und ein entsprechendes Risiko zu. Der Aufwand (gemessen in abstrakten Einheiten von 1 bis 6) gibt an, wieviel Arbeit das Teilgebiet bis zu seiner Komplettierung aufwirft. Das Risiko (auch gemessen von 1 bis 6) indiziert, welcher Schaden bei der Ver-

Interessenhalter	Aufwand A	Risiko R	Priorität
Betreuer (Uni)	3	6	$\sqrt{45}$
Entwickler	1	4	$\sqrt{41}$
GL	0	2	$\sqrt{40}$
Support	1	2	$\sqrt{29}$
Anwender	3	4	$\sqrt{25}$
Projekte	2	2	$\sqrt{20}$
Vertrieb	2	2	$\sqrt{20}$

Tabelle 2.3: Auflistung der Interessenhalter nach geschätzter Priorität. Diese ergibt sich aus $\sqrt{(6 - A)^2 + R^2}$, wobei $A, R \in \{1, 2, 3, 4, 5, 6\}$ und $A = 1$ geringen Aufwand und $R = 1$ geringes Risiko bedeuten.

nachlässigung des Teilgebietes auf mich als Diplomarbeiter zukommt. Die für die Einschätzung (in abgewandelter Form) verwendete Skala stammt von OESTEREICH [26].

Das Ergebnis dieser Überlegungen ist in Tabelle 2.3 zu begutachten.

Projekt-Ansprechpartner

Wir zählen die am Projekt in irgendeiner Form beteiligten Personen auf, und ordnen sie in die folgenden drei Gruppen ein.

Fachexperten ANTON RÖCKSEISEN (GeNUGate Entwicklungs-Koordinator), KAI DÖRNEMANN (GG-Entwicklungs-Koordinator), ALEXANDER BLUHM (GG-Entwickler), STEFFEN ULLRICH (GG-Entwickler), JÜRGEN KLEINÖDER (Betreuer, Uni Erlangen)

Systembetroffene MARCUS POPP (Produktmanager), CHRISTIAN JAMBOR (Schulungen), ULF RUDOLF (Support, Dokumentation)

Verantwortliche MAGNUS HARLANDER (techn. Geschäftsleitung), ANTON RÖCKSEISEN (GG-Koordinator), KAI DÖRNEMANN (GG-Koordinator), JÜRGEN KLEINÖDER (Betreuer, Uni Erlangen)

Interessen der beteiligten Gruppen

Die anfangs aufgezählten Interessensgruppen verfolgen jeweils verschiedene, möglicherweise sogar gegensätzliche Absichten. Um hiervon ein Bild zu bekommen, nun eine Auflistung der Ziele der jeweiligen Gruppen:

- Anwender** Gezielt Verbindungen terminieren können, ohne wissen zu müssen, auf welchem Relay diese aktuell bedient werden. Cluster zentral verwalten können. Einfachere Bedienbarkeit, schnellere Reaktion des Gates. Cluster "heiß" rekonfigurieren können
- Support** Einfache Bedienbarkeit, damit Schulung und Handbuch einfach gestaltet werden können. Einfach zu erlernende Debuggingmöglichkeit für den Einsatz beim Kunden
- Projekte** Flexible Einsatzmöglichkeiten für speziell angepasste Lösungen beim Kunden. Effizientere Rekonfiguration für Tests. Mächtigere Möglichkeit, Fehler zu suchen, auch im Laboreinsatz (*GeNUA Testbed*)
- Entwickler** Wartbarer, gut dokumentierter Code. Debuggingmöglichkeit für Laboreinsatz. Einfache Erweiterbarkeit. Technisch zukunftsweisendes Konzept
- Vertrieb** Geldwerte Vorteile für den Kunden als Verkaufsargument
- Geschäftsleitung** Bessere Marktstellung durch herausragende Funktionsmerkmale. Umsatzsteigerung
- Betreuer** Wissenschaftliche Herangehensweise. Abdeckung des gestellten Themas. Nachvollziehbarkeit, Anspruch, saubere Form

2.3.2 GeNUA-spezifische Anforderungen

1. Die Schnittstelle muss technisch in das GeNUGate Produkt integriert werden können. Sie muss auf der Zielplattform OpenBSD lauffähig und performant sein. Die zu erweiternden Relays sind in Perl implementiert.
2. GeNUGate strebt eine Sicherheitszertifizierung an. Es muss darauf geachtet werden, dass im Vergleich zur Ausgangssituation nicht zusätzliche, vermeidbare Dienste auf dem GeNUGate angeboten werden müssen, die ihrerseits Sicherheitsimplikationen nach sich ziehen. Von unsicheren Protokollen oder einer überladenen Implementation muss Abstand genommen werden.
3. Wichtig für die Zertifizierung ist die Überprüfbarkeit der Software. Dies setzt Quelloffenheit voraus. Wegen GeNUAs Unternehmensphilosophie muss jegliche eingesetzte Software dem Open Source Paradigma¹⁰ entsprechen.

¹⁰Für nähere Erläuterung der Begriffe *Freie Software* und *Open Source* empfiehlt sich die Lizenzseite von <http://www.gnu.org/licenses/> und die Homepage der Open Source Initiative, <http://www.opensource.org/>.

Mechanismus	OpenBSD	Reichw.	Namensraum
Pipe	•	<code>fork()</code> ^a	—
Named Pipe (FIFO)	•	lokal	Dateisystem-Pfad
System V msgq	•	lokal	<code>key_t</code>
POSIX msgq	—	lokal	POSIX IPC Name
System V shm	•	lokal	<code>key_t</code>
POSIX shm	—	lokal	POSIX IPC Name
Sun RPC	•	Netz	Programm/Version
Domain Sockets	•	lokal	Dateisystem-Pfad
TCP Sockets	•	Netz	IP:Port
UDP Sockets	•	Netz	IP:Port

^aNur zwischen Eltern-Kind Prozessen, weil namenlos

Tabelle 2.4: Kategorisierung der untersuchten Mechanismen zur Interprozesskommunikation

4. Die bisherige Benutzungssemantik erlaubt ein voneinander unabhängiges Starten der Relay-Prozesse. Deshalb muss die Schnittstelle auch Kommunikation zwischen neuen, vorher nicht gegenseitig bekannten Prozessen erlauben.
5. Die Schnittstelle muss fehlertolerant und nichtblockierend arbeiten. Ein ausgefallenes Relay darf nicht die Kommunikation mit den noch funktionierenden Prozessen behindern.

2.4 Verwandte Problemstellungen

In diesem Abschnitt behandeln wir der Problemstellung innewohnende allgemeine Prinzipien und beleuchten bereits vorhandene Lösungen gleicher oder sehr ähnlicher Probleme.

2.4.1 Interprozesskommunikation

Klassische UNIX-Mittel der IPC

W. RICHARD STEVENS [39] bespricht eine Vielzahl von IPC-Mechanismen. Diese sollen im Folgenden kurz aufgezählt und auf ihre Eignung für die Kommunikation der Relays untereinander untersucht werden (Tabelle 2.4).

TCP/UDP Sockets Es besteht die Möglichkeit, Relays auf bestimmten Ports lauschen zu lassen und somit einen klassischen TCP- bzw. UDP-Server zu implementieren [38].

Sun RPC Dieser Mechanismus hat auf praktisch allen UNIX-artigen Plattformen Implementation gefunden. RPC wird in Standardprotokollen wie NFS¹¹ verwendet. Ein RPC-bedingtes Problem, das vielen NFS-Anwendern auffällt, ist die Verklemmung der aufrufenden Maschine im Aufrufstumpf (*Stub*), wenn der NFS-Server gerade unerreichbar ist. Dieses Verhalten rührt vom RPC-inhärenten synchronen Methodenaufruf her und kann nicht durch ein nichtblockierendes Verhalten ersetzt werden. RPC erfordert den Start eines Portmappers für die Registration von RPC-Diensten. Dieser ist selbst wieder potenzielle Quelle von Fehlern und Sicherheitslücken. Deshalb ist der Einsatz von RPC zur Realisierung der Relay-Kommunikation nicht sinnvoll.

UNIX Pipes Eine Pipe ist prinzipiell unidirektional. Pipes existieren in namenloser und benannter Form. Manche Betriebssysteme bieten zwar bidirektionale Pipes an, diese sind aber nur wieder als Komposition zweier unidirektionaler Pipes realisiert. Für eine Kommunikation, die generell in beide Richtungen beabsichtigt ist, sind die Unix domain sockets den Pipes vorzuziehen. Laut STEVENS [37] sind auf den meisten BSDs die Pipes ohnehin durch *Streaming Sockets* implementiert.

UNIX domain sockets Diese Sockets sind auf den lokalen Rechner beschränkt, verhalten sich aber in Bezug auf die Programmierschnittstelle (*API*¹²) wie netzwerkgebundene TCP/UDP Sockets. Dies ist ein Vorteil, da netzwerkorientierter Quellcode mit minimalen Änderungen wiederverwendet werden kann, und die Domain Sockets bei lokaler Verwendung im Vergleich zu den Netzwerksockets weniger Ressourcen benötigen. Es existieren viele stabile und weitverbreitete Referenzimplementationen, die über Unix Domain Sockets kommunizieren, so etwa der X-Server oder die OpenBSD-Dienste *bgpd* und *bgpctl*, *bind* und *ndc*, *syslogd*. Wir werden in Abschnitt 2.4.2 einen kurzen Blick auf diese Implementationen werfen. Der Namensraum, unter dem die Sockets angesprochen werden, ist das UNIX-Dateisystem.

Message Queues Nachrichtenwarteschlangen sind Bestandteil des Betriebssystems. Sie existieren unabhängig von den Prozessen weiter und leeren sich auch nicht beim Beenden von Prozessen. Für unsere Zwecke unge-

¹¹Network File System: Version 1 (RFC 1094 [22]), Version 3 (RFC 1813 [7]) und Version 4 (RFC 3530 [35])

¹²Application Programming Interface

eignet, da der Vorteil einer derartige Persistenz gar nicht genutzt werden kann.

Shared Memory Zwischen Anwendungen geteilter Speicher ist generell die schnellste Art, auf einer Maschine zwischen Prozessen zu kommunizieren. Allerdings ist dann der Entwickler selbst zuständig, für eine angemessene Art von Synchronisation (Semaphoren etc.) zu sorgen. Shared Memory ist für unseren Anwendungsfall ungeeignet, da es explizit freigegeben werden muss und dadurch den Prozess überlebt, der es angelegt hat.

CORBA und Consorten

Bekannte Middleware wie CORBA¹³, Java RMI¹⁴ und Microsofts .net implementiert den entfernten Methodenaufruf. Die Vorteile dieser Kommunikationsschichten sind Unabhängigkeit von Hardwareplattform, Betriebssystem und verwendeter (objektorientierter) Programmiersprache sowie die Fähigkeit, Information in typisierter Form zu übertragen (im Gegensatz zu rohen Datenströmen auf Socket-Ebene). Das GeNUGate ist jedoch ein geschlossenes Anwendungsszenario, bei dem keinerlei Gebrauch von der genannten Flexibilität gemacht wird. Die Kommunikationsschicht ist Bestandteil der GeNUGate Sicherheitsarchitektur. Es ist nicht beabsichtigt, eine Anbindung an andere Komponenten außer an die Benutzeroberfläche durchzuführen.

Eine Verwendung derartiger Middleware würde Ressourcen beanspruchen, die für das GeNUGate nicht produktiv genutzt werden können. Es würden darüberhinaus alle Sicherheitsimplikationen der jeweiligen Middleware in das GeNUGate importiert. Für GeNUGate kämen ohnehin nur freie Implementierungen im Sinne der OpenBSD-Lizenz in Frage. Microsofts .net und Java RMI fallen damit schon einmal weg. Für CORBA existiert mindestens eine freie Implementation eines Object Request Brokers (ORB) namens ORBit¹⁵.

Darüberhinaus sind die genannten Middleware-Architekturen auf eine Verwendung mit objektorientierten Programmiersprachen ausgelegt. Gespräche bei GeNUGate haben ergeben, dass ein Umstieg von dem schwach typisierten Perl auf eine stark typisierte objektorientierte Sprache (wie sie in der *Interface Description Language* von CORBA beispielsweise gebraucht würde) in den nächsten Jahren nicht in Erwägung gezogen wird. Es existieren zwar im Bereich der Freien Software Module von Drittanbietern für

¹³Common Object Request Broker Architecture

¹⁴Remote Method Invocation

¹⁵Von einer zweiten Implementation namens *MICO CORBA* <http://www.mico.org/> konnte im Rahmen der Diplomarbeit nicht festgestellt werden, ob diese noch gepflegt wird, eine freie Lizenz aufweist und in aktuellen Projekten der Freien Software noch Verwendung findet.

Perl, die ein Benutzen von Middleware-Architekturen wie CORBA (z. B. durch ein Modul namens `CORBA::ORBit`) ermöglichen. Eine verlässliche Aussage über Stabilität und Sicherheit derartiger Erweiterungen kann allerdings nicht getroffen werden, außer durch eigene Untersuchungen.

Aus allen aufgezählten Punkten schließen wir, dass die Verwendung bekannter Middleware-Architekturen für diesen speziellen Anwendungsfall ungeeignet ist.

2.4.2 Beispiele für Frontend-Backend Kommunikation

In der UNIX-Welt gibt es viele Programme, die ebenfalls das Problem der Interprozesskommunikation zu lösen hatten. Es fällt auf, dass die wenigsten Programme Gebrauch von Sun RPC oder ähnlich schwergewichtiger Middleware machen, sondern auf eher konventionelle Mechanismen (Sockets, Pipes) zurückgreifen. Im Folgenden beleuchten wir einige repräsentative Beispiele.

Beispiel 2.5 (OpenBSD ntpd) *Der OpenNTPD [10] ist eine freie Implementation des Network Time Protokolls NTP¹⁶. Die Implementation führt Privilegien-separation [30] durch. Der unprivilegierte Teil des Dienstes kommuniziert über ein namenloses `socketpair(2)` mit dem privilegierten (Listing 2.1).*

Der Nachrichtenmechanismus `img` wurde für OpenBGPD entworfen und in OpenNTPD wiederverwendet. Mit dem relativ simplen Mechanismus werden innerhalb von in C Strukturen definiert, in denen typisierte Daten gespeichert werden. Die Strukturen werden dann serialisiert und durch die Sockets übertragen. Auf der anderen Seite stellen die ausgefüllten Strukturen Argumente zum entfernten asynchronen Methodenaufruf dar.

Der `IMSG` Mechanismus im OpenNTPD ist hochspezialisiert und eng verweben mit dem restlichen Quellcode. Deshalb existiert kein Stumpfgenerator – dieser ist aber wegen der nicht benötigten Flexibilität und den wenigen vordefinierten Methoden auch gar nicht notwendig.

```

1 int          pipe_chld [2];
  pid_t       chld_pid;
3
4 if (socketpair(AF_UNIX, SOCK_STREAM, PF_UNSPEC, \
5     pipe_chld) == -1)
6     fatal("socketpair");
7
8 /* fork child process */
9 chld_pid = ntp_main(pipe_chld, &conf);

```

¹⁶beschrieben in RFC 1305 [24]

```

11 setproctitle("[priv]");
   close(pipe_chld[1]);

```

Listing 2.1: Sinngemäßer Ausschnitt aus dem OpenNTPD Quellcode, in dem die Socketverbindung zwischen den beiden Prozessen erstellt wird

Beispiel 2.6 (OpenSSH privilege separation) *OpenSSH verwendet ebenfalls `socketpair(2)` zur Kommunikation. Es wird ein ssh-eigenes Nachrichtenformat verwendet, das auch zum entfernten Methodenaufruf zwischen zwei Prozessen auf dem selben Rechner dient.*

Laut Auskunft von OpenSSH-Entwickler MARKUS FRIEDL arbeitet der Methodenaufruf nur zwischen zwei lokalen Prozessen, und hat daher viel geringere Anforderungen als ein netzwerkfähiges Aufrufmodell. Die Aufrufe stehen auf der Client-Seite als entsprechende Wrapper-Funktionen bereit (siehe `monitor_wrap.c` im OpenSSH Quellcode), die ihrerseits das Marshalling für die spezielle Funktion übernehmen. Damit entfällt auch die generische Verwendung eines RPC-Protokolls und eines Stumpfgenerators.

Beispiel 2.7 (xpw) DUG SONGS Passwort-Dienst¹⁷ verwendet `rpcgen`, ohne tatsächlich RPC zum Methodenaufruf zu verwenden. Es wird lediglich die Aufgabe des Marshalling von Daten nicht selbst übernommen, sondern geschickt in die Hände einer standardisierten Anwendung gelegt. Die so in XDR¹⁸ eingepackten Daten werden per Wrapper-Funktion über eine Socket-Verbindung geleitet und bei der Gegenstelle wieder ausgepackt. `rpcgen` steht allerdings für Perl nicht zur Verfügung.

Serialisierung (On the wire Format)

Für die Netzwerkkommunikation mit anderen Netzwerkteilnehmern ist das Einhalten eines vordefinierten *on-the-wire Formats* entscheidend. C-Compiler versehen jedoch benutzerdefinierte Strukturen bei Bedarf mit einer Ausrichtung (*Alignment*) auf Maschinenwortgrenzen, um den Zugriff auf einzelne Elemente zu beschleunigen. Dieser Effekt kann ausgeschaltet werden, indem die einzelnen Felder einer Struktur mittels `memcpy(3)` sequentiell in einen Puffer kopiert werden, in dem sie dann garantiert dem Netzwerkformat entsprechen. Ein Beispiel für ein derartiges Vorgehen ist der OpenNTPD [10].

¹⁷Auf `xpw` hat mich MARKUS FRIEDL in einem Gespräch aufmerksam gemacht, vielen Dank. `xpw` ist nicht offiziell zum Download verfügbar, eine URL und Beschreibung kann aber einer Mail von DUG SONG entnommen werden, die er auf der Mailingliste `openbsd-tech` geschrieben hat: <http://marc.theaimsgroup.com/?l=openbsd-tech&m=104818343225696&w=2>

¹⁸eXternal Data Representation, RFC 1014 [21]

Kategorie	lokal	verteilt	Konsequenz
Interaktion	lokal	entfernt	Mehr Fehlerarten
Bindung	direkt	indirekt	Konfiguration wird zum dynamischen Vorgang
Ausführung	sequentiell	nebenläufig	Resequentialisierung nötig
Interaktion	synchron	asynchron	Pipelining
Umgebung	homogen	heterogen	Gemeinsame Datenrepräsentation nötig
Instanzen	einzel	repliziert	Maßnahmen zur Konsistenzwahrung erforderlich
Lokalität	fest	beweglich	Änderung der Lage von Schnittstellen zur Laufzeit
Namensraum	einheitlich	aggregiert	Anforderungen an Namensauflösung
Speicher	gemeinsam	getrennt	Shared memory Konzept obsolet

Tabelle 2.5: Gegenüberstellung von herkömmlichen Systemen und Verteilten Systemen mit Blick auf einzelne Aspekte und die daraus erwachsenden Konsequenzen
Quelle: SCHRÖDER-PREIKSCHAT [34]

2.4.3 Verteilte Systeme

Um die generellen Unterschiede und Anforderungen eines verteilten im Vergleich zu einem lokalen System darzustellen, zitiere ich in Tabelle 2.5 kurz einige wichtige Punkte aus der gleichnamigen Vorlesung [34] von Prof. WOLFGANG SCHRÖDER-PREIKSCHAT.

Fehlerbehandlung

Zu den Aufgaben eines Verteilten Systems gehört es, Fehler zu erkennen, zu maskieren (d. h. den Fehler abfangen und die fehlgeschlagene Operation wiederholen, ohne dass eine höhere Schicht Kenntnis davon erhält, oder von vornherein redundant arbeiten, oder Fehler selektiv ignorieren) und zu tolerieren. Hierbei wird zwischen transienten (behebaren) und permanenten Fehlern unterschieden.

Nachrichtensemantik

Es gibt unterschiedliche Stärken der Überprüfung von erfolgreicher Zustellung bei IPC Nachrichten. Diese Prinzipien sind schon sehr lange bekannt. Sie werden u. a. schon 1979 bei LISKOV [20] zitiert.

- **request:** Die Zustellung der Anfrage wird vom Protokoll zugesichert.
- **request-reply (HOARE):** Darüberhinaus versicherte Zustellung einer Empfangsbestätigung für die Anfrage.
- **request-reply/acknowledge-reply (HANSEN):** Versicherte Zustellung von Anfrage, Anfragebestätigung, Antwort und Antwortbestätigung.

2.4.4 Netzwerkmanagement

Praktisch alle der derzeit erhältlichen Router, Switches und ähnliche Netzwerkkomponenten von Industriequalität sprechen das Managementprotokoll SNMP.¹⁹

Zwar wurde einst CMIP²⁰ vom OSI als Ersatz für SNMP vorgeschlagen, fand aber trotz einiger technischer Verbesserungen nie weitreichenden Einsatz im Internetbereich, sondern nur im Bereich der Telekommunikation.

SNMP besteht aus drei Teilen:

1. dem Protokoll
2. der Darstellung bzw. Beschreibung von Werten mit ASN.1 bzw. BER
3. einer Management Information Base (MIB)

Die MIB ist eine hierarchisch angeordnete Struktur und kann verschiedene Datentypen speichern. Es existieren offizielle standardisierte MIBs, wie etwa RMON²¹, sowie die Möglichkeit, MIBs als Firma beim IETF-Konsortium einzureichen.

Vorteil SNMP und die dazugehörigen Techniken stellen Internet Standards dar, sie sind gut dokumentiert und weit verbreitet.

Nachteile SNMP benötigt einen ASN.1 Parser. Abgesehen von den zahlreichen Sicherheitsproblemen bekannter ASN.1-Implementationen²², müsste der Parser erst unter erheblichem Aufwand in das GeNUGate integriert werden. SNMP eignet sich zwar gut zum Setzen und Lesen von Werten und für vordefinierte Änderungen am Verhalten von Netzwerkkomponenten,

¹⁹SNMP existiert nominell in drei verschiedenen Hauptversionen, von denen aber nur SNMPv1 und SNMPv3 wirklich praktische Relevanz haben. Die Version 3 zeichnet sich vor allem durch die Einführung von Mechanismen zur Administration und zur Absicherung der Kommunikation aus. Beschrieben wird SNMP in einer großen Zahl von RFCs. Eine gute Übersicht über diese zahlreichen Dokumente findet sich in RFC 3584 [13].

²⁰Common Management Information Protocol, RFC 1189 [45]

²¹Remote Monitoring, RFC 2819 [44]

²²siehe hierzu sicherheitsrelevante Mailinglisten wie *Bugtraq*, *Full-Disclosure* oder <http://www.securityfocus.com/>

Application			
MPI	Sockets	Globus	Legion
VMI 2.0			
VIA	GM	TCP	
InfiniBand	Myrinet	Gigabit Ethernet	

Tabelle 2.6: Kurzübersicht über die Architektur der für Clustercomputing verwendeten Protokolle VMI und MPI. Die Applikation kann sich verschiedener Schnittstellen bedienen, die alle wieder auf VMI aufsetzen. VMI selbst ist wiederum unabhängig von der verwendeten Netzwerkhardware.

ist aber für unser Vorhaben zu unflexibel, weil es keinen echten entfernten Methodenaufruf bietet. Die Möglichkeit, größere Datenmengen (bulk transports) zu verschieben, besteht laut Protokoll auch nur in eine Richtung. Als Konsequenz kommt auch SNMP beim Entwurf unserer Kommunikationsschicht nicht in Frage.

2.4.5 Clustercomputing

Ein Hauptziel bei der effizienten Durchführung von Hochleistungsberechnungen mit Clustern ist die Abstraktion von der einzelnen Hardware und eventuellen Teilausfällen und die Schaffung einer virtuellen Maschine, die implizit durch Prozessmigration Lastverteilung schaffen kann. Wir betrachten bereits existierende Lösungsansätze für dieses Problem, da es zu dem unseren verwandt, aber nicht identisch ist.

VMI [6] und MPI (Tabelle 2.6) sind als Middleware zur Abstraktion und zum impliziten Verschicken von Nachrichten in Hochleistungsclustern mit großer Knotenzahl entworfen worden. Großer Wert wird auf Abstraktion von der jeweiligen Netzwerkarchitektur und -hardware (InfiniBand, Myrinet, Gigabit Ethernet) gelegt. Diese ist jedoch bei GeNUA immer auf die Verwendung von Ethernet beschränkt.

Wir können bereits jetzt feststellen, dass der Funktionsumfang von VMI und MPI die Anforderungen von GeNUA weit übersteigt, und für einen anderen Anwendungsfall entworfen wurde. Desweiteren übersteigt auch der Umfang von VMI/MPI die Grenzen dessen, was im Rahmen der Produktzertifizierung mit vertretbarem Aufwand verifiziert werden kann.

VMI

Von der Homepage von VMI:

„VMI 2.1 is a middleware communication layer that addresses the issues of availability, usability, and management in the context of large-scale SANs interconnected over wide-area grids. The rationale behind having the 2.1 version is the inclusion of novel features like the ability to stripe data across heterogeneous networks, the ability to fail over from one network onto a heterogeneous network, and the ability to add data filters and other features dynamically, remotely, and even on per-connection basis.“

VMI wird u. a. für das Grid Computing beim NCSA²³ eingesetzt. Dortiger Anwendungsfall sind Cluster mit einer Knotenzahl $n \approx 1000$, im Gegensatz zu GeNUA mit $n \leq 8$. Schwerpunkt des Einsatzes von VMI liegt auf dem Gewinn von Performanz durch Umgehung von Sockets und TCP Stack. Für GeNUAs Hochsicherheits-Anwendungsfall ist VMI eher ungeeignet.

MPI

Laut TRÖGER ist die erhältliche Implementation namens MPICH-G2 [3] überladen, unvollständig implementiert, und besitzt eine zweifelhafte Lizenz. Er zieht es daher vor, für seine Arbeit eine eigene Middleware zu entwerfen, anstatt von MPICH Gebrauch zu machen.

Da MPI/VMI für GeNUA ohnehin nicht in Frage kommen, verzichten wir auf eine Verifikation von TRÖGERS Aussagen. Für den Anwendungsfall bei GeNUA steht fest, dass auch hier zu viel Funktionalität geboten wird, z. B. Prozessmigration.

Beispiele

Während meiner Recherche bin ich auf ein sehr engagiertes Projekt gestoßen, das ich im Folgenden als Beispiel für die Implementation einer VMI/MPI-Middleware anführen möchte:

Beispiel 2.8 (Cactus) *Während der Recherche zu VMI/MPI stieß ich auf eine Middleware namens Cactus [1], die ebenfalls zur Verwaltung von HP-Clustern dienen kann. Cactus liefert ebenfalls eine Hardwareabstraktion für die Netzwerkschicht, und bietet zur Anwendung hin nicht nur VMI/MPI an, sondern bietet auch die Möglichkeit, eigene Erweiterungen als Plugins (Thorns) zu erstellen. Das Cactus Projekt bezeichnet seine Software selbst als eine „modulare multiplattform OpenSource Middleware für High Performance Computing“.*

2.4.6 Sicherheitskonzepte in der Softwaretechnik

Viele der heute in Freien Betriebssystemen verwendeten Sicherheitskonzepte stammen aus dem Dunstkreis des OpenBSD-Projekts [9]. Projektleiter

²³National Center for Supercomputing Applications [4]

THEO DE RAADT gibt auf vielen Konferenzen Vorträge zum Thema *Exploit Mitigation Techniques* [31]. Im Folgenden stellen wir einige richtungsweisende Konzepte vor, die auch in der Diplomarbeit Berücksichtigung finden.

Privilegienseparation

Dieses einfache, aber sehr effektive Konzept von PROVOS, FRIEDL und HONEYMAN [30] fand seinen ersten großen Einsatz in der weit verbreiteten SecureShell-Implementation OpenSSH [5]. Die prinzipielle Idee ist die Aufspaltung von Netzwerk-Serverprogrammen (*daemons*), die meist wegen spezieller privilegierter Operationen (Öffnen von TCP-Ports unter 1024, `chroot(2)`, `setuid(2)`) als Benutzer `root` gestartet werden müssen. Weil der Anteil der privilegierten Operationen am Gesamtgeschehen in einem Serverprozess eher gering ist, und der restliche Programmcode auch ohne Privilegien ausgeführt werden kann, verfolgt das Konzept der Privilegienseparation die Idee der Trennung in zwei Prozesse: Ein privilegierter Teil (dessen Codegröße im Vergleich sehr gering ist) arbeitet als *Monitor* für den großen nichtprivilegierten Teil. Die beiden Teile kommunizieren über IPC-Mechanismen (im Falle von OpenSSH mit `socketpair(2)`) miteinander über eine klar definierte Schnittstelle.

Unter der gängigen Annahme, dass sich die Wahrscheinlichkeit einer ausbeutbaren Schwachstelle im Quellcode proportional zur Codegröße (gemessen z. B. in LOC) verhält, ist der privilegierte Teil viel schwerer auszubeten als der unprivilegierte. Sollte es – bei aller gebotenen Vorsicht – trotzdem einem Angreifer gelingen, den unprivilegierten Teil (der meist direkt für Anfragen von Benutzern aus dem Netzwerk zur Verfügung steht) zu kompromittieren, so hat dieser Angreifer nur Benutzer- und nicht (wie früher) Administratorrechte auf der Maschine erlangt.

Inzwischen ist Privilegienseparation in der Welt der freien unixartigen Betriebssysteme ein weit verbreitetes und akzeptiertes Konzept [31]. Außer der OpenSSH [5] haben auch andere Programme diesen Weg verfolgt, wie u. a. OpenNTPD [10], OpenBGPD, der X Server, `xdm` oder `dhclient`.

Schutz des Stapelspeichers

Bekannte Exploits für Programme basieren oft auf stapelspeicherbasierten Pufferüberläufen (*stack-based buffer overflows*). Hierbei wird das nicht total korrekte Verhalten (Definition 2.9) des angegriffenen Programms ausgenutzt.

Definition 2.8 (partielle Korrektheit) *Ein Programmcode wird bezüglich einer Vorbedingung P und der Nachbedingung Q partiell korrekt genannt, wenn bei einer Eingabe, die die Vorbedingung P erfüllt, jedes Ergebnis die Nachbedingung Q erfüllt. Dabei ist es noch möglich, dass das Programm nicht für jede Eingabe ein Ergebnis liefert, also nicht terminiert.*

Quelle: Wikipedia [46]

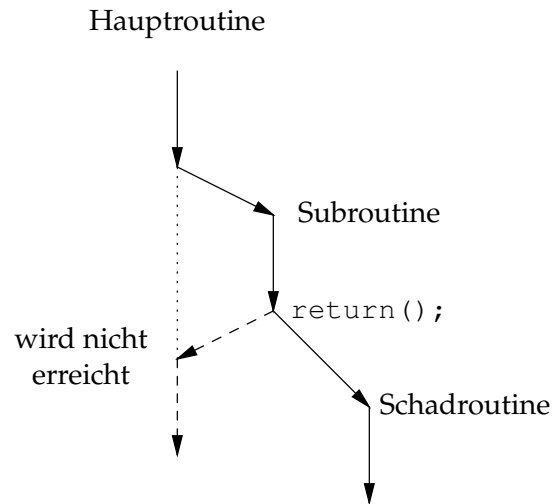


Abbildung 2.6: Nicht-Termination einer durch byzantinisches Fehlverhalten ausgebeuteten Subroutine: Der Handlungsfaden links verzweigt in die Unterfunktion, wo böswillig eingeschleuster Code zu einer Änderung des Kontrollflusses führt und in einen dritten (vom Angreifer beabsichtigten) Handlungsstrang wechselt. Aus Sicht des ersten Handlungsfadens terminiert die Unterfunktion nicht, sie verhält sich also nicht *total korrekt*.

Definition 2.9 (totale Korrektheit) *Ein Code wird total korrekt genannt, wenn er partiell korrekt ist und zusätzlich für jede Eingabe, die die Vorbedingung P erfüllt, terminiert. Aus der Definition folgt sofort, dass total korrekte Programme auch immer partiell korrekt sind.*
Quelle: Wikipedia [46]

Ein häufig benutzter Weg zum Ausbeuten von verwundbaren Programmen ist es, Strings von zu großer Länge zu übergeben, die von der Eingaberoutine des Programms unbeabsichtigterweise nicht korrekt überprüft werden. Das daraus resultierende Überschreiben des Stapelspeichers im Programm kann von Angreifern zum Einschleusen und Ausführen von Code genutzt werden. Dies entspricht der o. g. nicht-Termination (siehe Definition 2.8) der Routine (Abbildung 2.6).

Eine Möglichkeit, die Zahl dieser Angriffsmöglichkeiten drastisch zu reduzieren, ist das vorherige Speichern eines zufälligen Wertes (*Canary*) in der Nähe der Rücksprungadresse auf dem Stack der aufgerufenen Routine. Ein Rücksprung wird nur dann durchgeführt, wenn der Wert unverändert geblieben ist, weil dann auch mit hoher Wahrscheinlichkeit die Rücksprungadresse nicht manipuliert wurde.

Der besprochene Mechanismus ist seit OpenBSD 3.3 Bestandteil des Betriebssystems, wir profitieren also auch bei der Anfertigung dieser Diplomarbeit von den Vorteilen dieser Technik.

Ausführbar oder lesbar

Ein weiterer Schutz gegen Einbrüche durch stapelspeicherbasierte Pufferüberläufe ist es, die für die besprochene Manipulation notwendige Fähigkeit des gleichzeitigen Schreibens und Ausführens von Stack-Speicherseiten zu unterbinden.

Dieses Merkmal nennt sich *WX* (*write XOR execute*) und ist ebenfalls seit Version 3.3 Bestandteil von OpenBSD.

Kapitel 3

Middleware

Dieses Kapitel widmet sich dem konkreten Entwurf der Schnittstelle, zugeschnitten auf den Einsatz im GeNUGate. Im Bezug auf Abbildung 3.1 (S. 40) behandelt dieses Kapitel die Realisierung des Blockes *Kommunikationsschicht*. Es gliedert sich in die drei Stufen, die zur Realisierung der Schnittstelle in dieser Arbeit verwendet werden: Zunächst wird ein abstraktes Konzept (Abschnitt 3.1) aufgestellt, nach dem die jeweiligen Teile realisiert werden sollen. Anschließend wird ein Entwurf (Abschnitt 3.2) erstellt, der das abstrakte Konzept in konkrete Mechanismen übersetzt, allerdings die Implementationsdetails (Abschnitt 3.3) außen vor läßt.

3.1 Konzept

Relays

Die Relays sind hierarchisch einem knotenlokalen Relaydienst untergeordnet, der für diese eine Kommunikationsschnittstelle darstellt. Es wird das Entwurfsmuster *Beobachter* verwendet, d. h. der Knotencontroller weiß über die aktuell auf dem Knoten laufenden Relays bescheid. Laut *Beobachter*-Entwurfsmuster sollte sich der Knoten bei allen Entitäten, von denen er Informationen erhalten will (hier also bei den Relays) registrieren, um dann bei Ereignissen benachrichtigt zu werden. Weil aber die Relays im Vergleich zum Knotencontroller auf dem ALG die eindeutig kurzlebigeren Prozesse darstellen, kehren wir hierfür den Modus um und legen fest, dass sich jedes Relay nach dem Start beim Knotencontroller registrieren muss.

Wir fordern außerdem, dass das Initiieren der Kommunikation von jeder Seite aus möglich ist, so dass sowohl Informationen abgefragt werden können (*Polling*), als auch eine Benachrichtigung (*Trap*) bei eingetretenen Ereignissen geschickt werden kann.

Das hierarchische Kommunikationsmodell auf Knotenebene bietet keinerlei Redundanz. Diese fehlende Absicherung gegen Ausfälle ist aber kein

Verlust: Wir gehen in unserem *Fail-Stop* Modell von einer Knotengranularität aus, d. h. es wird davon ausgegangen, dass ein Knoten, bei dem ein Fehler bemerkt wird, seine Arbeit komplett einstellt. Der Ausfall einzelner Relays wird durch bereits vorhandene Mechanismen auf dem ALG behandelt. Ein Ausfall des Kommunikationsdienstes selbst kann durch Neustart desselben abgefangen werden, da wir fordern, dass dieser bis auf die gehaltenen Verbindungen zustandslos arbeitet. In einem solchen Fall müssen aber auch die Relay-Prozesse versuchen, sich nach Wegbrechen der Verbindung zum Kommunikationsdienst in unregelmäßigen Zeitabständen¹ neu zu verbinden.

Knotencontroller

Der Knotencontroller übernimmt die Kommunikation mit den lokalen Relay-Prozessen. Er lauscht aber auch auf dem Netzwerk, um die Kommunikation mit anderen Relays abzuwickeln. Normalerweise schreibt GeNUAs Sicherheitskonzept vor, dass jedem Dienst ein Relay zur Absicherung vorgeschaltet wird. In diesem speziellen Fall ist erwähntes Vorgehen allerdings noch zu diskutieren, da es sich um einen Dienst handelt, der eine Schicht niedriger arbeitet als die Relays.

Der Knotencontroller übernimmt hier die Aufgabe eines Guardian [20], bzw. er agiert als *reactor pattern* [32]. Das bedeutet, dass er nebenläufig eintreffende Anfragen von Clients serialisiert und bearbeitet (*dispatching*), indem er vorher registrierte Ereignisbehandlungsroutinen aufruft. SCHMIDT nennt folgende Vorteile dieses Entwurfsmusters:

Trennung von Zuständigkeiten und erhöhte Modularität Anwendungs-unabhängige Serialisierung und Bearbeitungsmechanismen (*Dispatching*) werden von der anwendungsspezifischen Funktionalität der Ereignisbehandlungsmethoden abgetrennt. Hierdurch werden beide Komponenten einfacher wiederverwendbar.

Erhöhte Portabilität Durch die Trennung von Zuständigkeiten werden auch betriebssystem- oder architekturabhängige Teile des Programms von generischen Mechanismen getrennt. Beim Portieren auf eine neue Plattform muss nur der abhängige Teil ersetzt bzw. angepasst werden, der Rest kann ohne Aufwand übernommen werden.

¹Ohne Pausen zwischen den Neuverbindungsversuchen würde das System nur unnötig belastet werden. Es empfiehlt sich, die Verwendung von zufälligen Intervallen (vergleichbar zum den bei CSMA/CD verwendeten) oder eines sog. *exponential backoff*, also das starke Anwachsen der Zeitabstände.

Kontrolle von Nebenläufigkeit Die Serialisierung als Kontrolle der Nebenläufigkeit ist ein sehr einfacher und stabiler Mechanismus, der komplexere Synchronisationsverfahren und Methoden für gegenseitigen Ausschluss unnötig macht.

Allerdings geht er auch auf mögliche Nachteile ein, die mit dieser Methode einhergehen:

Benötigt Deskriptoren Das *Reactor*-Entwurfsmuster kann nur auf denjenigen Betriebssystemen eingesetzt werden, die das Konzept der Deskriptoren mit Ereigniskopplung unterstützen. Nur auf diese Weise kann eine Liste mit denjenigen Deskriptoren erstellt und ans System übergeben werden, für die bei eintreffenden Ereignissen eine Reaktion ausgelöst werden soll.

Glücklicherweise ist dieses Konzept in allen unix-artigen Betriebssystemen mit den Mechanismen `select(2)`, `poll(2)` oder `kqueue(2)` (unvollständige Aufzählung) vorhanden, also im Speziellen auch im bei GeNUA verwendeten Basissystem OpenBSD.

Nicht präemptiv Wie schon angesprochen, findet eine *Serialisierung* der eintreffenden Ereignisse statt. Dies bedeutet unter anderem, dass von den aufgelaufenen Ereignissen jeweils ein einzelnes bearbeitet wird, bis dieses erledigt ist. Im Gegensatz zu Ereignisbehandlungen mit mehreren Aktivitätsträgern und komplexeren Synchronisationsmechanismen kann die Behandlung eines Ereignisses nicht einfach unterbrochen werden, selbst wenn inzwischen ein Ereignis aufgelaufen ist, das von höherer Priorität ist als das momentan behandelte.

Erschwerte Fehlersuche Die Struktur ereignisgesteuerter Programme weist einen invertierten Kontrollfluss auf, d. h. es ist kein durchgängiger Fluss vorhanden, sondern er wechselt zum einen ständig vom Ereignisframework (*Dispatcher*) zu den registrierten Callback-Routinen und zurück, und zum anderen ähnelt der Aufbau des Dispatchers mehr einem endlichen Automaten (wie auch die Kompilate von `lex/yacc` Parsern) als einem linearen Kontrollfluss.

Wir müssen deshalb also mit einer aufwändigeren Fehlersuche beim `commd` rechnen.

Um den Knotencontroller gegen Angriffe auf der Basis von Privilegieneskalaation abzusichern, benutzen wir das von PROVOS und FRIEDL vorgestellte Konzept der *Privilegienseparation* [30].

Kommunikationsschicht

Abbildung 3.1 zeigt eine Darstellung des beabsichtigten Kommunikationsmodells. Wir möchten mit den entfernten Methodenaufrufen eine *call-by-value/-result* Semantik bereitstellen, d. h. die Nachrichtenschicht muss lediglich Werte transportieren, aber keine komplexeren Konstrukte wie Referenzen, Objekte o. ä. Außerdem soll unsere Kommunikation eine *exactly-once*-Semantik bereitstellen. Jeder versendete Funktionsaufruf sollte also zu genau einem Aufruf auf der Gegenstelle führen. Wegen der Ähnlichkeit des von uns angestrebten Modells (IMSG über streaming Sockets) zu *call streams* [19] werden wir uns diesem Verhalten sehr gut annähern können.

Die teilnehmenden Knoten unterliegen einer *fail-stop* Semantik. Weil auf diese Weise aber bereits in Bearbeitung befindliche Transaktionen teilweise ausgeführt sein können, wenn ein Knoten wegbricht, können unsere Funktionsaufrufe nicht als atomare Operationen gelten.

Der Transport und die Bearbeitung von Kommunikationsnachrichten soll so weit wie möglich zustandslos ablaufen. Lediglich auf der Seite der Benutzerschnittstelle möchte man eine Zuordnung von Anfrage und asynchron erfolgendem Ergebnis. Für diese Aufgabe bedienen wir uns dem Mechanismus der *Promises* nach LISKOV [19]. Hierbei wird in der Benutzerschnittstelle pro gestellter asynchroner Anfrage ein Objekt („Versprechen“) erschaffen, auf das später zugegriffen werden kann. Entweder das Ergebnis der Anfrage ist inzwischen eingetroffen und findet sich in diesem Objekt wieder. Oder die Anfrage ist unbeantwortet geblieben, oder wurde sie aufgrund eines Fehlers nicht zugestellt bzw. bearbeitet. Dann ist der Fehlerstatus ebenfalls aus dem Objekt ablesbar.

Um nicht beliebig lange auf nicht mehr eintreffende, weil verlorene Antworten zu warten, versehen wir die *Promises* noch mit einem Zeitintervall, innerhalb dessen wir sinnvollerweise eine Antwort erwarten. Andernfalls gehen wir von einem Versagen der Gegenstelle aus.

Eine explizite Benutzung von Zuständen zur Realisierung der Konzepte *duplicate suppression* und *request resend* [34] ist nicht nötig, da wir uns für das Konzept der *call streams* (besprochen in [19]) entscheiden. Diese bringen die beiden o. g. Eigenschaften implizit mit sich.

Weiterleitung von Anfragen

Trifft eine Nachricht bei einem Knotenkontroller ein, die nicht für den lokalen Knoten bestimmt ist, so soll die Nachricht zum Zielknoten weitergeleitet werden. Ist der Zielknoten gerade nicht verfügbar, soll dies dem Aufrufer mit einer entsprechenden Fehlernachricht signalisiert werden.

Routing auf Anwendungsebene ist aus verschiedensten Peer-to-Peer Anwendungen bekannt [15]. Hierdurch wird von konkreten IP-Adressen des Clusters und Verfügbarkeit der HA-Knoten abstrahiert.

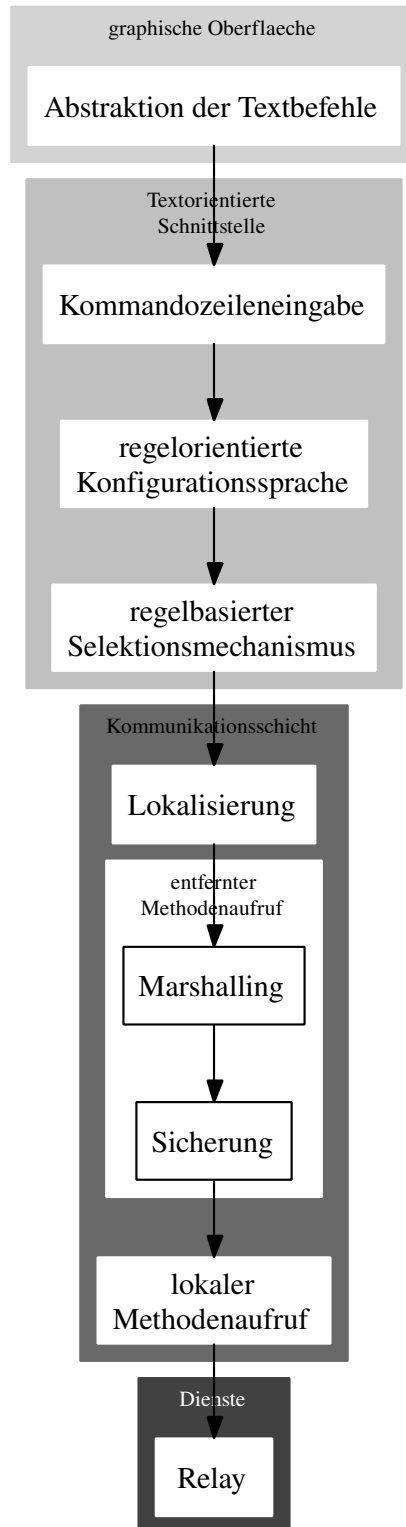


Abbildung 3.1: Schichtmodell für die zu implementierende Kommunikationsschicht

Kommunikation zwischen ALGs

Zwischen den Knoten im Cluster ist eine hierarchische Kommunikation mit einem ausgezeichneten Master-Knoten nicht sinnvoll, da wir von einer für jeden Knoten gleichen Ausfallwahrscheinlichkeit

$$p(a_1) = p(a_2) = \dots = p(a_n); \quad (a_1, \dots, a_n \in A)$$

ausgehen. Der Administrator soll sich auf einen beliebigen Knoten des Clusters verbinden können, und von jedem Knoten aus die identische Sicht auf das Netzwerk erhalten. Hierfür werden wir ein P2P-ähnliches Protokoll einsetzen.

3.2 Entwurf

Relays

In die vorhandene Relay-Implementierung wird die Behandlung eines Kommunikationsendpunktes mit Möglichkeit zum asynchronen entfernten Methodenaufruf eingebettet. Die korrekte Ablieferung der Nachrichten bei der Gegenstelle soll nachprüfbar sein. Dies entspricht der Erweiterung der Relay-Objekte um Methoden im UML Diagramm. Die Programmiersprache Perl ist durch den bereits existierenden Relay-Code festgelegt.

Der Einsatz herkömmlicher Middleware ist nach den Ergebnissen aus Abschnitt 2.4.1 für den speziellen Anwendungsfall im GeNUGate zu aufwendig, zu unflexibel und zu unsicher. Wegen der unterschiedlichen Anforderungen an die Kommunikationsschicht (statisch und effizient im lokalen Fall, redundant und fehlertolerant im globalen Fall) und den Sicherheitsanforderungen der Firma GeNUA überwiegen die Argumente für die Realisierung einer eigenen Kommunikationsschicht.

Die Abwesenheit von sprachintegrierten asynchronen Fernaufrufen in Perl zwingt uns zur Heranziehung einer eigenen Implementation. Von Dritten erstellte (z. B. von CPAN), eventuell verwendbare Module (z. B. Perl RPC) bieten nur blockierende Fernaufrufe, die sich in der bei GeNUA vorliegenden Situation mit nur einem Aktivitätsträger nicht einsetzen lassen. Da wir unsere Kommunikationsschicht selbst implementieren müssen, stehen uns auch keine Stumpfgeneratoren (*stub generators*) zur Verfügung, so wie sie von bekannten Middleware-Architekturen wie z. B. RPC (*rpcgen*), CORBA (IDL) etc. bereitgehalten werden.

Die Wahl des Kommunikationsmechanismus zum Knotencontroller wird als UNIX IPC² Primitiv realisiert, weil diese Mechanismen effizient einsetzbar und wegen ihrer starken Verbreitung exzellent getestet sind.

Die Abbildung unserer Kommunikationsschicht auf das ISO/OSI-Modell ist in Tabelle 3.1 dargestellt.

²*interprocess communication* [38]

Knotencontroller

Hier ist die Wahl der Programmiersprache prinzipiell frei, da der Knotencontroller nur über klar definierte Schnittstellen (Nachrichtenmechanismus) mit seinen Gegenstellen (Relays, Admin-Oberfläche) kommuniziert. Im Umkehrschluß (der sich bei der Implementation bewahrheitet hat) führt eine Verwendung verschiedener Programmiersprachen zu härteren Randbedingungen, die bei der Realisierung der Nachrichtenschicht beachtet werden müssen. Als Endergebnis erhalten wir also eine genauer geprüfte Schnittstelle, weil diese beim Testen mehr Unwägbarkeiten ausgesetzt war. Bei Verwendung gleichen Codes auf beiden Kommunikationsendpunkten können sich mögliche Fehler beim Ein- und Auspacken der Pakete (*Mars-halling*) gegenseitig aufheben.

Die Aufgabe des Knotencontrollers ist das Bereitstellen von Kommunikationswegen. Diese werden wie erwähnt mittels Unix IPC Primitiven realisiert. Durch die starke Separation vom Relay-Code bleibt der Modus für die Abwicklung (Abschnitt 2.2.5) der multiplen Kommunikationskanäle (lokale und Netzwerkverbindungen) auf Controllerseite frei wählbar. Die Entscheidung fällt auf I/O-Multiplexing, da dann die Kommunikation in einem einzigen Prozess abgewickelt werden kann, was einen geringeren Aufwand darstellt, als pro Verbindung einen eigenen Kindprozess zu erzeugen. Wir verwenden keine Threads, weil diese in der Praxis einige Unzulänglichkeiten zeigen [32]: Bei gleicher Aufgabe läßt sich durch Multiplexing in einem Prozess mehr Performanz gewinnen. Threads benötigen auch komplexere Kontrollmechanismen als der einfache Fall und sind im Debugger schwieriger zu behandeln. Darüberhinaus ist trotz POSIX-Standardisierung die Unterstützung nicht auf allen bekannten Betriebssystemen vorhanden und fehlerfrei.

Um die auf den verschiedenen Betriebssystemplattformen verfügbaren Multiplexing-Mechanismen (`select(2)`, `poll(2)`, `epoll(2)`, `devpoll(2)` und `kqueue(2)`) elegant zu abstrahieren, verwenden wir außerdem die Abstraktionsbibliothek `libevent` [29] von NIELS PROVOS. Diese ist auf allen BSD-Distributionen³ und unter Linux verfügbar. Vorteil der Verwendung von `libevent` ist, dass das Hauptprogramm auf die wesentliche Problemstellung reduziert werden kann und somit übersichtlich bleibt. Dadurch findet automatisch eine Kapselung der Zuständigkeiten zwischen den verschiedenen Schichten (reine Netzwerkbehandlung und IPC) statt.

³Der Autor war sogar selbst am Update der `libevent` unter OpenBSD von Version 0.8 auf Version 1.1a beteiligt.

	ISO/OSI Schicht	Kommunikationsdienst
7	Application	commd
6	Presentation	pack/unpack Datentypen
5	Session	IMSG
4	Transport	TCP
3	Network	IP bzw. IPsec
2	Data Link	POSIX Abstraktion
1	Physical Link	

Tabelle 3.1: Darstellung der zu implementierenden Kommunikationsschicht aus der Sicht des ISO/OSI Modells

Kommunikationsschicht

Die Kommunikationsschicht wird so entworfen, dass sie sowohl lokal, als auch im Netzwerk zwischen ALGs einsetzbar ist. Wiederverwendung von Code für zwei sehr ähnliche Aufgaben ist sinnvoller, als zwei getrennte Teile zu entwerfen und warten zu müssen.

Kommunikation zwischen ALGs

Für die Auswahl der P2P Schicht existieren schon umfangreiche Vorarbeiten in Form allgemeiner Kategorisierung und Terminologie [15]. Im Falle des GeNUGate Clusters liegt eine sehr spezielle Situation eines Netzes vor, die im Falle großer Filesharing-Netzwerke niemals gegeben ist:

- Die maximale Teilnehmermenge A mit $|A| = n$ ist bereits zu Beginn bekannt und verändert sich im weiteren Verlauf nicht.
- Abwesenheit byzantinischer Angreifer⁴ innerhalb des Knotenverbunds, weil geschlossenes System. Wir gehen davon aus, dass wegen der Sicherungsschicht nur die ALGs selbst und der Administrator Zugriff auf die P2P Schicht haben.
- Alle Kommunikationspartner liegen in einem einzigen Subnetz und haben (bis auf den Spezialfall mit ortsverteilten Clustern) sogar gleiche Größenordnung in der Netzwerklatenz.
- Die Teilnehmer wünschen explizit keine Anonymität in der Kommunikation miteinander.

⁴Teilnehmer an einem Kommunikationsprotokoll, die nicht nur aus einer Fehlfunktion heraus, sondern mutwillig falsche bzw. manipulierte Information versenden, um das Protokoll zu stören oder zu ihren Gunsten zu nutzen

n	$ K_n = \frac{n(n-1)}{2}$
1	0
2	1
3	3
4	6
5	10
6	15
7	21
8	28
9	36
10	45

Tabelle 3.2: Wachstum der Kantenzahl in den vollständig vernetzten Graphen K_n . Jeder Knoten selbst hält jeweils $n - 1$ Verbindungen zu den anderen Knoten.

- *Fail-Stop*-Semantik, d. h. ein Knoten beendet seine Dienste bei Auftreten eines Fehlers, stört aber die restlichen Knoten nicht mit fehlerhaften Beiträgen zur Kommunikation.
- *Homogenes* P2P [15], d. h. auf jedem ALG läuft die gleiche Software, die ALGs sind im Bezug auf das P2P Protokoll untereinander gleichberechtigt und nehmen gegenseitig die gleichen Aufgaben wahr.
- *Strukturiertes* P2P [15], d. h. es existiert für jeden Knoten eine eindeutige ID. Diese IDs unterliegen einer Totalordnung, und die Existenz eines beliebigen anderen Knotens ist anhand einer gegebenen ID eindeutig entscheidbar.
- Der Grad der Vernetzung im P2P Graphen kann je nach Wunsch sehr hoch bis vollständig sein, ohne dass die Anzahl der für das P2P Protokoll zu haltenden Verbindungen pro Knoten unbehandelbar wird. Zwar skaliert die Anzahl der Kanten im vollvernetzten Graphen mit $K_n = \mathcal{O}(n^2)$, allerdings ist für unseren Spezialfall auch nur mit maximal $n \leq 8$ zu rechnen (Abbildung 3.2 und Tabelle 3.2).

Grundsätzlich stehen für die Realisierung des Protokolls mehrere Möglichkeiten offen, die im Folgenden stichpunktartig diskutiert werden sollen:

Multicast Jedes ALG muss die für sich relevanten Nachrichten selbst aus dem Multicast-Strom extrahieren. Hierdurch muss auch jede Nachricht auf

dem Multicast-Kanal ein Ereignis in der Multiplexing-Schleife des Kommunikationsdienstes auslösen, selbst wenn die Nachricht gar nicht für den jeweiligen Knoten bestimmt ist. Durch die Multicast-Eigenschaft sind auch keine verbindungsorientierten, sondern nur datagrammbasierte Protokolle möglich. Deshalb ist nicht entscheidbar, ob ein Knoten ausgefallen ist, ohne auf Anwendungsebene einen Mechanismus zum Schicken und Empfangen von Lebenszeichen des Prozesses zu implementieren. Vorteil von Multicast ist, dass innerhalb des P2P Protokolls kein explizites Routing auf Anwendungsebene stattfinden muss, da ohnehin jeder Knoten alle Nachrichten erhält.

UDP Punkt-zu-Punkt Verbindungen Herkunft und Ziel der Netzwerkpakete kann eindeutig entschieden werden. Ankommende Nachrichten sind automatisch für den eigenen Knoten bestimmt und können ausgewertet werden. Allerdings muss beim Versenden von Nachrichten eine Routing-Entscheidung getroffen werden, für welchen Knoten die Nachricht jeweils bestimmt ist. Nachteil von UDP ist, dass keine garantierte Nachrichtenübertragung stattfindet und diese daher noch auf Anwendungsebene nachimplementiert werden müsste.

TCP Punkt-zu-Punkt Verbindungen Routing auf Anwendungsebene ist nötig, allerdings ist durch die Verbindungsbearbeitung der TCP-Schicht eine unmittelbare Rückmeldung über ausgefallene Gegenstellen möglich. Zum Aufsetzen der Verbindung ist allerdings im Gegensatz zu UDP eine Initialisierung (*Three-way Handshake* [36, Kapitel 18] und RFC 793 [28]) nötig.

Fazit

Es erscheint am günstigsten, für das Routing der IMSGs auf Anwendungsebene Punkt-zu-Punkt TCP-Verbindungen zu verwenden. Die Verbindungen sind sehr langlebig (vom Start der HA-Knoten bis zum Fehlerfall), so dass der Handshake nicht ins Gewicht fällt. Ein Wegbrechen der Verbindung wird allerdings durch TCP sehr schnell erkannt, ganz im Gegensatz zu den zustandslosen Datagrammverbindungen. Die jeweils bekannten Gegenstellen (angemeldete lokale Relays bzw. andere ALGs im Cluster) werden in einer dynamischen Listenstruktur verwaltet.

3.3 Implementierung

Die Abbildung des von uns erdachten Kommunikationsschichtmodelles auf reale Implementationsdetails ist in Tabelle 3.3 zu sehen.

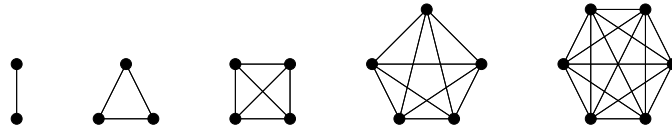


Abbildung 3.2: Die vollständig vernetzten Graphen K_2 bis K_6

Kommunikations-schicht	Implementation
Abstraktion der Textbefehle	Weboberfläche und evtl. interaktive Oberfläche, Rücksprache bei GeNUA nötig
Kommandozeileneingabe	Perl-Programm mit <code>Term::Shell</code> und IMMSG Mechanismus
Regelorientierte Konfigurationssprache	Erschaffung eines eigenen, erweiterbaren Befehlssatzes, abgestimmt auf Funktionalität im Backend
Regelbasierter Selektionsmechanismus	Inspiration bei SQL Mengenabfrage, Selektion nach Relay Art, Suche über PCB (Protocol Control Block, vgl. <code>netstat</code> und [47, Kapitel 22])
Lokalisierung	Tupel $\{[(Client-IP:Port) : (innere ALG-IP:Port)] : [(äußere ALG-IP:Port) : (Server-IP:Port)]\}$ für entfernte Verbindungen, $(IP:PID:RelayNr)$ für Relays
Marshalling	IMMSG in C und Perl
Sicherung	Lokal vertrauenswürdige Unix Domain Sockets, Netzwerkverbindungen durch generischen Mechanismus (SSL oder IPsec)
Lokaler Methodenaufruf	In jeweiliger Programmiersprache
Relay	Bereits existent bei GeNUA, wird im Zuge der DA erweitert

Tabelle 3.3: Gegenüberstellung der Schichten unseres theoretischen Kommunikationsmodells aus dem Entwurf mit der realen Implementierung im Rahmen der Diplomarbeit

Relays

Um die Erweiterung der vorhandenen Relays minimal intrusiv zu gestalten, abstrahieren wir die Nachrichtenschichten in einem eigenen Perl-Modul, das wir im folgenden `IMSG.pm` nennen. Die Implementation der Funktionalität erfolgt mittels objektorientierter Perl-Mechanismen. Hierbei wird die Kommunikationsschnittstelle `IMSG` selbst als Klasse begriffen, die pro Anwendung beliebig oft instanziiert werden kann. Das Relay, das sich grundsätzlich nur je einmal an den Knotencontroller verbindet, besitzt eine Instanz pro Prozess.

Die Entwicklung des Perl-Moduls geschah mit Rücksicht auf das Release 6.0 des GeNUGate im Juni 2005 zunächst getrennt, und mit klar definierten Schnittstellen. Die Einbringung in den GeNUA-Quellcodebaum erfolgte relativ spät. Eine Beschreibung der Programmierschnittstelle von `IMSG.pm` findet sich in Tabelle 3.4.

Die Sockets werden immer nicht-blockierend betrieben, damit nicht die Gefahr besteht, im Multiplexer an einer Stelle hängen zu bleiben und den gesamten Prozess unbrauchbar zu machen.

Knotencontroller

Wir bezeichnen den Knotencontroller im folgenden als `commd`. Er wird in C realisiert, weil dort alle POSIX Aufrufe bereitstehen und C als systemnahe Sprache gut für ein systemnahes Programm wie den Knotencontroller geeignet ist. Sowohl für die beabsichtigte Privilegienseparation, sowie für den ausgesuchten Nachrichtenmechanismus `IMSG` gibt es bereits gut dokumentierte und von einer breiten Benutzergemeinde getestete Referenzimplementationen in C. Wegen der freien Lizenz des OpenBSD Projekts fällt eine Wiederverwendung von Code (hauptsächlich aus dem OpenNTPD) sehr leicht.

Alternativ wäre noch eine Verwendung von C++ in Frage gekommen, da die früher angesprochenen Konzepte *Promises*, *Futures* [8] und *Guardians* in einer objektorientierten Sprache noch deutlicher hätten modelliert werden können. Allerdings hat bei meiner Implementation anfangs der Vorteil durch die Wiederverwendung des in C geschriebenen `IMSG`-Codes überwogen. Dass dieser Code zum Schluss der Diplomarbeit keinen großen Anteil des Gesamtprogramms mehr ausmacht (10% der Lines of Code), war zu Beginn noch nicht abzusehen.

Privilegienseparation

Abbildung 3.3 zeigt, dass jegliche Abwicklung des Netzwerkverkehrs abgetrennt ist von der eigentlichen Bearbeitung der Nachrichten. Durch eine stark eingeschränkte Schnittstelle zwischen den beiden Prozessen kann es

Methodenname	Beschreibung
<code>new(\$sockname)</code>	Erzeugt eine neue Instanz des IMMSG Subsystems.
<code>register_callback(\$type, \$callback)</code>	Registriert eine Callback-Routine, die eintreffende IMMSG-Nachrichten vom Typ <code>\$type</code> behandelt.
<code>read_dispatcher()</code>	Wird vom äußeren Event-Framework aufgerufen, wenn der IMMSG-Filedeskriptor zum Lesen bereit ist. Diese Funktion ruft die entsprechenden Callback-Routinen auf, die vorher registriert wurden.
<code>write_dispatcher()</code>	Wird vom äußeren Event-Framework aufgerufen, wenn der IMMSG-Filedeskriptor zum Schreiben bereit ist. Diese Funktion kümmert sich um das Versenden aller Nachrichten aus der lokalen Warteschlange.
<code>compose(\$type, \$ip, \$data)</code>	Erstellt eine neue IMMSG-Nachricht des Typs <code>\$type</code> mit den Nutzdaten <code>\$data</code> , und stellt diese in die Warteschlange. Der Parameter <code>\$ip</code> steht stellvertretend für viele Parameter, die im IMMSG-Header spezifiziert werden können.
<code>register_eventcb(\$key, \$handler)</code>	Registriert das Event-Framework betreffende Spezialfunktionen, die vom IMMSG-Subsystem aus zugreifbar sein müssen. Dadurch wird IMMSG unabhängig von der verwendeten äußeren Eventbehandlung.
<code>getfd()</code>	Gibt die vom IMMSG Subsystem intern benutzte Dateideskriptornummer zurück. Diese Funktion ist wichtig, weil IMMSG selbständig Wiederverbindungsversuche einleitet, wenn die vorherige Verbindung zusammenbricht. Hierdurch kann sich der Deskriptor ändern.

Tabelle 3.4: Auflistung der wichtigsten vom IMMSG-Subsystem Perl-Modul `IMMSG.pm` angebotenen Methoden mit Funktionsbeschreibung

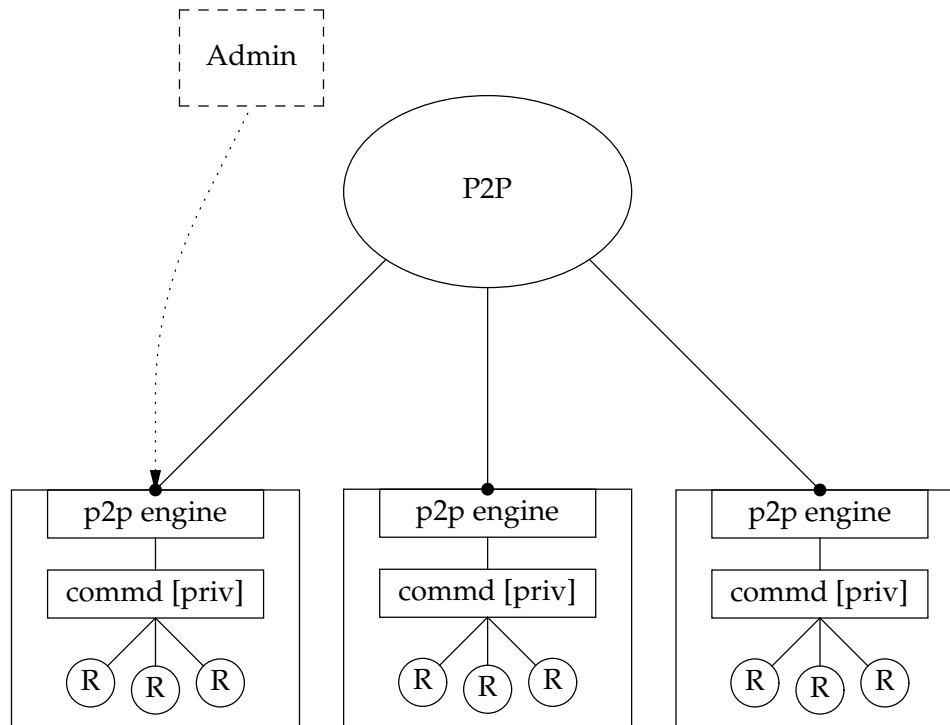


Abbildung 3.3: Modell der Kommunikation zwischen den Relays: Alle Relays R eines Knotens unterhalten sich hierarchisch mit dem lokalen Knotenmaster. Die Knotenmaster untereinander benutzen zur Kommunikation P2P Mechanismen. Der Administrator kann sich auf einen beliebigen Knoten verbinden, um an der Kommunikation teilzunehmen. Die P2P Schicht leitet seine Anfragen entsprechend weiter.

nicht zu einer Eskalation der Privilegien durch Einbruch von entfernter Seite kommen.

Konkret wird der unprivilegierte Prozess nicht als `root`, sondern als Benutzer `__comm` laufen. Die Unterstriche folgen einer Namenskonvention bei GeNUA für Benutzer-IDs privilegienseparierter Dienste. Das Arbeitsverzeichnis des unprivilegierten Teils, in das dieser per `chroot(2)` gesteckt wird, ist `/var/empty`, also ein (bis auf einen `syslog` Log Socket) per Definition leeres Verzeichnis.

Kommunikationsschicht

Genau wie bei der Kapselung der IMMSG-Funktionalität im Perl-Modul habe ich auch beim `commd` versucht, eine Kapselung von IMMSG in eine C-Library durchzuführen.

Eigene Erfahrungen, sowie Austausch mit mehreren OpenBSD-Entwicklern brachten zutage, dass eine `libimsg` im Projekt unerwünscht ist, ob-

wohl meine Implementation bereits die drei OpenBSD-Dienste `ntpd`, `bgpd` und `ospfd` unterstützt hätte. Die Gründe hierfür sind folgende:

- Es existiert keine eindeutige binäre Schnittstellendefinition, sondern jeder Dienst verwendet andere, auf seinen speziellen Anwendungsbereich zugeschnittene *on the wire* Formate.
- Es tritt je nach Einsatzzweck und Gestaltung des Message Passing in den OpenBSD Diensten eine spezielle Verwendung der internen Verwaltungsstrukturen auf. Eine Bibliothek hierfür müsste die Obermenge der vorhandenen Funktionalität bilden. Diese Menge würde dann bei keiner Anwendung vollständig benutzt werden, aber jedes Mal „toten Code“ produzieren.
- Die Dateien `buffer.c` und `imsg.c`, die den Mechanismus ausmachen, stellen in meiner Implementation 304 LOC dar – gerade einmal 10% des Codes.

Eine Kapselung in eine eigene Bibliothek hätte also weder eine einheitliche Schnittstelle vorzuweisen, noch würde die geringe Quellcodemenge diese Maßnahme rechtfertigen. Das Perl-Modul ist trotzdem gerechtfertigt, denn hier wird der Code immer mit der selben Schnittstelle, aber für unterschiedliche Anwendungen (verschiedene Relayarten) verwendet. `IMSG` ist eine Hauptaufgabe für den `commd`, aber nur eine Nebenaufgabe für die Relays. Der `commd` stellt die absehbar einzige C-Anwendung mit `IMSG` bei GeNUA dar, und kann damit von der Nichtauslagerung der ca. 300 Zeilen Code in eine Bibliothek sogar noch mit lokalen Optimierungen profitieren.

Protokollversionsfeld für `IMSG` Pakete

Aus Gründen der Portabilität zwischen verschiedenen Software-Versionen auf dem GeNUGate führen wir im `IMSG`-Protokoll ein Versionsfeld im `IMSG`-Header ein. Dieses soll immer am gleichen Offset vom Beginn der Nachricht stehen und anzeigen, welche Version die Protokollschicht des Absenders besitzt.

Auf diesem Weg halten wir uns die Möglichkeit von zukünftigen Änderungen am Protokoll offen und laufen nicht Gefahr, dass GeNUGates mit verschiedenem Softwarestand beim Versuch, miteinander zu kommunizieren, Fehler produzieren.

Denkbare Reaktionen auf verschiedene Protokollversionen sind einerseits die komplette Ablehnung der Kommunikation, oder (bei kleinerer Softwareversion des Gegenübers) der Rückfall in einen entsprechenden Kompatibilitätsmodus.

Kommunikation zwischen ALGs

Wir verwenden eine Vollvernetzung, weil diese bei der kleinen Knotenzahl sehr geringen Aufwand generiert und algorithmisch leicht zu behandeln ist. Dadurch kann von jedem Knoten des Clusters der Zielknoten einer Nachricht in einem Schritt (*Hop*) erreicht werden.

Der Administrator verbindet sich zum Ansteuern von Relays auf einen beliebigen ALG Knoten. Der dort laufende `commd` dient ihm als *Proxy* [14] in das P2P Netz und abstrahiert den Cluster, indem er die Anfragen des Benutzers transparent weiterleitet, falls sie nicht für den lokalen Knoten bestimmt sind.

Abstand der Wiederverbindungsversuche

Die Knoten im Cluster untereinander versuchen, immer die Vollvernetzung zu erreichen. Fällt ein Knoten aus, so wird regelmäßig versucht, die Verbindung zu ihm wieder herzustellen. Damit nicht sinnlos Netzwerkverkehr generiert wird, wächst das Zeitintervall zwischen den Wiederverbindungsversuchen exponentiell an (*exponential backoff*), bis zu einer konfigurierbaren Zeitschwelle. Mit den Einstellungen im Rahmen der Diplomarbeit starten die Wiederverbindungsversuche im Abstand von einer Sekunde, um bis auf 64 Sekunden maximal anzuwachsen. Eine Obergrenze ist sinnvoll, damit wieder zugeschaltete Knoten sich relativ schnell in den Cluster eingliedern können. So wird ein Wiederausammenfügen des Clusternetzes ohne großen Kommunikationsaufwand realisiert.

Die momentanen Kommunikationspartner werden im `commd` in einer dynamischen Liste gehalten. Hier profitiere ich von den bereits vorhandenen Listenmakros aus `<sys/queue.h>`.

Absicherung

Das HA-Netz, das die ALGs im Cluster intern verbindet, kann prinzipiell nicht als vertrauenswürdig angesehen werden. Dies liegt am möglichen ortsverteilten Aufbau des Clusters und der damit verbundenen Durchleitung des HA-Netzes durch mehrere Gebäude, unbefugt zugänglichen Switches etc.

Wichtig für eine Sicherheitsanalyse ist zunächst die Erarbeitung eines Angreiferszenarios. Ohne das Wissen, gegen welche Bedrohung sich Sicherheitsmaßnahmen richten, ist deren Einrichtung sinnlos („Secrets and Lies“ von BRUCE SCHNEIER [33]). Ein Angreiferszenario hierzu schliesst den Angreifer mit ein, der sich physikalischen Zugriff zum HA-Netz verschafft und sowohl passiv den Datenverkehr beobachtet, als auch durch eigene Injektionen von Paketen versucht, die Kontrolle über die Relays zu übernehmen. Es wird also sowohl eine vertrauliche als auch eine au-

thentifizierte Kommunikation benötigt, sowie die Verhinderung von sog. *Replay*-Attacken, also Schaden durch nochmaliges Schicken bereits versendeter Pakete durch den Angreifer. Dies ist eine Standardanforderung an heutige kryptographische Protokolle, also kann sie auch durch bereits existierende Standardprotokolle (IPsec, SSL) akzeptabel gelöst werden.

Das Problem der gesicherten Übertragung betrifft aber nicht nur die Kommunikationsschicht, die im Rahmen der Diplomarbeit erstellt wird, sondern auch andere Mechanismen des GeNUGates, die das HA-Netz zur Clusterkommunikation nutzen. Dies sind z. B. der *high availability daemon* `had` oder (bei der momentan vorliegenden Implementierung) die Weiterleitung von Mails zu einem ausgezeichneten VPN-Knoten. Deshalb ist ersichtlich, dass eine rein anwendungsspezifische Absicherung der Kommunikation auf OSI-Schicht 5 (z. B. SSL) zwar das Problem für unseren Mechanismus löst, aber für das Gesamtprodukt zu kurz greift.

Eine Rücksprache mit ANTON RÖCKSEISEN ergab, dass für diese Absicherung ein generischer Mechanismus wie z. B. IPsec vorgesehen ist und getrennt von dieser Diplomarbeit realisiert wird. IPsec arbeitet auf OSI-Schicht 3 und kann daher mehrere Dienste auf einmal absichern. Damit ist das Problem zumindest im Fokus der Diplomarbeit gelöst. Gleich, welche Form von Authentifizierung gewählt wird, es muss auch immer für eine sichere und vertrauenswürdige Verteilung der Authorisierungsinformation (SSL-Zertifikate, shared secrets etc.) gesorgt werden.

3.4 Funktionsumfang

Nachdem die Diplomarbeit nur einige wenige Befehle in `comm.pl` als *proof of concept* bereitstellt, zählen wir hier noch Funktionen auf, die bei einer vorhandenen und funktionierenden Kommunikationsschicht leicht ergänzt werden können.

- Aktuelle Netzwerkverbindungen eines Relays bzw. einer Relaymenge auflisten, anhand von (IP:PID) oder (IP:Relayart)
- Dynamisches Nachstarten von Relays und Erzeugen von Sockets
- Dynamisches Nachladen von signiertem Relay-Code, vergleichbar mit dem Hochladen eines Agenten, der im Relay eine bestimmte Aufgabe erledigt
- Möglichkeit zur Selektion einzelner Verbindungen aus dem Cluster anhand von zu definierenden Kriterien (z. B. `regex-match`, `pf-syntax`)
- Einzelne Verbindungen gezielt terminieren, verlangsamen oder Fehler injizieren

- Einsprungpunkte (sog. *hooks*) zum an bestimmte (erwartete) Verbindungen geknüpfte Ausführung von Code (vgl. Agenten s. o.), der seinerseits etwa die entsprechende Verbindung filtert, mitschneidet, interaktiv manipulieren läßt etc.
- Debugging-Information für einzelne Verbindungen mitschneiden. Evtl. Benutzen der Funktionalität von `tcpdump`, um das Rad nicht neu zu erfinden.
- Auslesen von Statistikinformation und Konfiguration von Netzknoten und Relays
- Übermittlung, Prüfung und Aktivierung einer geänderten Konfiguration, auch clusterzentral

Kapitel 4

Benutzeroberfläche

In diesem Kapitel weisen wir die Funktion der in Kapitel 3 implementierten Schnittstelle nach, indem wir als *proof of concept* eine einfache textuelle Oberfläche entwerfen. Darüberhinaus geben wir einen Ausblick auf die mögliche Realisierung einer graphischen Oberfläche.

4.1 Textuelle Oberfläche

4.1.1 Konzept

Die textuelle Oberfläche besteht aus einem Selektions- und einem Operationsmechanismus. Dieses Konzept ist vergleichbar mit dem von SQL¹. Bei SQL wird auch mit dem `WHERE`-Operator eine Einschränkung auf eine Untermenge aller Datensätze anhand von bestimmten Kriterien erzielt. Danach kann diese Menge mit Befehlen wie `SELECT`, `INSERT`, `UPDATE`, `DELETE` etc. bestimmten Operationen unterzogen werden. SQL fasst Selektion und Operation in einer Befehlszeile zusammen.

Die Kommandozeile der Kommunikationsschicht stellt eine Obermenge an Operationen bereit und ist zum einen für den erfahrenen Administrator oder Entwickler gedacht, zum anderen kann sie als Backend für eine graphische Oberfläche verwendet werden.

Die Oberfläche besteht aus einem Parser, vordefinierten Funktionen und der Bindung an das IMMSG-Subsystem (Tabelle 4.1).

4.1.2 Entwurf

Wir sind durch die Abstraktion über die Nachrichtenschicht frei in der Wahl der Programmiersprache. Weil die Oberfläche eher einen *proof-of-concept* Charakter hat und wissenschaftlich nicht sehr interessant ist, wird hier Wert auf eine schnelle Realisierung gelegt.

¹Structured Query Language

Für die Relays wurde bereits ein Perl-Modul erstellt, das eine Schnittstelle zur Nachrichtenschicht bereitstellt, das hier wiederverwendet werden kann. Für Perl existieren fertige Module zur Realisierung einer Kommandozeilenschnittstelle, außerdem ist die Sprache bei GeNUA sehr oft in Verwendung. Daher eignet sich in diesem Fall Perl für die Kommandozeile sehr gut.

Beschreibung des Befehlssatzes

Der Befehlssatz der Kommandozeileingabe sieht einen Mechanismus vor, mit dem Mengen spezifiziert werden können (`select`), einen Befehl, mit dem bereits spezifizierte (mit Namen versehene) Mengen als aktuelle Menge ausgewählt werden können (`use`), und Kommandos, die auf der jeweils aktuellen Menge bzw. auf angegebenen Mengen arbeiten.

Das Konzept der Kommandozeile stellen wir in Abbildung 4.1 in einer EBNF-Grammatik (*Erweiterte Backus-Naur-Form* [2]) vor. Untermengen der Gesamtmenge an Relay-Prozessen und darin enthaltener Relay-Objekten werden mithilfe von generischen und Relay-spezifischen Attributen spezifiziert. Das Dreiertupel (`IP:PID:Relay-Nr`) reicht beispielsweise aus, um einen Relay-Prozess eindeutig zu identifizieren. Die Angabe eines Relaytyps schränkt die Menge der ausgewählten Relays ebenfalls stark ein. Eine Erweiterung der Spezifikation um einzelne Attribute ist leicht möglich. Das den Verteilten Systemen [34] innewohnende Namensraum-Problem ist damit hinreichend gelöst.

Es wird übrigens nicht zwischen allgemeinen Untermengen und einelementigen unterschieden, daher können alle Operationen sowohl auf einzelnen Objekten, als auch auf einer Gruppe von Objekten ausgeführt werden.

4.1.3 Implementierung

Wir bedienen uns des CPAN-Paketes `Term::Shell`, das auch beim Textclient des Türschließsystems an der Universität Erlangen schon erfolgreich im Einsatz ist. Da das Paket noch nicht unter OpenBSD als Port verfügbar war, habe ich einen solchen Port erstellt (er ist jetzt offiziell unter OpenBSD als `devel/p5-Term-Shell` verfügbar). Die Kommandozeilenoberfläche ist also prinzipiell eine Textshell, die mit o. g. Paket realisiert wurde, und durch Einbinden des auch bei den Relays benutzten `IMSG.pm` die IPC-Kommunikation durchführen kann.

Beim Start des Programms (`comm.pl`) kann entweder ein Socket angegeben werden, oder es wird der Standardsocket des `commd`, nämlich `/var/run/commd/socket` verwendet.

```
start = { command }
      ;

command = select      (* specify a set *)
        | use        (* use as current set *)
        | status     (* one example for a command
                       working on current set *)
        ;

select = "select" { attribute } [ assignment ]
       ;

attribute = type
          | pid
          | relaynr
          | ha-ip
          ;

assignment = "as" identifier
           ;

use = "use" identifier
    ;

status = "status"
       ;
```

Abbildung 4.1: Erweiterte Backus-Naur-Form der Sprache, die in der Kommandozeilenoberfläche zum Einsatz kommt. Durch die Definition als EBNF bleibt die Sprache einfach zu überblicken und nachträglich gut erweiterbar. Es sind nur die syntaktischen Elemente der Sprache dargestellt, nicht die lexikalischen (Erfassen von Tokens und Identifiern). Die Möglichkeiten der Produktion `attribute` werden hier stellvertretend mit den nicht weiter definierten Regeln `type`, `pid`, `relaynr` und `ha-ip` aufgeführt. Sie stehen für Prädikate, mit denen *Key-Value*-Paare zur Eingrenzung der Menge übergeben werden können.

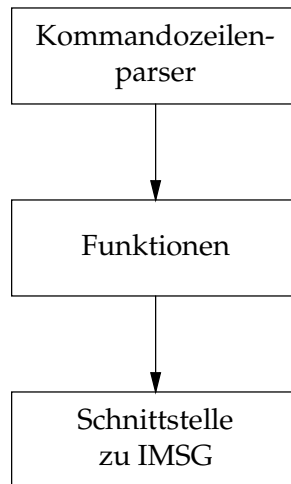


Tabelle 4.1: Schematischer Aufbau des Kommandozeilenwerkzeugs: Ein Parser verarbeitet die eingegebenen Kommandozeilen und ruft anhand dieser bestimmte Funktionen auf. Die Funktionen benutzen IMMSG-Funktionalität, um mit den Knotencontrollern zu kommunizieren.

4.2 Ansätze für eine Graphische Oberfläche

Dieser Abschnitt bietet bewusst nur einen Ausblick auf die Realisierung einer graphischen Oberfläche. Eine ernsthafte Beschäftigung mit dem Thema müsste eine ausführliche Bedarfsanalyse mit Betrachtung der Zielgruppe, sowie ergonomische Betrachtungen mit einschließen, was den Umfang der Diplomarbeit sprengen würde. Bei GeNUA wird der Entwurf und die Implementierung der Oberfläche als eigenes, separates Diplomarbeitsthema betrachtet.

4.2.1 Konzept

Die graphische Oberfläche zielt darauf ab, mit einer Untermenge der aus der Textkonsole verfügbaren Operationen den Hauptanwendungsfall für den Benutzer abzudecken, aber die Operationen in „intuitiver“² Weise zur Verfügung zu stellen.

Um die Gedanken für die Oberfläche aufzuzählen, bedienen wir uns des bekannten *Model-View-Controller*-Paradigmas. Hier einige Beispiele:

²Der Begriff *intuitiv* im Bereich von Software wird in vielen verschiedenen Definitionen verwendet und stellt zwischen Experten verschiedener Fachrichtungen (Ergonomie, Arbeitspsychologie, Wirtschaftsinformatik) einen Streitfall dar. Wahlweise kann er durch „benutzt eine Mehrheit der Anwender“, aber auch durch „ohne Vorkenntnisse bedienbar“ ersetzt werden.

Model Intern begreifen wir die Situation so, wie sie auch in der Kommandozeile beschrieben wird: Es liegt eine Menge von Relay-Objekten vor, die bestimmte Eigenschaften besitzen. Über diese Eigenschaften ist eine Gruppierung oder Differenzierung möglich, weiterhin sind die Relay-Objekte eindeutig unterscheidbar, weil jedes über eine eindeutig vergebene Relay-Nummer verfügt.

View Prozentsatz der aktuell ausgewählten Relays in der Gesamtmenge der Relays

View Graphische Darstellung des Clusters mit Abbildung der Relays (Zuordnung der Relays zu Knoten und Relay-Prozessen)

View Implizite Farbveränderung stark ausgelasteter Relays (Hitzepunkte)

Controller Maske zur interaktiven Selektion von Relays aus der Gesamtmenge per Attribut (Drop-Down Menüs, Checkboxes, Schieberegler). Aktualisierung der Views entweder sofort nach Einstellung oder auf Anforderung (Update-Knopf)

4.2.2 Entwurf

Grundsätzlich ist die Frage zu klären, ob eine interaktive Oberfläche benötigt wird, die dem Benutzer ein sich ständig aktualisierendes Bild bieten soll (Interrupt-Prinzip), oder ob eine statische, web-basierte Version genügt (Polling-Prinzip). Manche Teile der gewünschten Funktionalität (Konfiguration, Statistiken, Übersicht) lassen sich sehr einfach in einem Web-GUI darstellen. Andere dagegen haben hohe Interaktion und sind hierfür eher problematisch. Die Realisierung einer interaktiven Oberfläche schränkt die Plattformunabhängigkeit ein, da im Gegensatz zu einer per Web-Browser bedienbaren Oberfläche nicht mehr alle Betriebssysteme als mögliche Clientplattformen zur Verfügung stehen.

Weil bei GeNUA die komplette Bedienung des GeNUGate per Web-Oberfläche vorgesehen ist, wird die Antwort auf diese Entwurfsfrage wohl stark in Richtung Web tendieren.

4.2.3 Implementierung

Die Wahl der Programmiersprache bei GeNUA wird sehr wahrscheinlich wieder auf Perl fallen, da der Rest der graphischen Oberfläche auch in dieser Sprache implementiert ist. Perl bietet einfache Möglichkeiten, web-basierte oder interaktive grafische Oberflächen zu erstellen.

Kapitel 5

Fazit

Dieses Kapitel zieht Bilanz über die Ergebnisse der Arbeit sowie über die dabei entstandenen Erfahrungen. Die Resultate werden bewertet und einem kritischen Rückblick unterzogen. Es wird auch versucht, Anknüpfungspunkte für weiterführende Arbeit und Ausblicke in die Zukunft zu geben.

5.1 Resultate und Diskussion

5.1.1 Teststellungen

In diesem Abschnitt diskutieren wir hauptsächlich bereits erfolgte Tests an der Middleware, geben aber auch Ausblick auf mögliche weitere Tests.

Labor-Teststellung

commd Vernetzungstest Um die Funktion und Fehlertoleranz der Knotencontroller zu prüfen, habe ich in einer Teststellung zwei Rechner in einem Subnetz mit einem `commd` versehen, und in der Konfigurationsdatei die IP-Adresse des jeweils anderen Rechners als `peer` eingetragen. Ziel dieses Tests ist die Sicherstellung, dass

1. die Konfigurationsdatei korrekt interpretiert wird und der Dienst fehlerfrei startet
2. sich die Knotencontroller nach dem Start gegenseitig finden
3. nach gegenseitigem Auffinden die Wiederverbindungsversuche gestoppt sind
4. bei Wegfallen eines Partners der andere in einen Wiederverbindungsmodus fällt

5. die Wiederverbindungsversuche wie beabsichtigt in einem exponentiell wachsenden Abstand mit zeitlicher Obergrenze stattfinden

In diesem Labortest konnten alle genannten Punkte reproduzierbar und erfolgreich nachvollzogen werden.

commd lokaler IMMSG-Empfang Dieser Test konnte auf einer einzelnen Maschine ausgeführt werden. Ziel des Tests war es, den grundlegenden Empfang und das Versenden von IMMSG Nachrichten über die vom `commd` bereitgestellten Unix Domain Sockets zu verifizieren. Das bedeutet, dass

1. die Konfigurationsdatei korrekt interpretiert wird und die entsprechenden Sockets angelegt werden
2. eine Verbindung auf die Sockets möglich ist
3. der `commd` mit mehreren Verbindungen auf einmal zurechtkommt
4. die über den Socket versandten Daten als gültiges oder ungültiges Protokoll erkannt werden
5. der Socket im Fehlerfall (z. B. ungültiges Protokoll) von Seiten des `commd` geschlossen wird
6. Ereignisbehandlungsfunktionen für bestimmte IMMSG-Typen im `commd` angelegt werden können und bei Erhalt entsprechender IMMSG-Nachrichten auch aufgerufen werden
7. eine vom `commd` als Reaktion zurückgesandte IMMSG-Nachricht auch vom Frontend erkannt und entsprechend behandelt wird

Dieser Test involvierte den Kommunikationsdienst `commd` in Verbindung mit dem Kommandozeilenwerkzeug `comm.pl`. Als Nachrichtentyp für Tests wurde ein eigener Typ, `IMMSG_NONE` verwendet. Alle aufgezählten Punkte konnten im Labortest reproduzierbar nachvollzogen werden.

commd IMMSG-Weiterleitung Dieser Test soll die Routing-Funktionalität eines Verbunds von Knotencontrollern sicherstellen. Ziel des Tests ist, dass von einem Knoten empfangene Nachrichten, die aber nicht für den Knoten selbst bestimmt sind, zuverlässig an ihr Ziel weitergeleitet werden. Das bedeutet, dass

1. `commd` Vernetzung und lokaler IMMSG-Empfang funktionieren
2. aufgrund der Bestimmung der IMMSG-Pakete Routing-Entscheidungen getroffen und durchgeführt werden können

3. im Falle der Nichterreichbarkeit des Ziels eine Nachricht an den Absender geschickt wird

Für diesen Test sind bereits zwei GeNUGates im HA-Verbund erforderlich, sowie ein fertig entwickelter Knotencontroller `commd` und das Kommandozeilentool `comm.pl`. Der Test wurde bis jetzt noch nicht durchgeführt, da meine Implementierung noch nicht den erforderlichen Stand hat. Er ist aber unverzichtbar für das Funktionieren der Middleware, und wird deshalb noch nachgeholt.

Echte Teststellung

Im Gegensatz zum Laborversuch war eine echte Teststellung bis zum Ende der Diplomarbeit nicht vollständig realisierbar. In unserem Zusammenhang hätte eine echte Teststellung bedeutet, dass der im Labor funktionierende Quellcode vollständig und aktiviert in das GeNUGate Produkt gewandert wäre. Dies war aber wegen der sehr vorsichtigen Entwicklungsstrategie bei GeNUA nicht innerhalb eines halben Jahres durchführbar.

Es wurden aber durchaus Teilergebnisse erreicht. So ist etwa das Modul `IMSG.pm` bereits im GeNUGate ab 6.0p1 enthalten, nur nicht standardmäßig aktiviert. Zur Aktivierung führe man auf einem GeNUGate dieser Version nacheinander folgende Schritte aus:

1. Konfigurieren und Starten des `commd` mit `configfw -T commd`
2. Hinzufügen der Option
`IMSG = /var/run/commd/socket`
zur Konfiguration des TCP-Relays,
`/etc/configfw/local/relay/tcprelay.conf`
3. Neustart des TCP-Relays mittels `configfw -T proxy`.

Damit ist das TCP-Relay mit IMMSG-Schnittstelle aktiv, und verbindet sich nach dem Start auf den bereits im Labor getesteten Knotencontroller `commd`. Fällt dieser aus irgendwelchen Gründen aus, so versucht auch das TCP-Relay in exponentiell wachsenden Abständen, sich erneut über den in der Konfiguration angegebenen Socket wieder zum `commd` zu verbinden.

Ist der Knotencontroller erst einmal standardmäßig aktiviert und damit auch in die Prozessüberwachung aufgenommen, so laufen auch diese Wiederverbindungsversuche nicht mehr ins Leere, und es kann von einem System ausgegangen werden, in dem ausgefallene Komponenten nachgestartet werden und sich wieder funktionsfähig ins System einfügen.

5.1.2 Not "yet another" Middleware

In der Diplomarbeit haben wir eine komplett neue Middleware erstellt. Da im Fachgebiet der Informatik viel zu oft das Rad neu erfunden wird, muss

sich auch diese Arbeit der Frage stellen, ob nicht eine andere, bereits existierende Middleware die Anforderungen erfüllt und damit den doch hohen Aufwand einer Neuerstellung überflüssig gemacht hätte.

Diese Frage kann getrost mit „Nein“ beantwortet werden: Die Anforderungen von GeNUA waren zu speziell, als dass eine bereits vorhandene Middleware sie auf Anhieb erfüllt hätte. Der Arbeitsaufwand für nötige Anpassungsarbeiten und der dann immer wieder nötige Aufwand, den Spagat zwischen Weiterentwicklung der externen Middleware und eigenem Produkt zu schlagen, steht dem Aufwand für diese Diplomarbeit sehr wahrscheinlich in nicht viel nach.

Für GeNUA sind wesentliche Ziele erreicht worden: Es wurde ein dem Open Source Paradigma der Firma entsprechendes Modul geschaffen, das nicht nur gut in die bestehende Produktfamilie integriert werden kann, sondern über das GeNUA auch die volle Kontrolle der Weiterentwicklung hat. Dies ist nicht nur deshalb von Vorteil, weil dann eben keine Anpassung an externe Entwicklungen der Middleware stattfinden muss, sondern weil auch die Sicherheitszertifizierung mit einer selbst entwickelten, schlanken Middleware erheblich leichter fällt.

Dass die Arbeit bereits zu großen Teilen in den offiziellen Quellcode der Firma integriert wurde, zeigt dass die Anforderungen der Firma gut erfüllt wurden, und die Arbeit nicht nur eine wissenschaftliche Designstudie war, sondern konkrete Resultate geliefert hat.

5.2 Verwandte Arbeiten

5.2.1 Jackal DSM

Am Lehrstuhl 2 für Informatik der Universität Erlangen ist derzeit eine Forschungsanstrengung im Bereich des Programmierens von Rechnerbündeln (*Clustern*) im Gange. Schwerpunkt wird dort auf die Frage nach der Behandlung der nicht-uniformen Speicherzugriffs-Hierarchie gelegt. Eine Vermutung des Projekts ist, dass die Lösung des Problems ein Querschnittsthema für die beteiligten Bereiche Betriebssysteminteraktion, Übersetzerunterstützung und Programmiersprache/Systementwurf darstellt. Das Projekt erhält durch zahlreiche Studien- und Diplomarbeiten (wie etwa die zitierte DA von TRÖGER [43]) zusätzlichen Anschlag.

Unterschiede zwischen Jackal und dem hier verfolgten Projekte sind jedoch die Ausrichtung auf Hochleistungs-Grid-Computing einerseits und eine strenge Sicherheitsarchitektur mit wenigen Knoten andererseits. Während Performanz für Jackal sehr entscheidend ist, ist sie bei GeNUA eher zweitrangig. Strebt Jackal die Darstellung einer einzigen virtuellen Maschine an, so werden bei GeNUA die einzelnen Firewall-Knoten im Cluster durchaus als solche wahrgenommen.

5.2.2 Cactus

Auch das Cactus-Projekt [1] verfolgt den Betrieb von Hochleistungs-Clustern zur Unterstützung verschiedenster wissenschaftlicher Aufgaben, wie etwa der Klimaforschung, numerischer Berechnungen uvm. Im Vergleich zu Jackal ist festzustellen, dass dieses Projekt weniger versucht, neue Wege zu gehen, als vielmehr bestehende Technik unter freier Lizenz einsatzfertig zu realisieren. So werden z. B. die in der Welt der Clustercomputer gängigen Schnittstellen VMI und MPI unterstützt und großer Wert auf Anpassbarkeit und Erweiterbarkeit gelegt.

Die Unterschiede zwischen Cactus und dieser Diplomarbeit sind aber analog zu denen bei Jackal im vorigen Abschnitt.

5.2.3 AspectIX

Das Projekt AspectIX vom Lehrstuhl 4 für Informatik der Universität Erlangen beschäftigt sich auch mit Middleware, geht aber in eine vollkommen andere Richtung als diese Diplomarbeit. Bei AspectIX geht es um die Erweiterung einer vorhandenen Middleware (in diesem Fall CORBA) um vorher nicht vorhandene Merkmale, unter anderem ein fragmentiertes Objektmodell, Strategien zur impliziten Verteilung von Objekten, und Beschreibung und Möglichkeit zur Manipulation von Objekteigenschaften.

MARKUS FRIEDL behandelte schon 1999 mit dem Entwurf und der Implementierung eines Prototyps von AspectIX in seiner Diplomarbeit [12] die Grundsteine zur praktischen Realisierung der Middleware. Für die vorliegende Arbeit konnte jedoch wegen der vollständig anderen Zielsetzung von AspectIX kein weiterer Nutzen aus dem Projekt gezogen werden.

5.3 Erweiterungsmöglichkeiten

Keine Arbeit in der Informatik ist jemals wirklich vollendet – es gibt immer noch die Perspektive zur Verbesserung. Im Folgenden zählen wir einige Ausblicke für mögliche Fortführungen der Diplomarbeit auf.

5.3.1 Wissenschaftliche Perspektive

Stumpfgenerator

Je öfter eine Middleware-Schnittstelle um Funktionen zum Fernaufruf erweitert wird, desto deutlicher tritt zu Tage, dass sich durch eine automatisierte Erstellung von Funktionsrümpfen und durch eine automatische Generation des Marshallings viel Zeit sparen ließe, und sich Fehler vermeiden lassen würden. Hierzu müsste für unser IMSG-Protokoll ein eigener

Stumpfgenerator geschrieben werden, wie er auch in anderen Middleware-Architekturen vorhanden ist.

Allerdings ist hier zunächst eine Abschätzung zu treffen, ob der *break-even*-Punkt hinsichtlich des Aufwands überhaupt erreicht wird: Bleibt die Anzahl der Erweiterungen hinreichend gering, so ist das saubere Erstellen eines Stumpfgenerators mehr Aufwand als die Handarbeit, die bei allen Erweiterungen insgesamt zu leisten ist.

Abstrakte Schnittstelle

Trotz des Verzichts darauf, aus der IMMSG-Middleware innerhalb der Diplomarbeit eine allgemein verwendbare Bibliothek zu erstellen, ist die Idee, eine abstrakte Schnittstelle für IMMSG zu haben, eine weitere Beschäftigung wert. Ebenso wie beim Stumpfgenerator sollte aber vorher eine Aufwandsbetrachtung durchgeführt werden, ob die zu erwartenden Vorteile den nötigen Arbeitseinsatz rechtfertigen.

5.3.2 GeNUA

Für die Erstellung der Diplomarbeit waren zwar einige Punkte nicht sehr ergiebig, weil sie wissenschaftlich keine Herausforderung dargestellt haben. Beim praktischen Einsatz gibt es allerdings Stellen, an denen die Middleware noch abgerundet werden könnte.

Timeout-Grenzen in Konfigurationsdatei einstellbar machen

Momentan ist das Verhalten beim Wiederverbindungsversuch zwischen den Knotencontrollern fest in den `commd` einprogrammiert. Es wäre nur geringer Arbeitsaufwand nötig, um die Parameter wie initiale Wartezeit, Multiplikationsfaktor und zeitliche Obergrenze für das exponentielle Warteverhalten per Konfigurationsdatei einstellbar zu machen.

Nicht-interaktives Kommandozeilentool analog `sysctl`

Zwar ist es gut und nützlich, gleich zwei verschiedene interaktive Frontends für die IMMSG-Middleware zur Verfügung zu haben. Jedoch wäre gerade im Serverbereich eine zusätzliche Eingriffsmöglichkeit für Administratoren wünschenswert, die auch in nicht-interaktiven Skripten eingesetzt werden kann. Hierbei denkt man an ein Kommandozeilenwerkzeug, das (mit entsprechenden Schaltern aufgerufen) IMMSG-Nachrichten bestimmter Typen erzeugen und verschicken kann, und bei Bedarf auch auf deren Beantwortung warten kann.

Auch diese Idee erfordert lediglich etwas gutes Handwerk, und wäre ohne Probleme mit den bereits vorhandenen Mechanismen realisierbar.

5.3.3 Handwerk und guter Stil

commd auf privilegierten Port legen

Eine weitere Anregung, die auch bei GeNUA nützlich sein wird, ist das Betreiben grundlegender UNIX-Sicherheitskonzepte, wie das Betreiben von Diensten auf privilegierten Ports (also Ports mit einer Nummer unter 1024). Durch diesen Mechanismus kann der jeweils andere Kommunikationspartner davon ausgehen, dass der Besitzer des privilegierten Ports auch auf der Maschine Privilegien genießt, und dadurch eine schwache Vertrauensbeziehung ableiten.

Handwerklich ist diese Änderung ein Standardvorgehen, und ich werde sie nach der Diplomarbeit noch bei GeNUA realisieren.

Lexer überarbeiten

Die Konfigurationsdatei des `commd` wird mit Standardmechanismen unter UNIX eingelesen, namentlich `lex` und `yacc`. Wie man sich vorstellen kann, stand der Entwurf einer Konfigurationssprache und eines Parsers dafür nicht im Vordergrund dieser Diplomarbeit, also wurde nur eine grundlegende Funktionalität realisiert. Alles weitere, z. B. das exakte Parsing von gültigen IP-Adressen oder anderen Daten, muss noch verfeinert werden, um wirklich zufrieden mit dem Code sein zu können.

Timeout-Wert unscharf machen

Es ist von Vorteil, wenn nach einem Wegbruch von Kommunikationspartnern im Cluster nicht alle beteiligten Knoten gleichzeitig einen deterministischen Wiederverbindungsversuch einleiten. Dieses Verhalten erzeugt nur Kollisionen und Spitzenlast. Analog dem CSMA/CD-Verfahren, wie man es vom Einsatz in ungeswitchten Ethernets her kennt, kann auch dem Wiederverbindungsversuch jedes Knotens absichtlich etwas Unschärfe hinzugefügt werden. Denkbar ist etwa, jeden Timeout mit einem Aufruf von `arc4random(3)` zu „salzen“. Hierdurch wird eine bessere Streuung der Wiederverbindungen über die Zeit erreicht, und Spitzen bleiben aus.

SO_KEEPALIVE

Es wird sich voraussichtlich lohnen, die TCP-eigene Socket-Option `SO_KEEPALIVE` für die Verbindungen zwischen den `commd` Prozessen zu verwenden. Mit dieser Option wird auch in Zeiträumen, in denen keine Daten über die Verbindung gehen, die weitere Gültigkeit der Verbindung geprüft. Auf diese Weise können Ausfälle von Nachbarknoten noch früher erkannt werden.

5.4 Kritischer Rückblick auf die Projektplanung

Die ersten beiden Monate der Beschäftigung bei GeNUA (die als Einarbeitung in das Themenfeld, Kennenlernen des Betriebs und Ausloten von Machbarkeit gesehen werden), waren wie beabsichtigt eine reine Recherche und Stoffsammlung, sowie die Zeit theoretischer Überlegungen und des Entwurfs für das Gesamtkonzept der Programmierarbeit. Auch der dritte und vierte Monat, die im Wesentlichen aus der externen Implementierung des `commd`, sowie aus Verfeinerungen am Skript bestanden, waren von vornherein eingeplant.

Bei der Bemessung der Zeit für die Einbringung der erarbeiteten Mechanismen in die Endform (Integration des Codes in den GeNUA Quellcodebaum, Zusammenstellen des vorliegenden Skripts, sowie Erstellung von Präsentationsfolien) konnte jedoch der zunächst angepeilte Zeitrahmen von 3 Monaten nicht eingehalten werden. Wie sich herausstellte, war es weitaus aufwendiger, die Funktionalität in das bereits vorhandene Produkt einzubauen, als ursprünglich angenommen. Dies liegt an mehreren Gründen:

Strenges Peer Review durch Kollegen GeNUA entwickelt das GeNUGate unter Zuhilfenahme von *Peer Review*, d. h. keine Änderung am Quellcodebaum erfolgt ohne die Zustimmung eines Kollegen, der den Änderungsvorschlag vor der Einbringung in den Baum kritisch rekurriert. Dieses Vorgehen ist eine abgeschwächte Form des *extreme programming*. Dort sind immer zwei Leute an der Arbeit, egal bei welchem Arbeitsschritt. Vorteil der *Peer-Review* Technik für GeNUA ist eine geringere Fehlerhäufigkeit im Quellcode, die im Hinblick auf die Sicherheitszertifizierung den erhöhten Personalaufwand rechtfertigt.

Auch mein Beitrag im Rahmen der Diplomarbeit wurde des Öfteren im Rahmen eines Reviews mit dem Hinweis auf noch bestehende Unzulänglichkeiten zurückgewiesen, was die Dauer meiner Arbeit zusätzlich erhöht hat.

Komplexer Quellcode Das Produkt GeNUGate ist im Juni 2005 in Version 6.0 erschienen. Die Wurzeln des Projekts reichen zurück bis zur Firmengründung im Jahre 1992, und dementsprechend umfangreich und komplex ist der Quellcodebaum bis heute angewachsen. Innerhalb der Firma wird bei der Anstellung neuer Mitarbeiter durchaus mit einem halben Jahr Einarbeitungszeit gerechnet, bis diese sich im Codebaum genügend gut auskennen, um in allen Gebieten produktiv arbeiten zu können. Mit diesem Hintergrund war die Aufgabe der Integration durchaus anspruchsvoll.

Mit der DA kollidierender Releasetermin Bereits zu Beginn meiner praktischen Arbeit war spürbar, dass der Codebaum mit besonderer Vor-

sicht behandelt wurde. Das Release der Version 6.0 im Juni 2005 hat zwar nicht zu einem kompletten *Code Freeze* geführt, wie man ihn von vielen Freien Softwareprojekten kennt. Allerdings wurde darauf bestanden, dass meine Diplomarbeit nicht mehr in aktivierter Form in die Version 6.0 mit einfließt. Da es keinen alternativen Codebaum vor Abschluß der Release-Arbeiten gab, habe ich die erzwungene Wartezeit mit nicht-intrusiven Vorbereitungen am Codebaum, sowie mit Dokumentation (Arbeiten am vorliegenden DA-Skript und Folien) gefüllt. Trotzdem war der Zeitverlust durch die Produktrelease deutlich spürbar.

Belastung und Ausfälle des Entwicklungsservers Gerade zu Stoßzeiten kam es auf dem Server, der vom Team für Entwicklung und langwierige Build-Prozesse benutzt wurde, zu Lastwerten von 8 bis 10¹. Build-Prozesse, die unter normalen Umständen schnell erledigt gewesen wären, haben sich hierbei sehr in die Länge gezogen

Auch waren öfters Abstürze des Entwicklungsrechners zu beklagen, die von einer noch nicht einwandfreien Behandlung von NFS-Zugriffen sowie Multiprozessorunterstützung unter dem verwendeten Betriebssystem OpenBSD herrührten.

Aufwendige Entwicklungsumgebung Bedingt durch die eigenen hohen Qualitätsstandards, sowie eine von der Zertifizierungsstelle gewünschte Prozessdefinition innerhalb der Softwareentwicklung am Produkt, muss selbst für kleine Änderungen eine umfangreiche Prozedur absolviert werden:

1. Anlegen eines *Bug-Tickets* im Fehlerverwaltungssystem `bugzilla`
2. Öffnen eines sog. *Changes* in der Quellcodeverwaltung AEGIS [23]
3. Durchführung der beabsichtigten Änderungen am Quellcode in einem Schattenbaum
4. Schreiben von Tests
5. Erfolgreiches Absolvieren des Build-Prozesses mit o. g. Baum
6. Erfolgreiches Absolvieren der Tests: Bestehen des Tests mit geändertem Quellcode, und Fehlschlagen des Tests gegen die alte Baseline²

¹Durchschnittliche Anzahl von Prozessen, die gleichzeitig in einem bestimmten Zeitraum auf ein Betriebsmittel warten

²*Baseline* ist ein Begriff aus MILLERS AEGIS-Entwicklungsumgebung, die bei GeNUA im Einsatz ist. Mit *Baseline* wird der letzte offizielle Stand des Codes vor Öffnen eines neuen Branches bzw. eines neuen Changes bezeichnet. Das Fehlschlagen eines Tests gegen die alte Baseline beweist, dass der Test tatsächlich eine Änderung zwischen alter und neuer Funktionalität im Code feststellt, und nicht nur immer *wahr* zurückgibt.

Für GeNUA ist diese Formalisierung des Arbeitsprozesses unerlässlich. Aus akademischer Sicht ist es ebenfalls höchst erfreulich, dass diese in der freien Wirtschaft oftmals vermiedenen, weil teuren Methoden der Softwaretechnik angewendet werden.

Wechsel des GG-Koordinators Während meiner Diplomarbeit kündigte der zuerst für meine Arbeit zuständige Betreuer ANTON RÖCKSEISEN. Sein Platz, und damit die Betreuung meiner Arbeit wurden von KAI DÖRNE-MANN eingenommen. Dieser Punkt wirkte sich aber nicht besonders gravierend aus, da ich neben den Koordinatoren auch genügend andere Ansprechpartner für meine offenen Fragen im Entwicklerteam hatte.

Anhang A

Praktische Arbeit

Um dem Leser einen kurzen Eindruck über die entstandenen Ergebnisse vermitteln zu können, enthält dieser Anhang wesentliche Elemente des *Look and Feel*, das Benutzer und Entwickler bei Verwendung meiner Werkzeuge vorfinden werden.

A.1 Knotencontroller `commd`

A.1.1 Konfigurationsdatei `commd.conf`

Die hier gezeigte Konfiguration ist ein Beispiel, das alle wichtigen Anweisungen enthält.

```
# Configuration file for
# GeNUA Relay Communication Daemon commd(8)

# relay sockets
relay /var/run/commd.sock
#relay /cage/1.2.3.4/...

# network listeners
listen on 127.0.0.1
#listen on 10.1.254.89

# network peers
#peer 10.1.254.89

# bind to local interface
bind 10.1.254.89
```

A.1.2 Manualseite `commd(8)`

```
NAME
    commd - GeNUGate Relay Communication Layer daemon
```

SYNOPSIS

```
commd [-dnqv] [-C configfile] [-I pidfile] [-P port]
      [-S socket]
```

DESCRIPTION

The `commd` daemon passes messages for local running relay processes in a high-available GeNUA ALG cluster. For this, `commd` listens on both local UNIX domain sockets, and on network ports. Both can be configured via a supplied configuration file, which is by default looked up at `/etc/commd.conf`. The default network port is 4077.

Sending `SIGHUP` to `commd` causes it to restart, and re-read its configuration.

The options are as follows:

- `-C file` Uses `file` as the configuration file, instead of the default `/etc/commd.conf`.
- `-d` Runs `commd` in debug mode: Daemons stay on `tty` and print out debug messages there, instead of writing to `syslog`.
- `-I pidfile` Specify the `pid` file to use instead of the default `/var/run/commd.pid`
- `-n` Run non-privileged, i.e. don't `chroot()` and don't use ports `<1024`.
- `-P port` Specify a port on which `commd` should listen instead of the default port 4077.
- `-q` Lets `commd` behave more quiet. Only the absolute necessary messages about starting up and coming down are logged. Supplying the flag twice even cancels those messages. Note that this flag is the opposite of `-v`.
- `-S socket` Specify an additional unix domain socket path `commd` should listen on.
- `-v` Makes `commd` more verbose. This flag can also be supplied more than once. The `-q` and `-v` flags just manipulate an internal verbosity level variable, upon which decisions of printing a log entry are made.

ENVIRONMENT

Since `commd` makes use of Niels Provos' `libevent`, the environment variables for `libevent` apply here also.

FILES

`/etc/commd.conf` Default configuration file for `commd`.
`/var/run/commd.pid` Default PID file for `commd`.

SEE ALSO

`event(3)`

HISTORY

The `commd` program first appeared in GeNUGate 6.1

A.1.3 Verzeichnisansicht des Quellcodes

```
-rwxr-xr-x  1 gernler  genua  53396 Sep 15 18:42 commd
-r--r--r--  1 gernler  genua    442 Sep 15 18:40 Makefile
-r--r--r--  1 gernler  genua   3183 Sep 15 18:40 buffer.c
-r--r--r--  1 gernler  genua   1210 Sep 15 18:40 buffer.h
-r--r--r--  1 gernler  genua  12348 Sep 15 18:40 commd.c
-r--r--r--  1 gernler  genua   1629 Sep 15 18:40 commd.h
-r--r--r--  1 gernler  genua   2161 Sep 15 18:40 config.c
-r--r--r--  1 gernler  genua   1735 Sep 15 18:40 config.h
-r--r--r--  1 gernler  genua   6590 Sep 15 18:40 imsg.c
-r--r--r--  1 gernler  genua   2704 Sep 15 18:40 imsg.h
-r--r--r--  1 gernler  genua   1669 Sep 15 18:40 la_subr.c
-r--r--r--  1 gernler  genua   1155 Sep 15 18:40 la_subr.h
-r--r--r--  1 gernler  genua   1738 Sep 15 18:40 lex.l
-r--r--r--  1 gernler  genua   8900 Sep 15 18:40 listener.c
-r--r--r--  1 gernler  genua   1191 Sep 15 18:40 listener.h
-r--r--r--  1 gernler  genua   3656 Sep 15 18:40 log.c
-r--r--r--  1 gernler  genua   1176 Sep 15 18:40 log.h
-r--r--r--  1 gernler  genua  11300 Sep 15 18:40 p2p.c
-r--r--r--  1 gernler  genua    995 Sep 15 18:40 p2p.h
-r--r--r--  1 gernler  genua    987 Sep 15 18:40 parse.h
-r--r--r--  1 gernler  genua   2579 Sep 15 18:40 parse.y
-r--r--r--  1 gernler  genua   4605 Sep 15 18:40 relays.c
-r--r--r--  1 gernler  genua   1281 Sep 15 18:40 relays.h
```

A.1.4 Ausgabe eines Aufrufs von `commd -dvv`

```
# /usr/local/gg/sbin/commd -dvv
relay controller [priv] ready
p2p engine ready
new relay connection on /cage/proto/var/run/commd.sock
```

```
new relay connection on /var/run/commd.sock
closing connection to /cage/proto/var/run/commd.sock
closing connection to /var/run/commd.sock
p2p_listener_cb: new network connection from 172.16.100.14
closing connection to 172.16.100.14
```

A.1.5 Ansicht des commd in der Prozessliste

```
# ps -ax -O wchan | grep commd
9458 kqread ?? Is      0:00.00 commd: [priv] (commd)
26797 kqread ?? I       0:00.00 commd: p2p engine (commd)
```

Literaturverzeichnis

- [1] *The Cactus Code Server*. URL <http://www.cactuscode.org/>.
- [2] *ISO/IEC 14977 : 1996(E) – Extended Backus Naur Form*. URL <http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf>.
- [3] *MPICH-G2*. URL <http://www.hpclab.niu.edu/mpi/>.
- [4] *The National Center for Supercomputing Applications*. URL <http://www.ncsa.uiuc.edu/>.
- [5] *OpenSSH*. URL <http://www.openssh.com/>.
- [6] *Virtual Machine Interface 2.1*. URL <http://vmi.ncsa.uiuc.edu/>.
- [7] B. CALLAGHAN, B. PAWLOWSKI und P. STAUBACH. *NFS Version 3 Protocol Specification*. RFC 1813 (Informational), Jun 1995. URL <http://www.ietf.org/rfc/rfc1813.txt>.
- [8] A. CHATTERJEE. *FUTURES: a mechanism for concurrency among objects*. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Seiten 562–567. ACM Press, New York, NY, USA, 1989. ISBN 0-89791-341-8.
- [9] Das OpenBSD Projekt. *OpenBSD*. URL <http://www.openbsd.org/>, freies, multi-Plattform 4.4BSD-basiertes UNIX-ähnliches Betriebssystem.
- [10] Das OpenBSD Projekt. *OpenNTPD*. URL <http://www.openntpd.org/>, freie und portable Implementation des Network Time Protokolls.
- [11] MARCUS DEININGER, HORST LICHTER, JOCHEN LUDEWIG und KURT SCHNEIDER. *Studien-Arbeiten*. vdf Verlag, fünfte Auflage, 2005. ISBN 3-7281-3012-5. Ein Leitfaden zur Vorbereitung, Durchführung und Betreuung von Studien-, Diplom- und Doktorarbeiten am Beispiel Informatik.

- [12] MARKUS FRIEDL. *Entwurf und Implementierung eines Prototyps der objektorientierten Middleware AspectIX*. Diplomarbeit, FAU Erlangen-Nürnberg, Juni 1999.
- [13] R. FRYE, D. LEVI, S. ROUTHIER und B. WIJNEN. *Coexistence between Version 1, Version 2, and Version 3 of the Internet-standard Network Management Framework*. RFC 3584, August 2003. URL <ftp://ftp.internic.net/rfc/rfc3584.txt>.
- [14] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON und JOHN VLISIDES. *Entwurfsmuster*. Addison-Wesley, 1996. ISBN 3-89319-950-0.
- [15] ALEXANDER VON GERNLER. *Analyse und Vergleich von Peer-to-Peer basierten Algorithmen zur Lokalisierung von Ressourcen*. Studienarbeit, Universität Erlangen-Nürnberg, Lehrstuhl IV für Informatik: Verteilte Systeme und Betriebssysteme, Juni 2003. URL <http://pestilenz.org/~grunk/werke/2003/p2p.pdf>.
- [16] R. HINDEN und S. DEERING. *Internet Protocol Version 6 (IPv6) Addressing Architecture*. RFC 3513 (Proposed Standard), Apr 2003. URL <http://www.ietf.org/rfc/rfc3513.txt>.
- [17] IANA. *Special-Use IPv4 Addresses*. RFC 3330 (Informational), Sep 2002. URL <http://www.ietf.org/rfc/rfc3330.txt>.
- [18] BUNDESAMT FÜR SICHERHEIT IN DER INFORMATIK, Herausgeber. *IT-Grundschutzhandbuch*. Bundesanzeiger-Verlag. ISBN 3-88784-915-9. URL <http://www.bsi.bund.de/produkte/cdrom/index.htm>.
- [19] B. LISKOV und L. SHRIRA. *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, Seiten 260–267. ACM Press, New York, NY, USA, 1988. ISBN 0-89791-269-1.
- [20] BARBARA LISKOV. *Primitives for distributed computing*. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, Seiten 33–42. ACM Press, New York, NY, USA, 1979. ISBN 0-89791-009-5.
- [21] SUN MICROSYSTEMS. *XDR: External Data Representation standard*. RFC 1014, Juni 1987. URL <http://www.ietf.org/rfc/rfc1014.txt>.
- [22] SUN MICROSYSTEMS. *NFS: Network File System Protocol specification*. RFC 1094 (Informational), Mar 1989. URL <http://www.ietf.org/rfc/rfc1094.txt>.

- [23] PETER MILLER. *AEGIS*. URL <http://aegis.sourceforge.net/>.
- [24] DAVID L. MILLS. *Network Time Protocol (Version 3) Specification, Implementation*. RFC 1305, März 1992. URL <ftp://ftp.internic.net/rfc/rfc1305.txt>, obsoletes RFC0958, RFC1059, RFC1119.
- [25] J. MOY. *OSPF Version 2*. RFC 2328, April 1998. URL <ftp://ftp.internic.net/rfc/rfc2328.txt>, see also STD0054. Obsoletes RFC2178. Status: STANDARD.
- [26] BERND OESTEREICH. *Analyse und Design mit UML 2*. Oldenbourg, 7. Auflage, 2005. ISBN 3-486-57654-2. URL <http://www.oldenbourg-verlag.de/>.
- [27] The Open Group. *The Single Unix Specification*, v3 Auflage, 2004. URL <http://www.unix.org/version3/online.html>.
- [28] J. POSTEL. *Transmission Control Protocol*. RFC 793 (Standard), Sep 1981. URL <http://www.ietf.org/rfc/rfc793.txt>, updated by RFC 3168.
- [29] NIELS PROVOS. *libevent - an event notification library*. URL <http://www.monkey.org/~provos/libevent/>.
- [30] NIELS PROVOS, MARKUS FRIEDL und PETER HONEYMAN. *Preventing Privilege Escalation*. In *12th USENIX Security Symposium*. August 2003. URL <http://www.citi.umich.edu/u/provos/ssh/privsep.html>.
- [31] THEO DE RAADT. *Exploit Mitigation Techniques*, März 2005. URL <http://www.openbsd.org/papers/aug04/>.
- [32] DOUGLAS C. SCHMIDT. *Reactor – An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events*, 1994. URL <http://www.cs.wustl.edu/~schmidt/patterns-ace.html>.
- [33] BRUCE SCHNEIER. *Secrets & Lies – Digital Security in a Networked World*. John Wiley & Sons, Inc., New York, erste Auflage, 2000. ISBN 0-471-25311-1.
- [34] WOLFGANG SCHRÖDER-PREIKSCHAT. *Verteilte Systeme*. Vorlesung, 2005. URL http://www4.cs.fau.de/Lehre/SS05/V_VS/, Lehrstuhl für Informatik IV, Friedrich-Alexander-Universität Erlangen-Nürnberg.

- [35] S. SHEPLER, B. CALLAGHAN, D. ROBINSON, R. THURLOW, C. BEAME, M. EISLER und D. NOVECK. *Network File System (NFS) version 4 Protocol*. RFC 3530 (Proposed Standard), Apr 2003. URL <http://www.ietf.org/rfc/rfc3530.txt>.
- [36] W. RICHARD STEVENS. *TCP/IP Illustrated – The Protocols*, Band 1. Addison-Wesley, 1994. ISBN 0-201-63346-9.
- [37] W. RICHARD STEVENS. *TCP/IP Illustrated – TCP for Transactions, HTTP, NNTP, and the UNIX© Domain Protocols*, Band 3. Addison-Wesley, 1996. ISBN 0-201-63495-3.
- [38] W. RICHARD STEVENS. *UNIX Network Programming – Networking APIs: Sockets and XTI*, Band 1. Prentice Hall, zweite Auflage, 1998. ISBN 0-13-490012-X. URL <http://www.kohala.com/start/unpv12e.html>.
- [39] W. RICHARD STEVENS. *UNIX Network Programming – Interprocess Communication*, Band 2. Prentice Hall, zweite Auflage, 1999. ISBN 0-13-081081-9. URL <http://www.kohala.com/start/unpv22e/unpv22e.html>.
- [40] VOLKER STREHL. *Topics in Algorithms and Complexity*. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2003. <http://www8.cs.fau.de/IMMD8/Lectures/TAC/TAC05/skript.pdf>.
- [41] ANDREW S. TANENBAUM. *Computer Networks*. Prentice Hall, dritte Auflage, 1996. ISBN 0-13-394248-1.
- [42] Team Squid. *Squid Web Proxy Cache*. URL <http://www.squid-cache.org/>.
- [43] FRANK TRÖGER. *Design und Implementierung eines Kommunikationspaketes für Jackal*. Diplomarbeit, FAU Erlangen-Nürnberg, Jan 2005. <http://www2.cs.fau.de/Lehre/SA-DA/JackalCommPackage.xml>.
- [44] S. WALDBUSSER. *Remote Network Monitoring Management Information Base*. RFC 2819 (Standard), Mai 2000. URL <http://www.ietf.org/rfc/rfc2819.txt>.
- [45] U.S. WARRIER, L. BESAW, L. LABARRE und B.D. HANDSPICKER. *Common Management Information Services and Protocols for the Internet (CMOT and CMIP)*. RFC 1189 (Historic), Oktober 1990. URL <http://www.ietf.org/rfc/rfc1189.txt>.
- [46] Wikimedia Foundation. *Wikipedia*. URL <http://www.wikipedia.org/>, die freie Enzyklopädie.

- [47] GARY W. WRIGHT und W. RICHARD STEVENS. *TCP/IP Illustrated – The Implementation*, Band 2. Addison-Wesley, 1995. ISBN 0-201-63354-X.