

Framework zur Anomalie-basierten Angriffserkennung durch verteilte Instanzen

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

von

cand. inform.

Christoph Mayer

Betreuer:

Prof. Dr. M. Zitterbart
Dipl.-Inform. Thomas Gamer

Tag der Anmeldung: 15. Mai 2007
Tag der Abgabe: 09. November 2007

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

Karlsruhe, den 09. November 2007

Inhaltsverzeichnis

1	Einleitung	1
1.1	Zielsetzung der Arbeit	2
1.2	Gliederung der Arbeit	3
2	Grundlagen	5
2.1	Terminologie	5
2.2	Sampling und Filtering	6
2.3	Zufallszahlen	7
2.4	Pcap-System	10
2.5	Protokoll-Parser	12
2.6	Dynamische Bibliotheken	13
2.7	Netzwerk-Simulation	14
3	Analyse	19
3.1	Anforderungen	20
3.1.1	Skalierbarkeit	20
3.1.2	Erweiterbarkeit	21
3.1.3	Flexibilität	21
3.1.4	Robustheit	22
3.2	Existierende Lösungsansätze	23
3.3	Angriffserkennung in Produktiv-Umgebungen	25
3.3.1	Snort und PreludeIDS	25
3.3.2	Bro	27

4	Entwurf	29
4.1	Architektur	29
4.2	Netzwerkabstraktion	32
4.3	Modul-Umgebung	32
4.4	Lokales Nachrichtensystem	35
4.5	Verteilte Kommunikation	36
4.5.1	Umsetzung der verteilten Kommunikation	37
4.5.1.1	Zeitliche Aspekte entfernter Nachrichtenübertragung	37
4.5.1.2	Serialisierung und Deserialisierung von Nachrichten	38
4.5.1.3	Wahl der Nachrichtenempfänger	38
4.5.2	Ablauf der verteilten Kommunikation	39
4.5.3	Beispielhafte Protokolle	39
4.6	XML-basierte Konfiguration	41
4.7	Zusammenfassung	41
5	Implementierung	43
5.1	Netzwerkabstraktion	43
5.1.1	NetworkInterface	44
5.1.1.1	NetworkInterface für das Pcap-System	45
5.1.2	FrameBuffer	48
5.1.3	NetworkManager	49
5.2	Modul-Umgebung	49
5.2.1	ModuleManager	50
5.2.2	FrameDistribution	51
5.2.3	ChannelManager	51
5.2.4	Channel	51
5.2.5	Module	52
5.2.5.1	Konfiguration	53
5.2.5.2	DynamicLibrary	53
5.2.5.3	DistackModuleInterface	55
5.2.5.4	MessagingNode	57
5.2.5.5	Timer	59
5.2.5.6	Zusammenhang	61

5.3	Nachrichtensystem	62
5.4	Verteilte Kommunikation	64
5.4.1	MessagingNode	65
5.4.2	Message	67
5.4.3	RemoteMessagingSystem	69
5.4.4	MessagingSystem	71
5.4.5	Communication	71
5.4.5.1	TCP-Socket-basierte Kommunikationsschicht	72
5.5	Konfiguration	73
5.5.1	Konfigurationsformat und die Triggerconf-Bibliothek	73
5.5.2	Allgemeine Konfiguration	75
5.5.3	Konfiguration der Modul-Definitionen	75
5.5.4	Konfiguration zur Verschaltung der Module	77
5.6	Hilfsklassen	78
5.6.1	Logging	78
5.6.2	Routing-Tabelle	79
5.6.3	Patricia-Trie	81
5.6.4	Hilfsfunktionen	81
5.7	Standardmodule	81
5.7.1	Hilfs-Module	81
5.7.1.1	ModuleOutputterPlotter	82
5.7.1.2	ModuleUtilityIpLister	83
5.7.1.3	ModuleUtilityTimer	83
5.7.2	Sampling- und Filtering-Module	84
5.7.2.1	ModuleSampling_noutOfN	84
5.7.2.2	ModuleSamplingSystematicCountBased	85
5.7.2.3	ModuleSamplingSystematicTimeBased	85
5.7.2.4	ModuleSamplingUniformProbabilistic	86
5.7.2.5	ModuleFilteringProtocol	86
5.7.2.6	ModuleFilteringSubnet	86

6	Evaluierung	89
6.1	Skalierbarkeit	89
6.1.1	Lokale Skalierbarkeit	91
6.1.1.1	Durchsatz	92
6.1.1.2	CPU- und Speicherlast	96
6.1.2	Verteilte Skalierbarkeit	98
6.2	Erweiterbarkeit	100
6.2.1	Integration einer Angriffserkennung	100
6.2.1.1	Analyse der Angriffserkennung	100
6.2.1.2	Integration der Angriffserkennung	101
6.2.1.3	Evaluierung der Skalierbarkeit	105
6.2.1.4	Zusammenfassung	106
6.2.2	MPLS-Überwachung	106
6.2.2.1	MPLS Protokoll-Parser	107
6.2.2.2	MPLS-Angriffe und Überwachung	107
6.3	Flexibilität	108
6.3.1	Portierung nach OMNeT++	108
6.3.1.1	Kapselung des Framework im einem Simple-Module	109
6.3.1.2	Netzwerkabstraktion	109
6.3.1.3	Timer- und Zeit-Funktionalität	110
6.3.1.4	Threads	111
6.3.1.5	Routing-Tabelle	112
6.3.1.6	Prozessraum	112
6.3.1.7	Build-Umgebung und NED-Dateien	113
6.3.1.8	Simulation der Angriffserkennung	115
6.3.2	Beispielhafte Anwendungsfälle	116
6.3.2.1	Lokale Anwendungsfälle	116
6.3.2.2	Verteilte Anwendungsfälle	118
6.4	Robustheit	121
6.4.1	Korrupte Netzwerkpakete	122
6.4.2	Überlast	122
7	Zusammenfassung und Ausblick	125
	Literatur	127

1. Einleitung

Im Jahre 1988 brachte Robert Tappen Morris den ersten Internet-Wurm der Geschichte in Umlauf [Spaf04]. Die sehr schnelle Verbreitung des Wurms führte zum Zusammenbruch ganzer Netzbereiche und dem damit ersten Ausfall im Internet, welcher durch einen verteilten Angriff verursacht wurde. Heute, 19 Jahre nachdem der erste Internet-Wurm der Geschichte sein Unwesen trieb, haben sich Netzstrukturen und Mechanismen stark geändert. Wohingegen 1988 ein einzelner Wurm Teile des Internet zum Erliegen brachte, sind es heute Botnetze mit Größen bis zu 20 000 Bots [Post06], welche verwendet werden, um gezielt Systeme im Internet mit verteilten Denial-of-Service-Angriffen (DoS-Angriffe) auszuschalten. Der *Worldwide Infrastructure Security Report* [Arbo07] nennt Botnetze und verteilte DoS-Angriffe die größten Gefahren, welche zur Zeit das Internet bedrohen.

Die Erkennung von DoS-Angriffen und ihre Bekämpfung ist ein bis heute ungelöstes Problem der Netzsicherheit. Client-basierte Lösungen gegen DoS-Angriffe haben sich als ungeeignet erwiesen, da diese den Host selbst und auch das Netz nicht schützen können. Vielversprechende Lösungen liegen im Bereich der Angriffserkennung durch verteilte Instanzen, welche im Netzzinneren auf Vermittlungssystemen agieren und kollaborativ arbeiten. Dadurch können Angriffe frühzeitig erkannt und bekämpft werden, wodurch das Netz und die darin befindlichen Systeme besser geschützt werden können.

Um im Netzzinneren zu arbeiten, muss ein Angriffserkennungssystem hoch skalierbar sein, damit die dort beobachteten Verkehrsprofile verarbeitet werden können. Weiterhin muss das Angriffserkennungssystem auf Vermittlungssystemen mit geringen Ressourcen lauffähig sein, ohne diese selbst in ihrer Arbeit zu beeinflussen. Die sehr schnell wachsende Anzahl an neuen Protokollen macht eine leichte Erweiterbarkeit unabdingbar. Ferner fordert die hohe Forschungsaktivität im Bereich der Angriffserkennung eine leichte Erweiterbarkeit in Bezug auf neue Methoden und Verfahren zur Angriffserkennung. Um die Ressourcen des Vermittlungssystems geringstmöglich zu beeinflussen, müssen nicht benötigte Teile der Angriffserkennung automatisiert ge- und entladen werden. Des Weiteren ist eine hohe Robustheit notwendig, um zu verhindern, dass das Vermittlungssystem im Fall einer Fehlfunktion des Angriffser-

kennungssysteme oder eines Angriffs auf das Angriffserkennungssystem beschädigt oder in einer anderen Weise in seiner Arbeit behindert wird.

Die Verteilung und Kollaboration des Angriffserkennungssystems auf eine Menge von Vermittlungssystemen erlaubt die Nutzung von verteiltem Wissen durch die verschiedenen Instanzen. Dadurch können qualitativ hochwertigere Entscheidungen getroffen werden, was zu besseren Ergebnissen der Angriffserkennung führt. Weiterhin bietet die Verteilung und Kollaboration der Angriffserkennungssysteme die Möglichkeit, schwächere Knoten nur mit bestimmten Komponenten des Angriffserkennungssystems zu betreiben. Diese können somit spezifische Aufgaben übernehmen und als Auslagerung für andere Instanzen dienen.

Die zunehmende Heterogenität von Netzen, Knoten, Endsystemen und Anwendungen macht neue Methoden und Vorgehen zur Erkennung von Angriffen notwendig. Durch neue Netzstrukturen, Protokolle und Technologien entstehen neue Bedrohungen. Diese können nur durch ein geändertes Vorgehen bei der Erkennung von Angriffen behandelt werden. Visionen wie die des ubiquitären *Future Internet* [CPBD⁺05] machen ein Umdenken im Bereich der Angriffserkennung notwendig und erfordern Angriffserkennungssysteme, welche leicht auf geänderte Netzstrukturen und Mechanismen angepasst werden können.

1.1 Zielsetzung der Arbeit

Die Bekämpfung von Denial-of-Service-Angriffen ist durch koordinierte verteilte Systeme möglich, welche kollaborativ den Angriffsstrom identifizieren und effektiv stoppen. Ziel dieser Arbeit ist, ein Framework zu entwickeln, welches die einfache Implementierung eines Anomalie-basiert und verteilt arbeitenden Angriffserkennungssystems ermöglicht.

Um den Einsatz des verteilten Systems im Netzzinneren zu ermöglichen – wo große Mengen an Verkehr beobachtet werden –, müssen die Instanzen des Angriffserkennungssystems auf Vermittlungssystemen betrieben werden. Diese verfügen häufig über begrenzte Ressourcen und dürfen nicht in ihrer eigentlichen Arbeit – der Weiterleitung von Frames – beeinträchtigt werden. Daher muss ein System zur Angriffserkennung sehr robust und ressourcenschonend gebaut sein. Zu diesem Zweck wird ein Granularitäts-basierter Ansatz verfolgt, welcher die dauerhafte Analyse des Netzwerkverkehrs mit minimalen Ressourcen ermöglicht und nur bei Verdacht auf einen Angriff intensivere Analysen durchführt. Das Framework soll die Möglichkeit bieten, verschiedenartigste Methoden zur Angriffserkennung einfach und schnell zu implementieren und die Verteilung des Systems in seiner Organisation und Kollaboration bestmöglich zu unterstützen. Daher müssen wohldefinierte Schnittstellen definiert werden, welche die leichte Erweiterbarkeit sicherstellen. Zur Unterstützung der einfachen Entwicklung unterschiedlicher Analyse-Module wird eine hohe Abstraktion der verwendeten Laufzeitumgebung und des unterliegenden Netzwerks benötigt. Weiterhin muss das System Methoden zur Koordination von im Netz verteilten Instanzen der Angriffserkennung zur Verfügung stellen um eine dezentrale Angriffserkennung zu ermöglichen.

Um das entwickelte System und neue Methoden zur Angriffserkennung effektiv zu

testen, werden verschiedene Laufzeitumgebungen unterstützt wie beispielsweise reale Hardware- sowie Simulationsumgebungen. Durch Unterstützung einer Simulationsumgebung ist es möglich die Angriffserkennung einfach in großen Netzen zu evaluieren.

1.2 Gliederung der Arbeit

Die Arbeit ist wie folgt gegliedert: Kapitel 2 beschreibt die für die Arbeit notwendigen Grundlagen. Zu diesen gehören beispielsweise Sampling- und Filtering-Methoden zur Reduzierung des Verkehrsstroms sowie die Verwendung von dynamischen Bibliotheken. In Kapitel 3 werden die Anforderungen an ein verteilt arbeitendes Anomaliebasiertes Angriffserkennungssystem definiert und existierende Lösungsansätze vorgestellt. Die Grundlagen und der Entwurf des Frameworks zur Angriffserkennung werden in Kapitel 4 entwickelt. Es wird die Architektur des Frameworks entwickelt sowie Designentscheidungen verdeutlicht. Details zur Implementierung des in Kapitel 4 vorgestellten Entwurfs werden in Kapitel 5 beschrieben. Hier wird die Architektur umgesetzt und das Zusammenspiel der Komponenten erläutert. Das entworfene und implementierte System wird in Kapitel 6 evaluiert und die Anforderungen aus Kapitel 3 validiert. Eine abschließende Zusammenfassung sowie ein Ausblick auf offene Probleme und zukünftige Erweiterungen wird in Kapitel 7 gegeben.

2. Grundlagen

In diesem Kapitel werden Grundlagen beschrieben, welche für das Verständnis der weiteren Arbeit notwendig sind. In Abschnitt 2.1 werden Begriffe definiert, welche in der weiteren Arbeit verwendet werden. Abschnitt 2.2 beschäftigt sich mit Sampling- und Filtering-Methoden und erklärt ihre Notwendigkeit und Umsetzbarkeit. Da Methoden zur Angriffserkennung sowie Sampling- und Filtering-Methoden teilweise Zufallszahlen benötigen, werden diese in Abschnitt 2.3 behandelt und in diesem Zusammenhang die Boost Random Number Library [Maur00] erklärt. In Abschnitt 2.7 wird der Netzwerksimulator OMNeT++ [Varg01] vorgestellt um die Grundlagen für dessen spätere Verwendung zu geben. Die Anforderungen an Protokoll-Parser sowie deren Verwendung wird in Abschnitt 2.5 betrachtet. Die beiden Abschnitte 2.6 und 2.4 beschreiben abschließend die Verwendung von dynamischen Bibliotheken sowie das Pcap-System [Tcpd00].

2.1 Terminologie

Im Folgenden werden die Begriffe *Frame*, *Paket*, *Strom* und *Modul* definiert. Die Verwendung dieser Begriffe in der Literatur ist nicht einheitlich. Daher wird hier die in dieser Arbeit durchgängig verwendete Definition der Begriffe gegeben. Die hier verwendete Definition lehnt sich an Kurose und Ross [KuRo04] an, beinhaltet jedoch kleine Erweiterungen. Die Definitionen sind aus Sicht des Angriffserkennungssystems gegeben, nicht aus Sicht des ISO/OSI-Schichtenmodells.

- *Paket*: Ein Paket stellt den Header eines Protokolls dar. Beispiele dafür sind ein TCP-Paket oder ein IP-Paket. Diese definieren den Protokoll-Header des jeweiligen Protokolls. Ist die Struktur der Daten nicht bekannt, handelt es sich um ein *Payload*-Paket.
- *Frame*: Ein Frame stellt eine Hülle für Pakete dar. Er beinhaltet Metainformationen wie zum Beispiel einen Zeitstempel, welcher den Ankunftszeitpunkt der Daten angibt. Ein Frame selbst beinhaltet keine eigenen Protokoll-Header.
- *Strom*: Ein Frame-Strom stellt eine kontinuierliche Abfolge von Frames dar. Entsprechend ist ein Paket-Strom die Abfolge von Paketen.

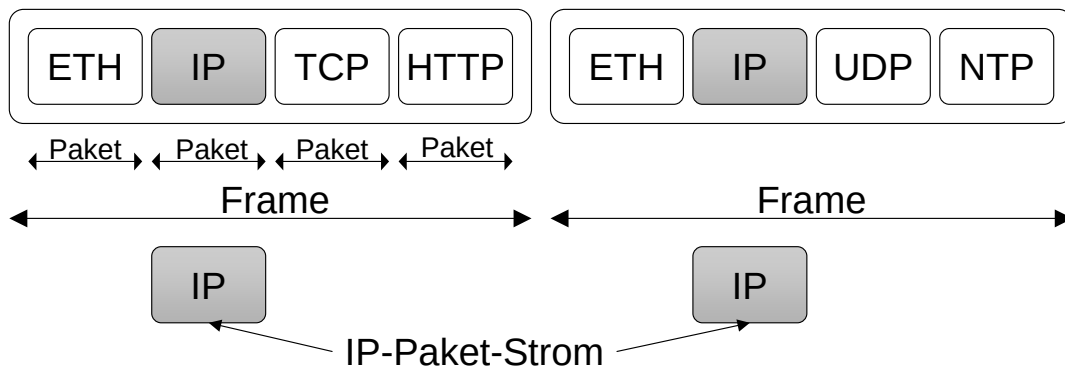


Abbildung 2.1: Terminologie für Frames, Pakete und Ströme

- *Modul*: Ein Modul stellt einen funktionellen Baustein dar, welcher Funktionalität kapselt. Hierbei kann es sich beispielsweise um einen Baustein handeln, welches Frames betrachtet, statistische Daten sammelt oder Sampling-Methoden anwendet.

Zur Verdeutlichung der Terminologie sind in Abbildung 2.1 zwei Frames dargestellt. Diese beinhalten Pakete verschiedener Protokolle, welche den jeweiligen Protokoll-Header implementieren. Die explizit gezeigten IP-Pakete stellen einen IP-Strom dar.

2.2 Sampling und Filtering

Systeme im Netzzinneren müssen – auf Grund des hohen Verkehrsaufkommens – sehr schnell arbeiten. Im Backbone-Bereich werden Geschwindigkeiten von mehreren GBit/s erreicht [Exch07]. Die für diesen Zweck eingesetzten Router sind hochspezialisierte Hardware-Appliances, welche einen Großteil ihrer Arbeit in programmierbare Bausteine auslagern. Um in solchen Hochgeschwindigkeitsnetzen Frame-Ströme zu überwachen und Daten zu sammeln, werden Sampling- und Filtering-Methoden angewendet [ZMDN⁺07].

Abbildung 2.2 zeigt beispielhaft die Anwendung von Sampling bzw. Filtering auf einem Frame-Strom. Das verwendete Sampling-Verfahren selektiert je zwei von drei Frames aus dem Strom. Durch das Filtering-Verfahren werden nur Frames selektiert, welche ein TCP-Paket enthalten.

Bei Anwendung einer Sampling-Methode auf einen Frame-Strom wird nicht jeder Frame zur Inspektion an höher liegende Module weitergegeben, so dass sich die Last auf das Gesamtsystem verringert. Dadurch können weit höhere Datenraten überwacht werden als es ohne Anwendung von Sampling-Methoden möglich wäre. Die Entscheidung, welche Frames zur Verarbeitung weitergegeben werden, wird anhand von statistischen oder zeitlichen Entscheidungen getroffen. Da aus gesampelten Daten gewonnene Informationen nur auf einem Teil der originalen Daten gewonnen werden, sind diese unscharf. Es entsteht somit ein Fehler zwischen den Informationen, welche aus originalen und gesampelten Daten gewonnen werden. Daher muss

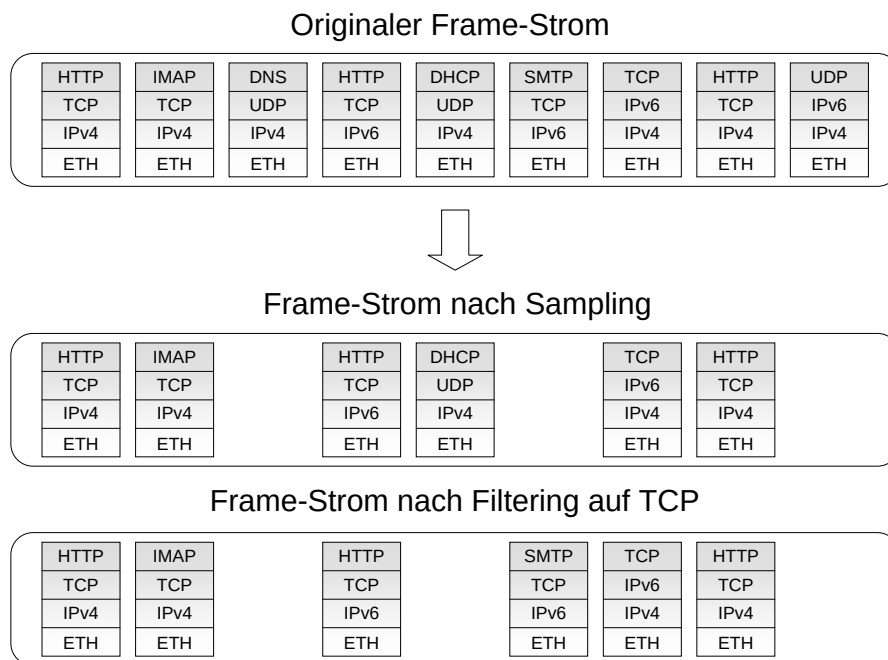


Abbildung 2.2: Sampling und Filtering auf einem Frame-Strom

die Sampling-Methode je nach Anwendungsfall so gewählt werden, dass der entstehende Fehler minimal ist.

Im Gegensatz zu Sampling-Methoden arbeiten Filter-Methoden auf spezifischen, in Frames vorhandenen Daten. Ein Beispiel für eine Filter-Methode wäre ein Filter, welcher nur Frames zur Inspektion weiterleitet, welche ein TCP-Paket enthalten. Im Gegensatz zu Sampling-Methoden wird hier also der Inhalt des Frames betrachtet und zur Entscheidung herangezogen. Filtering-Methoden können somit verwendet werden um Frames mit einer bestimmten Eigenschaft auszuwählen.

In Abschnitt 5.7.2 werden konkrete Sampling- und Filtering-Verfahren sowie deren Implementierung im Rahmen des Frameworks vorgestellt.

2.3 Zufallszahlen

Die Qualität von Pseudozufallszahlen spielt eine essentielle Rolle bei Mechanismen, welche auf Zufallszahlen angewiesen sind. Dies ist beispielsweise bei manchen Arten von Sampling-Verfahren der Fall.

Die C/C++-Standardbibliothek stellt die folgenden Funktionen zur Erzeugung von Pseudozufallszahlen zur Verfügung:

```
void srand (unsigned int seed);
int  rand  (void);
```

Die `srand`-Funktion wird verwendet, um den Pseudozufallszahlen-Generator zu initialisieren. Mit der `rand`-Funktion wird eine Pseudozufallszahl generiert. Die Implementierung dieser beiden Funktionen – welche ausschlaggebend für die Qualität der

Zufallszahlen ist – ist nicht im C/C++-Standard definiert und daher abhängig vom Hersteller des verwendeten Übersetzers und des Betriebssystems. Meist basiert die Implementierung auf Linear-Feedback-Shift-Registers (LFSR), welche von niedriger Qualität sind [CHKM⁺06]. Die Angreifbarkeit des Pseudozufallszahlen-Generators im Linux Betriebssystem wurde von Gutterman bereits nachgewiesen [Gutt06]. Qualitativ hochwertige Pseudozufallszahlen-Generatoren sind meist sehr schwer zu entwickeln und zu implementieren. Dieser Zustand wurde bereits 1988 von Park und Miller erkannt [PaMi88].

Um einheitlich hochqualitative Pseudozufallszahlen auf unterschiedlichen Plattformen zu gewährleisten und eine aufwändige und fehlerträchtige Eigenentwicklung zu vermeiden, wird die Boost Random Number Library [Maur00] verwendet. Sie ist komponentenweise aufgebaut und besteht aus:

- Generatoren für Pseudozufallszahlen
- Verteilungen für Pseudozufallszahlen
- Front-End-Generator zur Erzeugung von Pseudozufallszahlen mit gewähltem Generator und Verteilung

Als Generatoren für Pseudozufallszahlen stehen die Vorlagen Linear-Congruential, Additive-Combine, Inversive-Congruential, Shuffle-Output, Mersenne-Twister und Lagged-Fibonacci zur Verfügung. Sie basieren jeweils auf einem mathematischen Konzept zur Erstellung von Pseudozufallszahlen, wobei freie Parameter noch zu belegen sind. Im Gegensatz zu dem in der C/C++-Standardbibliothek verfügbaren Pseudozufallszahlen-Generator, welcher – wie oben bereits erwähnt – von niedriger Qualität ist, bieten die hier verfügbaren Pseudozufallszahlen-Generatoren hochqualitative und sehr flexible Implementierungen.

Die Boost Random Number Library bietet konkrete Implementierungen der oben genannten Vorlagen wie beispielsweise den Mersenne-Twister MT19937 [MaNi98]. Um die verfügbaren Implementierungen der Pseudozufallszahlen auf eine bestimmte statistische Verteilung abzubilden, existieren in der Boost Random Number Library Verteilungen. Die verfügbaren Verteilungen sind in Tabelle 2.1 aufgeführt. Über eine Verteilung wird die Ausgabe des Pseudozufallszahlen-Generators in einen bestimmten Zahlenbereich abgebildet. Somit kann der Pseudozufallszahlen-Generator in Bezug auf Geschwindigkeit und Qualität gewählt werden und mit einer für den Anwendungsfall angemessenen Verteilung verwendet werden. Im Gegensatz zum Modulo-Operator – welcher normalerweise in Zusammenhang mit der oben erwähnten `rand`-Funktion verwendet wird um Pseudozufallszahlen auf einen Zahlenbereich abzubilden – können hier nicht nur Zahlenbereiche, sondern auch statistische Verteilungen als Zielbereich verwendet werden. Dies erhöht die Flexibilität und ermöglicht die Verwendung in einem breiten Anwendungsfeld.

Abbildung 2.3 gibt einen Überblick der verfügbaren Pseudozufallszahlen-Generatoren. Der innere Kreis stellt dabei die allgemeinen Vorlagen dar. Elemente des äußeren Kreises sind konkrete Implementierungen der jeweiligen Vorlage, welche in der Boost Random Number Library zur Verfügung stehen.

Vorlage	Implementierung
Bernoulli-Verteilung	<code>bernoulli_distribution</code>
Cauchy-Verteilung	<code>cauchy_distribution</code>
Exponentialverteilung	<code>exponential_distribution</code>
Geometrisch Verteilung	<code>geometric_distribution</code>
Normalverteilung	<code>normal_distribution</code> <code>lognormal_distribution</code> <code>uniform_on_sphere</code>
Dreiecksverteilung	<code>triangle_distribution</code>
Gleichverteilung	<code>uniform_smallint</code> <code>uniform_int</code> <code>uniform_01</code> <code>uniform_real</code>

Tabelle 2.1: Verfügbare Zufallsverteilungen

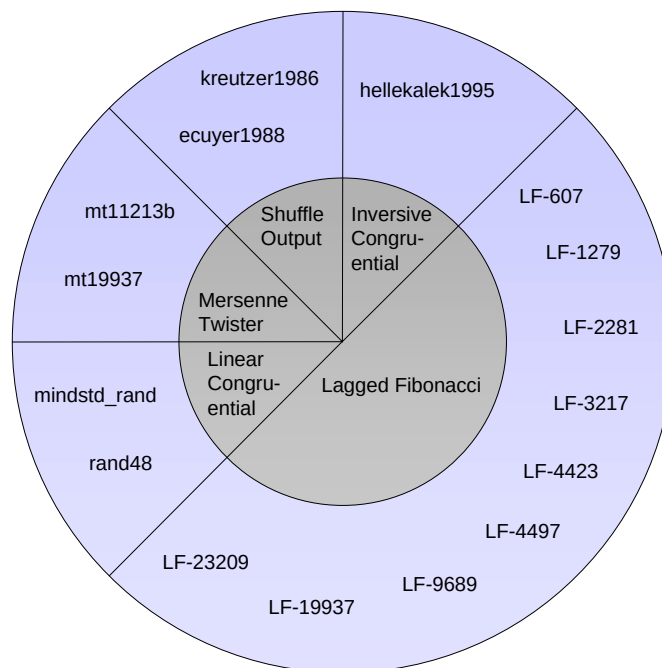


Abbildung 2.3: Übersicht von Pseudozufallszahlen-Generatoren in der Boost Random Number Library

Um einen gewählten Pseudozufallszahlen-Generator mit einer gewählten Verteilung zu verwenden und Pseudozufallszahlen aus der entsprechenden Verteilung zu erzeugen, wird das `variate_generator`-Template verwendet. Dieses wird mit dem konkreten Pseudozufallszahlen-Generator und der Verteilung einmalig initialisiert. Durch Aufruf des überladenen `()`-Operators werden Pseudozufallszahlen aus der entsprechenden Verteilung erzeugt.

Die Verwendung der Boost Random Number Library soll anhand des folgenden Beispiel verdeutlicht werden. Zuerst werden der besseren Lesbarkeit halber neue Typen definiert, welche die gewählte Klasse des Pseudozufallszahlen-Generators, der Verteilung und des konkreten Zahlen-Generators repräsentieren. Als Pseudozufallszahlen-Generator wird MT19937 [MaNi98] aus der Klasse der Mersenne-Twister Generatoren verwendet. Dies ist ein sehr schneller Pseudozufallszahlen-Generator, welcher Pseudozufallszahlen von hoher Qualität erzeugt [CHKM⁺06].

```
typedef boost::uniform_int<>          UNIFORM_DISTRIBUTION;
typedef boost::mt19937                RANDOM_GENERATOR;
typedef boost::variate_generator
    <RANDOM_GENERATOR, UNIFORM_DISTRIBUTION> VARIATE_GENERATOR;
```

Anhand dieser neu definierten Typen werden entsprechend Objekte erstellt, welche die benötigte Funktionalität bereitstellen. Es wird hier eine Gleichverteilung aus dem Intervall von Null bis Zehn verwendet.

```
UNIFORM_DISTRIBUTION uniform (0, 10);
RANDOM_GENERATOR      randomGenerator;
VARIATE_GENERATOR    variateGenerator(m_randomGenerator, m_uniform);
```

Um nun konkrete Pseudozufallszahlen zu erzeugen wird der `()`-Operator verwendet:

```
unsigned int value = variateGenerator();
```

Wie gezeigt wurde, bietet die Boost Random Number Library eine Vielzahl an Generatoren für Pseudozufallszahlen sowie statistische Verteilungen zur Abbildung der Pseudozufallszahlen in einen definierten Zahlenbereich an. Diese können sehr flexibel eingesetzt werden. Daher wird die Boost Random Number Library in der vorliegenden Arbeit zur Erzeugung von Pseudozufallszahlen verwendet.

2.4 Pcap-System

Das Pcap-System [Tcpd00] ist ein User-Mode-Interface, welches das Senden und Empfangen von rohen Netzwerk-Frames der Sicherungsschicht erlaubt. Auf UNIX-Systemen wird die Pcap-Bibliothek, welche das Interface zu Pcap bereitstellt, Libpcap genannt und ist durch den Berkeley-Packet-Filter [McJa93] implementiert. Die Portierung des Pcap-Systems auf Windows ist unter dem Namen Winpcap bekannt [Degi00].

Abbildung 2.4 zeigt den Aufbau des Pcap-Systems unter Windows und Linux. Intern

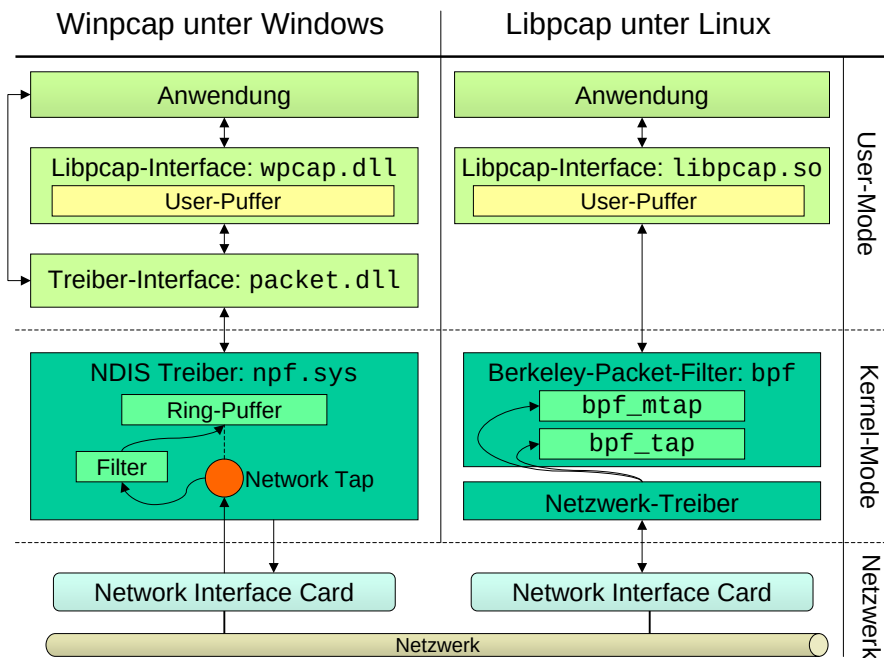


Abbildung 2.4: Architektur des Pcap-Systems unter Windows und Linux

unterscheiden sich die Implementierungen von Libpcap und Winpcap grundsätzlich, da sich das Treiber-Modell von Linux sehr von dem NDIS-basierten Treiber-Modell für Netzwerkgeräte unter Windows unterscheidet.

Unter Windows stellt Winpcap eine API zum NDIS-Treiber `npf.sys` zur Verfügung, welche in der Datei `packet.dll` implementiert ist. Weiterhin existiert eine Schnittstelle, welche mit der des Libpcap-Systems identisch ist. Sie stellt somit einen Wrapper dar, welcher in der Datei `wpcap.dll` implementiert ist und über das Interface in `packet.dll` operiert. Somit ist das eigentliche API – welches als User-Mode Interface dient – bei Libpcap und Winpcap identisch.

Die Implementierung des Pcap-Systems unter Linux ist durch den Berkeley-Packet-Filter realisiert. Dieser implementiert die folgenden beiden Funktionen:

```
void bpf_mtap (caddr_t arg, struct mbuf* m);
void bpf_tap (caddr_t arg, u_char* pkt, u_int pktlen);
```

Diese werden von dem Netzwerk-Treiber aufgerufen um Frames an den Berkeley-Packet-Filter weiterzureichen. Die `libpcap.so` Bibliothek liest die Frames aus dem Treiber aus und stellt sie der Anwendung zur Verfügung.

Im Pcap-System stehen drei Gruppen von Funktionen zur Verfügung: Erstens bietet Pcap Funktionalität um die im System vorhandenen Netzwerk-Interfaces anzusteuern und deren Eigenschaften abzufragen. Die zweite Gruppe ermöglicht das Lesen von Netzwerk-Frames, welche vom Netzwerk-Interface beobachtet werden. Das Schreiben von rohen Netzwerk-Frames bildet die dritte Gruppe. Die wichtigsten Typen und Funktionen des Pcap-Systems sind in Tabelle 2.2 aufgelistet.

Gruppe	Typ/Funktion	Beschreibung
1	<code>pcap_t</code>	Definiert einen Typ, welcher ein Netzwerk-Interface darstellt.
	<code>pcap_findalldevs</code>	Bietet eine Schnittstelle um alle im System verfügbaren Netzwerk-Interfaces zu ermitteln.
	<code>pcap_open_live</code>	Öffnet ein reales Netzwerk-Interface im Online-Modus.
	<code>pcap_open_offline</code>	Öffnet eine Tracedatei im Offline-Modus.
	<code>pcap_datalink</code>	Bietet Zugriff auf Informationen über den Typ der Sicherungsschicht, welche das Netzwerk-Interface verwendet.
	<code>pcap_close</code>	Schließt ein Netzwerk-Interface, welches durch <code>pcap_open_live</code> oder <code>pcap_open_offline</code> geöffnet wurde.
2	<code>pcap_next_ex</code>	Liest einen Netzwerk-Frame von der Netzwerk-Karte, welche mit <code>pcap_open_live</code> oder <code>pcap_open_offline</code> geöffnet wurde.
3	<code>pcap_write</code>	Schreibt einen rohen Netzwerk-Frame auf die Netzwerk-Karte, welche mit <code>pcap_open_live</code> oder <code>pcap_open_offline</code> geöffnet wurde.

Tabelle 2.2: Die wichtigsten Typen und Funktion des Pcap-Systems

Mit dem Pcap-System ist eine einheitliche Entwicklung einer Frames-Quelle für UNIX- und Windows-Systeme möglich. Es bietet somit ideale Möglichkeiten eine Frame-Quelle plattformübergreifend zu implementieren und wird daher in der vorliegenden Arbeit verwendet.

2.5 Protokoll-Parser

Die von einer Netzwerkabstraktion – wie zum Beispiel das in Abschnitt 2.4 vorgestellte Pcap-System – empfangenen Frames enthalten Informationen über die Größe der Daten und meist auch den Zeitpunkt, zu dem der Frame gelesen wurde. Um die im Frame gekapselten Pakete zu lesen – und somit die darin enthaltenen Informationen verfügbar zu machen – werden Protokoll-Parser benötigt.

Die Protokoll-Parser, die in dieser Arbeit verwendet werden, basieren auf [Maye06]. Sie folgen einem Objekt-orientierten Ansatz und wurden in der vorliegenden Arbeit erweitert und verbessert um den Anforderungen zu entsprechen. Das Parsen der Protokolle geschieht entsprechend dem ISO/OSI-Schichtenmodell, wobei mit der untersten Schicht begonnen wird. Für jedes unterstützte Protokoll existiert ein Protokoll-Parser. Dieser ist einmal für das Parsen des Protokolls verantwortlich und hat weiterhin die Aufgabe die gefundenen Protokoll-Attribute leicht verfügbar zu machen.

Auf Grund der losen Kopplung der einzelnen Protokoll-Parser sind beliebige Kapselungen von Protokollen möglich. Dieser Ansatz erlaubt es zum Beispiel getunnelte Protokolle wie IP-in-IP [Simp95] oder EtherIp [HoHo02] transparent zu unterstützen und damit eine hohe Flexibilität zu gewährleisten. Der Aufbau der Protokoll-Parser

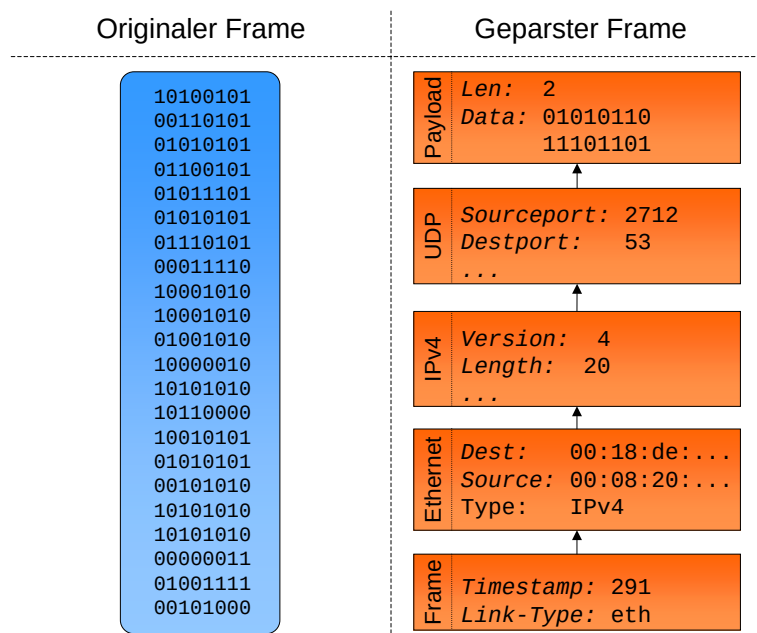


Abbildung 2.5: Parsen eines Frame durch die Protokoll-Parser

orientiert sich an den in Abschnitt 2.1 gegebenen Definitionen.

Abbildung 2.5 zeigt die Struktur eines geparsen Frames. Protokoll-Klassen bieten dabei Zugriff auf Protokoll-Attribute. Die Protokoll-Parser sind portierbar und besitzen eine sehr gute Performance, die durch die Verwendung der Boost Pool Library [Clea00] unterstützt wird. Sie können dadurch in verschiedenen – auch ressourcenbeschränkten – Laufzeitumgebungen verwendet werden. Weiterhin wurden die Protokoll-Parser erfolgreich mit dem ISIC-Testprogramm [Fran04] validiert.

2.6 Dynamische Bibliotheken

Moderne Betriebssysteme bieten die Möglichkeit Funktionalität in dynamisch ladbare Bibliotheken auszulagern. Dies hat unter anderem folgende Vorteile:

- Bessere Strukturierung von zusammengehöriger Funktionalität in eine gemeinsame Bibliothek
- Einfacher Austausch von Funktionalität
- Gemeinsame Verwendung von Funktionalität durch verschiedene Applikationen
- Laden und Entladen zur Laufzeit, führt zu Einsparung von Ressourcen

Obwohl das Konzept der dynamischen Bibliotheken einheitlich ist, unterscheiden sich die Betriebssysteme in der Implementierung und Verwendung von dynamischen Bibliotheken. Es werden im Folgenden die Betriebssysteme Windows und Linux betrachtet.

Dynamische Bibliotheken werden in Linux als *Shared Libraries* bezeichnet. Kennzeichnend für diese ist, dass sie grundsätzlich nicht alleine lauffähig sind. Dies bedeutet, dass es sich hier um passive Einheiten handelt, welche als Archiv für Funktionalität verwendet werden. Da Shared Libraries nicht alleine lauffähig sind, dürfen sie unaufgelöste Symbole beinhalten. Dies bedeutet, dass zur Kompilierzeit nicht alle Implementierungen in Form von Objektdateien verfügbar sein müssen. Hier genügt die Angabe der Definition von in der Shared Library verwendeter Funktionalität durch Schnittstellendefinitionen.

Im Vergleich zur Implementierung von dynamischen Bibliotheken unter Linux unterscheidet sich die Implementierung der *Dynamic Link Libraries* (DLL) unter Windows erheblich. Dynamic Link Libraries sind eigenständige Einheiten. Dies setzt voraus, dass alle Funktionalität, welche die DLL verwendet, zur Kompilierzeit entweder eingebunden sein oder ihr Ort in Form einer anderen Bibliothek angegeben werden muss. Dynamic Link Libraries unter Windows sind aktive Einheiten. Sie können einerseits als Archiv für Funktionalität verwendet werden, andererseits können sie selbst eine `DllMain`-Methode implementieren und somit als aktive Einheiten agieren [Isem00]. Diese wird automatisch von der Laufzeitumgebung aufgerufen sobald die DLL durch die `LoadLibrary`-Funktion geladen bzw. durch die `FreeLibrary`-Funktion entladen wird.

Jede Applikation besitzt in ihrem Prozessraum eine Menge an geladenen Symbolen. Diese definieren den Speicherort verfügbarer Funktionalität. Ist das Symbol für eine Funktion geladen, so kann diese Funktion verwendet werden. Gleiches gilt für Klassen und Typen. Wird eine Shared Library unter Linux geladen, so müssen unaufgelösten Symbole der Shared Library aufgelöst werden. Das Auflösen der Symbole geschieht innerhalb des eigenen Prozessraums der Anwendung. Dies bedeutet, dass Symbole dadurch aufgelöst werden, dass die Menge an bereits geladenen Symbolen durchsucht wird. Abbildung 2.6 verdeutlicht das Vorgehen des Auflöserns von Symbolen beispielhaft unter Linux: die Datei `Foo.cpp` und `Foo.h` werden zu der Anwendung `Foo` kompiliert. Ebenso werden die Dateien `Bar.cpp` und `Bar.h` zu einer Shared Library kompiliert. Die Datei `Bar.cpp` verwendet eine Funktion `func2` der Klasse `Foo`. Da nur die Datei `Bar.cpp` in die `Bar.so` Bibliothek kompiliert wurde, beinhaltet die `Bar.so` Shared Library ein unaufgelöstes Symbol auf `Foo::func2`. Das Kompilieren selbst ist erfolgreich, da die Datei `Foo.h` die Funktionsdeklaration von `Foo::func` beinhaltet und von `Bar.cpp` eingebunden wird. Die Implementierung selbst ist jedoch in der Shared Library nicht vorhanden. Zur Laufzeit wird erst die `Foo`-Anwendung selbst gestartet. Die in der Anwendung verfügbaren Symbole – unter anderem die Funktion `Foo::func2` – sind nun im Prozessraum bekannt. Wird nun die Shared Library `Bar.so` geladen, so kann das unaufgelöste Symbol auf `Foo::func2` aufgelöst und die Funktion durch die `Bar.so` Shared Library verwendet werden. Falls nicht alle in der Shared Library enthaltenen unaufgelösten Symbole aufgelöst werden können, schlägt das Laden der Bibliothek fehl.

2.7 Netzwerk-Simulation

An dieser Stelle wird eine kurze Einführung in den Netzwerk-Simulator OMNeT++ [Varg01] gegeben. OMNeT++ steht für *Objective Modular Network*

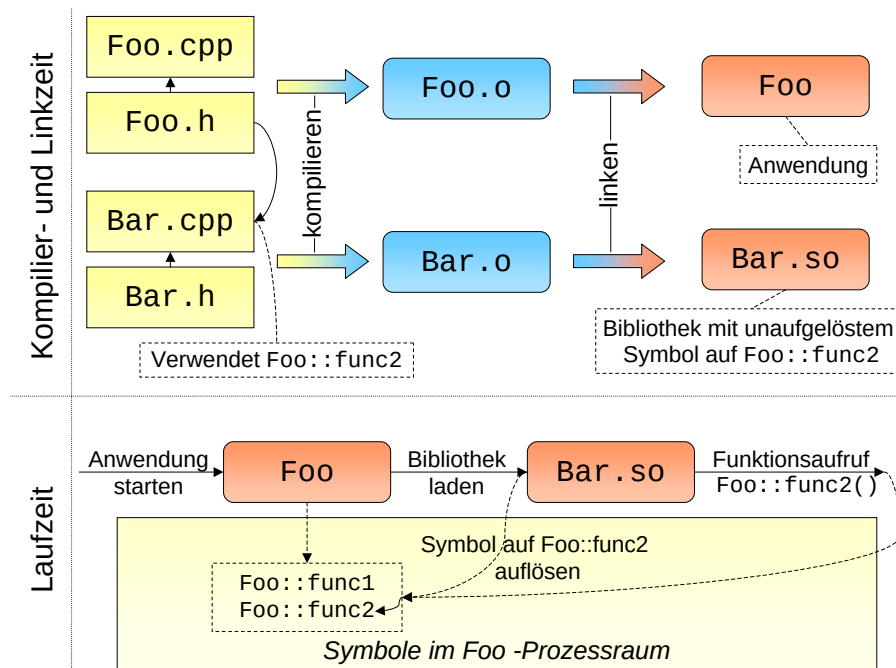


Abbildung 2.6: Dynamische Bibliotheken und Symbolauflösung unter Linux

Testbed in C++ und ist ein ereignisdiskreter Simulator für Netzwerke. Der Anwendungsbereich von OMNeT++ erstreckt sich dabei über alle Bereiche von verschalteten Systemen, welche Nachrichten untereinander austauschen wie beispielsweise Computer-Netzwerke oder Multiprozessor-Systeme.

Das OMNeT++-System besteht aus drei Komponenten:

- **Kernel:** Der Simulationskernel ist für das Ausführen von Simulationen verantwortlich und implementiert die Logik um erstellte Topologien zu simulieren.
- **NED:** Die *Network Description* (NED) ist eine Sprache zur Beschreibung der Topologie, auf welcher die Simulation ausgeführt werden soll. Zu dieser Komponente gehören die NED-Sprache selbst sowie der grafische Editor GNED und der Übersetzer NEDC.
- **Simulationsumgebung:** Zur Ausführung von Simulationen existieren zwei Interfaces: die grafischen Benutzeroberfläche Tkenv und das Kommandozeilen-Interface Cmdenv. Weiterhin beinhaltet die Simulationsumgebung die Programme Plove und Scalars zur Visualisierung von Simulationsergebnissen.

OMNeT++ basiert auf einer hierarchischen Modellierung. Diese ist in Abbildung 2.7 verdeutlicht. Ein Modell besteht aus einer Menge an Modulen, welche über Gates verbunden sind. Module können dabei ineinander verschachtelt sein um beispielsweise einen Netzwerk-Knoten mit einem hierarchisch aufgebauten Netzwerkprotokoll-Stack zu implementieren. Die kleinstmögliche Ebene stellt das *Simple Module* dar. Es kann in C++ programmiert werden und ist die eigentliche aktive Einheit. Beispielsweise kann der TCP-Zustandsautomat als Simple Module programmiert werden. Mehrere Simple Modules können zu einem *Compound Module* zusammengesetzt werden. Das Compound Modul kann beispielsweise einen Knoten im Netzwerk

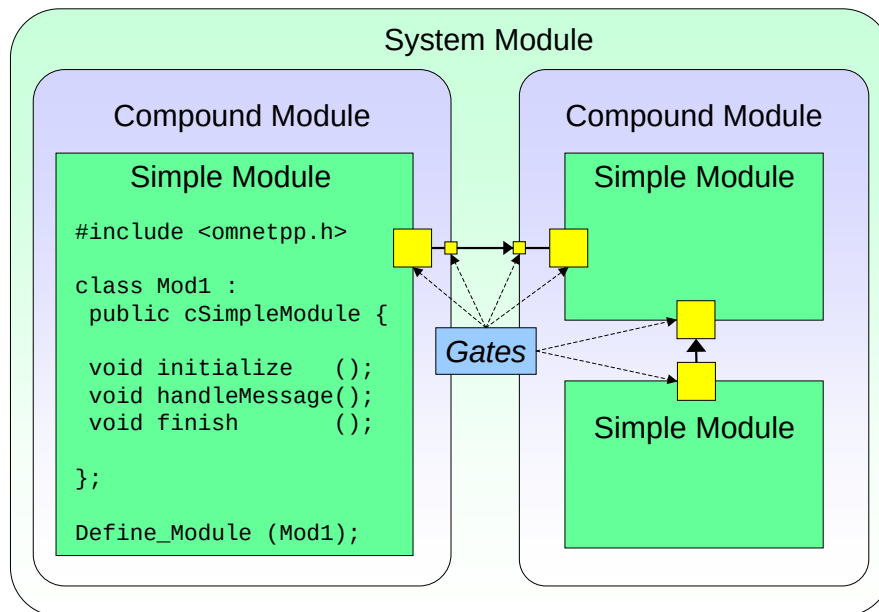


Abbildung 2.7: Hierarchische Modellierung in OMNeT++

darstellen. Die Zusammensetzung mehrerer Compound Modules ergibt ein *System Module*. Es stellt das Netzwerk aus Knoten dar.

Der Transport von Nachrichten wird in OMNeT++ über sogenannte *Gates* realisiert. Zwischen Simple Modules und Compound Modules können Gates definiert werden um Nachrichten zu übertragen. Verbindungen zwischen Gates können Verzögerungen sowie Fehlerwahrscheinlichkeiten zugeordnet werden. Nachrichten stellen die eigentliche Kommunikation des Simulationssystems dar und werden vom Kernel in einem zeitdiskreten Ansatz verarbeitet.

Die Verschachtelung von Modulen ist nicht begrenzt, womit es möglich ist, sehr große und komplexe Netze aufzubauen. Weiterhin spielt die Wiederverwendbarkeit von Modulen in OMNeT++ eine große Rolle. Bereits vorhandene Module können einfach in andere Simulationen eingebunden und dort verwendet werden. Dadurch können zum Beispiel Netzwerkprotokolle entwickelt werden, welche in anderen Simulationen wiederverwendet werden können.

Die NED-Beschreibung eines Simple-Modules besitzt Parameter und Gates. Parameter werden zur Konfiguration des Simple Modules verwendet und können direkt von der C++-Implementierung des Simple Modules verwendet werden. Gates beschreiben die Ein-/Ausgänge des SimpleModules und definieren die Verschaltung mit weiteren Modulen. Um ein Simple-Module in C++ zu programmieren wird eine Klasse erstellt, welche von `cSimpleModule` vererbt wird. Weiterhin müssen die folgenden Funktionen überschrieben werden:

- `initialize`-Funktion: Konfiguration des Moduls
- `handleMessage`-Funktion: Empfang von Nachrichten

- **finish**-Funktion: Beenden des Moduls

Die Simulationsumgebung bietet verschiedene Bibliotheken, welche in einer Simulation verwendet werden können. Zu diesen gehören beispielsweise:

- Generatoren für Zufallszahlen
- Container-Klassen wie zum Beispiel Warteschlangen
- Funktionalitäten zur Erstellung und Bearbeitung von Nachrichten
- Unterstützung von Routing-Funktionalität wie zum Beispiel Graphenfunktionen
- Statistische Funktionalität zur Auswertung von Simulationen

Da die genannten Generatoren für Zufallszahlen nur in einer OMNeT++-Simulation zur Verfügung stehen, werden diese in der vorliegenden Arbeit nicht verwendet. Stattdessen wird die in Abschnitt 2.3 vorgestellte Boost Random Number Library benutzt.

In Abschnitt 6.3.1 wird das in dieser Arbeit entwickelte Framework zur Angriffserkennung in die OMNeT++-Simulationsumgebung portiert. Somit ist es möglich die verteilte Angriffserkennung in großen Netzen zu evaluieren und neue Methoden zur Angriffserkennung zu entwickeln.

3. Analyse

Die ersten Computer-Netzwerke wurden Ende der 60er Jahre in Betrieb genommen [Abba94]. Der Kontext von Netzwerken war damals in erster Linie forschungsorientiert und es bestanden starke Vertrauensverhältnisse zwischen den beteiligten Instanzen, so dass Sicherheit initial nicht notwendig und vorgesehen war.

Die ersten Systeme, welche Ansätze von Netzwerksicherheit implementierten, waren Router der Firma Cisco [Cisc02]. Diese wurden mit Paketfiltern ausgestattet, welche es erlaubten, durch Regel-basierte Beschreibungen Netzwerk-Pakete gezielt zu filtern. Obwohl erste Implementierungen von Paketfiltern bereits 1985 verfügbar waren, dauerte es bis 1987 bis die erste wissenschaftliche Veröffentlichung erschien, welche die Mechanismen von Paketfiltern detailliert untersuchte [MoRA87]. Somit ist die Disziplin der Angriffserkennung in Computernetzen rund 20 Jahre alt. Sie hat in dieser Zeit vier Generationswechsel erlebt [Cisc02, InFo02]. Abbildung 3.1 gibt einen zeitlichen Überblick der Entwicklung von Systemen zur Angriffserkennung.

In den letzten 20 Jahren, in denen Systeme zur Angriffserkennung entwickelt wurden, haben sich Anforderungen und Umgebungen grundsätzlich geändert. Heutige Netzwerke weisen einen sehr hohen Grad an Heterogenität auf und sind sehr viel größer geworden. Durch diese geänderten Umgebungen entstehen grundsätzlich neue Anforderungen an ein System zur Angriffserkennung: Protokolle wie MPLS oder SS7

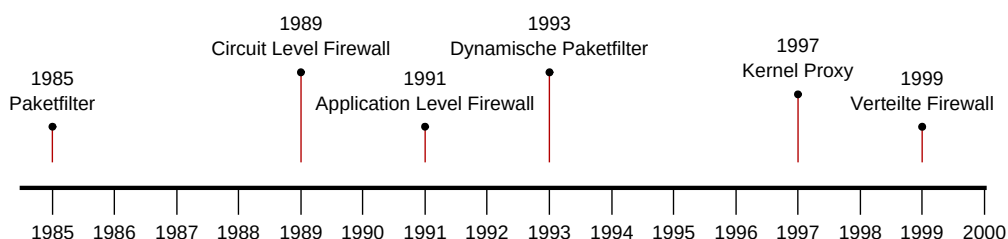


Abbildung 3.1: Zeitliche Entwicklung von Systemen zur Angriffserkennung nach [Cisc02]

müssen beispielsweise gleichermaßen in einem Angriffserkennungssystem überwacht werden können wie es heute für TCP und IP möglich ist[Rey06a]. Diese und weitere Anforderungen werden im folgenden Abschnitt 3.1 genauer definiert. Abschnitt 3.2 zeigt existierende Lösungsansätze mit ihren Vor- und Nachteilen auf. Angriffserkennungssysteme, welche heute in Produktivumgebungen eingesetzt werden, werden in Abschnitt 3.3 analysiert.

3.1 Anforderungen

Die Anforderungen an ein verteilt arbeitendes Framework zur Anomalie-basierten Angriffserkennung sind wie folgt:

- Skalierbarkeit
- Erweiterbarkeit
- Flexibilität
- Robustheit

Die genannten Anforderungen werden in den folgenden Abschnitten näher erläutert und ihre Notwendigkeit verdeutlicht.

3.1.1 Skalierbarkeit

Das System zur Angriffserkennung soll – wie in Kapitel 1 beschrieben – im Netzzinneren betrieben werden, wo sehr viel Verkehr beobachtet wird. Hierdurch können Angriffe schneller als am Rand des Netzes erkannt werden. Somit ist eine frühere Reaktion und ein besserer Schutz des Netzes möglich. Weiterhin kann durch den Betrieb im Netzzinneren ein größerer Netzabschnitt überwacht werden und somit die Anzahl der Angriffserkennungssysteme minimiert werden. Ein entscheidender Nachteil des Betriebs im Netzzinneren ist die hohe Last auf dem Angriffserkennungssystem durch die Menge an Verkehr, die überwacht werden muss. Skalierbarkeit und Performance ist hier eine Notwendigkeit um den Betrieb im Netzzinneren durchführen zu können.

Der Betrieb im Netzzinneren ist – wie oben beschrieben – sehr effektiv und ermöglicht eine schnelle Reaktion auf Angriffe. Weiterhin benötigt er einen geringen administrativen Aufwand, da die Infrastruktur der Vermittlungssysteme bereits existiert und in das Netz eingebunden ist. Somit ergibt sich auch kein zusätzlicher finanzieller Aufwand bezüglich der Beschaffung und Integration von Netzkomponenten. Da Vermittlungssysteme in ihren verfügbaren Ressourcen entsprechend ihrer eigentlichen Aufgaben beschaffen sind, darf eine zusätzliche Angriffserkennung diese nicht übermäßig beanspruchen. Daher muss bei der Entwicklung des Angriffserkennungssystems von Systemen mit begrenzten Ressourcen ausgegangen werden.

Zur Reduzierung der Ressourcenlast auf dem Vermittlungssystem wird in der vorliegenden Arbeit ein Granularitäts-basierter Ansatz verwendet: Die Angriffserkennung

betreibt im Normalzustand eine ressourcenschonende Grundüberwachung des Netzwerkverkehrs. Bei Verdacht auf einen Angriff können neue Methoden der Angriffserkennung geladen werden, welche den vermuteten Angriff näher untersuchen. Die so entstehenden Granularitätsstufen ermöglichen eine verfeinernde Angriffserkennung mit einer beliebigen Anzahl an Stufen.

Die bisher angesprochenen Punkte betreffen die lokale Skalierbarkeit. Da das Framework die verteilte Angriffserkennung ermöglichen soll, werden Mechanismen für die Kommunikation zwischen entfernten Instanzen integriert. Hier muss ein weiterer Punkt betrachtet werden: die verteilte Skalierbarkeit. Diese beschreibt die Skalierbarkeit *im Gesamten*. Sie schließt somit alle Instanzen zur Angriffserkennung sowie insbesondere die Kommunikation zwischen diesen ein. Die Last der verteilten Kommunikation ist sehr von der eigentlichen Realisierung der Angriffserkennung abhängig.

3.1.2 Erweiterbarkeit

Die Anzahl an Protokollen, Mechanismen und Formaten, welche in Netzen verwendet werden, lässt sich grob durch die Anzahl der von der IETF herausgegebenen RFCs bestimmen. Diese stellen zwar nur einen Teil der in Netzen verwendeten Standards dar, haben aber Gewicht und werden daher hier betrachtet. Die Gesamtzahl der veröffentlichten RFCs wächst exponentiell [RFC-06], was die immer stärker werdende Heterogenität und Komplexität von Netzen verdeutlicht. Abbildung 3.2 veranschaulicht das Wachstum der RFC-Veröffentlichungen. Erweiterbarkeit ist daher die zweite Anforderung an ein System zur Angriffserkennung: Neue Protokolle und Mechanismen, welche in Netzen verwendet werden, müssen leicht in das Framework integriert werden können.

Die bisher beschriebene Erweiterbarkeit bezieht sich auf neue Protokolle und Mechanismen, welche in Netzen verwendet werden. Ein weiterer Aspekt von Erweiterbarkeit bezieht sich auf Methoden zur Angriffserkennung. Da das in dieser Arbeit entwickelte Framework zur Angriffserkennung Anomalie-basierte Ansätze verwendet, wird Erweiterbarkeit in Bezug auf Methoden zur Angriffserkennung hier auf Anomalie-basierte Methoden bezogen. Es muss also möglich sein, neue Methoden zur Anomalie-basierten Angriffserkennung einfach und schnell in das Angriffserkennungssystem zu integrieren. Weiterhin muss die Integration neuer Sampling- und Filtering-Methoden möglich sein.

3.1.3 Flexibilität

Um das System zur Angriffserkennung an verschiedene Umgebungen anzupassen und effektiv betreiben zu können, muss es die Eigenschaft der Flexibilität besitzen. Beim Einsatz des Systems sind zum einen verschiedene Laufzeitumgebungen zu beachten. Dies können reale Router-Hardware, Simulatoren oder beispielsweise auch Sensor-knoten sein. Eine generische Abstraktion von der unterliegenden Technologie und Umgebung bietet die Flexibilität, das System in einer Vielzahl von Umgebungen zu betreiben ohne grundlegende Änderungen am System selbst vornehmen zu müssen. Dies betrifft vor allem die Module zur Angriffserkennung: Diese müssen nicht angepasst werden um in einer anderen Laufzeitumgebung verwendet werden zu können.

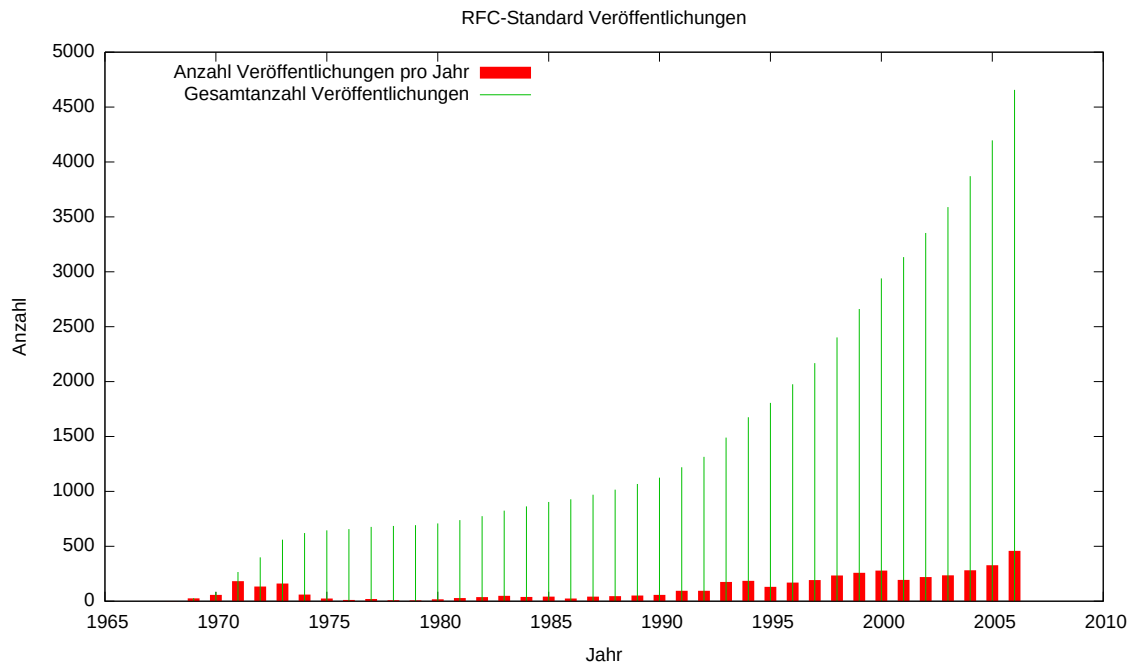


Abbildung 3.2: Anzahl der RFC-Veröffentlichungen pro Jahr und Gesamtzahl

Ein weiterer Punkt, welcher Einfluss auf die Flexibilität hat, ist die Konfigurierbarkeit. Ein hohes Maß an Konfigurierbarkeit bietet ein größeres Anwendungsfeld für den praktischen Einsatz und die Forschung auf dem Gebiet der Angriffserkennung. Es muss ein Grad der Konfigurierbarkeit sichergestellt werden, welcher es Benutzern erlaubt, auf einfache und sichere Weise Änderungen an der Konfiguration des Systems vorzunehmen.

Um den internen Ablauf und die Einsatzmöglichkeiten in Bezug auf die Funktionalität des Systems zu optimieren ist eine hohe Konfigurierbarkeit der Methoden zur Angriffserkennung notwendig. Dadurch kann das System auf verschiedene Netzwerkumgebungen angepasst werden bzw. sich selbständig den dort vorhandenen Bedingungen anpassen. Dies kann idealerweise durch Verwendung von Bausteinen erreicht werden, welche durch eine Konfiguration verschaltet und zur Laufzeit dynamisch ge- und entladen werden können. Durch diese Modularisierung der einzelnen Bausteine ist eine hohe Flexibilität sichergestellt.

3.1.4 Robustheit

Wie bereits in Abschnitt 3.1.1 beschrieben, soll das System zur Angriffserkennung auf Vermittlungssystemen betrieben werden. Diese dürfen in keinem Fall in ihrer eigentlichen Arbeit – der Weiterleitung von Netzwerk-Frames – behindert werden. Um dies zu gewährleisten ist eine gute Skalierbarkeit (siehe Abschnitt 3.1.1) und Performance notwendig. Da ein System zur Angriffserkennung komplexe Funktionalität implementiert, ist es ein potentieller Kandidat für Angriffe und Ausfälle. Daher darf das Angriffserkennungssystem nicht das Auftreten von Stausituationen, beispielsweise durch eine zu langsame Verarbeitung von Frames, fördern. Dies würde den Durchsatz des Vermittlungssystems verringern und könnte somit von einem Angreifer zur Ausführung eines DoS-Angriffs gegen das Vermittlungssystem ausgenutzt

werden. Weiterhin muss das System Schutzmechanismen vor Angriffen besitzen, welche auf das Angriffserkennungssystem selbst gerichtet sind. Dies bedingt vor allem den Schutz vor korrupten Netzwerkframes.

3.2 Existierende Lösungsansätze

Emerald [PoNe97] ist eines der ersten verteilt arbeitenden Angriffserkennungssysteme. Es ist hierarchisch aufgebaut und besteht aus drei Ebenen. Auf der untersten Ebene stehen die Servicemonitore. Sie sammeln Informationen und stellen ihre Daten über Push- und Pull-Mechanismen bereit. Die nächsthöhere Schicht sind die Domain-Monitore, welche auf Domänen-Ebene betrieben werden. Sie sammeln Alarmmeldungen und Daten der Servicemonitore. Die letzte Ebene ist unternehmensweit. Sie übernimmt die Aufgabe die Daten der Domain-Monitore darzustellen. Durch die hierarchische Struktur stellt jede Emerald-Instanz auf jeder Ebene einen Angriffspunkt dar. Beispielsweise existiert für jede Domäne genau ein Domain-Monitor. Ein Ausfalls des Domain-Monitors hat zur Folge, dass diese Domäne nicht mehr überwacht werden kann. Gleiches gilt für eine Überlastsituation der hierarchischen Verbindung. Weiterhin endet die Hierarchie bei der unternehmensweiten Ebene. Es ist somit nicht möglich Instanzen der Angriffserkennung an beliebigen Punkten in mehreren Netzen und auf mehrere Unternehmen oder Autonome Systeme verteilt zu platzieren.

Die CITRA-Umgebung [SHSD⁺01] erlaubt die Integration verschiedenartigster Sicherheitssysteme, mit dem Ziel, Angriffe zur ihrem Ursprung zurück zu verfolgen. Dabei wird eine sogenannte CITRA-Community erstellt, welche aus den Komponenten Detectors, Responders, Audit-Prozesse und Discovery Coordinator besteht. Detectors erkennen Angriffe und melden diese an die Responders sowie an den Discovery Coordinator. Die Aufgabe der Responders ist, Gegenmaßnahmen einzuleiten und den Angriff entsprechend zurück zu verfolgen. Audit-Prozesse speichern Netzwerk-Aktivität und erstellen so eine Historie. Die zentrale Instanz ist der Discovery Coordinator. Er empfängt alle Angriffe sowie die eingeleiteten Gegenmaßnahmen und erstellt eine Strategie zur Einleitung von Gegenmaßnahmen auf globaler Ebene. Somit stellt der Discovery Coordinator eine Instanz dar, dessen Ausfall das System möglicherweise wirkungslos macht, da keine globale Strategie erarbeitet werden kann.

Janakiraman et al. entwickelten das Indra-System [JaWZ03], einen Peer-to-Peer-basierten Ansatz, in welchem unterschiedliche Systeme von Netzkomponenten und Angriffserkennungssystemen über einen Registrierungs-basierten Ansatz Informationen austauschen. Über ein Plugin-System können zusätzliche Komponenten auf einer Instanz zur Laufzeit geladen werden. Zur Implementierung der Plugins wird Java verwendet. Eine Java Laufzeitumgebung ist jedoch nicht zwingend auf einem Vermittlungssystem vorhanden. Es ist fraglich, ob es wünschenswert ist eine Java Laufzeitumgebung auf einem Vermittlungssystem zu betreiben. Somit ist es nicht direkt möglich das Indra-System auf einem Vermittlungssystem zu betreiben. Weiterhin betreibt Indra keine Granularitäts-basierte und ermöglicht so keinen ressourcenschonenden Betrieb.

Frincke und Wilhite schlagen ein auf dem HUMMER-System aufbauendes, hierarchisch strukturiertes System aus leichtgewichtigen Knoten und mobilem Code vor [FrWi01]. Das System basiert auf mobilen Agenten, welche sich aktiv im Netz fortbewegen können. Die Agenten sind dabei in drei Klassen unterteilt: Tool-Agenten, Investigative-Agenten und Abwehr-Agenten. Damit sich Agenten aktiv durch das Netz bewegen können, müssen sehr starke Vertrauensverhältnisse zwischen den beteiligten Instanzen existieren, da aktiv Code auf Fremdgeräten ausgeführt werden soll. Dies ist meist nicht gegeben und schwer zwischen verschiedenen Organisationen und Unternehmen zu etablieren.

Die SHOMAR-Umgebung [UnPN02] stellt ein Framework zur Integration von Angriffserkennung und verteilter Kommunikation bereit und ist damit der hier vorgestellten Arbeit sehr ähnlich. Jedoch liegt das Hauptaugenmerk der SHOMAR-Umgebung auf der Kommunikation zwischen Instanzen. Es wird nicht beschrieben wie SHOMAR die Angriffserkennung selbst unterstützt bzw. vereinfacht. Die Kommunikation zwischen den Instanzen wird über ein XML-basiertes Format realisiert. Hiermit müssen Datenstrukturen in das XML-basierten Format konvertiert und wieder extrahiert werden. Weiterhin stellt eine XML-basiertes Kommunikationsformat zusätzlichen Aufwand bei der Definition neuer Nachrichten dar und enthält Redundanz, da einerseits lokale Datenstrukturen existieren müssen und weiterhin diese in XML abgebildet werden müssen. Die Struktur des Systems kann unter anderem hierarchisch oder als Peer-to-Peer-System aufgebaut sein.

Das COSSACK-System [PLMH⁺03] beschreibt eine Architektur von sogenannten Watchdogs. Diese werden am Rand eines Netzes betrieben. Somit wird an jedem Verbindungspunkt zwischen zwei Netzen ein Watchdog installiert. Die Kommunikation zwischen Watchdog-Instanzen ist über Multicast realisiert. COSSACK stellt keine eigenständige Umgebung bereit. Statt dessen wurde das Angriffserkennungssystem Snort gewählt und die Watchdogs als Snort-Plugins realisiert. Somit stellt das COSSACK-System kein Framework zur Verfügung, mit welchem eine verteilte Angriffserkennung integriert werden kann.

Das STAT-System [ViVK03] besteht aus einer Menge an sogenannten STAT-Komponenten. Die Basis stellt das STAT-Framework dar. Es bietet die Grundlage für die Verarbeitung von Ereignissen sowie die Entwicklung von Sensoren. Neben dem Framework selbst existieren verschiedene Arten von Netz- und Host-basierten Sensoren. Diese unterscheiden sich im unterliegenden Betriebssystem sowie dem Ziel der Datenausgabe. Die STATL-Beschreibungssprache wird verwendet um Angriffsszenarien zu modellieren. Diese können mit einem Übersetzer in lauffähigen Code – sogenannte Szenario-Plugins – transformiert werden, welche die eigentliche Angriffserkennung implementieren. Die Szenario Plugins basieren auf endlichen Automaten. Übergänge zwischen den Zuständen der Automaten werden durch die überwachten Verkehrsströme ausgelöst. Die Autoren nennen die Anomalie-basierte Angriffserkennung als zukünftiges Ziel. Die Zustandhaltung und Verwaltung großer Mengen an Automaten ist möglicherweise sehr ressourcenaufwändig. Leider liefern die Autoren keine ausführlichen Messdaten der Systemlast. Daher ist unklar, ob das STAT-System im Netzzinneren betrieben werden kann.

Dressler et al. beschreiben das CATS-System [Falk04]: autonome Instanzen von

Angriffserkennungssystemen, welche kooperativ arbeiten. Die Kooperation zwischen den Instanzen wird über den Versand von Verkehrsstrom-Statistiken sowie PSAMP-, IPFIX-Daten und Informationen über verdächtige Aggregate erreicht. Die Arbeit gibt leider keine weiterführende Einsicht darüber, wie die Kommunikation zwischen den Instanzen realisiert ist und welche weiteren Informationen versendet werden. Das System bietet Signatur- und Anomalie-basierte Angriffserkennung. Jedoch sind die einzelnen Methoden zur Angriffserkennung nicht auf einen Granularitäts-basierten Ansatz ausgelegt. Weiterhin bietet das CATS-System keine Modularisierung von Funktionalität und ist somit nicht über Bausteine erweiterbar.

Das von Braun und Münz im Rahmen des Diadem Firewall Projekts [Cons07] vorgestellte TOPAS-System [BrMü06] betreibt sogenannte Monitore an verschiedenen Punkten im Netz. Die Monitore wenden dabei durch Filtering- und Sampling-Mechanismen eine Vorverarbeitung des Netzverkehrs an. Nach erfolgter Vorverarbeitung senden sie die beobachteten Verkehrsdaten – in einem der Formate Netflow, IPFIX oder PSAMP – an eine zentrale Instanz. Das System ist modular aufgebaut und erlaubt einen Granularitäts-basierten Ansatz durch die Verwendung von dynamischen Modulen. Das TOPAS-System implementiert keine kooperative Angriffserkennung. Die Monitore sammeln lediglich Verkehrsdaten, welche von einer zentralen Instanz analysiert werden. Diese stellt einen Angriffspunkt dar und macht somit die Angriffserkennung im Falle eines erfolgreichen Angriffs nutzlos.

3.3 Angriffserkennung in Produktiv-Umgebungen

In Produktiv-Systemen werden seit vielen Jahren Systeme zur Angriffserkennung eingesetzt. Die bekanntesten werden im Folgenden kurz vorgestellt. Hierbei wird klar, dass die vorhandenen Systeme nicht den Anforderungen genügen, welche in dieser Arbeit verfolgt werden.

3.3.1 Snort und PreludeIDS

Snort [Roes01] ist das am weitesten verbreitete Angriffserkennungssystem für IP-basierte Netze und erfreut sich einer sehr großen Nutzerzahl. Die Standardversion von Snort ist unter einer Open-Source Lizenz verfügbar. Weiterhin existieren kommerzielle Management-Konsolen für Snort wie zum Beispiel Applied Watch Console der Firma Applied Watch Technologies [Tech07] und Sourcefire Management Console von Sourcefire [Sour07]. Diese können die Angriffs- und Log-Ausgaben mehrerer Snort-Instanzen abfragen und einheitlich darstellen. Um Snort in Netzen mit hoher Bandbreite zu betreiben existieren mehrere Ansatzpunkte: Zum einen bieten kommerzielle Hersteller wie Sourcefire integrierte Hardwarebausteine an, welche einen höheren Durchsatz erreichen. Zum anderen können Systeme zur Lastverteilung verwendet werden um die Verkehrslast zwischen mehreren Snort-Instanzen aufzuteilen [Netw07].

PreludeIDS [Zara03] arbeitet – wie auch Snort – Signatur-basiert. Wo Snort nur als Network Intrusion Detection System (NIDS) verwendet wird, kann PreludeIDS sowohl als NIDS als auch als Host Intrusion Detection System (HIDS) verwendet werden. Somit bietet PreludeIDS mehr Anwendungsmöglichkeiten. Weiterhin ist PreludeIDS von Haus aus für den Einsatz als Sensor/Manager-System ausgelegt und

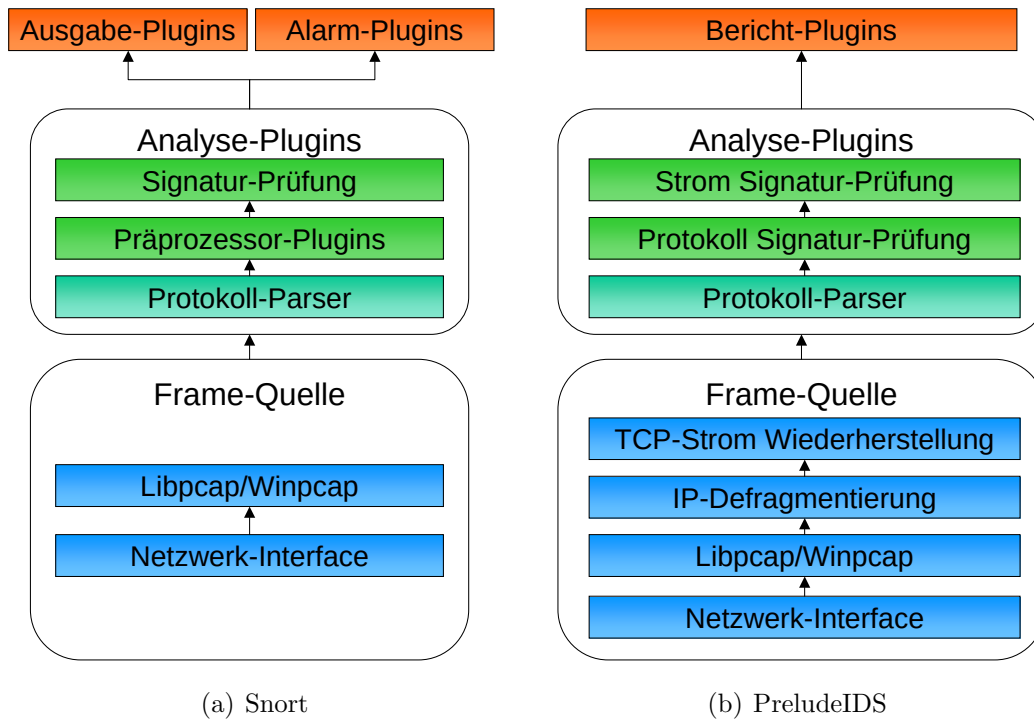


Abbildung 3.3: Architektur von Systemen zur Angriffserkennungssystemen

beinhaltet somit Sensor- und Manager-Instanzen: Mehrere Sensoren melden die von ihnen erkannten Angriffe an eine Manager-Instanz, welche eine einheitliche Darstellung der Daten aller Sensoren ermöglicht. PreludeIDS beinhaltet zwei Sensor-Typen: Prelude-NIDS zur Überwachung des Netzwerks und Prelude-LML zur Überwachung von Hosts. LML steht für Log Monitoring Lackey und bezeichnet die Überwachung von Log-Dateien, ähnlich dem Programm Tripwire [Trip07]. Weiterhin existieren fremde Sensoren, welche mit PreludeIDS zusammenspielen und ihre Meldungen den Manager-Instanzen zur Verfügung stellen. Zu diesen gehören zum Beispiel der Sicherheitsscanner Nessus [Ness07] oder die Dateintegritätsprüfung Samhain [Samh07].

Abbildung 3.3 zeigt die Architekturen von Snort und PreludeIDS. Wie zu sehen ist, sind beide Systeme sehr ähnlich aufgebaut. Die Schnittstelle zum Netzwerk wird in beiden Fällen durch ein Libpcap-Interface hergestellt. Höhere Schichten enthalten Protokoll-Parser und Teile zur Prüfung des Netzwerkverkehrs durch Signaturen. Benutzerinteraktion ist durch Plugins gegeben, welche zum Beispiel eine Email an den Administrator senden oder Alarm-Meldungen grafisch anzeigen. Snort bietet über Präprozessor-Plugins die Möglichkeit Frames zu manipulieren oder zu kennzeichnen. Diese Kennzeichnung wird in höheren Schichten verwendet um spezielle Überwachungen und Tests auf diesen Frames durchzuführen.

Snort und PreludeIDS sind Signatur-basierte Angriffserkennungssysteme mit einem sehr großen Stamm an Signatur-Daten, welche häufig aktualisiert werden. Auf Grund der Signatur-basierten Arbeitsweise ist es nicht sinnvoll Sampling-Mechanismen einzusetzen, da bereits ein einzelnes Paket Schaden anrichten kann. In einem Snort- oder PreludeIDS-System sind grundsätzlich immer alle Signaturen geladen. Dies ist auch sinnvoll, da jede Signatur einen eigenen Angriff darstellt, welcher erkannt bzw. verhindert werden soll. Durch die nicht änderbare Tatsache, dass stets alle Signatu-

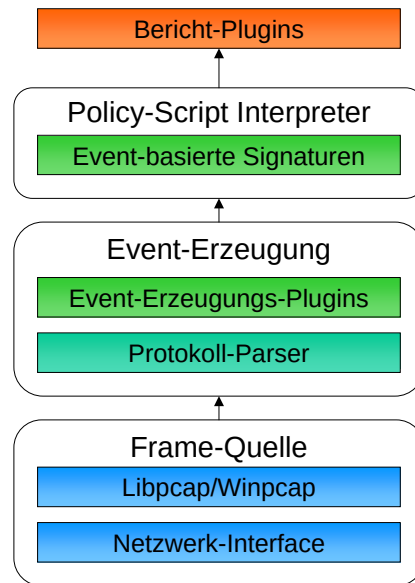


Abbildung 3.4: Architektur des Angriffserkennungssystems Bro

ren geladen sind, entsteht jedoch eine hohe Last auf dem Angriffserkennungssystem. Daher ist ein Betrieb auf einem Vermittlungssystem ohne zusätzliche und teure Hardwareunterstützung ungeeignet ist, da dort geringe Ressourcen vorhanden sind und eine hohe Verkehrslast überwacht werden muss.

Snort und PreludeIDS können als Sensor/Manager-Architektur betrieben werden: Eine Menge an Sensoren liefert aus verschiedenen Netzsegmenten Angriffsmeldungen an eine zentrale Manager-Station, welche von einem Administrator überwacht wird. Beide Systeme ermöglichen jedoch nicht die kollaborative Zusammenarbeit verschiedener Instanzen.

Beide Systeme sind für den Einsatz in realen Umgebungen konzipiert und entwickelt. Die Integration in einen Simulator wurde bei der Entwicklung nicht vorgesehen. Bisher existiert von beiden Systemen keine Version, die in einem Simulator lauffähig ist.

3.3.2 Bro

Das Angriffserkennungssystem Bro [Somm03] verfolgt einen Ereignis-basierten Ansatz und ist vor allem im Forschungsumfeld bekannt. Die Architektur von Bro ist in Abbildung 3.4 gezeigt. Auf Grund von Paketen und deren Eigenschaften werden Ereignisse erzeugt. Diese werden durch die Event-basierten Signaturen analysiert und entsprechend verarbeitet. Im Falle eines Alarms werden die Bericht-Plugins aktiviert um den Benutzer entsprechend zu informieren.

Die Event-basierten Signaturen werden in einer speziellen Sprache geschrieben und stellen Regeln dar, welche die erlaubte Aktivität der Protokolle beschreibt. Der Ereignis-basierte Ansatz beruht darauf, dass Pakete je nach Typ und Status ein bestimmtes Ereignis wie zum Beispiel das Ereignis *Versuch eines Verbindungsaufbaus* erzeugen. Damit ist eine weitere Abstraktion von der Netzwerkschicht gegeben, welche es erleichtert Signaturen in einer höheren Sprache zu verfassen und die Last

auf das System zu reduzieren, da nicht jedes Paket von den Signaturen überprüft werden muss. Jedes Paket muss jedoch geparkt und analysiert werden, um entsprechende Ereignisse zu erstellen.

Auf Grund des Ereignis-basierten Ansatzes ist es nicht möglich ganze Aggregate – wie zum Beispiel alle UDP-Pakete – zu überwachen, da nicht für jedes Paket überhaupt ein Ereignis erzeugt wird. Weiterhin ist es – wie auch bei Snort und PreludeIDS – nicht sinnvoll, Sampling-Mechanismen zu verwenden, da diese die Ereignisse und damit den Status von Verbindungen verfälschen bzw. komplett korrumpieren würden: Die Überwachung des Status einer TCP-Verbindung ist beispielsweise bei Verwendung von Sampling-Mechanismen nicht möglich. Ebenso wie Snort und PreludeIDS wurde Bro bisher nicht in einen Simulator portiert.

4. Entwurf

In diesem Kapitel wird das den Anforderungen in Abschnitt 3.1 entsprechende Framework zur Anomalie-basierten Angriffserkennung *Distack* (Distributed Attack Detection System) entwickelt und dessen Entwurf vorgestellt. In Abschnitt 4.1 wird ein erster Überblick über die entwickelte Architektur gegeben und auf Besonderheiten eingegangen. Die darauffolgenden Abschnitte beschreiben die Architektur detaillierter: Ausgehend von den unteren Komponenten werden die Zusammenhänge zu höher liegenden Komponenten dargestellt und diese aufbauend erläutert. Dabei werden Entwurfsentscheidungen verdeutlicht und der Gesamtaufbau erklärt.

4.1 Architektur

Die Architektur von Distack ist modular aufgebaut und besitzt wohldefinierten Schnittstellen zwischen den einzelnen Komponenten. Abbildung 4.1 zeigt die Architektur des Systems.

Die Architektur ist in sechs Komponenten aufgeteilt:

- *NetworkManager*
- *ModuleManager*
- *RemoteMessaging*
- *Communication*
- *Utilities*
- *Configuration*

NetworkManager

Um eine bestmögliche Abstraktion von dem unterliegenden Netzwerk und der Laufzeitumgebung zu erreichen, existiert die *NetworkManager*-Komponente. Sie bietet eine einheitliche Schnittstelle in Richtung des *ModuleManager*. Somit sorgt sie für

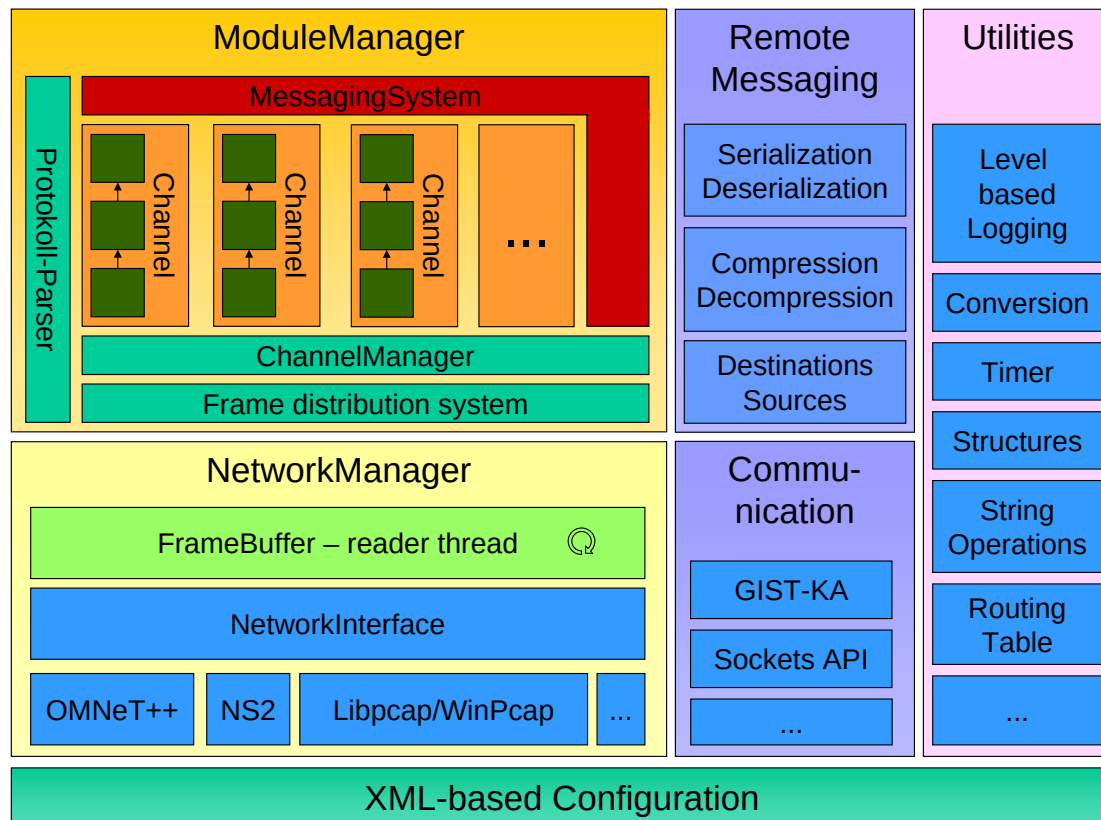


Abbildung 4.1: Architektur von Distack

eine Abstraktion des Netzwerks und der Netzwerk-Hardware, auf welcher das System läuft. Auf diese Weise ist eine leichte Portierbarkeit auf andere Laufzeitumgebungen und Netze sichergestellt. Die durch den *NetworkManager* ermöglichte Abstraktion vom unterliegenden Netzwerk wird in Abschnitt 4.2 detailliert beschrieben.

ModuleManager

Die eigentliche Funktionalität der Angriffserkennung ist im *ModuleManager* integriert. Er bietet die Möglichkeit sogenannte *Channels* zu laden und entladen. Ein *Channel* stellt eine Gruppierung von zusammengehöriger Funktionalität dar, welche in *Modules* implementiert ist. *Modules* wiederum sind kleine funktionelle Einheiten, welche bestimmte Operation auf Frames, Paketen oder Strömen ausführen. Sie implementieren somit die eigentliche Logik zur Angriffserkennung. Durch die Modularisierung von Funktionalität ist die Komponierbarkeit von Bausteinen möglich. Somit ist eine hohe Flexibilität gewährleistet. Um eine Kommunikation zwischen den *Modules* zu ermöglichen, existiert ein datenzentriertes Nachrichtensystem. Dieses wird durch das *MessagingSystem* realisiert. Es erlaubt den lokalen Versand von Nachrichten innerhalb des Angriffserkennungssystems. *ModuleManager* und *MessagingSystem* werden in Abschnitt 4.3 und Abschnitt 4.4 erläutert.

RemoteMessaging

Um den Versand von Nachrichten zwischen entfernten Instanzen zu ermöglichen, existiert die *RemoteMessaging*-Komponente. Das *RemoteMessaging* verarbeitet lokale Nachrichtenobjekte zum Versand und Empfang über Byte-orientierte Kanäle.

Dies beinhaltet die Serialisierung und Deserialisierung von Nachrichten. Der Aufbau des Systems zur verteilten Kommunikation ist in Abschnitt 4.5 dargestellt.

Communication

Serialisierte Nachrichten werden über die *Communication*-Komponente versendet. Diese Komponente ist austauschbar und realisiert den Versand sowie Empfang von Nachrichten. Weiterhin müssen hier die konkreten entfernten Instanzen bestimmt werden an welche die Daten übertragen werden sollen. Die *Communication*-Komponente arbeitet rein Byte-orientiert. Die Implementierung dieser Komponente ist von der verwendeten Laufzeitumgebung abhängig. Hier kann zum Beispiel eine Kommunikation über TCP-Sockets oder über ein Signalisierungsprotokoll – wie beispielsweise GIST [ScHa07b] – realisiert werden. Beispielhafte Protokolle zur Realisierung der *Communication*-Komponente werden in Abschnitt 4.5.3 vorgestellt.

Configuration

Die systemweite Konfiguration wird über eine XML-basierte Konfigurationsdatei gesteuert, welche durch die *Configuration*-Komponente gekapselt ist. Die Konfigurationsdatei muss persistent auf dem System vorliegen. Sie unterstützt die Verwendung von *Modules* als Bausteine: verfügbare *Module*-Elemente können über die Konfiguration zu einem Angriffserkennungssystem zusammengesteckt werden. Der Aufbau der Konfigurationsdatei ist in Abschnitt 4.6 erklärt.

Utilities

Den letzten Baustein bilden die *Utilities*. Sie stellen häufig verwendete Funktionalität, wie zum Beispiel ein Logging-System, zur Verfügung. Weiterhin werden unter anderem Datenstrukturen und Funktionen zur Konvertierung bereitstellt. Da die *Utilities* eine lose Ansammlung von Funktionen darstellen, werden sie nicht im Entwurf erklärt. Die Implementierung der *Utilities* ist in Abschnitt 5.6 beschrieben.

Zusammenspiel der Komponenten

Der *NetworkManager* ermöglicht es soweit möglich von der verwendeten Netzwerkumgebung und der unterliegenden Hardware zu abstrahieren. Dadurch ist die Verwendung der Angriffserkennung sowohl in einer realen Hardwareumgebung als auch in einem Simulator möglich. Anpassungen, welche für die Verwendung in unterschiedlichen Laufzeit- und Hardwareumgebungen vorgenommen werden müssen, werden somit komplett vom Framework selbst gekapselt. Die höher liegende Komponente des *ModuleManager* bietet das Rahmenwerk für die Integration einer Granularitätsbasierten Angriffserkennung. Die eigentliche Funktionalität ist dabei in *Modules* gekapselt. Diese können in mehreren Stufen gruppiert werden und ermöglichen somit die Definition von Granularitätsstufen. Die Kommunikation zwischen den *Modules* zur Angriffserkennung wird über das *MessagingSystem* realisiert, welches sich ebenfalls im *ModuleManager* befindet. Die Kommunikation mit entfernten Instanzen der Angriffserkennung wird über das *RemoteMessaging* realisiert. Dabei stellt dies eine zweigeteilte Architektur dar: die Verarbeitung von Nachrichten wird vom *RemoteMessaging* vorgenommen. Der eigentliche Versand wird über die austauschbare *Communication*-Komponente vorgenommen. Weiterhin existieren *Utilities* zur Unterstützung der Entwicklung. Die Konfiguration des Systems sowie die Verschaltung von *Modules* als Bausteine wird über die *Configuration*-Komponente realisiert.

4.2 Netzwerkabstraktion

Um eine Vielzahl an verschiedenen Laufzeitumgebungen und Netzen transparent zu unterstützen, muss von dem unterliegenden Netz und der Hard- und Softwareumgebung bestmöglich abstrahiert werden. Beispielsweise unterscheidet sich eine reale Router-Hardware grundlegend von einer Software-basierten Simulationsumgebung oder einem Sensorknoten. Daher muss eine einheitliche Schnittstelle zur Verfügung gestellt werden, um Zugriff auf diese unterschiedlichen Umgebungen zu ermöglichen.

Abbildung 4.2 zeigt die logische Unterteilung des NetworkManager in den von der Laufzeitumgebung abhängigen und unabhängigen Teil. Die von der Laufzeitumgebung abhängigen Teile stellen ein Interface vom Typ `NetworkInterface` zur Verfügung, um einen generischen Zugriff auf unterschiedliche Laufzeitumgebungen zu ermöglichen. Die im `NetworkInterface` definierten Schnittstellen bieten die Möglichkeit die Netzschnittstelle zu öffnen und zu schließen, Informationen über sie abzufragen und Frames von ihr zu lesen. Die Schnittstelle zum Lesen von Frames muss dabei immer eine Kopie des Frames anlegen. Dadurch wird sichergestellt, dass die Operation des Netzwerks unbeeinflusst bleibt.

Die eigentliche Implementierung des `NetworkInterface` ist von der Laufzeitumgebung abhängig. Sie implementiert die vorgegebenen Schnittstellen und beinhaltet die eigentliche Funktionalität zum Zugriff auf das Netzwerksystem. Auf Grund der Abstraktion sind verschiedene Implementierungen möglich, welche den Zugriff auf das Netzwerksystem beinhalten.

Neben der `NetworkInterface`-Schnittstelle besteht der von der Laufzeitumgebung unabhängige Teil des NetworkManager aus dem `FrameBuffer`. Er liest kontinuierlich Frames über die `NetworkInterface`-Schnittstelle. Diese speichert er in einer Warteschlange. Somit wird sichergestellt, dass alle Frames schnellstmöglich von der Netzschnittstelle gelesen werden und Bursts abgefangen werden können. Die Größe der Warteschlange kann eingestellt und somit den auf dem Vermittlungssystem verfügbaren Ressourcen angepasst werden.

4.3 Modul-Umgebung

Die Netzwerkabstraktion stellt eine Frame-Quelle für unterschiedliche Laufzeitumgebungen zur Verfügung. Die Nutzung dieser Frames geschieht vornehmlich in der Modul-Umgebung. Diese verarbeitet Frames und bietet das Rahmenwerk um ein Angriffserkennungssystem zu integrieren. Abbildung 4.3 zeigt einen Ausschnitt aus Abbildung 4.1. Anhand dieser Abbildung wird die Modul-Umgebung erläutert und die Anbindung an andere Komponenten verdeutlicht.

Die Schnittstelle zu der in Abschnitt 4.2 beschriebenen Netzwerkabstraktion stellt das `FrameDistributionSystem` dar. Es agiert als eigenständiger Thread, welcher aktiv Frames aus dem `FrameBuffer` der Netzwerk-Umgebung liest und diese in die höheren Komponenten weiterleitet. Damit stellt das `FrameDistributionSystem` diejenige Komponente dar, welche dafür verantwortlich ist Frames von der Netzwerk-Umgebung in die Modul-Umgebung zu transportieren.

Die nächsthöhere Komponente ist der `ChannelManager`. Er ist für die Verwaltung der

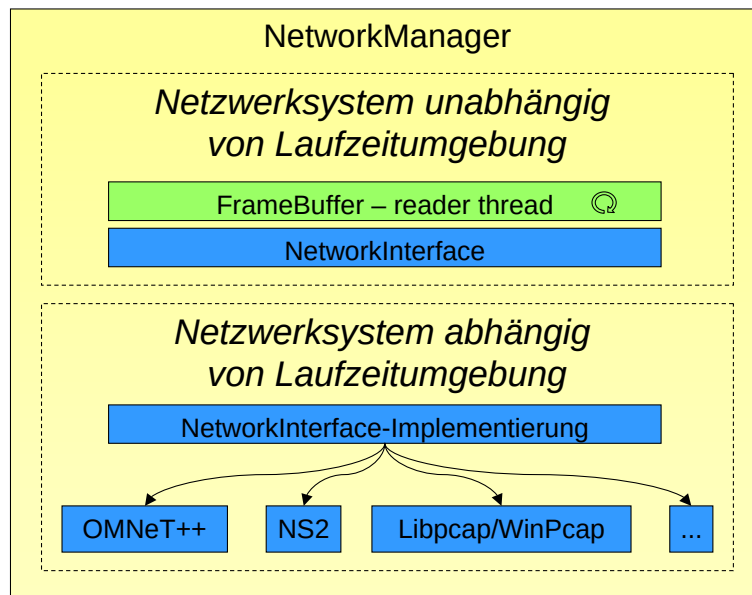


Abbildung 4.2: Detailansicht des NetworkManager

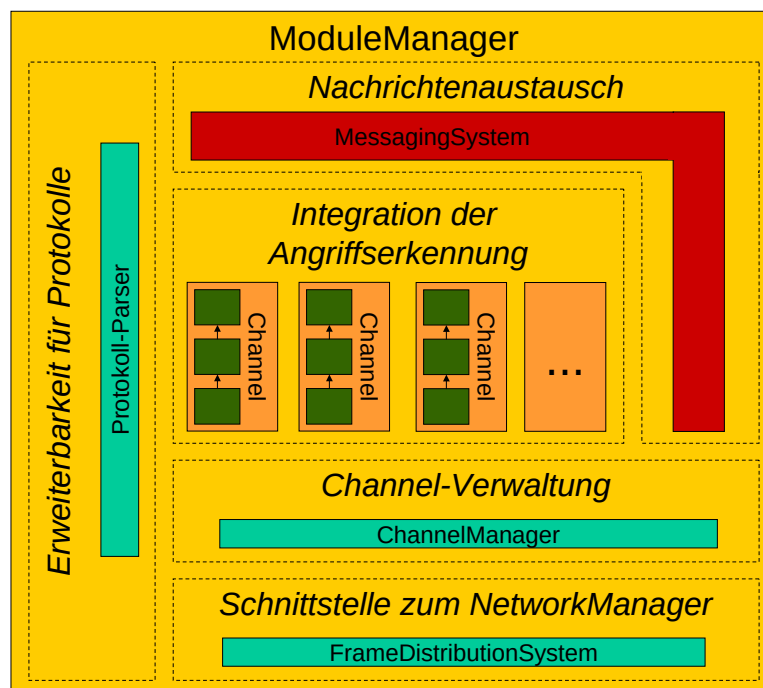


Abbildung 4.3: Übersicht der Modul-Umgebung

einzelnen Channel verantwortlich und kann diese laden und entladen. Ein Channel stellt eine logische Gruppierung von Funktionalität dar, welche in Modulen gekapselt ist. Beispielsweise könnte ein Sampling-Modul mit einem Modul welches statistische Daten sammelt in einen gemeinsamen Channel gesetzt werden. Somit wird sichergestellt, dass zusammengehörige Module – welche aufeinander aufbauen oder in einer anderen Nutzenrelation zueinander stehen – auch simultan geladen und entladen werden. Module liegen linear verkettet in einem Channel. Dabei ist die Reihenfolge der Module von Bedeutung: die Information, welche durch den Channel – und damit durch die einzelnen Module – fließt, sind Frames. Jedes Modul hat dabei die Möglichkeit den Frame zu analysieren und Informationen für seine eigene Arbeit zu gewinnen. Nach Verarbeitung eines Frames durch ein Modul kann das Modul entscheiden, ob der Frame im Channel – entsprechend der linearen Reihenfolge der Module – an das nächste Modul weitergegeben wird oder die Verarbeitung im Channel abgeschlossen wird. Ist die Verarbeitung in einem Channel abgeschlossen, so wird der Frame an den nächsten verfügbaren Channel weitergegeben und dort entsprechend durchgeleitet. Wurde der Frame durch alle Channels bearbeitet, so wird er gelöscht und der nächste Frame wird verarbeitet.

Die Gruppierung und Reihenfolge von Modulen in einem Channel sowie die Existenz von Channels sind rein konfigurationsabhängig und nicht fest vorgegeben. Diese Gruppierung verwirklicht einen Granularitäts-basierten Ansatz: zu Beginn der Angriffserkennung werden nur wenige Channels betrieben, deren Module eine ressourcenschonende Grundüberwachung des Netzwerkverkehrs vornehmen. Wurden Anzeichen für einen Angriff erkannt, so werden weitere Channels geladen, deren Module eine intensivere Analyse des Netzwerkverkehrs vornehmen. Somit können unterschiedliche Granularitätsstufen realisiert werden, welche entsprechend den verfügbaren Ressourcen und dem Status der Angriffserkennung ge- und entladen werden können. Die Granularität kann dabei beliebig gewählt werden, da die Gruppierung in Channels und die Sensitivität der einzelnen Channels rein konfigurationsabhängig und nicht begrenzt sind.

Durch die Entwicklung von Modulen ist es möglich, diese als Bausteine zu verwenden und beliebig in Channels zu gruppieren. Dabei ist auch eine mehrmalige Nutzung eines Moduls – beispielsweise mit unterschiedlichen Parametern – möglich. Weiterhin können Channels mit gleicher Granularitätsstufe per Konfiguration gruppiert werden. Diese werden entsprechend des derzeitigen Status der Angriffserkennung gemeinsam geladen und entladen. Module stellen somit funktionelle leichtgewichtige Einheiten dar welche entsprechend einem Baukastenprinzip verwendet werden können: eine Menge an Modulen kann der individuellen Umgebung entsprechend per Konfiguration zusammengebaut werden und kommuniziert selbständig. Weiterhin wird durch dieses Design erreicht, dass – abhängig von den auf dem Vermittlungssystem zur Verfügung stehenden Ressourcen – die zu betreibenden Module entsprechend angepasst werden können.

Die Integration eines Angriffserkennungssystems findet durch die Entwicklung von Modulen statt. Diese können dann durch die Konfiguration zusammengesetzt und entsprechend einem Granularitäts-basierten Ansatz gruppiert werden. Beim Laden eines Channels werden auch alle zu diesem Channel gehörigen Module explizit in den Speicher geladen. Gleiches gilt für das Entladen eines Channels: die Module

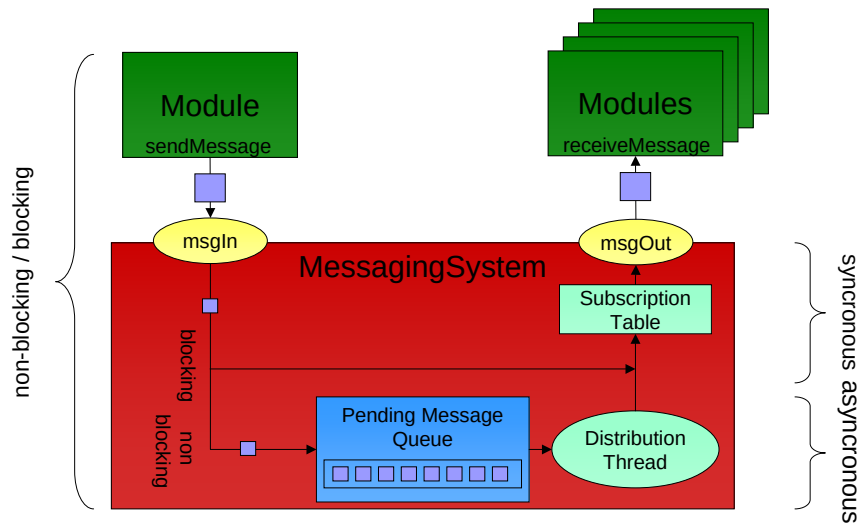


Abbildung 4.4: Detailansicht des MessagingSystem

welche in den Channel geladen wurden werden explizit aus dem Speicher entladen. Somit ist sichergestellt, dass zu jedem Zeitpunkt eine geringstmögliche Belastung der Ressourcen durch das Angriffserkennungssystem besteht. Dies ist notwendig, damit das Vermittlungssystem nicht in seiner Arbeit beeinträchtigt wird (siehe auch Abschnitt 3.1.1).

4.4 Lokales Nachrichtensystem

Module haben die Möglichkeiten in einem datenzentrierten Ansatz Nachrichten auszutauschen. Dieser Ansatz ähnelt dem von Gopalakrishna vorgestellten Verfahren der *Interessen-basierten Kommunikation* [GoSp01]. Das datenzentrierte Nachrichtensystem wird durch das MessagingSystem ermöglicht. Abbildung 4.4 zeigt eine Detailansicht des MessagingSystem.

Das MessagingSystem arbeitet Registrierungs-basiert. Es bietet den Modulen die Möglichkeiten sich für Nachrichten an- und abzumelden, Nachrichten zu empfangen und zu versenden. Jedes Modul registriert sich für die Nachrichten, welche es empfangen will. Dadurch wird vermieden, dass durch das MessagingSystem ein zu großer Overhead durch Broadcasting entstehen würde. Somit empfangen Module nur die Nachrichten, welche sie wirklich benötigen und für welche sie sich registriert haben. Beim Versand einer Nachricht durch ein Modul wird kein Empfänger angegeben. Stattdessen werden die Empfänger – abhängig von früheren Registrierungen durch Module – vom MessagingSystem bestimmt und die Nachricht entsprechend an die Module geliefert. Die Bestimmung der Empfänger muss dabei sehr schnell sein. Die Implementierung dieses Mechanismus zur schnellen Auffindung aller für eine Nachricht registrierten Module wird in Abschnitt 5.3 erläutert.

Das MessagingSystem arbeitet standardmäßig asynchron. Dies bedeutet, dass ein Sender einer Nachricht beim Versand nicht warten muss bis die Nachricht von allen Empfängern verarbeitet wurde. Stattdessen nimmt das MessagingSystem die Nachricht in eine Warteschlange auf und verarbeitet sie asynchron durch einen Thread. Somit wird sichergestellt, dass ein erhöhter Nachrichtenversand nicht das ganze System blockiert oder womöglich die Angriffserkennung verfälscht. Will ein Modul eine

Nachricht synchron versenden, so kann dies durch die Angabe spezieller Optionen beim Versand erreicht werden. Die Kontrolle geht somit erst dann an das sendende Modul zurück, nachdem die Nachricht von allen dafür registrierten Modulen verarbeitet wurde.

Der Versand von Nachrichten findet vornehmlich durch Module statt. Beispielsweise kann ein Modul, welches eine statistische Verteilung überwacht, periodisch Nachrichten mit aktuellen Verteilungswerten versenden. Diese können von anderen Modulen, welche sich für diesen Nachrichtentyp registriert haben, empfangen und verarbeitet werden. Falls sich kein Modul für den entsprechenden Nachrichttyp registriert hat, so wird diese vom MessagingSystem verworfen. Eine andere Möglichkeit ist, Nachrichten explizit anzufordern. Falls ein Modul Informationen nicht periodisch in das System sendet – beispielsweise falls die Daten sehr groß sind und somit hohe Ressourcenkosten hervorrufen würden – können diese in einem datenzentrierten Ansatz angefordert werden. Dazu sendet ein Modul eine Anfrage nach den gewünschten Informationen in das MessagingSystem. Module, welche die angeforderten Informationen bereitstellen können – und sich daher für den Nachrichtentyp registriert haben – senden eine Nachricht mit den angeforderten Informationen zurück in das MessagingSystem. Diese Nachricht wird entsprechend an alle für den Nachrichtentyp registrierten Module – und somit auch an den Anfragenden – gesendet.

4.5 Verteilte Kommunikation

Die lokale Kommunikation zwischen Modulen wird über das in Abschnitt 4.4 beschriebene MessagingSystem realisiert. Dies ermöglicht jedoch nicht die Kommunikation zwischen verteilten Instanzen der Angriffserkennung. Da lokales Wissen initial beschränkt ist, führt ein Austausch von Informationen zwischen verteilten Instanzen zu einem erhöhtem Wissen einer Instanz. Somit kann diese eine bessere Angriffserkennung durchführen, da sie nicht rein auf lokales Wissen beschränkt ist. Dies führt wiederum zu einer Verringerung von false positives – dem fehlerhaften Melden eines Angriffs – und false negatives – dem fehlerhaften Nicht-Melden eines Angriffs. Weiterhin können über Mechanismen zur verteilten Kommunikation Upstream- und Downstream-Signalisierung von Angriffen ermöglicht werden [GaSS07].

Die hier vorgestellte verteilte Kommunikation unterscheidet sich grundlegend von Sensor/Manager-Ansätzen, welche zum Beispiel in PreludeIDS (siehe Abschnitt 3.3.1) realisiert sind. Bei einem Sensor/Manager-Ansatz arbeitet eine Menge an Sensoren autonom und sendet ihre erkannten Angriffe an die Manager-Station. Diese empfängt die von den Sensoren erkannten Angriffe und zeigt sie dem Benutzer an. Bei diesem Ansatz wird *kein* verteiltes Wissen verwendet um eine bessere Angriffserkennung zu ermöglichen. Die vom Manager empfangenen Angriffe werden nicht dazu verwendet um sein – oder das Verhalten anderer Sensoren – zu ändern. Sie dienen rein zur Darstellung für den Benutzer.

Das in Abschnitt 4.4 beschriebene MessagingSystem für die lokale Kommunikation wurde auf Basis eines datenzentrierten Ansatzes entworfen. Dieses Prinzip soll auch für die verteilte Kommunikation verwendet werden. Der Austausch von Nachrichten zwischen verteilten Instanzen soll sich nicht vom lokalen Nachrichtenaustausch

unterscheiden: Nachrichtenobjekte, die lokal versendet werden, sollen auch an entfernte Instanzen versendet werden können. Damit wird der Aufwand zum Versand und Empfang von entfernten Nachrichten auf Seiten der Module bestmöglich minimiert.

Der Versand und Empfang von entfernten Nachrichten wird über das RemoteMessagingSystem ermöglicht. Es erlaubt den transparenten Versand und Empfang von Nachrichten durch Module. Hierzu ist es notwendig beliebige Nachrichten zu serialisieren und zu deserialisieren und sie damit in einen Bytestrom umzuwandeln oder aus einem Bytestrom zu erstellen.

Das RemoteMessagingSystem arbeitet über der austauschbaren Communication-Komponente. Diese ermöglicht den Transport von serialisierten Nachrichten. Damit ist die RemoteMessagingSystem-Komponente unabhängig vom konkret verwendeten Protokoll. Die verwendete Communication-Komponente kann zum Beispiel durch GIST oder eine Socket-Schnittstelle realisiert werden (siehe auch Abschnitt 4.5.3).

In dem folgenden Abschnitt 4.5.1 werden Probleme dargestellt, welche bei der Umsetzung der verteilten Kommunikation zu beachten sind. Abschnitt 4.5.2 beschreibt den Ablauf der verteilten Kommunikation sowie die Integration in das Framework. Schließlich werden in Abschnitt 4.5.3 zwei beispielhafte Protokolle zur Realisierung der Communication-Komponente vorgestellt: das Signalisierungsprotokoll GIST und die Socket-Schnittstelle.

4.5.1 Umsetzung der verteilten Kommunikation

Wie in Abschnitt 4.5 beschrieben, wird das RemoteMessagingSystem so entwickelt, dass es sich nahtlos in das lokale Nachrichtensystem (siehe Abschnitt 4.4) integriert. Hierfür müssen mehrere Punkte beachtet werden welche in den folgenden Abschnitten behandelt werden.

4.5.1.1 Zeitliche Aspekte entfernter Nachrichtenübertragung

Das in Abschnitt 4.4 beschriebene Nachrichtensystem bietet die Möglichkeit Nachrichten sowohl synchron wie auch asynchron an Module zu verteilen. Bei synchronem Versenden von Nachrichten wird der Sender so lange blockiert, bis alle Empfänger, welche sich für die Nachricht registriert haben, die Nachricht entsprechend verarbeitet haben. Hierfür ist es zum einen notwendig, dass Empfänger einer Nachricht diese grundsätzlich schnell verarbeiten und währenddessen nicht auf andere Ereignisse warten. Zum anderen ist es notwendig, dass das Nachrichtensystem Nachrichten schnell an die Empfänger weiterleiten kann. Dies ist beim lokalen Nachrichtenversand gewährleistet, da die Weiterleitung Prozess-intern stattfindet. Bei einer verteilten Kommunikation kann nicht gewährleistet werden, dass eine Nachricht innerhalb einer begrenzten Zeit an entfernte Module übertragen wird. Somit ist es nicht möglich einen synchronen Versand von Nachrichten zu ermöglichen. Alle Nachrichten welche an entfernte Instanzen versendet werden sollen, werden daher asynchron versendet. Dies betrifft jedoch nicht zwingend Nachrichten welche lokal *und* entfernt verarbeitet werden sollen: diese können weiterhin lokal synchron versandt werden. Die entfernte Nachrichtenübertragung geschieht aber in jedem Fall asynchron.

4.5.1.2 Serialisierung und Deserialisierung von Nachrichten

Zur lokalen Vermittlung von Nachrichten genügt es, diese als Objekte im lokalen Adressraum der Instanz zu übergeben. Um ein verteiltes Nachrichtensystem zu unterstützen ist die Serialisierung und Deserialisierung von beliebigen Nachrichten notwendig. Serialisierung beschreibt dabei die Umwandlung eines Nachrichtenobjektes in einen Datenstrom. Deserialisierung entsprechend den umgekehrten Vorgang der Erstellung eines Objektes im lokalen Adressraum der Instanz aus einem Datenstrom. Als Datenstrom dient hier eine Bytefolge, welche über eine Kommunikationsverbindung zwischen entfernten Instanzen des Angriffserkennungssystems übertragen wird.

Um Serialisierung und Deserialisierung zu unterstützen, müssen die Nachrichtenklassen angepasst werden. In Abschnitt 3.1.2 wurde die Erweiterbarkeit als Anforderung an das Framework zur Angriffserkennung genannt. Da neue Module auch neue Nachrichten benötigen, müssen sich die für die Serialisierung und Deserialisierung notwendigen Änderungen einer Nachrichtenklasse so gering wie möglich halten. Hierfür bietet das Framework entsprechende Unterstützung und hilft somit die Änderungen zu minimieren. Die notwendigen Erweiterungen werden in Abschnitt 5.4.2 detailliert und anhand von Beispielen beschrieben.

4.5.1.3 Wahl der Nachrichtenempfänger

Der Frage, ob eine Nachricht lokal bearbeitet oder an entfernte Instanzen versendet wird, kann durch zwei Möglichkeiten begegnet werden. Die erste Möglichkeit ist, dem Sender der Nachricht Mechanismen zur Verfügung zu stellen um selbst zu entscheiden, ob die Nachricht an lokale, entfernte oder beide Arten von Empfängern gesendet werden soll. Dies ermöglicht volle Flexibilität auf der Modul-Seite, da die Kontrolle über lokale oder entfernte Nachrichtenzustellung ganz auf der Seite der Entwickler von Modulen liegt. Die zweite Möglichkeit ist, für jede Nachricht zu definieren, ob diese grundsätzlich nur an lokale, nur an entfernte oder grundsätzlich an lokale und entfernte Empfänger versendet wird. Mit diesem Ansatz kann somit definiert werden, ob bestimmte Nachrichten grundsätzlich für alle Instanzen der Angriffserkennung oder nur für die lokale Instanz interessant sind.

Der zweite Ansatz stellt ein Problem dar: es kann nicht pauschal definiert werden welche Nachrichten für entfernte oder nur für das lokale Angriffserkennungssystem interessant sind. Daher wird hier der erstgenannte Ansatz verwendet und beim Versand einer Nachricht explizit angegeben, ob die Nachricht nur lokal, lokal und entfernt, oder nur entfernt versendet werden soll.

Eine weitere Frage der entfernten Nachrichtenübertragung ist, an welche entfernten Instanzen Nachrichten übertragen werden sollen. Prinzipiell sind hier alle benachbarten Angriffserkennungssysteme interessant. Weiterhin sind – besonders im Falle eines erkannten Angriffs – alle Angriffserkennungssysteme auf dem Pfad zum Angreifer von Interesse. Welche Möglichkeiten zur Nachrichteübertragung verfügbar sind, hängt von dem verwendeten Protokoll zur Realisierung der Communication-Komponente ab. Beispielsweise unterstützt eine TCP-Socket-basierte Communication-Komponente keine automatisierte Nachbarfindung.

Module müssen beim entfernten Versand einer Nachricht angeben, ob die Nachricht

an alle benachbarten Instanzen oder an eine definierte Menge an Instanzen gesendet werden soll. Dies Versandoptionen werden der Communication-Komponente übergeben. Diese ist dafür verantwortlich die Nachricht entsprechend zu übertragen. Hierzu können weitere Aufgaben notwendig sein – wie zum Beispiel eine Nachbarfindung – welche von der Communication-Komponenten realisiert werden müssen.

4.5.2 Ablauf der verteilten Kommunikation

Der Ablauf der verteilten Kommunikation ist in Abbildung 4.5 aus Sicht einer einzelnen Instanz dargestellt. Die Verwendung von Nachrichtenobjekten ist für die Module transparent: beliebige Nachrichtenobjekte können sowohl entfernt versendet als auch empfangen werden.

Nachrichtenversand

Durch Angabe von Versandoptionen kann ein Modul bestimmen ob eine Nachricht lokal, lokal und entfernt, oder nur entfernt versendet wird. Weiterhin kann angegeben werden, ob eine Nachricht asynchron oder synchron versendet werden soll. Im Falle eines Versands an eine entfernte Instanz wird diese Option nicht beachtet. Sie hat daher nur für den lokalen Versand Relevanz. Soll eine Nachricht an entfernte Instanzen versendet werden, so kann dies auf zwei Arten geschehen:

- Angabe einer Menge an entfernten Instanzen
- Versand an alle benachbarten Instanzen

Wurde durch das MessagingSystem festgestellt, dass eine Nachricht an entfernte Instanzen versendet werden soll, so wird sie mitsamt den Versandoptionen an die RemoteMessagingSystem-Komponente übergeben. Diese serialisiert die Nachricht und gibt die entstandenen Daten und Versandoptionen an die Communication-Komponente weiter. Hier wird der eigentliche Versand vorgenommen. Der Ablauf ist dabei stark von dem verwendeten Protokoll abhängig.

Nachrichtenempfang

Wird eine Nachricht von der Communication-Komponente empfangen, so wird diese an die RemoteMessagingSystem-Komponente weitergegeben. Dort findet eine Deserialisierung der Daten in ein Nachrichtenobjekt statt. Die Nachricht wird dann zur Verarbeitung an das MessagingSystem weitergegeben und – entsprechend dem lokalen Versand – an die für diesen Nachrichtentyp registrierten Module übergeben.

4.5.3 Beispielhafte Protokolle

Im Folgenden werden beispielhaft zwei Möglichkeiten zur Realisierung der Communication-Komponente vorgestellt: Das Signalisierungsprotokoll GIST und die Socket-Schnittstelle. Da die Communication-Komponente nur Schnittstellen zur Kommunikation vorgibt, ist die Verwendung beliebiger Protokolle möglich.

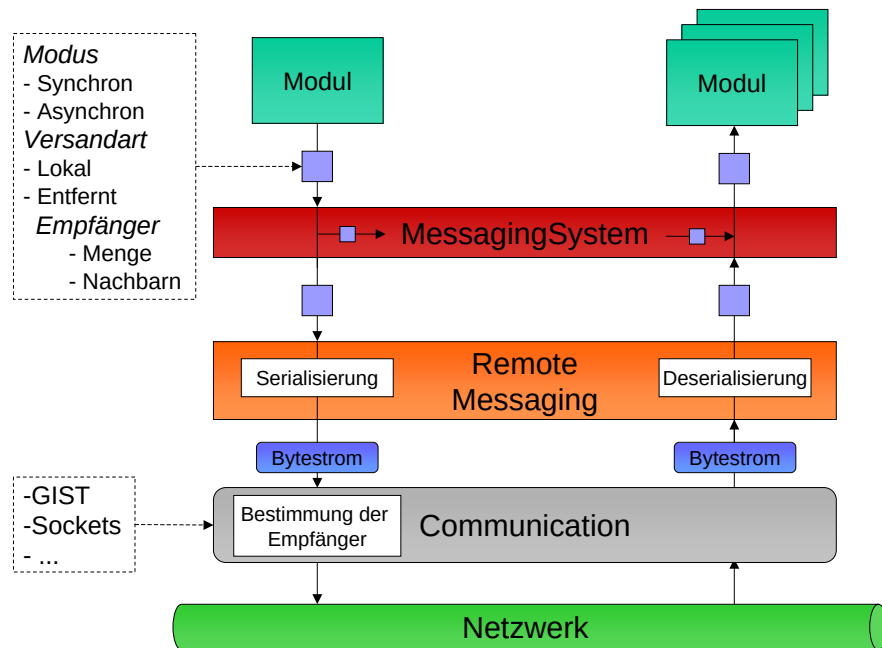


Abbildung 4.5: Ablauf der verteilten Kommunikation aus Sicht der Instanz

GIST

Das Signalisierungs-Framework Next Steps in Signaling (NSIS) [HKLB05] stellt Mechanismen zur pfadgebundenen Signalisierung im Internet zur Verfügung. Bei NSIS handelt es sich um eine zweigeteilte Architektur mit dem Ziel, die Signalisierung von Nachrichten von ihrem Transport zu trennen. Somit besteht NSIS aus den folgenden Protokollen:

- NSLP: NSIS Signaling Layer Protokoll
- NTLP: NSIS Transport Layer Protokoll

Ein Beispiel für ein NTLP-Protokoll ist das General Internet Signaling Transport Protokoll, kurz GIST [ScHa07b]. Es bietet eine automatisierte Auffindung von Nachbarknoten und ermöglicht somit die Signalisierung in *Richtung* einer IP-Adresse. Die Zwischensysteme – im GIST Kontext als *Middleboxes* bezeichnet – sind meist Vermittlungssysteme. Da auf den Middleboxes keine konkreten Server-Instanzen zur Signalisierung betrieben werden und die Signalisierungsnachricht meist nicht speziell an eine Middlebox gerichtet ist – sondern an alle Middleboxes, welche sich auf dem Pfad befinden – muss eine Möglichkeit bereitgestellt werden, um die Middleboxes von der Signalisierungsnachricht zu informieren. Hierfür wird bei GIST die IP Router Alert Option [Katz97] verwendet. Sie veranlasst jedes Vermittlungssystem dazu, das zur Weiterleitung erhaltene Paket zu inspizieren und gegebenenfalls an einen aktiven GIST-Prozess weiterzureichen. Der aktive GIST-Prozess stellt nun mit der Middlebox, von welcher das Paket empfangen wurde, explizit eine Verbindung über die Signalisierungsschicht NSLP her. Diese kann beispielsweise auch auf TLS basieren und somit gesichert und authentifiziert werden. Die eigentliche Signalisierungsinformation wird nun in der hierfür erstellten Verbindung über NSLP

übertragen.

Zur Kommunikation zwischen verteilten Instanzen der Angriffserkennung ergibt sich durch Verwendung von GIST die Möglichkeit zur automatisierten Auffindung von benachbarten Angriffserkennungssystemen, womit eine statische Vorkonfiguration der benachbarten Angriffserkennungssysteme entfällt. Nachrichten zwischen verteilten Instanzen der Angriffserkennung können somit einfach und effizient an Nachbarinstanzen weitergegeben werden.

Sockets-Schnittstelle

Die Verwendung der Sockets-Schnittstelle stellt eine weitere Möglichkeit zur Implementierung der Communication-Komponente für die verteilte Kommunikation dar. Es wird hier die zuverlässige Kommunikation über TCP [Post81b] betrachtet. Jede Instanz der Angriffserkennung agiert bei Wahl dieser Communication-Komponente als Client und Server. Die Server-Instanz wartet auf eingehende Nachrichten und gibt diese an die höhere Komponente weiter. Entsprechend wird die Client-Instanz verwendet um Nachrichten aktiv an entfernte Systeme zu versenden. Bei Verwendung der Sockets-Schnittstelle ist eine manuelle Konfiguration der entfernten Angriffserkennungssysteme notwendig.

4.6 XML-basierte Konfiguration

Die Konfiguration einer Instanz des Angriffserkennungssystems geschieht über eine XML-basierte Konfigurationsdatei, welche persistent auf dem Hostsystem vorliegen muss. Die Konfiguration definiert drei Abschnitte: zum einen werden grundsätzliche Parameter definiert, wie zum Beispiel die zu verwendende Frame-Quelle (siehe auch Abschnitt 4.2). Der zweite Abschnitt der Konfigurationsdatei definiert die zur Verfügung stehenden Modul-Instanzen sowie ihre Parameter. Die im zweiten Abschnitt definierten Modul-Instanzen werden im dritten Abschnitt der Konfiguration in Channels gruppiert. Channels können weiterhin in Stufen gruppiert werden. Channels mit gleicher Stufen werden gemeinsam geladen und definieren somit eine gemeinsame Granularitätsstufe des Angriffserkennungssystems.

Die XML-basierte Konfiguration erlaubt die Verwendung von Modulen als Bausteine: entwickelte Modul-Bausteine können durch die Konfiguration zusammengesteckt werden. Dieses Baustein-basierte Konzept unterstützt die in Abschnitt 3.1 geforderten Anforderungen der Erweiterbarkeit und Flexibilität.

4.7 Zusammenfassung

Der in diesem Kapitel erarbeitete Entwurf für das Framework zur Angriffserkennung durch verteilte Instanzen ermöglicht die flexible Integration von Modulen zur Angriffserkennung über einen Baustein-basierten Ansatz. Die Abstraktion von der Laufzeitumgebung und unterliegenden Hardware ermöglicht die Verwendung in einer Reihe von verschiedenen Systemen. Die Integration einer Angriffserkennung in Form von Modulen kann zum Beispiel ohne Änderungen auf einem Vermittlungssystem oder einem Simulator betrieben werden.

Die verteilte Kommunikation bietet einen sehr transparenten Ansatz, so dass Module nur durch Angabe zusätzlicher Parameter Nachrichten an entfernte Instanzen versenden können. Dabei können für die Angriffserkennung definierte Nachrichten beliebige Komplexität zur Darstellung ihrer Information verwenden. Der Versand von Nachrichten an entfernte Instanzen wird komplett vom Framework organisiert, so dass für Entwickler von Modulen zur Angriffserkennung minimale Änderungen notwendig sind um die verteilte Kommunikation zu unterstützen.

Durch die XML-basierte Konfiguration wird die Verwendung von Modul-Bausteinen ermöglicht. Module können durch die Konfiguration mit verschiedenen Granularitätsstufen zusammengesetzt und betrieben werden. Dies gewährleistet eine hohe Flexibilität und erlaubt die einfache Auslagerung von Modulen auf entfernte Instanzen. Somit ist nicht nur die verteilte Angriffserkennung sondern auch der verteilte Betrieb des Framework zur Angriffserkennung möglich.

5. Implementierung

In diesem Kapitel wird die Implementierung des in Kapitel 4 entworfenen Distack-Systems beschrieben. Es wird auf Details und Probleme eingegangen welche beachtet werden müssen um ein solches System – entsprechend den Anforderungen in Abschnitt 3.1 und dem Entwurf in Kapitel 4 – zu implementieren.

In Abschnitt 5.1 wird eine Netzwerkabstraktion implementiert welche die Verwendung in verschiedenen Umgebungen wie beispielsweise einem realen Router oder einem Simulator erlaubt. Die Implementierung der Modul-Umgebung wird in Abschnitt 5.2 beschrieben. Sie stellt die Umgebung dar in welcher Mechanismen zur Angriffserkennung implementiert werden. Für die Kommunikation zwischen Modulen einer lokalen Instanz sowie zur Unterstützung der verteilten Kommunikation wird in Abschnitt 5.3 das Nachrichtensystem implementiert. Um die Kommunikation zwischen verteilten Instanzen zu ermöglichen wird in Abschnitt 5.4 die Implementierung der verteilten Kommunikation beschrieben. Die auf Bausteinen basierende Konfiguration des Systems wird in Abschnitt 5.5 implementiert. Abschließend werden Hilfsfunktionen in Abschnitt 5.6 sowie im Framework implementierte Standardmodule in Abschnitt 5.7 beschrieben.

5.1 Netzwerkabstraktion

In diesem Abschnitt wird die Abstraktion von einer spezifischen Frame-Quelle beschrieben. Dabei wird eine beispielhafte Implementierung einer Frame-Quelle für reale Netzwerkhardware und für Tracedateien erläutert. In Abschnitt 6.3.1.2 ist weiterhin eine Netzwerkabstraktion für die OMNeT++-Simulationsumgebung beschrieben.

Wie in Abschnitt 4.2 beschrieben, ist die Netzwerkabstraktion dreigeteilt. Sie besteht aus den Klassen **FrameBuffer**, **NetworkInterface** und **NetworkManager** welche in den folgenden Abschnitten genauer erklärt werden. Hier sei angemerkt, dass diese drei Teile der Netzwerkabstraktion von spezifischen Umgebungen unabhängig sind und das allgemeine Framework zur Integration neuer Frame-Quelle darstellen. In Abschnitt 5.1.1.1 wird eine spezifische Implementierung des **NetworkInterface** für

Name	Beschreibung	Variablenname	Typ
Lese-Typ	Sagt aus, ob das Interface im Online- oder Offline-Modus arbeitet.	<code>m_captureType</code>	<code>CAPTURE_TYPE</code>
Status	Der derzeitige Öffnungsstatus des Interface.	<code>m_state</code>	<code>INTERFACE_STATE</code>
Promiscuous	Gibt an, ob das Interface im Promiscuous Mode geöffnet werden soll.	<code>m_promisc</code>	<code>bool</code>
Lese-Länge	Die Anzahl an Bytes welche von einem Frame gelesen werden sollen.	<code>m_frameCaptureLen</code>	<code>unsigned int</code>
Name	Der Name des Interfaces welches geöffnet wird.	<code>m_name</code>	<code>string</code>
Beschreibung	Einen beschreibenden Text für das Interface	<code>m_description</code>	<code>string</code>
Speedup	Faktor um der eine Tracedatei im Offline-Modul schneller oder langsamer angespielt werden soll	<code>m_replaySpeedup</code>	<code>double</code>

Tabelle 5.1: Allgemeine Eigenschaften eines NetworkInterface

das in Abschnitt 2.4 beschriebene Pcap-System beschrieben.

Das Parsen von Frames durch die in Abschnitt 2.5 beschriebenen Protokoll-Parser wird hier nicht durchgeführt, sondern auf höhere Schichten des Frameworks verlagert. Manche Frames werden durch Sampling- und Filtering-Methoden gelöscht und werden somit durch kein Modul betrachtet. Somit macht es Sinn nur Frames zu parsen, deren Informationen auch wirklich benötigt werden. Falls ein Modul die Inhalte eines Frames betrachten will, kann es das Parsen des Frames auslösen. Der Vorgang muss für jeden Frame nur einmal durchgeführt werden. Somit stehen Inhalte des Frame anderen Modulen sofort zur Verfügung.

5.1.1 NetworkInterface

Die `NetworkInterface`-Klasse stellt ein Interface für die Implementierung von realen Netzwerk-Schnittstellen zur Verfügung. Es werden dabei die in Tabelle 5.1 aufgelisteten Eigenschaften der Netzwerk-Schnittstelle verwaltet. Hierfür bietet die `NetworkInterface`-Klasse entsprechende Funktionen an.

Um ein reales Netzwerk-Interface zu implementieren, muss eine neue Klasse erstellt werden und diese von `NetworkInterface` abgeleitet werden. Weiterhin müssen die

folgenden drei Funktionen implementiert werden welche in der Klasse `NetworkInterface` als virtuell definiert sind:

```
bool      NetworkInterface::open      ();  
bool      NetworkInterface::close     ();  
Frame*    NetworkInterface::read      ();
```

Die Funktion `NetworkInterface::open` muss die Netzwerk-Schnittstelle mit dem Namen `m_name` (siehe Tabelle 5.1) öffnen. Dabei muss die Eigenschaft `m_promisc` beachtet werden welche die Art des Öffnens für das Interface beschreibt. Weiterhin muss unter Umständen die Eigenschaft `m_captureType` beachtet werden, falls das Interface einen einheitlichen Zugriff zu Online und Offline-Quellen ermöglicht. Nach der Bearbeitung der `NetworkInterface::open`-Funktion muss die Netzwerk-Schnittstelle in einem Zustand sein, welcher es erlaubt Frames zu lesen.

Entsprechend der `NetworkInterface::open`-Funktion muss durch einen Aufruf der `NetworkInterface::close`-Funktion das Interface geschlossen werden. Es muss dabei ein Zustand erreicht werden, so dass ein erneuter Aufruf der `NetworkInterface::open`-Funktion möglich ist und das Interface damit wieder geöffnet ist.

Als dritte Interface-Funktionalität muss die Funktion `NetworkInterface::read` implementiert werden. Sie ermöglicht das Lesen von Frames von dem unterliegenden Netzwerk. Ein Frame besteht aus Metainformationen und Daten. Die Metainformationen beschreiben unter anderem den Ankunftszeitpunkt des Frame sowie die Länge der enthaltenen Daten. Die Daten sind noch ohne Struktur und bestehen aus einer Abfolge von Bytes. Die Interpretation der Bytefolge wird auf höhere Schichten des Framework – unter Verwendung der in Abschnitt 2.5 beschriebenen Protokoll-Parser – verlagert. Um die im Frame enthaltene Bytefolge nicht bei der Weitergabe an höhere Schichten zu kopieren, wird diese in einen neu reservierten Speicherbereich gelegt. Die Allokation des Speichers liegt in der Verantwortung des `NetworkInterface`. Die Freigabe geschieht hingegen durch höhere Schichten. Somit muss das `NetworkInterface` keinen Status bezüglich Frames und Speicher verwalten.

Ein über die drei vorgestellten Funktionen implementiertes Netzwerk-Interface integriert sich nahtlos in das Framework und kann ohne zusätzlichen Aufwand direkt verwendet werden.

5.1.1.1 `NetworkInterface` für das Pcap-System

In diesem Abschnitt wird eine konkrete Implementierung der `NetworkInterface`-Schnittstelle vorgestellt. Sie verwendet das in Abschnitt 2.4 vorgestellte Pcap-System. Das Pcap-System erlaubt zum einen das Lesen von rohen Netzwerk-Frames von einem realen Netzwerk-Interface (Online-Modus), zum anderen können Tracedateien geöffnet und diese als Frame-Quelle verwendet werden (Offline-Modus). Hier werden nun die für beide Modi benötigten Operationen beschrieben.

Entsprechend Abschnitt 5.1.1 implementiert das hier vorgestellte

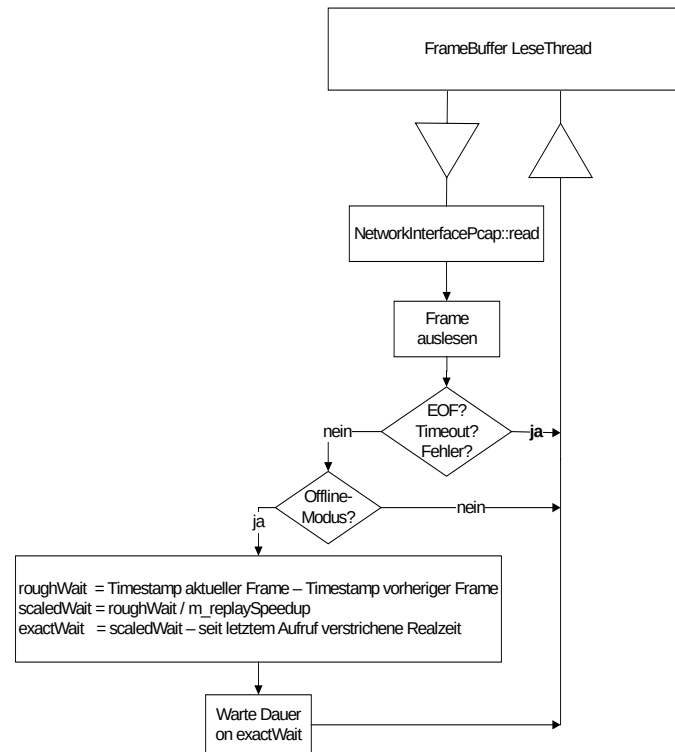


Abbildung 5.1: Ablauf der Implementierung für das NetworkInterface des Pcap-Systems

NetworkInterfacePcap die drei Funktionen `open`, `read` und `close`. Anhand dieser Funktionen wird die Implementierung in den folgenden Abschnitten vorgestellt. Abbildung 5.1 zeigt den Ablauf der Implementierung für das Pcap-System. Die Leseoperationen werden, wie in Abschnitt 5.1.2 beschrieben, vom **FrameBuffer** durchgeführt.

Open-Funktion

Bei der Wahl der Open-Funktion wird zwischen dem Online- und Offline-Modus unterschieden. Dieser wird entsprechend der Variable `m_captureType` aus Tabelle 5.1 bestimmt. Das Öffnen im Offline-Modus ist über die `pcap_open_offline`-Funktion realisiert. Als Parameter wird der Name der Tracedatei angegeben, welche geöffnet werden soll.

Das Öffnen eines Netzwerk-Interfaces im Online-Modus ist aufwendiger: Zuerst werden durch Verwendung der `pcap_findalldevs`-Funktion alle auf dem System vorhandenen Netzwerk-Interfaces gefunden. Somit kann bestimmt werden, ob das zu öffnende Netzwerk-Interface auf dem System vorhanden ist und geöffnet werden kann. Dann wird durch Aufruf der `pcap_open_live`-Funktion das Netzwerk-Interface geöffnet. Dabei sind mehrere zusätzliche Parameter anzugeben. Zum einen muss die Anzahl der von jedem Frame zu lesenden Bytes bestimmt werden. Weiterhin muss angegeben werden, ob das Netzwerk-Interface im Promiscuous-Modus geöffnet werden soll. Des Weiteren benötigt die `pcap_open_live`-Funktion einen Read-Timeout Parameter. Dieser bestimmt, wie lange ein Lese-Aufruf blockiert, falls kein Frame verfügbar ist. Ein zu kleiner Timeout-Wert führt dazu, dass die Funktion sehr kurz

blockiert und somit ein erneuter Funktionsaufruf zum Lesen eines Frames durchgeführt wird. Ein zu großer Timeout-Wert bewirkt, dass das Beenden des ganzen Frameworks blockiert wird. Als guter Kompromiss zeigt sich ein Timeout-Wert von 500 Millisekunden.

Read-Funktion

Nach Öffnen des Netzwerk-Interface im Online- oder Offline-Modus ist das `NetworkInterfacePcap` in einem Zustand, in welchem ein nachfolgender Aufruf von `NetworkInterface::read` möglich ist. Dieser wird vom in Abschnitt 5.1.2 beschriebenen `FrameBuffer` durchgeführt.

Ein `NetworkInterface::read`-Aufruf auf dem `NetworkInterfacePcap`-System unter Verwendung eines realen Netzwerk-Interfaces ist wie folgt realisiert: durch Verwendung der `pcap_next_ex`-Funktion kann vom Pcap-System der nächste verfügbare Frame angefordert werden. Der Funktionsaufruf blockiert bis ein Frame verfügbar ist. Da der interne Speicher des Pcap-Systems begrenzt ist, ist es sehr wichtig, dass verfügbare Frames schnell ausgelesen werden. Dies wird durch den in Abschnitt 5.1.2 beschriebenen `FrameBuffer` realisiert und unterstreicht noch einmal dessen Notwendigkeit. Werden Frames nicht schnell genug aus dem Pcap-System gelesen, so werden sie von diesem verworfen und sind somit für die spätere Analyse nicht verfügbar. Nach Einlesen eines Frames werden benötigte Datenstrukturen aus einem Speicherpool allokiert um den Frame und seine Daten zu verwalten (siehe auch Abschnitt 2.5). Dann wird der Frame entsprechend an den Aufrufer der `NetworkInterface::read`-Funktion zurückgegeben.

Falls es sich um eine Tracedatei im Offline-Modus handelt ist das Vorgehen anders. Hier wird über die `pcap_next_ex`-Funktion sofort der nächste Frame der Tracedatei zurückgegeben. In diesem Fall muss der Zeitstempel des Pcap-Formats beachtet werden. Im Offline-Modus besteht kein Zusammenhang zwischen dem Lesezeitpunkt und dem ursprünglichen Ankunftszeitpunkt eines Frames. Um eine Tracedatei entsprechend ihrer ursprünglichen Zeitdomäne wieder abzuspielen, ist es notwendig das Zeitdelta zwischen zwei Frames zu beachten. Um dies zu realisieren wird die Differenz der Zeitstempel zwischen zwei Frames beachtet und entsprechend dieser Differenz gewartet. In Abbildung 5.1 ist diese Zeitdauer als *roughWait* bezeichnet.

Das Abspielen von Tracedateien unter Beachtung der Frame-Zeitstempel ermöglicht das beschleunigte bzw. verlangsamte Abspielen ohne zeitliche Verzerrung. Durch Angabe des `m_replaySpeedup`-Parameters entsprechend Tabelle 5.1 ist es somit möglich, eine Tracedatei um einen Faktor schneller oder langsamer abzuspielen. Hierfür wird entsprechend Abbildung 5.1 die Zeitdauer *roughWait* durch den Wert des Parameters `m_replaySpeedup` dividiert. Die resultierende Zeitdauer wird als *scaledWait* bezeichnet. Von dieser Wartezeit muss jedoch zusätzlich die seit dem letzten Lesevorgang verstrichene Realzeit subtrahiert werden, womit sich die *exactWait*-Dauer ergibt. Somit wird sichergestellt, dass das Abspielen einer Tracedatei keiner zeitlichen Verzerrung unterliegt.

Offensichtlich kann eine Tracedatei nicht beliebig schnell abgespielt werden. Wurde ein zu hoher Speedup gewählt, so berechnet sich die *exactWait*-Variable zu Null.

Close-Funktion

Um das `NetworkInterfacePcap` wieder zu schließen, wird die `pcap_close`-Funktion verwendet. Das `NetworkInterfacePcap` wird so in einen Zustand versetzt, der einen erneuten Aufruf von `NetworkInterface::open` ermöglicht.

5.1.2 FrameBuffer

Die `FrameBuffer`-Klasse stellt einen Zwischenpuffer dar, welcher über das Netzwerk-Interface gelegt wird. Seine Aufgabe besteht darin, über einen eigenen asynchronen Thread Frames aus dem Netzwerk-Interface zu lesen und diese in einem Zwischenpuffer zu speichern. Die Notwendigkeit für einen solchen Zwischenpuffer besteht zum einen darin mögliche Bursts abzufangen. Zum anderen soll das implementierte `NetworkInterface` selbst keinerlei Zwischenpuffer verwalten müssen. Somit wird sichergestellt, dass verfügbare Frames schnellstmöglich aus dem `NetworkInterface` gelesen und dort nicht verworfen werden. Höhere Schichten – welche das eigentliche Angriffserkennungssystem implementieren – können Frames somit asynchron abarbeiten. Dies erhöht den Schutz gegen Angriffe und Überlastsituationen. Abbildung 5.2 zeigt den Aufbau des `FrameBuffer`, welcher die genannten Mechanismen implementiert.

Es sei hier erwähnt, dass der Zwischenpuffer lediglich Speicheradressen verwalten muss. Wie in Abschnitt 5.1.1 erläutert, wird der Speicherbereich für einen Frame vom `NetworkInterface` allokiert und durch höher liegende Schichten wieder freigegeben. Daher muss der `FrameBuffer` nur diese Adressen speichern und verursacht daher einen sehr geringen Overhead. Pro Frame muss der `FrameBuffer` somit vier Bytes zwischenspeichern.

Die Anzahl der im `FrameBuffer` eingereihten Frames entspricht auch dem auf dem Hostsystem allokierten Speicher, da jeder im `FrameBuffer` gespeicherte Frame auch im Speicher liegt. Daher gibt es zwei Mechanismen um die Anzahl der im `FrameBuffer` speicherbaren Frames zu begrenzen. Hierdurch kann auch effektiv der Speicherverbrauch auf dem System begrenzt werden. Zum einen ist es möglich die Anzahl der im `FrameBuffer` zwischengespeicherten Frames zu begrenzen. Eine weitere Möglichkeit stellt die Begrenzung des durch die Frames eingenommenen Speicherplatzes dar. Hiermit sind nicht der notwendige Speicherplatz für die Verwaltung der Speicheradressen gemeint, sondern der durch die Frames selbst eingenommene Speicherplatz. Es können auch beide Begrenzungen – die maximale Anzahl an Frames sowie der maximale Speicherverbrauch – angegeben werden. In diesem Fall wird ein Frame nur in den `FrameBuffer` aufgenommen, wenn beide Bedingungen erfüllt sind. Kann ein Frame nicht in den `FrameBuffer` aufgenommen werden, so wird er gelöscht und sein Speicherplatz freigegeben. Falls alle Frames verarbeitet werden sollen ist es möglich beide Begrenzungsmechanismen zu deaktivieren. Dies ist allerdings nicht empfehlenswert, da hierdurch Überlastsituationen hervorgerufen werden können, welche von einem Angreifer für einen DoS-Angriff gegen das Angriffserkennungssystem bzw. das Vermittlungssystem, auf welchem das Angriffserkennungssystem betrieben wird, ausgenutzt werden können.

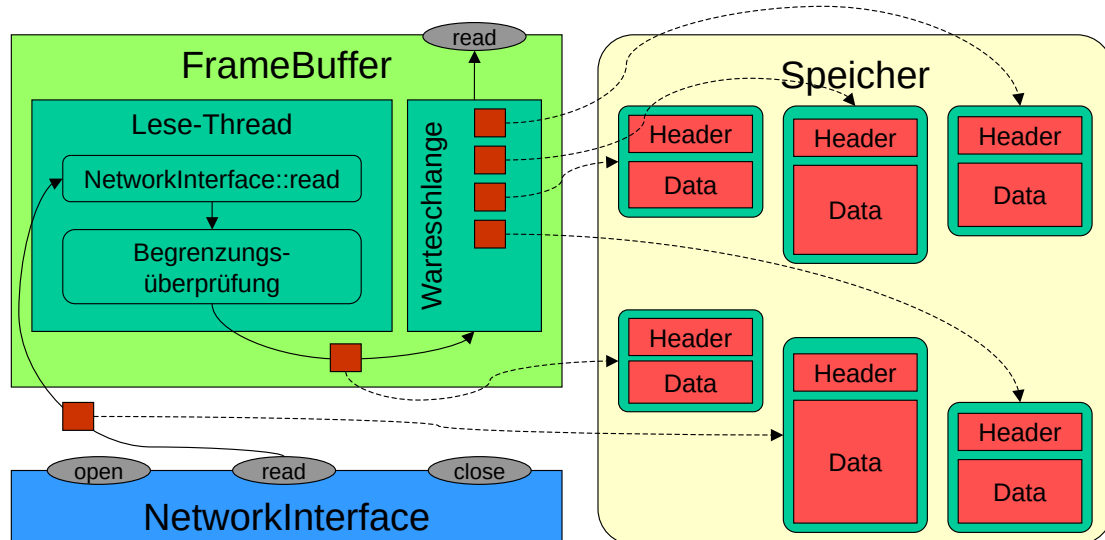


Abbildung 5.2: Übersicht der Funktionalität des FrameBuffer

5.1.3 NetworkManager

Die Aufgabe des **NetworkManager** besteht in der Konfiguration und Verwaltung des **FrameBuffer**- und **NetworkInterface**-Objektes. Somit stellt der **NetworkManager** die in Abschnitt 4.2 beschriebene Netzwerkabstraktion dar.

Zur Konfiguration greift der **NetworkManager** auf die XML-basierte Konfiguration zu (siehe auch Abschnitt 5.5). Dem **NetworkInterface**-Objekt wird die zu öffnende Netzwerk-Schnittstelle bzw. Tracedatei übergeben. Im Falle einer realen Netzwerk-Schnittstelle wird weiterhin angegeben, ob diese im Promiscuous Mode geöffnet werden soll oder nicht. Falls es sich bei der Frame-Quelle um eine Tracedatei handelt, wird der zum Abspielen der Frames verwendete Speedup angegeben (siehe Abschnitt 5.1.1.1). Weiterhin kann für beide Arten der Frame-Quellen angegeben werden, ob die Prüfsummen von Protokollen validiert werden sollen. Im Falle von ungültigen Prüfsummen werden die **Packet**-Objekte entsprechend gekennzeichnet.

Nach der Konfiguration werden das **NetworkInterface** und der **FrameBuffer** erstellt und verbunden. Somit ist es dem **FrameBuffer** möglich Frames aus dem **NetworkInterface** zu lesen und diese zwischenspeichern. Der **FrameBuffer** liest asynchron und kontinuierlich Frames auf dem **NetworkInterface**. Somit können Bursts abgefangen werden. Das Auslesen der Frames aus dem **FrameBuffer** geschieht durch höhere Schichten, welche in Abschnitt 5.2 erläutert werden.

5.2 Modul-Umgebung

Die Modul-Umgebung stellt alle Mechanismen zur Verfügung um eine Angriffserkennung mit minimalem Aufwand zu integrieren. Wie bereits in Abschnitt 4.3 aufgezeigt, beinhaltet die Modul-Umgebung mehrere Teilbereiche. Abbildung 5.3 zeigt einen funktionalen Überblick über die Modul-Umgebung. Dies ist keine architekturelle Darstellung, sondern beschreibt die Zusammenhänge zwischen Komponenten des Frameworks, der Konfiguration und Bibliotheken. Die aufgezeigten Komponenten

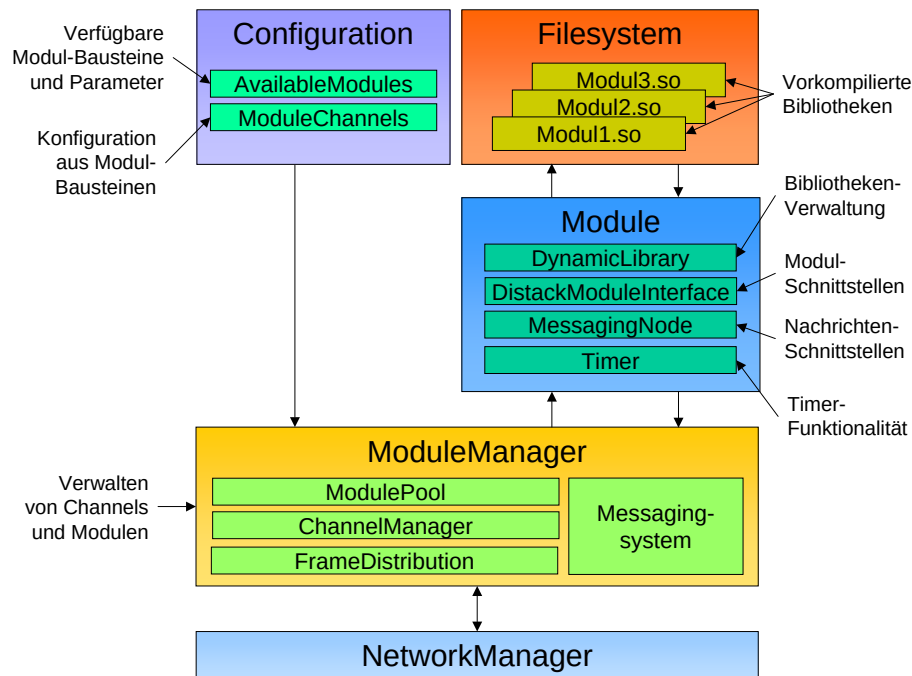


Abbildung 5.3: Funktionale Übersicht der Modul-Umgebung

werden in den folgenden Abschnitten detailliert erläutert und ihre Implementierung beschrieben.

5.2.1 ModuleManager

Neben dem in Abschnitt 5.1.3 beschriebenen **NetworkManager** stellt der **ModuleManager** den zweiten großen Block der Architektur dar. Er beinhaltet einen Großteil der Objekte und bildet das Rahmenwerk zur Verwaltung der Angriffserkennung. Der **ModuleManager** enthält die Objekte **FrameDistribution**, **ChannelManager** und **MessagingSystem**. Die Aufgabe des **ModuleManager** ist somit, diese Objekte zu verwalten und zu überwachen sowie die Konfiguration zu übernehmen.

Zu Beginn werden vom **ModuleManager** die verfügbaren Module sowie ihre Konfigurationsparameter eingelesen. Diese dienen als Schablonen für die spätere Erstellung von konkreten Instanzen der Module. Weiterhin registriert sich der **ModuleManager** für Nachrichten vom Typ `Message::MESSAGE_TYPE_ALERT`. Diese werden von Modulen im Falle eines erkannten Angriffs oder des Endes eines Angriffs versendet und vom **ModuleManager** verwendet um neue Stufen zu laden bzw. aktive Stufen zu entladen. Dann wird geprüft, welches die niedrigste konfigurierte Stufe in der Konfiguration darstellt, und die zu dieser Stufe gehörigen Channels zu laden. Wurde die niedrigste Stufe geladen, wird der **FrameBuffer** mit dem **FrameDistribution**-Objekt verbunden. Hiermit beginnt die Abarbeitung von Frames welche vom **FrameBuffer** gelesen wurden: das **FrameDistribution**-Objekt liest in einem Thread Frames, aus dem **FrameBuffer**. Diese werden den aktiven Channels übergeben und von diesen in einem seriellen Ansatz an ihre Module übergeben.

Das Laden einer Stufe läuft wie folgt ab: Zuerst werden alle Channels bestimmt, welche der angegebenen Stufe angehören. Jeder Channel wird autonom geladen. Hierfür

werden für jeden Channel die zugehörigen Module bestimmt. Die Konfiguration der Module wird entsprechend der zu Beginn eingelesenen Modul-Konfiguration zusammengestellt und dem **ChannelManager** übergeben. Dieser ist dafür verantwortlich die Channels zu konfigurieren und zu verwalten. Beim Entladen einer Stufe wird wiederum dem **ChannelManager** die Aufgabe übertragen die entsprechenden Channels zu entladen.

5.2.2 FrameDistribution

Die **FrameDistribution**-Klasse ist das Verbindungsstück zwischen der in Abschnitt 5.1 beschriebenen Netzwerkabstraktion und der Modul-Umgebung. Sie operiert direkt auf dem **FrameBuffer** (siehe Abschnitt 5.1.2). Durch Verwendung eines eigenen Threads werden Frames aktiv aus dem **FrameBuffer** gelesen und in der Modul-Umgebung verteilt. Dabei wird jeder Frame über den **ChannelManager** nacheinander in alle aktiven Channels geliefert. Diese wiederum übergeben allen Modulen im Channel den Frame in einer seriellen Reihenfolge. Nachdem ein Frame in allen Channels bearbeitet wurde wird er aus dem Speicher gelöscht: hier endet die Lebenszeit des Frames. Will ein Modul den Frame dauerhaft speichern, so muss explizit eine Kopie angelegt werden.

5.2.3 ChannelManager

Alle im System geladenen Channels werden vom **ChannelManager** verwaltet. Somit ist die Aufgabe des **ChannelManager** zweigeteilt: einmal muss er das Laden und Entladen der Channels implementieren, zum anderen stellt er die Schnittstelle zum **FrameDistribution**-Objekt dar und muss jeden Frame in die geladenen Channels weiterleiten.

Das Erstellen eines Channels ist sehr einfach, da der Channel und die Module selbst dafür verantwortlich sind, sich durch eine ihnen übergebene Konfiguration entsprechend zu konfigurieren. Somit muss nur ein neues **Channel**-Objekt erstellt werden. Diesem wird die Konfiguration der Module und des Channels selbst übergeben. Hiermit ist die Aufgabe des **ChannelManager** beendet. Der neu erstellte Channel wird in ein Vektor gespeichert und ist somit aktiv.

Zum Entladen einer Stufe kann dem **ChannelManager** die zu entladende Stufe übergeben werden. Jeder Channel, welcher der angegebenen oder einer höheren Stufe angehört, wird somit entladen. Dies bedeutet, dass zum Beispiel bei geladenen Stufen 1, 2 und 3 ein Entladen der Stufe 2 auch ein automatisches Entladen aller Channels der Stufe 3 veranlasst.

5.2.4 Channel

Wie in Abschnitt 4.3 beschrieben, stellt ein Channel eine logische Gruppierung von Modulen dar. Eine Channel-Konfiguration besteht aus dem Namen des Channels, der Stufe, welcher der Channel angehört, und den Modul-Konfigurationen, welche in den Channel geladen werden sollen.

Wird ein Channel gestartet, so wird ihm die Channel-Konfiguration übergeben. Für

jede durch den `ModuleManager` übergebene Modul-Konfiguration wird ein neues Modul erstellt. Die interne Konfiguration der Module wird entsprechend dem Modul weitergegeben. Dadurch kann sich das Modul intern selbst erstellen und konfigurieren. Somit wird jede Konfiguration der verantwortlichen Schicht überlassen.

Nachdem ein Modul erstellt und konfiguriert wurde, wird es an das `MessagingSystem` gebunden (siehe Abschnitt 5.2.5.4). Hierzu wird die Boost Signals Library [Greg01] verwendet. Dem Module werden somit folgende Funktionen des `MessagingSystems` bekannt gemacht:

- Registrieren von Nachrichten
- Senden von Nachrichten
- Deregistrieren von Nachrichten

Details hierzu werden in Abschnitt 5.2.5 vorgestellt. Nach erfolgreicher Konfiguration der Nachrichtenfunktionen wird das Modul dazu aufgefordert sich beim Nachrichtensystem für Nachrichten zu registrieren. Danach ist das Modul aktiv und wird gespeichert.

Wird ein Channel entladen, so wird für jedes enthaltene Modul wie folgt vorgegangen: Zuerst wird das Modul vom Nachrichtensystem entkoppelt und alle Registrierungen werden aufgehoben. Danach wird das Modul angehalten und schließlich zerstört.

Neben dem Laden und Entladen von Modulen ist ein Channel dafür verantwortlich Frames, welche vom `ChannelManager` übergeben werden, an die aktiven Module weiterzuleiten.

5.2.5 Module

Die von einem Channel verwalteten Elemente sind Module. Diese können zur Laufzeit geladen und entladen werden. Sie sind die eigentlichen funktionellen Einheiten und implementieren die Logik der Angriffserkennung. Um dynamisches Laden und Entladen zu ermöglichen, werden Module als Dynamic Link Libraries (unter Windows) bzw. als Shared Libraries (unter Unix/Linux) entwickelt (siehe Abschnitt 2.6). Das Laden und Entladen bezieht sich auch auf den Speicher des Hostsystems: das Entladen eines Moduls entlädt die Bibliothek auch aus dem Speicher. Damit arbeitet das System ressourcenschonend, da nur wirklich benötigte Module Speicherplatz beanspruchen.

Module sind als eigenständige Einheiten im Framework lauffähig. Die Gruppierung von Modulen zu Channels, sowie die Konfiguration der Module ist konfigurationsabhängig und wird in Abschnitt 5.2.5.1 erläutert. Durch die im Framework enthaltenen Channel- und Modul-Objekte wird die Konfiguration direkt nachgebildet. Damit Module durch Bibliotheken ge- und entladen werden können, müssen sie Betriebssystem-spezifische Funktionalität verwenden. Diese ist in Abschnitt 5.2.5.2 erklärt. Zur Integration eines Moduls in das Framework müssen zwei Schnittstellen beachtet werden. Dies ist zum einen das `DistackModuleInterface`-Interface zum

Gruppe	Name	Beschreibung
Allgemein	Name	Name des Moduls.
	Bibliothek	Name der Bibliothek, aus der das Modul geladen werden soll.
	Stufe	Stufe des Channels, in dem das Modul betrieben wird.
Parameter	RoutingTable	Name der zu verwendenden Routing-Tabelle oder ein leerer String, falls die System-Tabelle verwendet werden soll
	TimerModule	Name des Timer-Module, dessen Timer-Nachrichten zur Synchronisierung verwendet werden sollen
	HistoryLength	Anzahl der zu speichernden Intervalle

Tabelle 5.2: Beispielhafte Modul-Konfiguration

Laden, Entladen, Konfigurieren und Verarbeiten von Frames und zum anderen das `MessagingNode`-Interface zur Integration in das `MessagingSystem`. Beide Schnittstellen werden in Abschnitt 5.2.5.3 sowie Abschnitt 5.2.5.4 erläutert.

5.2.5.1 Konfiguration

Zur Konfiguration eines Moduls wird dem Modul vom Channel die für dieses bestimmte Konfiguration übergeben. Diese beinhaltet die folgenden allgemeinen Informationen:

- Der Name, unter welchem das Modul bekannt gemacht werden soll
- Die Bibliothek, aus welcher das Modul erstellt werden soll
- Die Stufe, zu welcher der Channel gehört, in welchem das Modul arbeitet

Weiterhin beinhaltet die Konfiguration eine Liste mit Parametern, welche der Implementierung des Moduls übergeben werden. Dies sind Name-Wert-Paare, welche direkt zur Konfiguration der spezifischen Implementierung verwendet werden. Alle Konfigurationselemente stammen aus der XML-basierten Konfiguration (siehe Abschnitt 5.5) und wurden initial vom Modul-Manager eingelesen (siehe Abschnitt 5.2.1).

Eine beispielhafte Konfiguration für ein Modul besitzt die in Tabelle 5.2 aufgeführten Parameter. Die Parameter der Gruppe *Allgemein* beschreiben den Namen des Moduls, die Bibliothek aus welcher das Modul erstellt werden soll sowie die Stufe des Channels in welcher das Modul konfiguriert ist. Spezifische Parameter beschreiben die interne Konfiguration des Moduls, wie zum Beispiel die in Tabelle 5.2 genannte Länge der Historie von Aggregatwerten.

5.2.5.2 DynamicLibrary

Um ein Modul aus einer Bibliothek zu laden und zu betreiben, existiert die `DynamicLibrary`-Klasse. Sie erlaubt das dynamische Laden von Bibliotheken unter Windows und Linux (siehe auch Abschnitt 2.6). Hierzu werden die jeweiligen

Aufgabe	Windows	Linux
Laden einer Bibliothek	LoadLibrary	dlopen
Entladen einer Bibliothek	FreeLibrary	dlclose
Funktionsadresse beziehen	GetProcAddress	dlsym
Abfrage von Fehlern	GetLastError	dlerror
Header	<Windows.h>	<dlfcn.h>

Tabelle 5.3: Funktionen zur Verwendung von Bibliotheken

Betriebssystem-spezifischen Funktionen verwendet. Tabelle 5.3 gibt einen Überblick über die verwendeten Funktionen der Betriebssysteme.

Weiterhin ermöglicht die `DynamicLibrary`-Klasse über Template-Mechanismen den Aufruf von beliebigen Funktionen aus der Bibliothek. Zur Verdeutlichung dieser Methodik wird der Code-Abschnitt in Listing 5.1 verwendet.

```
template<class R, class P>
R functionCall (string functionName, P param)
{
    R ret = R ();                                //
    typedef R(*FUNCCALL)(P);                      // (1)
    FUNCCALL func = NULL;                         //

    // (2)

#ifdef WIN32
    func = (FUNCCALL) GetProcAddress (
                                m_module,
                                functionName.c_str());
#else
    func = (FUNCCALL) dlsym (m_module,
                                functionName.c_str());
#endif // WIN32

    if (func != NULL) ret = func (param);          // (3)
    return ret;                                    //
}
```

Listing 5.1: Template-basierter Funktionsaufruf für Bibliotheken

Die `functionCall`-Funktionen ermöglicht es beliebige Funktionen der Bibliothek aufzurufen und dabei Typsicherheit bezüglich dem Parameter und des Rückgabewertes zur Kompilierzeit sicherzustellen. Die Marken im Code-Abschnitt beschreiben die folgende Funktionalität:

1. Erstellung des Rückgabewertes und Definition des Funktionspointers auf eine Funktion mit Parameter vom Typ `P` und Rückgabewert vom Typ `R`
2. Beziehen des Funktionspointers aus der Bibliothek über den Funktionsnamen

3. Falls die Funktion mit dem entsprechenden Namen gefunden wurde, wird die Funktion mit dem Parameter aufgerufen und der Rückgabewert zurückgegeben

5.2.5.3 DistackModuleInterface

Die **Module**-Klasse stellt die Verwendung und Verwaltung von Modul-Bibliotheken zur Verfügung stellt (siehe Abschnitt 5.2.5). Ein konkretes Modul wird durch Implementierung der **DistackModuleInterface**-Schnittstelle realisiert. Diese definiert die für ein Modul notwendigen Schnittstellen zur Konfiguration und zur Frame-Verarbeitung:

```
bool configure (ModuleConfiguration& configList);  
bool frameCall (Frame& frame );
```

Die **configure**-Methode übergibt der Modul-Implementierung eine Konfiguration, ähnlich der in Tabelle 5.2 beispielhaft vorgestellten. Die Modul-Implementierung hat nun zur Aufgabe die Konfiguration zu prüfen und sicherzustellen, dass alle notwendigen Konfigurationsparameter verfügbar und korrekt sind. Entsprechend werden alle Konfigurationsparameter ausgelesen. Um die Konfiguration flexibel zu halten, werden hier reine Zeichenketten betrachtet. Es ist Aufgabe der Modul-Implementierung diese in die entsprechenden Datentypen umzuwandeln. Ob die Konfiguration erfolgreich war, wird von durch den Rückgabewert angezeigt.

Durch Verwendung der **frameCall**-Methode wird der Modul-Implementierung vom Channel ein Frame zur Inspektion übergeben. Die Inspektion sollte schnellstmöglich geschehen, da sonst die Frame-Verarbeitung im Framework blockiert wird. Es sei hier zu erwähnen, dass dies jedoch durch Verwendung des **FrameBuffer** nicht zu einer Blockierung des ganzen Systems führt.

Welche Operationen in der **frameCall**-Methode durchgeführt werden, hängt stark von der Aufgabe der spezifischen Modul-Implementierung ab. Beispielsweise könnte ein Sampling- oder Filtering-Modul prüfen, ob der Frame durch andere Module im selben Channel analysiert werden soll oder nicht. Dies kann durch den Rückgabewert bestimmt werden. Inspektions-Module analysieren in der **frameCall**-Methode den Frame und aktualisieren zum Beispiel ihre statistischen Verteilungen. Hierbei kann durch Verwendung der Protokoll-Parser (siehe Abschnitt 2.5) auf alle in dem Frame gekapselte Protokolle zugegriffen werden. Es wird hier eine passive Strategie beim Parsen eines Frames verwendet: im Normalfall wird der Frame nicht geparkt. Erst wenn ein Modul dies durch Aufruf der **Frame::parsePackets**-Methode veranlasst, wird der Frame und alle seine Protokolle geparkt und über einfache Schnittstellen zur Verfügung gestellt. Ein einmaliges Parsen eines Frames bleibt für die Lebenszeit des Frames im gesamten System erhalten, so dass weitere Aufrufe der **Frame::parsePackets**-Methode sofort beendet werden und keine weitere Laufzeit benötigen.

Durch Vererbung und Implementierung der **DistackModuleInterface**-Schnittstelle erhält das Modul zwei weitere Funktionalitäten: Zugriff auf das Nachrichtensystem sowie die Möglichkeit zur Verwendung von Timern. Da diese beiden Funktionalitäten in vielen Modulen benötigt werden, sind sie direkt in die Schnittstelle integriert.

Die Implementierung der Timer-Funktionalität wird in Abschnitt 5.2.5.5 erläutert und der Zugriff aus das Nachrichtensystem im folgenden Abschnitt 5.2.5.4.

Module-Implementierungen sind nicht gegen das Framework selbst gelinkt. Dies bedeutet, dass sich die Modul-Bibliotheken und das Framework initial nicht kennen. Vorteil dieses Vorgehens ist, dass beliebige Module vom Framework geladen und verwendet werden können. Beispielsweise kann das Framework mit Modulen arbeiten, die von einer anderen Instanz über das Netzwerk gesendet wurden und möglicherweise mit einer anderen Version des Frameworks erstellt wurden. Dem Framework ist lediglich eine vordefinierte Funktionen bekannt, welche in den Modul-Bibliotheken exportiert wird. Sie kann über den in Abschnitt 5.2.5.2 beschriebenen Mechanismus verwendet werden. Diese spezielle Funktion wird über ein Makro definiert und kann so leicht in die Modul-Implementierung eingebunden werden. Dieser Mechanismus wird nun näher erläutert.

Unter dem Linux Betriebssystem sind grundsätzlich alle Symbole einer Bibliothek exportiert und können verwendet werden. Dies ist unter Windows nicht der Standard. Dort müssen zu exportierende Funktionen speziell gekennzeichnet werden. Dies geschieht durch Verwendung des `DLLEXPORT`-Makros, welches wie folgt definiert ist:

```
#ifndef WIN32
    #define DLLEXPORT __declspec(dllexport)
#else
    #define DLLEXPORT
#endif // WIN32
```

Somit kann nun eine Schnittstellenfunktion definiert werden mit welchem eine Modul-Implementierung ihre Funktionalität dem Framework zur Verfügung stellen kann:

```
#define MODULE_INTERFACE(x) \
    extern "C" DLLEXPORT DistackModuleInterface* \
    createModule () \
    { \
        return new x(); \
    }
```

Bemerkenswert ist hier noch der Zusatz `extern "C"`. Er bewirkt, dass die Funktion als C- und nicht als C++ Funktion exportiert wird. Dies verhindert das sogenannte *Name-Mangling* welches normalerweise bei C++-Code stattfindet und den Namen kodiert. Somit kann sichergestellt werden, dass die Funktion direkt über den Namen `createModule` gefunden werden kann.

Eine Modul-Implementierung welche das `DistackModuleInterface` implementiert kann nun durch Verwendung dieses Makros vom Framework eingebunden werden. Ein weiterer Vorteil dieses Vorgehens ist, dass keine Speicherprobleme auftreten: normalerweise darf Speicher, welcher in einer Bibliothek allokiert wurde nicht vom Framework freigegeben werden. Dies bringt große Schwierigkeiten in Bezug auf komplexe Rückgabewerte mit sich. In dem hier vorgestellten Ansatz wird ein Modul

durch Aufruf des vorgestellten Makros *im* Framework selbst erstellt. Damit entstehen keine der beschriebenen Speicherprobleme. Die Bibliothek stellt somit nur eine Fabrik dar aus welcher eine Modul-Implementierung angefordert und erzeugt werden kann.

5.2.5.4 MessagingNode

Wie in Abschnitt 5.2.5.3 erläutert, sind Module zum Framework lose gekoppelt. Somit sind einer Modul-Implementierung auch die Funktionen für die Verwendung des Nachrichtensystems initial unbekannt. Die Funktionen zum Zugriff auf das Nachrichtensystem sind zwar durch eine Signatur definiert, jedoch noch nicht mit dem Nachrichtensystem selbst verbunden. Ein grobes Vorgehen zu diesem Problem wurde bereits in Abschnitt 5.2.4 erläutert. Hier soll nun detailliert auf die implementierte Lösung eingegangen werden.

Die Boost Signals Library [Greg01] bietet Funktionalität zum dynamischen Verbinden von Funktionen. Damit kann ein Aufrufer dynamisch an eine Menge von Aufgerufenen gebunden werden. Solche Möglichkeiten sind in der C++-Sprache nicht vorgesehen. Dort ist es nicht möglich Member-Funktionen einer Klasse gezielt auf einem Objekt über Funktionszeiger aufzurufen. Dies ist nur bei statischen Member-Funktionen möglich. Durch Verwendung der Boost Signals Library ist es möglich Funktionssignaturen zu definieren und diese an beliebige Objekte zu binden. Hierbei muss keine konkrete Klasse des Zielobjekts bekannt sein.

Wie bereits in Abschnitt 5.2.4 erklärt, werden mehrere Funktionen benötigt um mit dem Nachrichtensystem zu arbeiten. Die Implementierung wird hier exemplarisch anhand einer Funktion – der `sendMessage`-Funktion – erklärt, welche für den lokalen und entfernten Versand und Empfang von Nachrichten verwendet wird. Die anderen Funktionen sind ähnlich aufgebaut.

Es müssen drei Komponenten betrachtet werden um das Zusammenspiel zu verdeutlichen:

- **MessagingNode:** Stellt der Modul-Implementierung Funktionalität zur Verwendung des Nachrichtensystems bereit
- **MessagingSystem:** Das Nachrichtensystem selbst
- **Channel:** Konfiguriert das Modul zur Verwendung des Nachrichtensystems

Wenn ein Modul eine Nachricht senden will, so verwendet es direkt die Funktionalität des **MessagingNode**. Dieser besitzt eine Variable `m_conSendFun` vom Typ `boost::signal`, welche eine Quelle definiert. Die genaue Definition lautet:

```
typedef boost::signal<void(Message*,SEND_OPTIONS)> MESSAGE_SEND_FUNC;
MESSAGE_SEND_FUNC m_sendFunction;
```

Auf diese Weise wird eine Quelle definiert, welche über eine `boost::signals::connection` mit einer Senke verbunden werden kann. Dabei muss die Senke eine Funktion vom Rückgabety `void` sein und zwei Parameter

vom Typ `Message*` bzw. `SEND_OPTIONS` akzeptieren. Eine solche Funktion ist im `MessagingSystem` wie folgt implementiert:

```
void MessagingSystem::sendMessage (
    Message*          msg      ,
    MessagingNode::SEND_OPTIONS options )
```

Nachdem ein `Module` durch den übergeordneten `Channel` erstellt und mit Parametern konfiguriert wurde, wird es mit dem Nachrichtensystem verbunden. Hierzu bindet der `Channel` die Senke `MessagingSystem::sendMessage` an das `MessagingSystem`-Objekt `msgsystem` und übergibt es dem `MessagingNode`:

```
MessagingNode* messenger = dynamic_cast<MessagingNode*>
                           (module.getInternal());
messenger->configure (boost::bind(
                       &MessagingSystem::sendMessage,
                       &msgsystem, _1, _2));
```

Der `MessagingNode` verbindet nun die Quelle mit der übergebenen Senke durch eine Verbindung:

```
m_conSendFun = m_sendFunction.connect (sendFunc);
```

Somit sind nun Quelle und Senke miteinander verbunden. Ein Aufruf der Quelle mit entsprechenden Parametern wird über die Verbindung auf die Senke weitergeleitet. Somit kann ein Modul die Funktion zum Versand von Nachrichten verwenden, ohne das Objekt des Nachrichtensystems selbst zu kennen. Ein Aufruf durch ein Modul sieht beispielsweise so aus:

```
MessageAlert* msg = new MessageAlert (
    MessageAlert::ALERT_TYPE_RAISE_LEVEL,
    m_stage,
    alertMessage);

MessagingNode::SEND_OPTIONS option =
    MessagingNode::SEND_OPTIONS::createLocalAsync ();

MessagingNode::sendMessage (msg, option);
```

Dieser wird von der `MessagingNode::sendMessage`-Funktion über die Verbindung an das Nachrichtensystem weitergeleitet:

```
void MessagingNode::sendMessage (Message* msg, SEND_OPTIONS options)
{
    m_sendFunction (msg, options);
}
```

Für diesen Zweck wurde in der Boost Signals Library der `()`-Operator der Template-basierten `boost::signal`-Klasse überschrieben. Daher ist ein Funktionsaufruf möglich als ob `m_sendFunction` eine Funktion darstellt. In Wirklichkeit erweitert der Compiler diese Zeile zu

```
m_sendFunction.operator()(msg, options);
```

um den überladenen Operator `()` aufzurufen welcher als Funktion `boost::signal::operator()` implementiert ist.

Nachdem ein Modul durch seinen Channel mit dem Nachrichtensystem konfiguriert wurde, wird das Modul zum Registrieren seiner Nachrichten aufgefordert. Dies geschieht durch Aufruf der `MessagingNode::registerMessages`-Funktion, welche von einer Modul-Implementierung bereitgestellt werden muss. In dieser Funktion registriert sich die Modul-Implementierung für alle Nachrichtentypen, welche für sie von Interesse sind.

5.2.5.5 Timer

Wie auch die in Abschnitt 5.2.5.4 vorgestellten Schnittstellen für den Nachrichtenaustausch, stellen Timer-Mechanismen eine häufig benötigte Funktionalität dar. Daher wurden sie direkt in die einem Modul grundsätzlich zur Verfügung stehenden Funktionen eingebaut.

Ein Timer ruft in zeitlich periodischen Intervallen eine vordefinierte Funktion auf. Diese Funktion hat die folgende Signatur:

```
void Timer::eventFunction ();
```

Will eine Modul-Implementierung die Timer-Funktionalität verwenden, so überschreibt sie diese Funktion. Zur Verwaltung des Timers stehen die folgenden Funktionen zur Verfügung:

- `Timer::setInterval`: Setzt das Timer-Intervall in Millisekunden
- `Timer::start`: Startet den Timer
- `Timer::stop`: Stoppt den Timer

Eine wichtige Eigenschaft der Timer-Implementierung ist die Unterbrechbarkeit. Dies bedeutet, dass zu jedem Zeitpunkt der Timer abgebrochen werden kann und nicht gewartet werden muss, bis das nächste Timer-Intervall abgelaufen ist. Dies wird durch die hier vorgestellte Implementierung erfüllt. Hierzu wird eine neue Klasse `InterruptableSleeper` erstellt. Diese erlaubt es, einen Thread für eine definierte Zeit anzuhalten. Weiterhin ist es möglich das Pausieren des Threads zu jedem Zeitpunkt abzubrechen.

Der Aufruf der `Timer::start`-Funktion bewirkt die Erstellung eines Threads. In

einer Schleife wird die `InterruptedException`-Klasse verwendet um den Thread für die Zeitdauer des Timer-Intervalls anzuhalten. Ist das Intervall abgelaufen, so wird die `Timer::eventFunction`-Funktion aufgerufen. Da diese von dem entsprechenden Modul überschrieben wird, wird der Aufruf an das Modul weitergeleitet.

Die Problematik der Unterbrechbarkeit hat sich also auf die Implementierung der `InterruptedException`-Klasse verlagert. Unterbrechbare Warte-Mechanismen sind nicht Teil der C++-Sprache. Sie sind plattformabhängig und müssen daher für Windows und Linux getrennt betrachtet werden. Das grundlegende Vorgehen ist jedoch bei beiden Plattformen identisch: Durch die Verwendung von Ereignissen wird auf das Eintreten des Unterbrechungs- bzw. Timeout-Ereignis gewartet.

InterruptedException unter Windows

Die hier verwendeten Funktionen sind allesamt aus der Win32-Synchronization-API des Windows Betriebssystems. Über die `CreateEvent`-Funktion wird eine Ereignis erstellt. Soll nun der übergeordnete Timer-Thread für eine definierte Zeitdauer angehalten werden, so wird die `WaitForSingleObject`-Funktion verwendet. Dieser wird die Ereignis-Variable sowie ein Timeout übergeben. Soll der Timer unterbrochen werden, so kann mit der `SetEvent`-Funktion die Ereignis-Variable gesetzt werden. In diesem Fall bricht die `WaitForSingleObject`-Funktion sofort ab und gibt die Kontrolle an den übergeordneten Thread zurück. Falls der Timer nicht abgebrochen wird, endet die `WaitForSingleObject`-Funktion mit Ablauf des Timer-Intervalls. Nach Ablauf des Timer-Intervalls kann über die `ResetEvent`-Funktion die Ereignis-Variable wieder rückgesetzt werden und ist somit für den nächsten `WaitForSingleObject`-Aufruf bereit. Wird das `InterruptedException`-Objekt zerstört, so wird die Ereignis-Variable unter Verwendung der `CloseHandle`-Funktion gelöscht.

InterruptedException unter Linux

Die Implementierung des `InterruptedException`-Klasse ist unter Linux ein bisschen komplexer. Dies hat zwei Gründe: zum einen sind unter Windows Ereignis-Variablen automatisch durch einen Mutex geschützt. Unter Linux muss dies von Hand realisiert werden. Zum anderen ist die Handhabung des Timeouts unter Linux komplexer. Die verwendeten Funktionen stammen hier aus der pthreads-Bibliothek.

Als Ereignis-Variable wird ein neuer Typ definiert. Dieser enthält eine Bedingungs-Variable sowie einen Mutex um diese zu schützen:

```
typedef struct _MUTEX_COND {
    pthread_cond_t  cond;
    pthread_mutex_t mutex;
} MUTEX_COND, *PMUTEX_COND;
```

Die Bedingungs-Variable sowie der Mutex werden zuerst mit den Funktionen `pthread_mutex_init` sowie `pthread_cond_init` initialisiert. Das Anhalten des übergeordneten Threads geschieht durch die `pthread_cond_timedwait`-Funktion. Da diese Funktion kein Timeout in Millisekunden akzeptiert, müssen vorhergehende

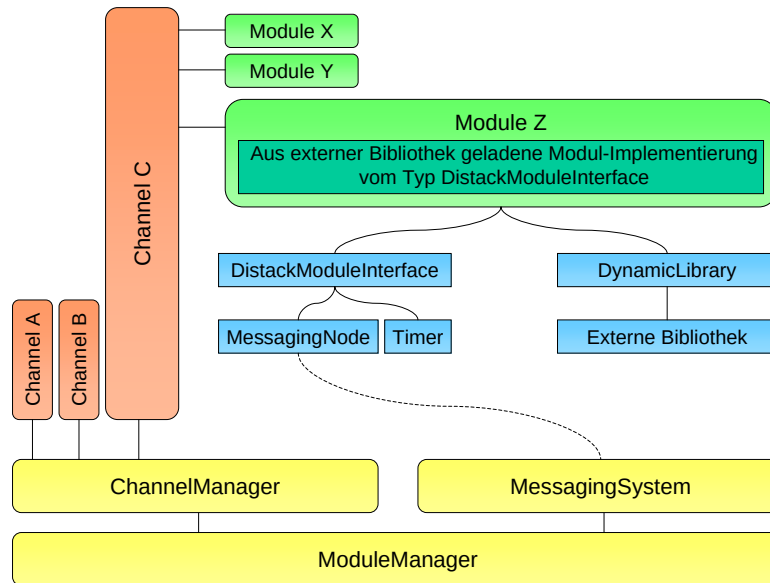


Abbildung 5.4: Zusammenhang der Komponenten eines Moduls

zeitliche Berechnungen angestellt werden. Der Aufruf der `pthread_cond_timedwait`-Funktion muss durch den Mutex entsprechend geschützt werden. Ein Rücksetzen der Bedingungs-Variable ist unter Linux nicht notwendig. Soll der Schlafvorgang unterbrochen werden, so wird die `pthread_cond_signal`-Funktion verwendet. Diese bricht den blockierenden Wartevorgang der `pthread_cond_timedwait`-Funktion ab. Bei Zerstörung des `InterruptableSleeper`-Objektes wird die Bedingungsvariable durch die `pthread_cond_destroy`-Funktion und der Mutex durch die `pthread_mutex_destroy`-Funktion zerstört.

5.2.5.6 Zusammenhang

Die in den letzten Abschnitten beschriebenen Mechanismen sind zum Teil komplex und der Gesamtzusammenhang daher nicht offensichtlich. Daher soll dieser hier genauer betrachtet und verdeutlicht werden. Das Zusammenspiel der einzelnen Komponenten eines Moduls ist in Abbildung 5.4 gegeben. Sie zeigt den **ModuleManager** mit den Komponenten **ChannelManager** und **MessagingSystem**. Die restlichen Komponenten werden hier der Übersichtlichkeit wegen nicht gezeigt. Der **ChannelManager** beinhaltet drei **Channel**-Objekte mit dem Namen *A*, *B* und *C*. **Channel C** besitzt drei Module *X*, *Y* und *Z*. Der Aufbau wird anhand Modul *Z* erläutert. Durch die Konfiguration erhält Modul *Z* von **Channel C** den Namen der zu verwendenden externen Bibliothek. Über das **DynamicLibrary**-Objekt ist es dem Modul möglich die dynamische Bibliothek zu laden. Das in der Bibliothek enthaltene Objekt ist vom Typ **DistackModuleInterface**. Diese Modul-Implementierung wird aus der geladenen Bibliothek erstellt und im Modul gespeichert. Die geladene Modul-Implementierung vom Typ **DistackModuleInterface** besitzt Funktionalität zur Verwendung von Timern sowie Schnittstellendefinitionen zur Verwendung des **MessagingSystem**. Diese Funktionen sind jedoch initial noch nicht mit dem **MessagingSystem** verbunden. Die Herstellung der Verbindung geschieht über Verwendung der Boost Signals Library. Somit ist das **MessagingNode**-Objekt konfiguriert und kann das Nachrichtensystem verwenden.

5.3 Nachrichtensystem

In Abschnitt 5.2.5.4 wurde bereits – aus Sicht eines Moduls – auf die Implementierung des Nachrichtensystems eingegangen. Es wurde beschrieben, wie ein Modul an die Funktionalität des Nachrichtensystems gekoppelt wird um diese zu nutzen. Hier wird nun die interne Funktionalität des Nachrichtensystems erläutert. Zuerst wird auf die Implementierung von Nachrichten selbst eingegangen. Dann wird die Implementierung des Nachrichtensystems erläutert und der Gesamtzusammenhang zwischen Nachrichten, Modulen und dem Nachrichtensystem erläutert.

Nachricht

Eine Nachricht wird von der Klasse `Message` abgeleitet. Weiterhin besitzt sie einen *Typ* und eine *Art*. Der Typ kennzeichnet die Nachricht und macht sie damit eindeutig. Die Art gibt an, ob es sich bei der Nachricht um eine Anfrage oder Antwort handelt. Soll eine neue Nachricht definiert werden, so sind folgende Schritte notwendig: es wird eine neue Klasse definiert und von der Basisklasse `Message` abgeleitet. Dann wird in der `Message`-Klasse der Typ der neuen Klasse definiert. Weiterhin definiert die abgeleitete Klasse eine Funktion `getType`. Diese gibt den für diese Message-Klasse eindeutig definierten Nachrichtentyp zurück. Es sei hier angemerkt, dass für die verteilte Kommunikation zusätzliche Erweiterungen an der `Message`-Klasse notwendig sind. Diese werden in Abschnitt 5.4 erläutert.

Nachrichtensystem und Registrierung

Wie in Abschnitt 4.4 beschrieben, arbeitet das Nachrichtensystem Registrierungs-basiert. Jedes Modul kann sich somit für ein oder mehrere Nachrichten an- und ab-melden. Wird eine Nachricht entsprechenden Typs im Nachrichtensystem verarbeitet, so wird sie jedem für diesen Typ registrierten Modul zur Bearbeitung übergeben.

Die Registrierung durch ein Modul geschieht – wie in Abschnitt 5.2.5.4 beschrieben – bei Erstellung eines Moduls. Hierzu stellt das Nachrichtensystem folgende Funktion bereit:

```
void MessagingSystem::subscribe (MessagingNode*      node,
                                Message::MESSAGE_TYPE type)
```

Ein Modul wird nach erfolgter Erstellung und Konfiguration von seinem Channel dazu aufgefordert sich für Nachrichten zu registrieren. Dies ist notwendig, da eine Registrierung für eine Nachricht erst möglich ist nachdem das Modul Zugriff auf das Nachrichtensystem hat. Der Channel ruft hierzu die Funktion `registerMessages` der `MessagingNode`-Klasse auf um das Modul zur Registrierung von Nachrichten aufzufordern:

```
MessagingNode* messenger = dynamic_cast<MessagingNode*>
                                (module.getInternal());
messenger->registerMessages ();
```

Die Modul-Implementierung registriert sich nun für alle Nachrichten welche für sie von Interesse sind:

```
void ModulTest::registerMessages ()
{
    MessagingNode::subscribeMessage (Message::MESSAGE_TEST);
}
```

Dieser Aufruf wird von der `MessagingNode`-Schnittstelle entsprechend auf das Nachrichtensystem weitergeleitet:

```
void subscribeMessage (Message::MESSAGE_TYPE type)
{
    assert (m_conSubscFun.connected ());
    m_subscribeFunction (this, type);
}
```

Der Aufruf des `()`-Operators des `m_subscribeFunction`-Objektes wird durch die Boost Signals Library auf das Nachrichtensystem weitergeleitet (siehe auch Abschnitt 5.2.5.4) und ruft die oben genannte `MessagingSystem::subscribe`-Funktion auf.

Zur Verwaltung der registrierten Nachrichten wird eine Tabelle verwendet. Sie speichert zu jedem Nachrichtentyp alle für diese Nachricht registrierten Module. Entsprechend wird bei Registrierung bzw. Deregistrierung das entsprechende Modul in der Tabelle ein- bzw. ausgetragen. Somit ist die Tabelle wie folgt definiert:

```
typedef vector<MessagingNode*> MESSAGING_NODES;
typedef map<Message::MESSAGE_TYPE, MESSAGING_NODES> MESSAGING_TABLE;
```

Wie zu sehen ist, wurde für den Index der Tabelle der Nachrichtentyp gewählt. Dies ermöglicht eine schnelle Suche aller registrierten Module zu einem gegebenen Nachrichtentyp. Hierfür ist in Kauf zu nehmen, dass – im Falle von Registrierung und Deregistrierung eines Moduls – mehr Zeit in Anspruch genommen werden muss. Die Fälle des Registrieren, Deregistrieren und die Zustellung einer Nachricht werden hier kurz in Bezug auf die vorgestellte Tabelle untersucht und die Laufzeit betrachtet. Hierbei kennzeichnet n die Anzahl der Nachrichten und m die Anzahl an Modulen.

- *Registrierung durch ein Modul für einen definierte Nachrichtentyp:* alle für einen Nachrichtentyp registrierten Module können in $\mathcal{O}(1)$ abgerufen werden. Es muss geprüft werden, ob in der Liste der Module, welche für den gegebenen Nachrichtentyp registriert sind, das Modul noch nicht vorhanden ist. Dies kann durch lineares Durchsuchen der Liste in $\mathcal{O}(m)$ erreicht werden.
- *Deregistrierung durch ein Modul für einen definierten Nachrichtentyp:* Hier dauert – ähnlich der Registrierung – das Aufsuchen aller für den Nachrichtentyp definierten Module $\mathcal{O}(1)$. Das Durchlaufen der Module für den Nachrichtentyp benötigt wiederum $\mathcal{O}(m)$.

- *Deregistrierung durch ein Modul für alle registrierten Nachrichten:* Hierzu müssen alle Nachrichtentypen durchlaufen werden, was $\mathcal{O}(n)$ benötigt. Weiterhin muss für jeden Nachrichtentyp die Liste an Modulen durchlaufen werden, was $\mathcal{O}(m)$ benötigt. Somit ist der Gesamtaufwand $\mathcal{O}(n * m)$.
- *Zustellung einer Nachricht an alle registrierten Module:* Das Bestimmen aller für eine Nachricht registrierten Module benötigt $\mathcal{O}(1)$. Die Zustellung der Nachricht an alle Module entsprechend $\mathcal{O}(m)$.

Wie zu sehen ist, wurde die Tabelle entsprechend so gewählt, dass die Zustellung von Nachrichten an Module schnellstmöglich verarbeitet werden kann, da dies die häufigste Methode ist. Die Anzahl der Registrierungen und Deregistrierungen ist im Vergleich zum Versand und zur Zustellung von Nachrichten als vernachlässigbar zu sehen.

Das Nachrichtensystem bietet die Möglichkeit Nachrichten synchron oder asynchron zu versenden. Die gewünschte Versandart wird im Aufruf der `MessagingNode::sendMessage`-Funktion durch die Modul-Implementierung angegeben. Standardmäßig erfolgt der Versand asynchron. Auf weitere Versandoptionen für die verteilte Kommunikation wird in Abschnitt 5.4 eingegangen. Wird eine Nachricht synchron versendet, so werden alle Module bestimmt welche sich für den Nachrichtentyp registriert haben. Auf jedem dieser Module wird dann die `MessagingNode::receiveMessage`-Funktion aufgerufen und die Nachricht übergeben.

Für den asynchronen Versand existiert eine Warteschlange. Zur asynchronen Zustellung gesendete Nachrichten werden dort eingereiht. Ein Thread verarbeitet die Nachrichten aus der Warteschlange und übergibt sie den für diese Nachricht registrierten Modulen.

Ein Modul sollte in seiner `MessagingNode::receiveMessage`-Funktion die Verarbeitung einer Nachricht schnellstmöglich durchführen. Denn während dieser Zeit ist der synchrone bzw. asynchrone Zweig des Nachrichtensystems entsprechend blockiert und kann keine weiteren Nachrichten zustellen. Im Falle eines synchronen Nachrichtenversands ist neben der synchronen Verarbeitung durch das Nachrichtensystem zusätzlich der Sender blockiert.

5.4 Verteilte Kommunikation

In Abschnitt 5.3 wurde auf den lokalen Versand von Nachrichten zwischen Modulen eingegangen. Da das Framework zur Angriffserkennung – wie in Abschnitt 4.5 beschrieben – auch die verteilte Kommunikation ermöglichen soll, ist weiterhin der Nachrichtenversand zwischen verteilten Instanzen der Angriffserkennung notwendig. Wie auch beim lokalen Versand soll der entfernte Versand beliebige Nachrichtenobjekte verwenden können. Hierzu sollen keine speziellen Paketformate definiert werden müssen. Weiterhin soll die bisher ermöglichte Komplexität der Nachrichten beibehalten werden können: Die in einer Nachricht, welche für die lokale Verwendung entwickelt wurde, enthaltenen Datentypen sollen weiterhin verwendet werden können. Dies betrifft auch komplexe Elemente, wie z.B. die Verwendung von Typen

der Standardbibliothek: `std::list`, `std::vector` oder `std::map`.

Um den Nachrichtenversand zwischen verteilten Instanzen zu ermöglichen, sind Erweiterungen an mehreren Teilen des Frameworks notwendig:

- **MessagingNode**: Zum einen ist eine Erweiterung der Schnittstelle notwendig über welche Modul-Implementierungen Nachrichten versenden können. Dies betrifft auch die Erweiterungen der zur Kopplung der Module an das Nachrichtensystem verwendeten `boost::signals`-Definitionen.
- **Message**: In der Basisklasse **Message** werden neue Definitionen integriert. Diese erleichtern es konkreten Nachrichten-Klassen, sich für den verteilten Nachrichtenversand anzupassen. Die Erweiterung an bestehenden Nachrichten-Klassen ist somit minimal.
- **RemoteMessagingSystem**: Diese neue Klasse ermöglicht den Versand und Empfang von Nachrichtenobjekten. Hierzu müssen beim Versand die Empfänger der Nachricht bestimmt werden. Weiterhin ist eine Serialisierung und Deserialisierung von Nachrichtenobjekten notwendig. Zusätzlich kann hier eine Komprimierung bzw. Dekomprimierung der Byteströme implementiert werden.
- **Communication**: In dieser neuen Klasse wird der eigentliche Versand und Empfang der Byteströme veranlasst. Sie beschreibt eine abstrakte Schnittstelle mit welcher reale Kommunikationsschichten erstellt werden können.
- **MessagingSystem**: Das Nachrichtensystem selbst muss auf zwei Arten erweitert werden. Zum einen müssen Schnittstellen geändert werden und der Versand von Nachrichten an entfernte Instanzen an das **RemoteMessagingSystem** weitergeleitet werden. Zum anderen ist es notwendig dem **RemoteMessagingSystem** Zugriff auf die Nachrichten-Warteschlange zu gewähren um empfangene Nachrichten intern zu verteilen.

Abbildung 5.5 zeigt die genannten Komponenten welche Änderungen oder Erweiterungen unterliegen. Die genannten Änderungen werden in den nun folgenden Abschnitten detailliert erläutert.

5.4.1 MessagingNode

Das **MessagingNode**-Interface bietet einem Modul die Möglichkeit das Nachrichtensystem zu verwenden. Entsprechende Verwendungen der Boost Signals Library – wie in Abschnitt 5.2.5.4 vorgestellt – sind hier gekapselt. Zur Verwendung des verteilten Nachrichtensystems müssen hier Erweiterungen an den `boost::signal`-Typdefinitionen sowie an den Parametern vorgenommen werden, welche von einem Modul verwendet werden um eine Nachricht zu versenden.

Die `sendMessage`-Funktion welche der Modul-Implementierung zur Verfügung gestellt wird ist wie folgt definiert:

```
void MessagingNode::sendMessage (Message* msg, SEND_OPTIONS options);
```

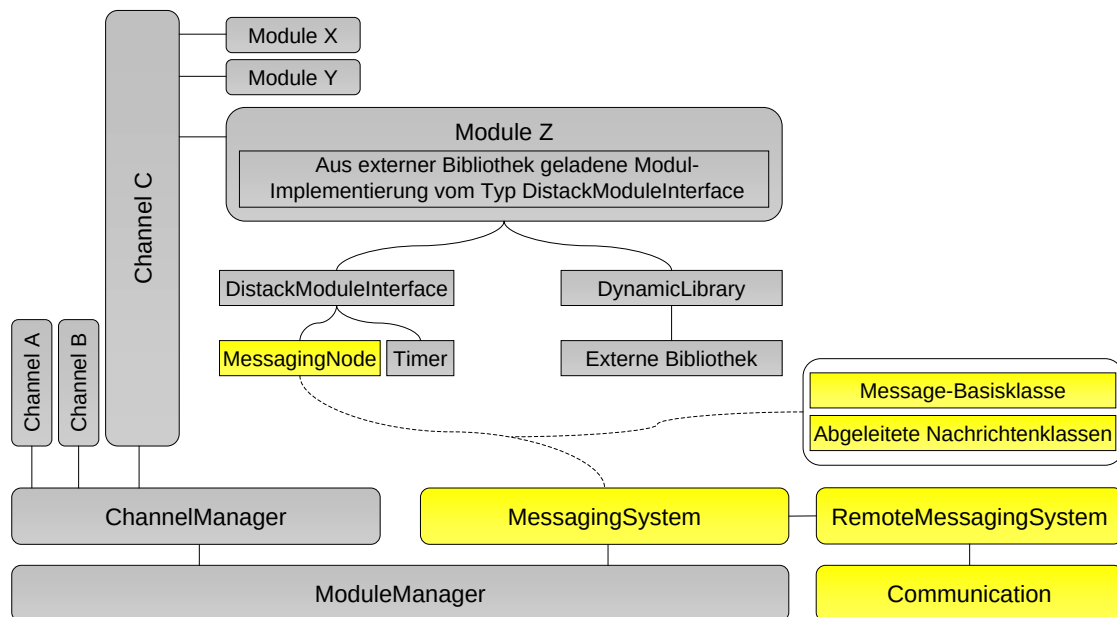


Abbildung 5.5: Notwendige Änderungen entsprechend Abbildung 5.4

Die Funktion ist dieselbe wie beim Versand von lokalen Nachrichten, so dass hier keine Änderungen notwendig sind. Der `SEND_OPTIONS`-Typ hat dabei die folgende Definition:

```
typedef struct _SEND_OPTIONS {
    SEND_SYNC_MODE      syncMode;
    SEND_REMOTE_MODE    remoteMode;
    REMOTE_DESTINATIONS remoteDest;
    IP_ADDR_LIST        remoteList;
} SEND_OPTIONS;
```

Dieser definiert alle Modi, welche für den Versand einer Nachricht – ob lokal oder entfernt – verwendet werden können. Wie bereits in Abschnitt 4.4 erläutert, können Nachrichten synchron oder asynchron versendet werden. Dies wird durch den `SEND_SYNC_MODE`-Typ definiert. Der `SEND_REMOTE_MODE`-Typ gibt an, ob die Nachricht lokal, entfernt oder lokal und entfernt versendet werden soll. Im Falle eines entfernten Versands kann durch den `REMOTE_DESTINATIONS`-Typ bestimmt werden, ob die Nachricht an alle benachbarten Angriffserkennungssysteme oder an eine vordefinierte Menge an Angriffserkennungssysteme gesendet werden soll. Diese Menge kann durch den `IP_ADDR_LIST`-Typ angegeben werden. Soll die Nachricht an alle Nachbarsysteme der Angriffserkennung weitergeleitet werden, so werden keine IP-Adressen angegeben. Es ist dann Aufgabe der verwendeten `Communication`-Komponente die Nachbarfindung durchzuführen und die Nachricht an die entsprechenden Instanzen auszuliefern.

Nachrichten werden von einem Modul immer erst dem lokalen Nachrichtensystem übergeben. Dieses prüft anhand der oben gezeigten Parameter, wie die Nachricht versendet werden soll. Im Falle eines entfernten Versands wird die Nachricht dem in Abschnitt 5.4.3 erläuterten `RemoteMessagingSystem` übergeben.

Neben der beschriebenen `sendMessage`-Funktion existiert eine überladene Funktion, welche wie folgt definiert und implementiert ist:

```
void MessagingNode::sendMessage (Message* msg)
{
    sendMessage (msg, MessagingNode::SEND_OPTIONS::createLocalAsync());
}
```

Hier wird der Versand einer Nachricht standardmäßig lokal und asynchron durchgeführt. Es existieren weitere Funktionen, welche die Generierung von oft verwendeten `SEND_OPTIONS`-Optionen unterstützen.

5.4.2 Message

Das `RemoteMessagingSystem` (siehe Abschnitt 5.4.3) realisiert die Serialisierung der Nachrichtenobjekte. Hierfür wird die Boost Serialization Library [Rame02] verwendet. Sie ermöglicht die Serialisierung beliebiger Objekte in einen Bytestrom. Jede zu serialisierende Klasse muss mit einem kleinen Code-Fragment erweitert werden, welches bei Serialisierung und Deserialisierung aufgerufen wird. Die Basisklasse, von welcher alle neuen Nachrichtenklassen abgeleitet werden, stellt die `Message`-Klasse dar. In dieser wurden zwei Makros definiert, welche die Serialisierung in spezialisierten `Message`-Klassen erleichtern:

```
#define REMOTE_MESSAGING_BEGIN(x,y) \
    friend class boost::serialization::access; \
    template<class Archive> \
    void serialize(Archive& y, const unsigned int version) \
    { \
        boost::serialization::base_object<Message>(*this); \
        boost::serialization::void_cast_register<x, Message> (); \
    } \

#define REMOTE_MESSAGING_END }
```

Unter Verwendung dieser Makros kann die Serialisierung für abgeleitete `Message`-Klassen sehr einfach realisiert werden. Dies wird an einem Beispiel verdeutlicht: wir betrachten eine Nachrichtenklasse `MessageAggregateIntervalValues`, welche Zähler und Schwellenwert der letzten x zeitlichen Intervalle für ein spezielles zu überwachendes Aggregat definiert:

```
typedef pair<unsigned int, double>    COUNT_THRESHOLD_PAIR;
typedef vector<COUNT_THRESHOLD_PAIR> INTERVAL_ITEMS;
```

Die Nachricht besitzt die folgenden Member Variablen, welche die Aggregat-Überwachung, das zu überwachende Protokoll sowie eine Beschreibung definieren:

```

INTERVAL_ITEMS                m_intervals;
Packet::PROTOCOL              m_protocol;
string                        m_description;

```

Mit Hilfe der oben beschriebenen Makros lässt sich der Code zur Serialisierung wie folgt schreiben:

```

REMOTE_MESSAGING_BEGIN(MessageAggregateIntervalValues, remote)
    remote & m_intervals;
    remote & m_protocol;
    remote & m_description;
REMOTE_MESSAGING_END

```

Dieses Code-Fragment wird von der Boost Serialization Library aufgerufen, wenn die entsprechende Klasse serialisiert bzw. deserialisiert werden soll. Die Serialisierung von komplexen Typen – wie den hier verwendeten `std::pair` und `std::vector` – wird durch die Boost Serialization Library durch Einfügen der Header-Dateien `<boost/serialization/utility.hpp>` bzw. `<boost/serialization/vector.hpp>` direkt unterstützt.

Wie in dem Makro `REMOTE_MESSAGING_BEGIN` zu sehen ist, wird auch die Serialisierung der Basisklasse `Message` veranlasst. Dies geschieht durch folgende Zeile:

```

boost::serialization::base_object<Message>(*this);

```

Somit muss auch die `Message`-Basisklasse die Serialisierung unterstützen. Hierzu müssen die Member-Variablen der `Message`-Klasse serialisiert werden:

```

REMOTE_MESSAGING_BEGIN_BASE(remote)
    MESSAGE_TYPE type = getType ();
    remote & type;
    remote & m_kind;
    remote & m_fromRemote;
REMOTE_MESSAGING_END

```

Wie zu sehen ist, wurde die `Message`-Basisklasse weiterhin um das Attribut `m_fromRemote` erweitert. Mit diesem wird gekennzeichnet, ob die Nachricht von einer entfernten Instanz empfangen wurde.

Das `MessagingSystem` verarbeitet Nachrichtenobjekte rein durch die Verwendung von Pointern auf den Basistyp `Message`. Damit die Serialisierung durch einen Pointer auf den Basistyp die Serialisierungsfunktion auf dem spezialisierten `Message`-Objekt – wie zum Beispiel in dem oben beispielhaft eingeführten `MessageAggregateIntervalValues` – ausführt, muss dieses als polymorph gekennzeichnet sein. Dies wird dadurch erreicht, dass mindestens eine Funktion in der Basisklasse `Message` als rein virtuell gekennzeichnet ist. Dies ist hier durch die `getType`-Funktion definiert, welche wie folgt in der Basisklasse `Message` definiert ist:

```
virtual MESSAGE_TYPE getType () = 0;
```

Jede abgeleitete Nachrichtenklasse muss diese Funktion überschreiben und als Rückgabewert den Typ der Nachricht zurückgeben.

5.4.3 RemoteMessagingSystem

Bisher wurden in Abschnitt 5.4.1 die Änderungen an den Schnittstellen für das Senden von Nachrichten durch ein Modul und in Abschnitt 5.4.2 die notwendigen Erweiterungen in der `Message`-Klasse sowie allen spezialisierten Nachrichtenklassen unter Verwendung der Boost Signals Library erläutert. Nun wird darauf eingegangen wie Nachrichten konkret serialisiert bzw. deserialisiert werden und über eine Kommunikationsschicht versendet und empfangen werden.

Jede zu serialisierende Klasse muss der Boost Serialization Library bekannt gemacht werden. Dies wird durch Verwendung des `BOOST_CLASS_EXPORT_GUID`-Makro ermöglicht. Weiterhin wird das Tracking der Nachrichtenobjekte durch das `BOOST_CLASS_TRACKING`-Makro deaktiviert. Dies erleichtert die Implementierung der Serialisierung in den spezialisierten Nachrichtenklassen: normalerweise können nur `const`-Member-Variablen serialisiert werden. Dadurch wird innerhalb der Boost Serialization Library sichergestellt, dass Member-Variablen nicht nach der Serialisierung geändert werden. Da hierdurch jedoch die Handhabung erschwert wird, wird das Tracking deaktiviert. Somit kann ein Makro definiert werden, welches die Registrierung von Nachrichtenklassen ermöglicht sowie das angesprochene Tracking deaktiviert:

```
#define REMOTE_MESSAGING_REGISTER(x)    \
    BOOST_CLASS_EXPORT_GUID(x, #x)      \
    BOOST_CLASS_TRACKING    (x, boost::serialization::track_never)
```

Für jede Nachrichtenklasse, welche durch das verteilte Nachrichtensystem versendet und empfangen werden soll, wird nun dieses Makro verwendet. An der beispielhaften `MessageAggregateIntervalValues`-Klasse sieht dies wie folgt aus:

```
REMOTE_MESSAGING_REGISTER(MessageAggregateIntervalValues)
```

Das Versenden einer Nachricht durch das `RemoteMessagingSystem` läuft wie folgt ab. Zuerst wird das Nachrichtenobjekt serialisiert. Hierzu wird ein Objekt vom Typ `std::stringstream` definiert welches an ein Output-Serialisierungs-Archiv übergeben wird:

```
stringstream archiveStream (ios_base::binary | ios_base::out);
boost::archive::polymorphic_binary_oarchive outputArchive (archiveStream);
```

Der zu serialisierende **Message**-Pointer kann nun in das Archiv serialisiert werden und in Form einer Zeichenkette zur Weiterverarbeitung abgefragt werden:

```
outputArchive & msg;
string data = archiveStream.str ();
```

Durch Verwendung des in der Boost Serialization Library überladenen **&**-Operators wird das in der Nachrichtenklasse – unter Verwendung des **REMOTE_MESSAGING_BEGIN_BASE**-Makros – definierte Code-Fragment aufgerufen und alle Mitglieder der Nachrichtenklasse – wie auch der **Message**-Basisklasse – serialisiert.

Die so erhaltene Zeichenkette wird mitsamt den durch **MessagingNode::SEND_OPTIONS** (siehe Abschnitt 5.4.1) definierten Versandoptionen – an die unterliegende Kommunikationsschicht (siehe Abschnitt 5.4.5) übergeben.

Das **RemoteMessagingSystem** verwendet keine konkrete Kommunikationsschicht. Diese Schicht ist frei austauschbar und durch die in Abschnitt 5.4.5 beschriebene Schnittstelle definiert.

Entsprechend der Serialisierung von zu versendenden Nachrichten wird auch die Deserialisierung von empfangenen Nachrichten realisiert. Die unterliegende Kommunikationsschicht übergibt dem **RemoteMessagingSystem** einen Speicherbereich, in welchen die empfangenen Daten gelegt wurden. Nachdem der Speicherplatz in eine Zeichenkette eingelesen wurde, kann das **Message**-Objekt erstellt werden. Hierzu wird wieder ein Objekt vom Typ **std::stringstream** definiert und einem Input-Serialisierungs-Archiv übergeben.

```
stringstream archivStream (data, ios_base::binary | ios_base::in);
boost::archive::polymorphic_binary_iarchive inputArchive (archivStream);
```

Hierbei stellt die **data**-Variable die empfangene Zeichenkette dar. Die Deserialisierung wird durch Verwendung des **>>**-Operators durchgeführt:

```
Message* msg = NULL;
inputArchive >> msg;
```

Eine empfangene und deserialisierte Nachricht wird direkt in die Warteschlange des **MessagingSystem** (siehe Abschnitt 5.3) eingefügt und dort an die registrierten Module weitergegeben. Weiterhin wird das **m_fromRemote**-Attribut der **Message**-Klasse gesetzt, um den Modulen die Möglichkeit zur Unterscheidung zwischen lokal und entfernt empfangenen Nachrichten zu ermöglichen (siehe Abschnitt 5.4.2).

5.4.4 MessagingSystem

Es bleibt die Frage offen, wie Nachrichten an das `RemoteMessagingSystem` übergeben werden. In dem in Abschnitt 5.4.1 beschriebenen `MessagingNode::SEND_OPTIONS`-Typ kann ein Modul beim Versand einer Nachricht angeben, wie eine Nachricht versendet werden soll. Hier kann ein Modul unter anderem angeben, ob die Nachricht an entfernte Instanzen der Angriffserkennung versendet werden soll. Ist dies der Fall, so wird die Nachricht dem `RemoteMessagingSystem` übergeben. Weiterhin beinhaltet der `MessagingNode::SEND_OPTIONS`-Typ Informationen über die entfernten Empfänger der Nachricht. Diese können auf zwei Arten angegeben werden: entweder anhand einer konkreten Liste von IP-Adressen der Empfänger oder durch Angabe, dass die Nachricht an alle benachbarten Instanzen des Angriffserkennungssystems versendet werden soll. Die vom Modul angegebenen `MessagingNode::SEND_OPTIONS` werden vom `RemoteMessagingSystem` weiter an die unterliegende Kommunikationsschicht übergeben.

5.4.5 Communication

Als unterste Schicht des verteilten Nachrichtensystems muss eine Kommunikationsschicht realisiert werden. Diese sorgt für den eigentlichen Versand und Empfang der Byteströme, welche vom `RemoteMessagingSystem` (siehe Abschnitt 5.4.3) serialisiert bzw. deserialisiert werden.

Um die Kommunikationsschicht einfach austauschen zu können, wird die Schnittstelle `Communication` definiert. Reale Kommunikationsschichten müssen diese implementieren. Sie definiert Datenstrukturen sowie Funktionen, mit welchen das `RemoteMessagingSystem` auf die Kommunikationsschicht zugreifen kann. Die Schnittstelle beinhaltet unter anderem die `write`-Funktion. Sie muss von einer Kommunikationsschicht implementiert werden und ist wie folgt definiert:

```
virtual bool write (DISTACK_REMOTE_DATA      data      ,
                  MessagingNode::SEND_OPTIONS options);
```

Ihr werden die zu versendenden Daten – welche vom `RemoteMessagingSystem` serialisiert wurden – sowie die vom sendenden Modul definierten Sendeoptionen vom Typ `SEND_OPTIONS` übergeben. Die Sendeoptionen beschreiben wie die Nachricht versendet werden soll. Hierbei gibt es zwei Möglichkeit: einmal kann durch Angabe einer Liste an IP-Adressen genau definiert werden, an welche Instanzen die Nachricht versendet werden soll. Wird hier nur eine IP-Adresse angegeben, so wird die Nachricht entweder direkt an diese Instanz gesendet oder, falls eine pfadgebundene Signalisierung von der `Communication`-Komponente unterstützt wird, entlang des Pfades zu dieser IP-Adresse. Als zweite Möglichkeit ist es möglich anzugeben, dass die Nachricht an alle Nachbarinstanzen versendet werden soll. In diesem Fall ist es Aufgabe der verwendeten `Communication`-Komponente die Nachbarfindung durchzuführen und die Nachricht an die entsprechenden benachbarten Instanzen zu übertragen.

Wurden Daten von der `Communication`-Komponente empfangen, so können diese durch Aufruf der in `Communication` definierten `read`-Funktion an das `RemoteMessagingSystem` weitergegeben werden. Dort werden sie deserialisiert und

in das lokale Nachrichtensystem eingespeist und wie lokale Nachrichten verarbeitet.

Für den Versand über das Netzwerk wird ein Paketformat definiert. Es ist hier zu bemerken, dass nur *ein* solches Paketformat existieren muss. Wie bereits in Abschnitt 5.4 erläutert, muss nicht pro Nachricht ein Paketformat definiert werden. Das hier definierte Paketformat ist das einzig notwendige.

```
typedef struct _DISTACK_REMOTE_HEADER {
    unsigned long    magic;
    unsigned long    payloadlen;
    IP_ADDR          source;
    IP_ADDR          dest;

    bool valid () {
        return (magic == DISTACK_REMOTE_MESSAGING_MAGIC);
    }
} DISTACK_REMOTE_HEADER;
```

Der hier definierte `DISTACK_REMOTE_HEADER` wird den zu sendenden Daten vorangestellt. Da der Header eine konstante Länge von 16 Byte besitzt, ist es der empfangenden Instanz möglich zuerst den Header zu lesen und dann zu bestimmen wie lange die zu lesenden Daten sind. Das `magic`-Attribut im `DISTACK_REMOTE_HEADER` wird vor Versand auf einen vordefinierten Wert gesetzt. Somit kann bei Empfang sichergestellt werden, dass auch wirklich ein beginnender `DISTACK_REMOTE_HEADER` gelesen wird und nicht fälschlicherweise andere Daten. Weiterhin enthält das Paketformat Informationen über die Länge der beinhalteten Daten. Dies ist notwendig, da die Serialisierung unterschiedlicher Nachrichten in unterschiedlich langen Bytefolgen resultiert. Als letztes Attribut enthält das Paketformat die IP-Adressen von Sender und Empfänger.

Im Folgenden wird eine beispielhafte Implementierungen der Kommunikationsschicht – ein auf TCP-Sockets basierendes Client-Server-System – vorgestellt.

5.4.5.1 TCP-Socket-basierte Kommunikationsschicht

Wie in Abschnitt 4.5 erwähnt, kann die Communication-Komponente durch eine Vielzahl an Protokollen realisiert werden. Beispielsweise ist eine Implementierung über die TCP-Socket-Schnittstelle möglich, welche hier vorgestellt wird. Jede Instanz der Angriffserkennung agiert im Falle einer TCP-Socket-basierten Kommunikationsschicht als Server – zum Empfang von Nachrichten – und als Client – zum Versand von Nachrichten. Zur Implementierung der TCP-Socket-Schnittstelle wird die Boost Asio Library [Kohl03] verwendet.

Serverseitige Implementierung

Die Server-Instanz wird bei Erstellung der Communication-Komponente gestartet und wartet auf dem Port 53599 auf eingehende Verbindungen. Hierzu wird ein Objekt vom Typ `asio::io_service` – zur Darstellung des Dienstes – sowie ein `asio::tcp::acceptor` – zur Verarbeitung eingehender Verbindungen – benötigt.

Als Objekt für eingehende Verbindungen wird eine Klasse `tcpConnection` erstellt, welche das von `asio::basic_socket_acceptor` definierte Interface implementiert. So kann auf eingehende Verbindungen gewartet und diese in einem neuen Verbindungsobjekt verarbeitet werden.

Bei eingehender Verbindung wird die `handle_accept`-Methode aufgerufen und die Verbindung akzeptiert. Im `m_connection`-Objekt können dann, durch Verwendung der `asio::read`-Funktion, Daten aus der Verbindung gelesen werden. Auf diese Weise wird zuerst der Header vom Typ `DISTACK_REMOTE_HEADER` (siehe Abschnitt 5.4.5) eingelesen und über die `valid`-Funktion validiert. Diese prüft, ob die ersten vier Byte der Daten dem definierten Wert `DISTACK_REMOTE_MESSAGING_MAGIC` entsprechen. Wurde der Header als gültig validiert, so wird, entsprechend der im Header gegebenen Länge, der eigentliche Inhalt des Pakets eingelesen. Die gelesenen Daten werden dann an die `RemoteMessaging`-Komponente nach oben gegeben. Dort werden sie entsprechend Abschnitt 5.4.3 deserialisiert und zur Verarbeitung durch Module in das `MessagingSystem` transportiert.

Clientseitige Implementierung

Wird eine Nachricht von der `RemoteMessaging`-Komponente an die Communication-Komponente übergeben, so wurde diese bereits serialisiert und die Empfängeradressen bestimmt. Wurden mehrere Empfänger ausgemacht, so wird die Communication-Komponente mehrfach aufgefordert die Nachricht zu übertragen. Somit muss die Communication-Komponente eine Menge an Bytes an eine definierte Adresse übertragen. Hierzu muss zuerst der Endpunkt der Verbindung zum Server bestimmt werden. Dieser ist im Falle eines Hostnamen nicht zwingend eindeutig. Wurde der Endpunkt gefunden, so wird dieser geöffnet.

Zuerst wird ein Header vom Typ `DISTACK_REMOTE_HEADER` erstellt und über die Verbindung gesendet. Dann werden die eigentlichen Daten gesendet welche die serialisierte Nachricht darstellen. Somit ist die Übertragung der Nachricht abgeschlossen. Sie kann auf der Server-Seite entsprechend entgegengenommen, deserialisiert und an höhere Schichten weitergeleitet werden.

5.5 Konfiguration

In Abschnitt 4.6 wurde bereits grob auf den Aufbau der Konfiguration eingegangen. In diesem Abschnitt wird das Format und die Implementierung der Konfiguration beschrieben. Abschnitt 5.5.1 stellt das allgemeine Format und die zur Verwaltung verwendete Triggeronf-Bibliothek dar. Unter Verwendung dieses Formats wird dann in den Abschnitten 5.5.2-5.5.4 die Struktur der Konfiguration für das Framework dargestellt. Hierbei wird noch einmal das Baukastenprinzip – welches durch Module realisiert wird – verdeutlicht.

5.5.1 Konfigurationsformat und die Triggeronf-Bibliothek

Die Triggeronf-Bibliothek wurde in Fortführung der Arbeiten von [Maye06] entwickelt um eine XML-basierte Konfiguration von Systemen zu ermöglichen. Dabei wurde ein hierarchischer Ansatz verfolgt. Dieser erlaubt die Gliederung der

Konfiguration in die Ebenen `module`, `submodule` und `configitem`. Die `module`-Konfigurationsebene darf nicht mit den in dieser Arbeit vorgestellten Modulen verwechselt werden. Eine Konfiguration kann beliebig viele `module`-Elemente besitzen. Um sie nicht mit Modulen der Angriffserkennung zu verwechseln, wird der Name `module` klein geschrieben. Die genannten `module`-Elemente spiegeln die Architektur des Systems wieder und dienen zur ersten Gliederung der Konfiguration. Ein `module`-Element stellt im Fall des Distack-Frameworks z. B. die allgemeine Konfiguration (siehe Listing 5.2). Die zweite Gliederungsebene ist das `submodule`-Element. Auch hier sind beliebig viele `submodule`-Elemente in einem `module`-Element möglich. In Listing 5.2 existiert hier das `submodule`-Element *Network* zur Konfiguration der verwendeten Framequelle. Die letzte Gliederungsebene stellen die `configitem`-Elemente dar. Sie enthalten die eigentliche Information, welche zur Konfiguration verwendet wird. Das Distack-Framework beinhaltet zum Beispiel ein `configitem`-Element *Interface* zur Angabe des zu verwendenden Netzwerk-Interface.

Jedes der vorgestellten Elemente `module`, `submodule` und `configitem` besitzt einen Namen. Die Elemente `submodule` sowie `configitem` können weiterhin eine beliebige Anzahl an Attributen besitzen. Dies sind Name-Wert-Paare, die zusätzliche Informationen beinhalten. Die eigentliche Konfigurationsinformation ist in den `configitem`-Elementen gespeichert. Sie besitzen einen Wert, welcher die Konfigurationsinformation darstellt.

Die Triggerconf-Bibliothek bietet ein breites Spektrum an Funktionen um die vorgestellte Konfigurationsstruktur zu bearbeiten. Intern wird dabei automatisch sichergestellt, dass die verwendete XML-Struktur stets gültig ist. Der Benutzer hat damit alle Vorzüge einer hierarchischen Konfiguration, ohne sich um das eigentliche XML-basierte Format kümmern zu müssen. Der Zugriff auf `configitem`-Elemente ist unter anderem über ein Pfad-Format möglich. Das Pfad-Format besitzt folgende Form: `module/submodule/configitem`.

Weiterhin besitzt die Triggerconf-Bibliothek einen Modus zur automatisierten Erstellung von Elementen. Soll beispielsweise ein `configitem`-Element mit einem Wert gespeichert werden, so werden `submodule`- und `module`-Elemente – falls diese noch nicht existieren – automatisch erstellt.

Für die Angriffserkennung wurde eine Hierarchie aus drei `module`-Elementen gewählt:

- *General*: Definiert allgemeine Einstellungen zum Betrieb des Frameworks (siehe Abschnitt 5.5.2).
- *Modules*: Beschreibt alle als Bibliotheken verfügbare Module sowie ihre Parameter (siehe Abschnitt 5.5.3).
- *Channels*: Definiert die verfügbaren Channels, welche aus einer Menge an Modulen bestehen (siehe Abschnitt 5.5.4). Dies ist somit eine Zusammensetzung aus Elementen der *Modules*-Hierarchieebene.

5.5.2 Allgemeine Konfiguration

Die *General*-Konfiguration besteht aus zwei **submodule**-Elementen. Diese Beschreiben die Konfiguration des Netzwerk-Interfaces – **submodule**-Element *Network* – sowie Informationen über benachbarte Instanzen der Angriffserkennung – **submodule**-Element *Remote*). Die XML-basierte Struktur der *General*-Konfiguration ist in Listing 5.2 dargestellt.

Das **submodule**-Element *Network* enthält die folgenden Elemente:

- *Interface*: Der Name des Netzwerk-Interface welches als Frame-Quelle geöffnet werden soll. Soll eine Tracedatei gelesen werden, so muss dieses Element leer sein.
- *Promiscuous*: Falls ein Netzwerk-Interface geöffnet werden soll, kann hier angegeben werden, ob das Netzwerk-Interface im Promiscuous Mode geöffnet werden soll.
- *TraceFile*: Der Name der Tracedatei, welche als Frame-Quelle geöffnet werden soll. Soll ein Netzwerk-Interface verwendet werden, so muss dieses Element leer sein.
- *TraceReplaySpeedup*: Falls eine Tracedatei als Frame-Quelle verwendet werden soll, so kann hier ein Beschleunigungsfaktor angegeben werden (siehe auch Abschnitt 5.1.1.1). Entsprechend diesem Faktor wird das Abspielen der Tracedatei beschleunigt (Faktor > 1.0) bzw. verlangsamt ($0.0 < \text{Faktor} < 1.0$).
- *ChecksumCheck*: Gibt an, ob die Prüfsummen von Protokollen überprüft werden sollen (Wert 1) oder nicht (Wert 0). Pakete mit ungültigen Prüfsummen werden dann entsprechend gekennzeichnet.

Das **submodule**-Element *Remote* enthält Informationen über benachbarte Instanzen der Angriffserkennung. Dies ist unter Umständen für die Konfiguration des verteilten Nachrichtensystems notwendig, falls die verwendete Kommunikationsschicht keine Nachbarfindung bereitstellt (siehe Abschnitt 4.5). Das **configitem**-Element *Neighbors* enthält eine durch Kommas getrennte Liste mit IP-Adressen der entfernten Instanzen.

Nachfolgend ist eine beispielhafte Konfiguration gezeigt. In dieser wird als Frame-Quelle ein Netzwerk-Interface mit dem Namen **eth1** konfiguriert. Weiterhin sind Informationen über benachbarte Instanzen der Angriffserkennung vorhanden. Dies ist notwendig, falls die verwendete **Communication**-Komponente keine Nachbarfindung unterstützt.

5.5.3 Konfiguration der Modul-Definitionen

Das in Listing 5.3 dargestellte **module**-Element *Modules* beschreibt die verfügbaren Module. Ein als Bibliothek erstelltes Modul kann dabei durch ein **submodule**-Element instanziiert werden. Dies bedeutet, dass eine verfügbare Modul-Bibliothek mehrfach verwendet werden kann.

```

<module name="General">
  <submodule name="Network">
    <configitem name="Interface">eth1</configitem>
    <configitem name="Promiscuous">1</configitem>
    <configitem name="TraceFile"></configitem>
    <configitem name="TraceReplaySpeedup"></configitem>
    <configitem name="ChecksumCheck">1</configitem>
  </submodule>
  <submodule name="Remote">
    <configitem name="Neighbors">82.165.97.37,
                                141.3.70.4</configitem>
  </submodule>
</module>

```

Listing 5.2: Aufbau der allgemeinen Konfiguration

```

<module name="Modules">
  <submodule library="ModuleSampling_noutOfN"
              name="SamplingChannel1">
    <configitem name="n">5</configitem>
    <configitem name="N">200</configitem>
  </submodule>
  ...
</module>

```

Listing 5.3: Konfiguration zur Verwendung einer Modul-Bibliothek

Es wird ein Modul mit dem Namen `SamplingChannel1` definiert. Dieses wird aus der Bibliothek `ModuleSampling_noutOfN` erstellt. Bei Erstellung des Moduls werden die beiden Parameter `n` und `N` mit ihren Werten 5 und 200 übergeben. Falls ein weiteres Modul aus der gleichen Bibliothek `ModuleSampling_noutOfN` benötigt wird, so kann dies entsprechend mit einem anderen Namen und anderen Parametern durch die Konfiguration erstellt werden. Die Bibliothek muss dabei persistent als Shared Library bzw. Dynamic Link Library (siehe auch Abschnitt 2.6) verfügbar sein. In dem oben genannten Beispiel bedeutet dies, dass eine Datei `ModuleSampling_noutOfN.so` bzw. `ModuleSampling_noutOfN.dll` existieren muss. Diese implementiert das in Abschnitt 5.2.5.3 beschriebene Interface.

Die Anzahl der möglichen Parameter ist unbegrenzt und richtet sich entsprechend nach der verwendeten Bibliothek. Die Parameter werden bei Erstellung des Moduls diesem übergeben. Es ist Aufgabe der Modul-Implementierung diese Parameter auf Vollständigkeit und Korrektheit zu testen. Das Laden eines Moduls kann somit – im Falle einer fehlerhaften Konfiguration – auch scheitern.

Die *Modules*-Konfiguration entsprechend dem `module`-Element definiert noch keine konkrete Verwendung des durch die Bibliothek erstellten und entsprechend konfigurierten Moduls. Sie beschreibt lediglich die zur Verfügung stehenden Modul-Instanzen, welche aus Bibliotheken erstellt werden können.

5.5.4 Konfiguration zur Verschaltung der Module

Bisher wurden in Abschnitt 5.5.3 die verfügbaren Instanzen von Bibliotheken samt Parameter definiert und durch einen Namen gekennzeichnet. Der vergebene Name wird nun verwendet um die Module in Channels (siehe auch Abschnitt 5.2.4) zusammenzubauen. Ein Channel gruppiert Module und benötigt somit keine externen Elemente wie zum Beispiel Bibliotheken. Jedes im `modules`-Element *Modules* definierte Modul kann nun über seinen Namen verwendet werden. Hierzu wird ein `submodule`-Element definiert. Dieses besitzt die Attribute `name` – welches den Namen des Channels definiert – sowie `stage`. Das `stage`-Attribut definiert die Zugehörigkeit des Channels zu einer Stufe (siehe auch Abschnitt 4.1). Mehrere Channels können somit über die Stufennummer weiter gruppiert werden. Da das Angriffserkennungssystem immer ganze Stufen lädt bzw. entlädt, werden somit immer alle Channels mit gleicher Stufennummer ge- bzw. entladen.

Das in Listing 5.4 beispielhaft dargestellte `submodule`-Element definiert einen Channel mit Namen *StageOne*. Er gehört der Stufe Eins an und beinhaltet die Module *SamplingChannel1* und *ThresholdCollector*. Die hier verwendeten Namen wurden in der *Modules*-Konfiguration in Listing 5.3 definiert.

```
<module name="Channels">
  <submodule name="StageOne" stage="1">
    <configitem name="1">SamplingChannel1</configitem>
    <configitem name="2">ThresholdCollector</configitem>
  </submodule>
  ...
</module>
```

Listing 5.4: Konfiguration zur Verschaltung von Modulen

Die verwendeten Module *SamplingChannel1* und *ThresholdCollector* müssen entsprechend im `module`-Element *Modules* definiert sein. Beim Laden des Channels, werden die beiden Module entsprechend aus den Bibliotheken geladen und konfiguriert. Die Reihenfolge der Module im Channel *StageOne* entspricht der konfigurierten Reihenfolge im `submodule`-Element mit Namen *StageOne*. Daraus folgt, dass die Verarbeitung von Frames zuerst durch das Modul *SamplingChannel1* und danach durch das Modul *ThresholdCollector* vorgenommen wird. Daher ist es für das Modul *SamplingChannel1* möglich, zu verhindern, dass bestimmte Frames zur weiteren Verarbeitung an Modul *ThresholdCollector* weitergegeben werden. Durch die lineare Verkettung von Modulen in einem Channel sind somit Sampling- und Filtering-Module implementierbar.

Die Anzahl der definierbaren Channels durch `submodule`-Elemente ist nicht beschränkt. Auch die Gruppierung in Stufen ist beliebig anwendbar. Somit entsteht eine hoch flexible Konfiguration welche die Verwendung von Bausteinen ermöglicht. Bibliotheken können mehrfach durch die Konfiguration verwendet und konfiguriert werden. Die Zusammensetzung von Modulen zu Channels ist wiederum rein konfigurationsabhängig und sehr flexibel.

5.6 Hilfsklassen

Neben den eigentlichen Funktionalitäten zur Integration von Angriffserkennung, bietet das Framework zusätzliche Hilfsklassen. Diese werden einmal innerhalb des Framework selbst verwendet, weiterhin erleichtern sie die Entwicklung von Modulen, welche die eigentliche Angriffserkennung implementieren. Diese Utilities werden in den folgenden Abschnitten kurz vorgestellt.

5.6.1 Logging

Die Ausgabe von Programminformationen zur Laufzeit stellt einen der wichtigsten Mechanismen für Entwickler und Anwender zur Verfügung. Durch ein ausgereiftes Logging-System ist es möglich – durch Funktionsaufrufe innerhalb des Quellcodes – nach Wichtigkeit abgestufte Informationen in verschiedene Senken zu schreiben.

Damit das Logging von allen Programmteilen einheitlich verwendet werden kann, wurde es als Singleton implementiert. Dies bedeutet, dass im gesamten Prozess ein einziges Objekt des Typs `Log` existiert. Die `Log`-Klasse bietet statische Methoden, welche intern auf das Singleton-Objekt weitergeleitet werden. Somit sind einfache Aufrufe möglich.

Die `Log`-Klasse bietet vier verschiedene Funktionen zur Ausgabe von Logging-Informationen:

- `Log::debug`-Funktion: Zur Ausgabe von Programminformationen welche vor allem für Entwickler von Interesse sind um die Programmfunktionalität zu überwachen
- `Log::info`-Funktion: Für Informationen, welche für das Verständnis des Ablaufs der Angriffserkennung von Interesse sind
- `Log::warning`-Funktion: Ausgabe von Warnungen, welche auf einen nicht einwandfreien Ablauf des Systems schließen lassen
- `Log::error`-Funktion: Für die Ausgabe von Fehlermeldungen bei schwerwiegenden Fehlern, welche eine fehlerhafte Funktion des Systems bewirken

Die Ausgabe der hier vorgestellten Funktionen der `Log`-Klasse können stufenweise angepasst werden. Hierfür wird das Logging mit einem Loglevel konfiguriert. Entsprechend der vorgestellten Log-Funktionen existieren die Loglevel `TYPE_DEBUG`, `TYPE_INFO`, `TYPE_WARNING`, `TYPE_ERROR`. Die Loglevel schließen sich gegenseitig ein (siehe Abbildung 5.6). Das Loglevel kann durch die Funktion `Log::setLogLevel` gesetzt werden. Unter Verwendung von `TYPE_WARNING` werden beispielsweise alle Logausgaben der Funktionen `Log::debug` und `Log::info` ignoriert.

Bisher wurden Funktionen vorgestellt, welche zur Konfiguration und zur Ausgabe von Logging-Informationen verwendet werden. Als Datenziel der Logging-Informationen existiert die Möglichkeit entweder vordefinierte Klassen zu verwenden, oder eigene zu entwickeln. Vordefinierte Klasse sind einmal `StdOutLogOutputter` zur Ausgabe von Logging-Informationen auf der Konsole und `FileLogOutputter` zur

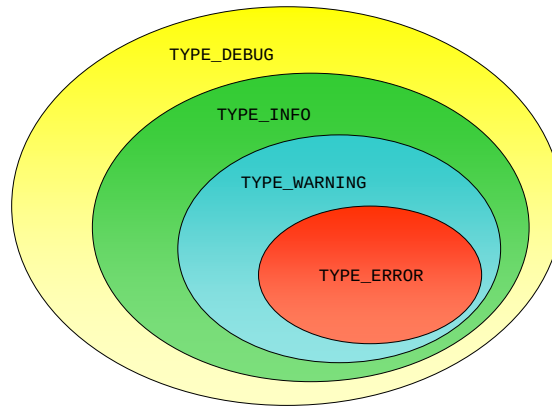


Abbildung 5.6: Verfügbare Loglevel und ihre Einschließung

Ausgabe der Logging-Informationen in eine Datei. Zur Entwicklung neuer Ausgabefunktionalität muss die `LogOutputter`-Schnittstelle beachtet werden. Sie definiert die folgende Schnittstellenfunktion:

```
void output (OUTPUT_OBJ* outobj);
```

Der `OUTPUT_OBJ`-Typ ist hierbei wie folgt definiert:

```
typedef struct _OUTPUT_OBJ {
    string          date; // output date
    string          time; // output time
    LogLevel::LOG_LEVEL type; // type of the output
    string          text; // main text of the output
} OUTPUT_OBJ, *POUTPUT_OBJ;
```

Durch die Funktionen `addLogOutput` und `removeLogOutput` der `Log`-Klasse können beliebig viele Objekte, welche die Schnittstelle des `LogOutputter` implementieren, hinzugefügt werden. Allen registrierten Objekten werden die Logausgaben übergeben, so dass diese ihre Operationen darauf ausführen können.

5.6.2 Routing-Tabelle

Ein Modul – welches einen Teil der Angriffserkennung implementiert – benötigt möglicherweise Zugriff auf die lokale Routing-Tabelle. Daher müssen Funktionen vorhanden sein, um Zugriff auf die Routing-Tabelle – auf verschiedenen Plattformen – komfortabel zur Verfügung zu stellen. Es werden hier drei verschiedene Möglichkeiten implementiert um unter Verwendung der `GenericRoutingTable`-Klasse Zugriff auf die Routing-Tabelle zu erhalten. In Abschnitt 6.3.1.5 wird die `GenericRoutingTable`-Klasse für die OMNeT++-Umgebung erweitert.

Um einheitlichen Zugriff für die hier vorgestellten verschiedenen Möglichkeiten der Routing-Tabelle zu ermöglichen, werden neue Typen definiert, welche die Information der Routing-Tabelle beinhalten:

```
typedef struct _ROUTING_ENTRY {
    IP_ADDR addr;
    IP_ADDR mask;
} ROUTING_ENTRY, *PROUTING_ENTRY;

typedef vector<ROUTING_ENTRY>ROUTING_TABLE;
```

Diese gewähren den Zugriff auf die Routing-Tabelle über die folgende Funktion:

```
ROUTING_TABLE getRoutingTable (ROUTING_SOURCE source,
                                string filename = "");
```

Die Quelle für die Routing-Tabelle kann über das `source`-Argument vom Typ `ROUTING_SOURCE` bestimmt werden:

```
typedef enum _ROUTING_SOURCE {
    ROUTING_SOURCE_SYSTEM,
    ROUTING_SOURCE_FILE,
} ROUTING_SOURCE;
```

Somit kann bestimmt werden, ob die Routing-Information aus einer Datei oder aus der Laufzeitumgebung ausgelesen werden soll.

Implementierung für Dateien

Unter Verwendung der in Abschnitt 5.5.3 beschriebenen Modul-Konfiguration kann einem Modul eine Datei angegeben werden, welche Routing-Informationen beinhaltet. Diese kann dann verwendet werden, um durch die `GenericRoutingTable`-Klasse die Routing-Informationen auszulesen. Das Format der Datei muss in der CIDR-Notation [FuLi06] gegeben sein und einen Eintrag pro Zeile enthalten.

Implementierung für Linux

Linux ermöglicht den Zugriff auf die Routing-Tabelle unter anderem über die Datei `/proc/net/route`. Von Interesse sind hier vor allem die zweite und dritte Spalte des Formats der Datei. Sie geben die Ziel-IP-Adresse sowie die IP-Adresse des Gateways an. Über diese Informationen kann wieder entsprechend die Routing-Tabelle – ähnlich wie für eine externe Datei – ausgelesen und zur Verfügung gestellt werden.

Implementierung für Windows

Die Windows Win32-API stellt direkte Funktionen zum Auslesen der Routing-Tabelle zur Verfügung. Unter Verwendung der `GetIpForwardTable`-Funktion [Isem00] aus der IP-Helper-API kann diese ausgelesen werden.

5.6.3 Patricia-Trie

Ein Patricia-Trie [Morr68] ist eine in der Telematik häufig benötigte Datenstruktur zur Speicherung von Daten unter Verwendung von IP-Adressen als Schlüssel. Nach außen hin arbeitet ein Patricia-Trie ähnlich wie ein normaler Baum. Jedoch sind die Elemente in einem Patricia-Trie entsprechend dem längsten Präfix komprimiert. Daher eignet sich ein Patricia-Trie sehr gut um zum Beispiel Routing-Informationen zu strukturieren.

Die C++-Standardbibliothek bietet keine Implementierung eines Patricia-Trie. Die verfügbaren Implementierungen können nicht mit beliebigen Schlüssel-/Wert-Typen verwendet werden. Daher wurde die in [Grui05] entwickelte Implementierung weiterentwickelt um Template-basierte Schlüssel- und Werte-Typen zu ermöglichen.

5.6.4 Hilfsfunktionen

Neben den bisher besprochenen Hilfsklassen gibt es noch die Hilfsklasse `Helper`. Sie bietet vor allem Funktionen zur Konvertierung von Datentypen. Eine in der C++-Standardbibliothek fehlende Gruppe an Funktionalität ist die einfache Konvertierung von und nach Zeichenketten. Daher bietet die `Helper`-Klasse einfach zu verwendende Funktionen, welche die teilweise aufwendigen Konvertierungen mit Hilfe der in der Standardbibliothek zur Verfügung stehenden Funktionen kapselt. Weiterhin befinden sich in der `Helper`-Klasse Funktionen zur Manipulation von Zeichenketten. Tabelle 5.4 gibt einen Überblick über die verfügbaren Funktionen sowie ihren Verwendungszweck.

5.7 Standardmodule

Neben der Möglichkeit zur Entwicklung von Modulen zur Angriffserkennung (siehe auch Abschnitt 5.2.5), bietet das Framework eine Menge an fertigen Modulen, welche direkt verwendet werden können. Dies sind Module, welche häufig verwendete Funktionalität implementieren. Die hier vorgestellten Module sind nicht mit den in Abschnitt 5.6 beschriebenen Utilities zu verwechseln: Utilities stellen eine Sammlung an Funktionalität zur Verfügung, welche an jeder Stelle des Framework verwendet werden kann. Standardmodule implementieren konkrete Module, welche in Channels gruppiert und betrieben werden können.

Im folgenden Abschnitt 5.7.1 werden allgemeine Hilfs-Module, in Abschnitt 5.7.2 Sampling- und Filtering-Module erläutert.

5.7.1 Hilfs-Module

In diesem Abschnitt werden drei Module vorgestellt, welche die allgemeine Arbeit mit dem Angriffserkennungssystem erleichtern. Sie bieten Funktionalität für die folgenden Aufgaben:

- `ModuleOutputterPlotter`: Rahmenwerk für Live-Plotting über gnuplot
- `ModuleUtilityIpLister`: Erstellung von Routing-Tabellen für eine Tracedatei oder Live-Verkehr
- `ModuleUtilityTimer`: Timer-Modul, welches über Timer-Nachrichten mehrere Module synchronisiert

Name	Beschreibung
<code>ultos</code>	Konvertiert einen <code>unsigned long</code> in einen <code>string</code>
<code>ptos</code>	Template-basierte Methode um die Speicheradresse eines Objektes in hexadezimaler Schreibweise als <code>String</code> zu konvertieren
<code>ltos</code>	Konvertiert einen <code>long</code> in einen <code>string</code>
<code>ultohexs</code>	Konvertiert einen <code>unsigned long</code> in einen <code>string</code> in hexadezimaler Schreibweise
<code>ltohexas</code>	Konvertiert einen <code>long</code> in einen <code>string</code> in hexadezimaler Schreibweise
<code>stol</code>	Konvertiert einen <code>string</code> in einen <code>long</code>
<code>stoi</code>	Konvertiert einen <code>string</code> in einen <code>int</code>
<code>stod</code>	Konvertiert einen <code>string</code> in einen <code>double</code>
<code>hstoui</code>	Konvertiert einen <code>string</code> in hexadezimaler Schreibweise in einen <code>unsigned int</code>
<code>dtos</code>	Konvertiert einen <code>double</code> in einen <code>string</code>
<code>trim</code>	Schneidet vor und nach einem <code>string</code> Leerzeichen und nicht sichtbare Zeichen ab
<code>split</code>	Zerteilt einen <code>string</code> entsprechend einem übergebenen Trennzeichen in eine Liste
<code>replace</code>	Ersetzt in einem <code>string</code> einen <code>string a</code> durch einen <code>string b</code>
<code>getTime</code>	Gibt die Zeit in einem lesbaren Format als <code>string</code> zurück
<code>getDate</code>	Gibt das Datum in einem lesbaren Format als <code>string</code> zurück

Tabelle 5.4: Hilfsfunktionen aus der Helper-Klasse

5.7.1.1 ModuleOutputterPlotter

Das `ModuleOutputterPlotter`-Modul ermöglicht Live-Plotting über das `gnuplot`-Programm [WiKe86]. Zusätzlich werden die Messwerte in eine Ausgabedatei geschrieben um sie später weiterverwenden zu können. Als Konfigurationselement wird dem Modul ein `configitem`-Element mit Namen `Filename` übergeben. Dies ist die anzulegende Datei, in welche die Messwerte geschrieben werden.

Nachdem das Modul über die `configure`-Methode (siehe auch Abschnitt 5.2.5.3) mit dem Dateinamen konfiguriert wurde, wird das `gnuplot`-Programm gestartet. Um `gnuplot` aus dem Modul heraus zu steuern, wird die Funktion `popen` verwendet. Diese öffnet eine Pipe zum `gnuplot`-Programm und ermöglicht so die Steuerung. Nach Öffnen der Pipe wird `gnuplot` für die Anzeige über ein Fenster und mit dem `Plot`-Kommando konfiguriert. Zur grafischen Anzeige wird unter Linux das `x11`-Terminal und unter Windows das `windows`-Terminal verwendet. Das `Plot`-Kommando muss entsprechend der gewünschten Verwendung konfiguriert werden. Die Messwerte können nun über eine Nachricht empfangen werden – unter Verwendung der Funktionen `MessagingNode::subscribeMessage` – oder selbst durch die `frameCall`-Methode aus dem `Frame`-Strom extrahiert werden. Dies ist sinnvoll, falls zum Beispiel die Anzahl an `Frames` oder andere einfache Aggregate visualisiert werden sollen. Nachdem die Messwerte in die durch `Filename` vorgegebene Datei geschrieben wurden, kann das Diagramm neu geplottet werden.

Es wird hier deutlich, dass das `ModuleOutputterPlotter`-Modul für die eigene Ver-

wendung angepasst werden muss, da die Datensammlung Anwendungsfall-spezifische ist.

5.7.1.2 ModuleUtilityIpLister

In Abschnitt 5.6.2 wurde erläutert, wie die Routing-Tabelle aus dem System oder einer Datei einheitlich ausgelesen und verwendet werden kann. Will man Tracedateien als Frame-Quelle verwenden (siehe auch Abschnitt 5.5.2), so sind unter Umständen keine Routing-Informationen verfügbar. Diese können mit dem `ModuleUtilityIpLister`-Modul grob erstellt werden. Hierzu analysiert das Modul in seiner `frameCall`-Methode alle IP-Pakete und sammelt alle IP-Adressen. Über eine vordefinierte Bitmaske können die IP-Adressen zu Subnetzen gruppiert werden.

Die Konfiguration des Moduls beinhaltet die beiden `configitem`-Elemente `OutFilename` und `Masking`. Das Konfigurationselement `OutFilename` definiert die Ausgabedatei für die Routing-Tabelle. Diese wird entsprechend dem in Abschnitt 5.6.2 erläuterten Format geschrieben. Durch das Konfigurationselement `Masking` wird eine Bitmaske in Form einer IP-Adresse angegeben. Wird dort beispielsweise

255.255.255.0 angegeben, so werden alle IP-Adressen zu Class C Netzwerken zusammengefasst.

Das `ModuleUtilityIpLister`-Modul bietet somit die Möglichkeit bei nicht vorhandener Routing-Tabelle diese grob zu generieren. Es ist hervorzuheben, dass die resultierende Routing-Tabelle nur unter bestimmten Umständen der realen Routing-Tabelle entspricht. In den meisten Fällen wird dies nicht der Fall sein. Eine gewisse Ähnlichkeit mit der realen Routing-Tabelle ist jedoch in den meisten Fällen anzunehmen.

5.7.1.3 ModuleUtilityTimer

Wie in Abschnitt 5.2.5.5 beschrieben, hat jedes Modul die Möglichkeit Timer-Funktionalität zu verwenden. Sollen mehrere Module bezüglich ihrer Timer synchron laufen, so genügt es nicht, diese mit dem gleichen Timer-Intervall zu konfigurieren. Grund dafür ist, dass die Module zu unterschiedlichen Zeiten geladen werden können. Auch das Laden eines Channels lädt die enthaltenen Module in einem seriellen Ansatz. Die Synchronisation von Modulen aus verschiedenen Channels oder sogar unterschiedlichen Stages kann nur über das hier vorgestellte `ModuleUtilityTimer`-Modul erreicht werden. In Verbindung mit dem in Abschnitt 5.4 vorgestellten Nachrichtensystem ergibt sich die Möglichkeit auch Module entfernter Instanzen zu synchronisieren.

Das `ModuleUtilityTimer`-Modul besitzt einen eigenen Timer. In seiner `Timer::eventFunction`-Funktion des Moduls versendet das Modul eine Nachricht vom Typ `MessageTimer`. Diese beinhaltet den Namen des Timers – welcher in der Konfiguration des `ModuleUtilityTimer`-Moduls gesetzt werden kann – sowie einen Zähler, welcher nach jedem Timer-Intervall erhöht wird. Module können nun dadurch synchronisiert werden, dass sie sich für die `MessageTimer`-Nachricht registrieren, und den Namen des verwendeten `ModuleUtilityTimer`-Modul überprüfen. Somit wird in den Modulen die Funktionalität der `Timer::eventFunction`-Methode

in die `MessagingNode::receiveMessage`-Methode übertragen.

Mit dem beschriebenen Mechanismus können mehrere Module synchronisiert werden. Jedes Modul kann jedoch auch zusätzlich seinen eigenen Timer verwenden, falls dies benötigt wird. Dieser ist von dem vorgestellten Verfahren unabhängig. Eine weitere interessante Möglichkeit, welche sich ergibt, ist die Synchronisation von entfernten Modulen. Wie in Abschnitt 5.4 erläutert, können Nachrichten an entfernte Instanzen des Frameworks versandt werden. Somit können entfernte Module mit dem gleichen `ModuleUtilityTimer`-Modul synchronisiert werden. Offensichtlich handelt es sich hier um eine grobe Synchronisation die nicht verwendet werden kann, falls die Module sich im Bereich von Millisekunden synchronisieren müssen.

Auf welcher Instanz das `ModuleUtilityTimer`-Modul betrieben wird ist unerheblich. Werden mehrere verschiedene Timer benötigt um verschiedene Module einer Gruppe von Instanzen unterschiedlich zu synchronisieren, kann zum Beispiel eine Instanz betrieben werden, welche keine Funktionalität zur Angriffserkennung verwendet, sondern mehrere `ModuleUtilityTimer`-Moduls betreibt. Diese synchronisieren über Nachrichten die Module der verteilten Instanzen. Hier wird deutlich, dass die verteilte Kommunikation weit mehr Möglichkeiten bietet als die verteilte Erkennung von Angriffen: sie bietet die Möglichkeit das System selbst zu verteilen und somit die auf einem System betriebenen Module der Ressourcenkapazität des Vermittlungssystems anzupassen. Es entsteht auf diese Weise ein verteiltes System.

5.7.2 Sampling- und Filtering-Module

In Abschnitt 2.2 wurde die Notwendigkeit von Sampling- und Filtering-Mechanismen beschrieben. Durch das in Abschnitt 4.3 und 5.2.4 beschriebene Konzept der linearen Verkettung von Modulen in einem Channel sind Sampling- und Filtering-Module problemlos in das Framework integrierbar. Sie kennzeichnen durch den Rückgabewert ihrer `frameCall`-Methode, ob ein Frame in ihrem Channel an die verbleibenden Module weitergeleitet werden soll oder nicht. Durch die lineare Verkettung sind auch Kombinationen von Sampling- und Filtering-Modulen möglich. Weiterhin können Module, welche sich weiter vorne im Channel befinden, anders gefilterte bzw. gesampelte Daten verwenden, als Module weiter hinten im Channel. Hiermit existiert die Möglichkeit, die Funktionalität in einem Channel abgestuft mit Frames zu bedienen: Je nach Ressourcenintensität der Module können hier Abstufungen vorgenommen werden.

In den folgenden Abschnitten werden Sampling- und Filtering-Module vorgestellt, welche bereits mit dem Framework mitgeliefert werden und somit fertig einsetzbar sind. Es wird hier vor allem auf die Umsetzung der Sampling- und Filtering-Mechanismen eingegangen. Grundlagen zur Funktionsweise und Qualität können in [ZMDN⁺07] nachgelesen werden.

5.7.2.1 ModuleSampling_noutOfN

Das `ModuleSampling_noutOfN`-Sampling-Modul wählt aus N Elementen n Elemente zufällig aus. Das Frame-Strom wird somit in Intervalle der festen Länge N unterteilt. Aus jedem dieser Intervalle der Länge N werden zufällig n Elemente ausgewählt.

Die für dieses Modul benötigten Zufallszahlen werden unter Verwendung der in Abschnitt 2.3 vorgestellten Boost Random Library berechnet. Nach Konfiguration des Moduls mit den Parametern N und n wird eine Gleichverteilung mit Werten zwischen 0 und $N - 1$ erstellt. Unter Verwendung dieser Verteilung wird ein Pseudozufallszahlen-Generator vom Typ Mersenne-Twister erstellt. Wie in Abschnitt 2.3 beschrieben, bietet dieser hochqualitative Pseudozufallszahlen bei hoher Geschwindigkeit.

Für die Auswahl von n Elementen aus einem Intervall der Länge N wird das `dynamic_bitset`-Template aus der Boost Bibliothek verwendet. Durch ein `dynamic_bitset`-Objekt der Länge N werden durch Verwendung des Pseudozufallszahlen-Generators n Pseudozufallszahlen zwischen 0 und $N - 1$ errechnet. Alle Bits des `dynamic_bitset`-Objektes werden aus 0 gesetzt. Die Bitpositionen der errechneten Pseudozufallszahlen werden auf 1 gesetzt.

Dieses Verfahren muss zu jedem Beginn eines neuen Intervalls der Länge N durchgeführt werden. Jeder eingehende Frame verwendet – entsprechend seinem Index im derzeitigen Intervall – die nächste Zufallszahl im `dynamic_bitset`-Objekt. Entsprechend dem Bitwert 1 oder 0 wird entschieden, ob der Frame im Channel weitergeleitet wird oder nicht.

5.7.2.2 ModuleSamplingSystematicCountBased

Das `ModuleSamplingSystematicCountBased`-Modul ist ein sehr leicht umzusetzendes Sampling-Modul. Es definiert zwei Phase: eine Selektionsphase und eine Sampling-Phase. In den Selektionsphase werden Frames selektiert – also im Channel weitergeleitet. In der Sampling-Phase werden Frames verworfen und somit nicht im Channel weitergeleitet. Es existieren somit zwei disjunkte Intervalle.

Die Konfiguration des Moduls geschieht durch zwei Ganzzahlwerte. Diese sind in der Konfiguration als `SamplingCount` – zur Definition der Samplingphase – und `SelectionCount` – zur Definition der Selektionsphase – vorgegeben. Diese Werte werden in der Implementierung abwechselnd hochgezählt und entsprechen Frames verworfen – Rückgabewert der `frameCall`-Methode `false` – bzw. weitergeleitet – Rückgabewert der `frameCall`-Methode `true`.

5.7.2.3 ModuleSamplingSystematicTimeBased

Das `ModuleSamplingSystematicTimeBased`-Modul arbeitet ähnlich wie das in Abschnitt 5.7.2.2 vorgestellte `ModuleSamplingSystematicCountBased`-Modul. Im Unterschied zum `ModuleSamplingSystematicCountBased`-Modul arbeitet es jedoch nicht auf Intervallen von Frames, sondern auf zeitlichen Intervallen. Die Konfiguration des Moduls geschieht über die zwei Konfigurationswerte `SamplingDuration` und `SelectionDuration`. Hiermit werden wieder zwei disjunkte Intervalle – hier jedoch zeitliche Intervalle – definiert, in denen Frames weitergeleitet werden – `SelectionDuration` – bzw. verworfen werden – `SamplingDuration`. Zur Messung der verstrichenen Zeiträume werden die Funktionen `GetTickCount` unter Windows bzw. die `times`-Funktion unter Linux verwendet.

5.7.2.4 ModuleSamplingUniformProbabilistic

Das `ModuleSamplingUniformProbabilistic`-Modul arbeitet, wie auch das in Abschnitt 5.7.2.1 vorgestellte `ModuleSampling_noutOfN`-Modul auf Pseudozufallszahlen. Über einen Konfigurationsparameter `Probability` wird dem Modul eine Wahrscheinlichkeit übergeben, mit welcher ein Frame weitergeleitet oder verworfen wird. Diese Wahrscheinlichkeit ist für jeden zu überprüfenden Frame gültig. Es findet hier also keine Unterteilung in Intervalle statt.

Als Pseudozufallszahlen-Generator wird, wie in Abschnitt 5.7.2.1, ein Mersenne-Twister verwendet. Die verwendete Verteilung liefert Zahlen aus dem Intervall $(0, 1]$. Für jeden eingehenden Frame wird eine Pseudozufallszahl aus diesem Intervall errechnet. Ist diese höher als die in der Konfiguration angegebene Selektionswahrscheinlichkeit, wird der Frame weitergeleitet, sonst verworfen.

5.7.2.5 ModuleFilteringProtocol

Im Gegensatz zu den beschriebenen Sampling-Verfahren arbeiten Filter-Module auf den Daten der Frames. Im Falle des `ModuleFilteringProtocol`-Moduls wird überprüft, ob ein vorgegebenes Protokoll in dem Frame enthalten ist. Dies kann im positiven wie im negativen Sinne überprüft werden. Die Konfiguration des Moduls geschieht über zwei Konfigurationselemente `Protocol` und `Positive`. Über das `Protocol`-Element wird das zu prüfende Protokoll angegeben. Dies ist eine Zeichenkette, welche das Protokoll kennzeichnet, wie beispielsweise TCP oder IPv6. Das `Positive`-Element gibt an, wie das Vorhandensein bzw. Nichtvorhandensein des angegebenen Protokolls behandelt werden soll. Die folgenden Beispiel verdeutlichen den Sachverhalt.

Es wurde als `Protocol` der Wert TCP angegeben und als `Positive` der Wert 1. Alle im Frame vorhandenen Protokolle werden in der `frameCall`-Methode durchlaufen. Wird ein Paket entsprechend dem TCP-Protokoll gefunden, so wird der Frame im Channel weitergeleitet, ansonsten verworfen.

Wird für `Positive` der Wert 0 angegeben, dann *darf* das angegebene Protokoll nicht im Frame vorhanden sein. Ist es vorhanden, wird der Frame verworfen, sonst weitergeleitet.

5.7.2.6 ModuleFilteringSubnet

Das `ModuleFilteringSubnet`-Modul arbeitet sehr ähnlich dem in Abschnitt 5.7.2.5 vorgestellten `ModuleFilteringProtocol`-Modul. Hier wird auf Grund der im IP-Paket vorhandenen Quell- und Zieladresse über die Weiterverarbeitung des Frame im Channel entschieden. Die Konfiguration des Moduls geschieht über die drei Konfigurationsparameter `Subnet`, `Mask` und `Positive`. Durch den `Subnet`- und `Mask`-Parameter wird das Subnetz definiert, auf welches die IP-Adressen des IP-Pakets geprüft werden. Der `Positive`-Parameter hat die gleiche Funktion wie bei dem in Abschnitt 5.7.2.5 vorgestellten `ModuleFilteringProtocol`-Modul.

Bei Prüfung eines Frames wird zuerst nach dem enthaltenen IP-Paket gesucht. Ist ein solches nicht vorhanden, wird der Frame im Falle eines positiven Filters verworfen und entsprechend im Falle eines negativen Filters zur Weiterverarbeitung im Channel

freigegeben. Die Prüfung der IP-Adressen wird durch den Oder-Operator verknüpft: sobald die Quell- oder Ziel-IP-Adresse die Bedingungen des positiven oder negativen Filters erfüllt, gilt die ganze Prüfung als erfüllt. Das bedeutet, dass im Falle eines positiven Filters mindestens eine der beiden IP-Adressen im Subnetz liegen muss. Im Falle eines negativen Filters darf keine der beiden IP-Adressen im Subnetz liegen.

6. Evaluierung

In Abschnitt 3.1 wurden Anforderungen an ein verteilt arbeitendes Anomalie-basiertes Angriffserkennungssystem definiert. Auf Grund dieser wurde dann in Kapitel 4 der Entwurf des Angriffserkennungssystems und die Implementierung in Kapitel 5 vorgestellt. Es bleibt zu zeigen, dass das System die in Abschnitt 3.1 erarbeiteten Anforderungen erfüllt. Dieses Kapitel ist wie folgt gegliedert: in Abschnitt 6.1 wird die Skalierbarkeit des Systems betrachtet. Auf Grund von Messungen wird hier gezeigt, dass das System lokal und verteilt skaliert. Die Erweiterbarkeit des Systems wird in Abschnitt 6.2 betrachtet und durch beispielhafte Integrationen von Angriffserkennung und Erweiterungen gezeigt. Durch die Portierung des Systems in die OMNeT++-Simulationsumgebung (siehe Abschnitt 6.3.1) und beispielhafte Anwendungsfälle wird die Flexibilität des Systems untersucht. Die geforderte Robustheit wird in Abschnitt 6.4 betrachtet.

6.1 Skalierbarkeit

Die Skalierbarkeit eines Systems zur Angriffserkennung ist sehr wichtig um bei hohen und schwankenden Datenraten korrekt zu arbeiten. Da das Distack-Framework zum Beispiel auf einem Vermittlungssystem im Inneren eines Netzes betrieben werden soll, ist es sehr wichtig den Betrieb des Vermittlungssystems nicht zu beeinflussen. Weiterhin ist im Inneren eines Netzes ein sehr hohes und burstartiges Verkehrsaufkommen zu beobachten.

Die Skalierbarkeit wird in zwei Gruppen betrachtet: einmal wird die lokale Skalierbarkeit einer autonomen Instanz geprüft. Des Weiteren wird die Skalierbarkeit bezüglich der Verteilung betrachtet. Hier werden Mechanismen, welche für die Kommunikation zwischen verteilten Instanzen verwendet werden, auf ihre Skalierbarkeit getestet.

Die Evaluierung findet durchgängig auf zwei Linux Systemen statt. Die Systemdaten von SystemA sind in Tabelle 6.1, die von SystemB in Tabelle 6.2 zu finden.

Gruppe	Eigenschaft	Wert
Betriebssystem	Typ	GNU/Linux 2.6.15.1
	Distribution	Debian 4.1.1-21
Festplatte	Gesamtgröße	18 GByte
	Frei	7 GByte
	Geschwindigkeit	350 MBit/s
Prozessor	Architektur	i686, Intel Pentium III, GenuineIntel
	Geschwindigkeit	800 MHz
	Cache	256 KByte
Speicher	Gesamtgröße	256 MByte
	Frei	12 264 KByte
Netzwerk	Typ	Ethernet
	Geschwindigkeit	100 MBit/s
Entwicklung	Übersetzer	GNU g++, 4.1.2
	Boost Bibliotheken	1.34.1

Tabelle 6.1: Systemdaten des Evaluierungsrechners SystemA

Gruppe	Eigenschaft	Wert
Betriebssystem	Typ	GNU/Linux 2.6.15.1
	Distribution	Debian 4.1.1-21
Festplatte	Gesamtgröße	56 GByte
	Frei	52 GByte
	Geschwindigkeit	360 Mbit/s
Prozessor	Architektur	i686, Intel Pentium 4, GenuineIntel
	Geschwindigkeit	2,60 GHz
	Cache	512 KByte
Speicher	Gesamtgröße	512 MByte
	Frei	16 608 KByte
Netzwerk	Typ	Ethernet
	Geschwindigkeit	100 MBit/s
Entwicklung	Übersetzer	GNU g++, 4.1.2
	Boost Bibliotheken	1.34.1

Tabelle 6.2: Systemdaten des Evaluierungsrechners SystemB

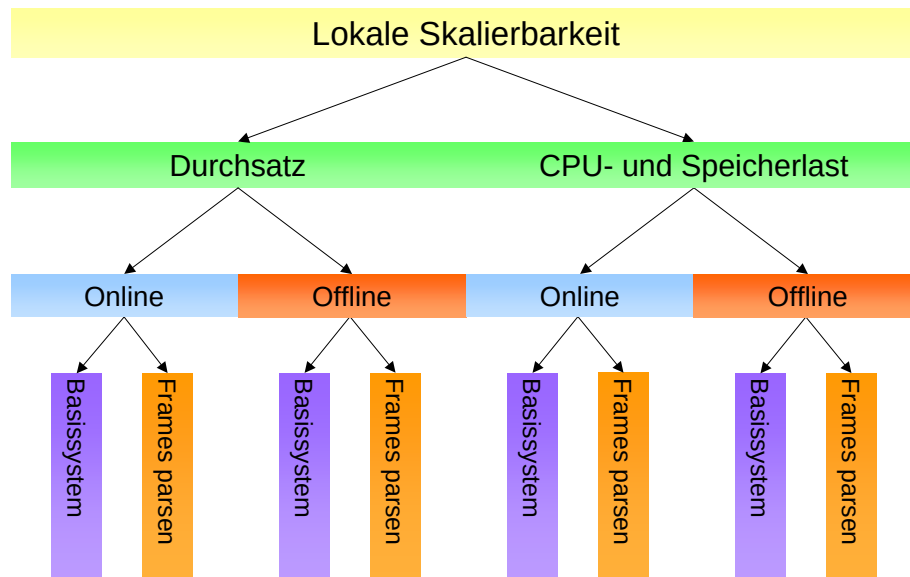


Abbildung 6.1: Aufbau der Evaluierung für lokale Skalierbarkeit

6.1.1 Lokale Skalierbarkeit

Der Aufbau zur Evaluierung der lokalen Skalierbarkeit ist in Abbildung 6.1 dargestellt. Im Folgenden wird die Skalierbarkeit einer Instanz der Angriffserkennung durch Messung von Durchsatz in Abschnitt 6.1.1.1 und der CPU- und Speicherlast in Abschnitt 6.1.1.2 gezeigt. Jede der Gruppen wird dabei in Online- und Offline-Evaluierung unterteilt. Messungen werden einmal nur mit dem Basissystem sowie einmal mit einem Modul, welches Frames parst, durchgeführt. Das Parsen von Frames ist eine der ressourcenintensivsten Operationen, da sie auf jedem eingehenden Frame ausgeführt werden.

Verwendete Netzwerktrace

Die Evaluation der lokalen Skalierbarkeit benötigt Netzwerkverkehr. Dieser wird entweder von einem anderen System abgespielt – Online-Messung – oder aus einer lokalen Tracedatei gelesen – Offline-Messung. Zur Evaluation der lokalen Skalierbarkeit mit einer Netzwerktrace müssen folgende Aspekte beachtet werden, welche die Messung beeinflussen:

- Im Verkehr vorhandene Protokolle
- Größe der Frames im Falle von abgeschnittenen Paketen
- Geschwindigkeit der Festplatte bei Verwendung einer Netzwerktrace
- Geschwindigkeit der Netzkomponenten

Die zur Durchsatzmessung verwendete *New York*-Netzwerktrace wurde im Rahmen der Arbeiten von Mayer [Maye06] am Mirror-Port eines Gateways – welcher ein großes Netzwerk an das Internet anbindet – aufgenommen und entsprechend den datenschutzrechtlichen Vorgaben anonymisiert. In der Trace ist eine repräsentative

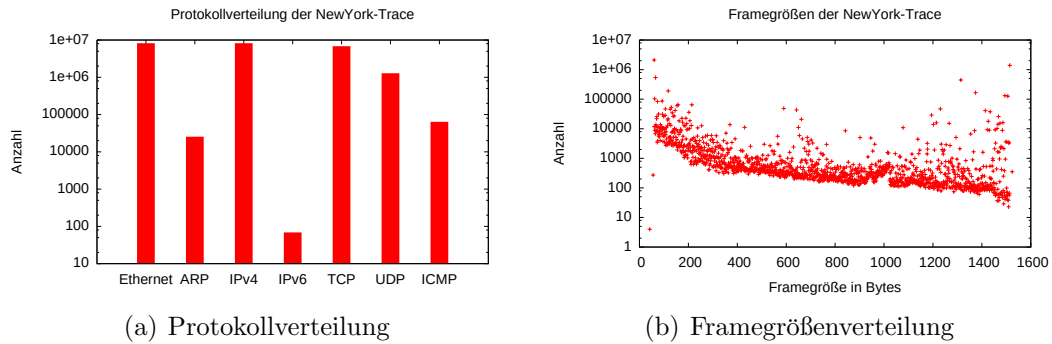


Abbildung 6.2: Analyse der NewYork-Trace

Vielfalt an Protokollen gegeben. Dies ist notwendig, da die Komplexität der Protokolle entscheidenden Einfluss auf die Dauer des Parsens der Frames hat. Bei Aufnahme einer Netzwerktace mit Tcpcdump [Tcpc00] werden standardmäßig nur die ersten 68 Byte jedes Frame gespeichert. Somit gehen Informationen über höhere Protokollschichten verloren. Die hier verwendete NewYork-Trace beinhaltet jedoch den kompletten Frame. Somit kann der Durchsatz genau bestimmt werden.

Bei Verwendung einer Netzwerktace ist die Geschwindigkeit der Festplatte eine wichtige Größe, da große Datenmengen schnellstmöglich gelesen werden müssen. Die Geschwindigkeit der jeweiligen Festplatte der zur Evaluierung verwendeten Systeme ist in Tabelle 6.1 und 6.2 gegeben.

Abbildung 6.2 zeigt die in der NewYork-Trace vorhandenen Protokolle sowie die Größenverteilung der Frames. Die logarithmisch dargestellte Protokollverteilung zeigt eine ausgewogene Zusammensetzung an Protokollen, welche überdurchschnittlich oft in Netzwerkverkehr vorhanden sind. Auch die Größenverteilung spiegelt realen Verkehr wieder (siehe auch [ThMW97]) und besitzt repräsentative Spitzen bei 60 und 1514 Bytes. Die Spitze bei 60 Bytes repräsentiert kleine TCP-Kontrollpakete, welche durch das Ethernet-Padding auf 60 Byte aufgefüllt werden. Die Spitze bei 1514 Bytes stellt die konkrete Datenübertragung dar. Der Grund, warum in Abbildung 6.2(b) auch einige Frames kleiner als 60 Bytes zu sehen sind, ist, dass das Ethernet-Padding von der Netzwerkkarte durchgeführt wird und somit ausgehende Frames noch nicht aufgefüllt wurden.

Abbildung 6.3 zeigt die Datenraten der Protokolle TCP, UDP und ICMP. Sowohl für die Online- wie auch für die Offline-Messungen wird die Trace mit maximaler Geschwindigkeit abgespielt bzw. eingelesen. Da hier keine Angriffserkennung evaluiert wird, ist die maximale Geschwindigkeit in Bezug auf Messungen zur Skalierbarkeit besser geeignet als die originale Geschwindigkeit. Da zum Abspielen die maximale Geschwindigkeit verwendet wird, entstehen in den folgenden Messungen andere Datenraten als original in der Trace vorhanden sind. Weitere Analysen der NewYork-Trace werden in Abschnitt 6.3.2.1 gezeigt.

6.1.1.1 Durchsatz

Ein wichtiger Anhaltspunkt für die Performance und Skalierbarkeit des Systems ist der Durchsatz. Dies ist die maximale Geschwindigkeit des Netzwerkverkehrs, welcher

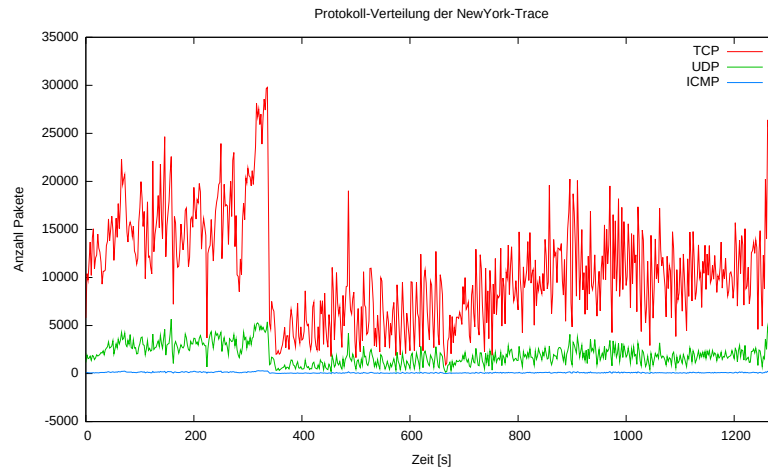


Abbildung 6.3: Originaler zeitlicher Verlauf der Protokolle TCP, UDP und ICMP

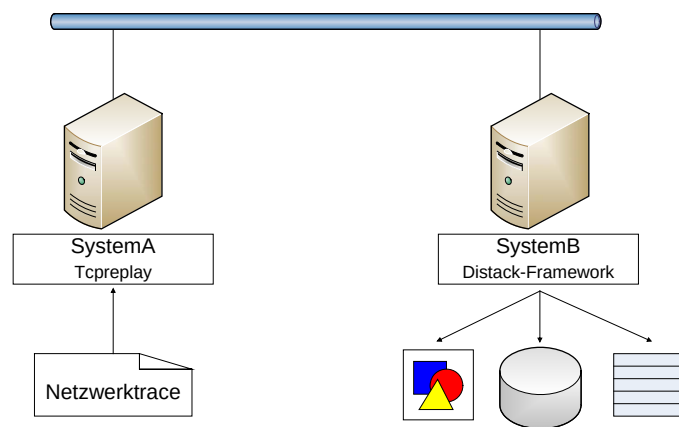


Abbildung 6.4: Systemaufbau zur Durchsatzmessung mit Online-Verkehr

vom Framework verarbeitet werden kann. Um den maximal erreichbaren Durchsatz zu messen werden zwei Methoden angewendet:

- Durchsatzmessung mit Online-Verkehr unter Verwendung eines realen Netzwerk-Interface
- Durchsatzmessung mit Offline-Verkehr unter Verwendung einer Netzwerktrace

Diese beiden Verfahren werden im Folgenden vorgestellt und entsprechende Messungen durchgeführt.

Durchsatzmessung mit Online-Verkehr

Zur Erzeugung von Netzwerkverkehr wird unter Verwendung von Tcpreplay [Turn00] die NewYork-Trace von einem zweiten System aus abgespielt. Abbildung 6.4 zeigt den Systemaufbau zur Durchsatzmessung mit Online-Verkehr. Unter Verwendung von Tcpreplay wird die NewYork-Trace mit maximaler Geschwindigkeit von SystemA abgespielt. SystemB betreibt das Distack-Framework. Dieses liest die Frames vom Netzwerk und erstellt Statistiken zum Durchsatz.

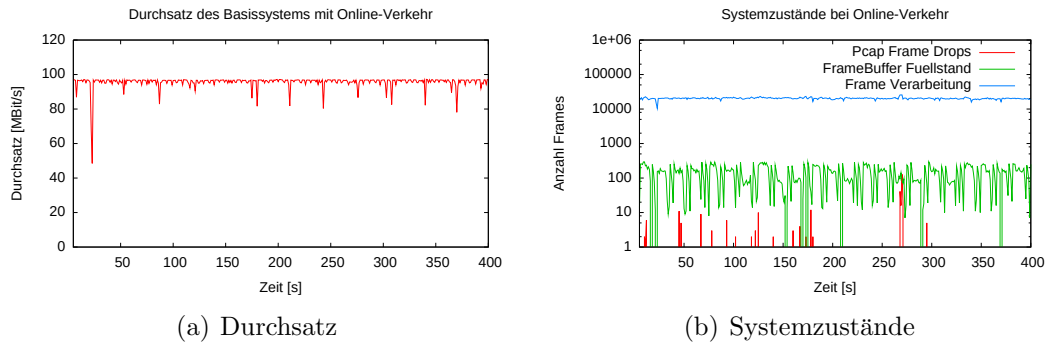


Abbildung 6.5: Messung des Basissystems mit Online-Verkehr

Beim Abspielen der NewYork-Trace durch SystemA unter Verwendung von Tcpreplay wird eine maximale Geschwindigkeit von 90 MBit/s erreicht. Diese beschreibt die Sendegeschwindigkeit welche von SystemA erreicht wurde. Hierbei werden alle in der Trace vorhandenen Frames schnellstmöglich auf die Netzwerkkarte geschrieben und entsprechend des vorhandenen Schicht 2-Protokolls weitergeleitet. In diesem Fall wurde die Trace vorher mit Tcprewrite [Turn02] bearbeitet und die Ziel-MAC-Adressen aller Schicht 2-Protokolle entsprechend der MAC-Adresse von SystemB geändert. Somit ist sichergestellt, dass alle Frames bei SystemB gelesen werden können.

Bei der Durchsatzmessung mit Online-Verkehr ist es möglich, dass bei hohen Raten Frames bei Empfang durch SystemB bereits vom im Distack-Framework verwendeten Pcap-System (siehe auch Abschnitt 2.4 und 5.1.1.1) verworfen werden. Dies kann zwei Gründe haben: Erstens werden Frames vom Pcap-System verworfen, falls die Verarbeitung innerhalb des Pcap-Systems selbst zu langsam ist und Frames nicht schnell genug aus dem Kernel-Puffer in den User Mode-Puffer kopiert werden können. Zweitens werden Frames verworfen, falls der **FrameBuffer** die Frames nicht schnell genug – unter Verwendung des **NetzwerkInterfacePcap** – aus dem Pcap-System auflösen kann. Das Pcap-System bietet Funktionen an um statistische Informationen über die Anzahl der verworfenen Frames abzufragen. Diese werden im Folgenden verwendet, um die Anzahl der vom Pcap-System verworfenen Frames zu messen.

Zur Durchsatzmessung mit Online-Verkehr wurde das Distack-Framework an mehreren Stellen erweitert und zusätzliche Messfunktionalität implementiert. Folgende Eigenschaften werden auf diese Weise gemessen:

- Frame-Drops des Pcap-Systems im **NetworkInterfacePcap**
- Füllstand des **FrameBuffer** Zwischenspeichers
- Durchsatzmessung und Ressourcenlast unter Verwendung eines Moduls

Der **FrameBuffer** wird ohne Begrenzungsmechanismen (siehe auch Abschnitt 5.1.2) betrieben, um zu prüfen, ob sich der Füllstand in einem angemessenen Bereich befindet oder womöglich stetig anwächst.

Abbildung 6.5(a) zeigt die Datenrate der durch SystemB vom Netzwerk gelesenen Verkehrsdaten. Dabei wurde als einzige Funktionalität im Distack-Framework ein

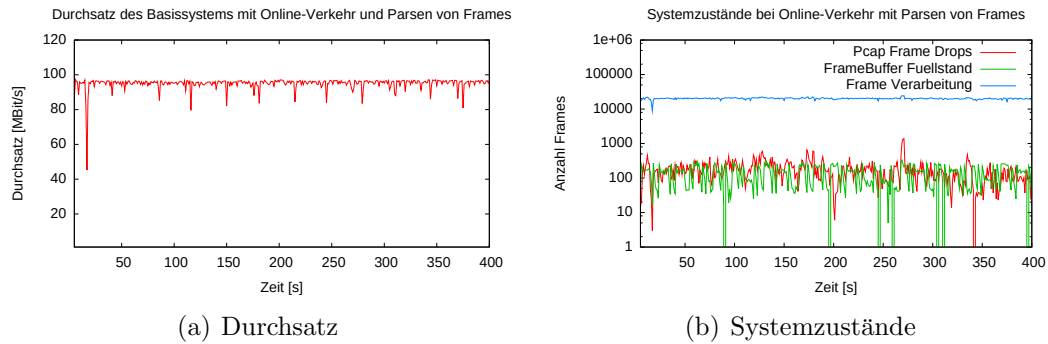


Abbildung 6.6: Messung mit Online-Verkehr und Parsen von Frames

Modul betrieben, welches die Datenrate misst. Wie oben genannt, erreicht SystemA eine durchschnittliche Sendegeschwindigkeit von 90 MBit/s. Diese spiegelt sich in der empfangenen Datenrate wieder. Dies zeigt, dass das Distack-Framework die gesendete Datenrate verarbeiten kann. In Abbildung 6.5(b) sind Systemdaten visualisiert, welche während der Durchsatzmessung aufgenommen wurden. Sie zeigen – auf einer logarithmischen Skala aufgetragen – die internen Systemzustände des Distack-Frameworks bei Verwendung von Online-Verkehr. Der Füllstand des **FrameBuffer**-Objektes befindet sich bei durchschnittlich 133 Frames und maximal 296 Frames. Die maximale Framegröße der NewYork-Trace beträgt 1 524 Byte. Somit beträgt der durch den **FrameBuffer** maximal benötigte Speicherverbrauch 451 KByte. Dies ist akzeptabel und beschränkt nicht das System auf welchem das Distack-Framework betrieben wird. Durch den **FrameBuffer** können die meisten der im Netzwerkverkehr enthaltenen Burst abgefangen. Daher ist der zusätzliche Speicherverbrauch akzeptabel. Weiter sind in Abbildung 6.5(b) die durch das Pcap-System verworfenen Frames zu sehen. Diese treten nicht kontinuierlich und selten auf. Werden Frames verworfen, so ist die Anzahl der verworfenen Frames unter 0,7 % der Framerate. Die Messungen zeigen, dass Netzwerkverkehr schnell verarbeitet wird und die internen Systemzustände sich in einem akzeptablen Bereich bewegen.

Die in Abbildung 6.6 gezeigten Messdaten wurden mit dem gleichen Systemaufbau wie oben aufgezeichnet. Der Unterschied bei dieser Messung besteht darin, dass jeder eingehende Frame geparkt wird und somit alle Protokolle verfügbar gemacht werden. Der Füllstand des FrameBuffer liegt durchschnittlich bei 135 Frames und maximal bei 334 Frames. Bei einer maximalen Framegröße von 1 524 Byte liegt der durch den FrameBuffer benötigte Speicherverbrauch somit bei 509 KByte. Dies ist akzeptabel und beeinflusst das unterliegende Vermittlungssystem nicht. Ein deutlicher Unterschied zu der oben beschriebenen Messung des Basissystems ohne das Parsen von Frames ist in Abbildung 6.6(b) zu sehen: Die Anzahl der Pcap Frame Drops ist kontinuierlich und die Anzahl der jeweils verworfenen Frames ist höher. Es werden durchschnittlich 177 Frames pro Sekunde und maximal 1378 Frames pro Sekunde vom Pcap-System verworfen. Dies entspricht, bei einer maximalen Framegröße von 1 524 Byte der NewYork-Trace, durchschnittlich 2 MBit/s und maximal 17 MBit/s. Die Gründe für das Verwerfen von Frames durch das Pcap-System liegen in einer erhöhten Systemlast, welche sich durch das Parsen von Frames ergibt. Die Systemlast betrifft hier jedoch nicht die CPU- oder Speicherlast, daher müssen weiterführende Untersuchungen angegangen werden um die Gründe für die Pcap Fra-

me Drops zu untersuchen. Die durchschnittlich 177 verworfenen Frames pro Sekunde entsprechen 0,9 % der durchschnittlichen Framerate. Der maximale Frameverlust von 1378 Frames pro Sekunde entspricht 6,8 % der durchschnittlichen Framerate. Da sich der durchschnittliche Frameverlust unter 1 % befindet ist das Verhalten akzeptabel.

Durchsatzmessung mit Offline-Verkehr

Zur Durchsatzmessung mit Offline-Verkehr ist es notwendig die in Abschnitt 5.1.1.1 beschriebenen Mechanismen zum Abspielen einer Netzwerktrace in seiner originalen Zeitdomäne mit entsprechenden Wartezeiten zwischen den Frames zu deaktivieren. Somit wird die Netzwerktrace schnellstmöglich verarbeitet und dadurch der maximale Durchsatz erreicht. Bei Verwendung einer Tracedatei wird der **FrameBuffer** intern ausgeschaltet und jeder Frame rein seriell gelesen und verarbeitet. Obwohl das Pcap-System zum Lesen von Tracedateien verwendet wird, werden in diesem Fall keine Frames vom Pcap-System verworfen, da es sich um die Verwendung einer Tracedatei handelt. Somit ist die einzig interessante Messgröße der letztendlich erreichte Durchsatz an Daten. Dieser ist in Abbildung 6.7 einmal mit Betrieb des Basissystems sowie unter Verwendung eines Moduls, welches die eingehenden Frames parst, dargestellt.

Die maximale Lesegeschwindigkeit der Festplatte wurde mit dem `hdparm`-Programm [Lord05] bestimmt und beträgt 360 MBit/s. Dies stellt die obere Grenze für Leseoperationen auf der Festplatte dar, welche aus den folgenden Gründen nicht erreicht werden kann:

- Das `hdparm`-Programm liest vom Anfang der Festplatte. Dies ist der schnellste Bereich.
- Die vom `hdparm`-Programm verwendete Lesegröße wird entsprechend der Blockgröße der Festplatte optimal gewählt.

Beim Lesen von Frames aus einer Tracedatei unter Verwendung des Pcap-Systems wird wie folgt vorgegangen: Zuerst wird für den nächsten Frame der Pcap-Frame-Header gelesen. Dieser besitzt eine feste Größe. In diesem steht die Größe des folgenden Frames, welcher dann als zweite Leseoperation gelesen wird. Somit sind für jeden Frame zwei Leseoperationen notwendig. Die zweite Leseoperation hat zusätzlich eine nicht feste und somit potentiell ungünstige Länge in Bezug auf die Blockgröße der Festplatte. Weiterhin wird der gelesene Pcap-Frame-Header nicht zum Datendurchsatz des gespeicherten Netzverkehrs addiert. Das Lesen von Frames aus einer Tracedatei erreicht daher grundsätzlich nicht die maximale Geschwindigkeit von 360 MBit/s. Wie deutlich zu sehen ist, unterscheiden sich die beiden Messungen in Abbildung 6.7 kaum. Dies liegt daran, dass die Lesegeschwindigkeit – wie eben erläutert – begrenzt ist und dadurch genügend Zeit zur Verfügung steht um Frames zu parsen.

6.1.1.2 CPU- und Speicherlast

In diesem Abschnitt wird die Belastung der CPU und des Speichers betrachtet, welche bei den in Abschnitt 6.1.1.1 durchgeführten Messungen auftritt. Die Belastung

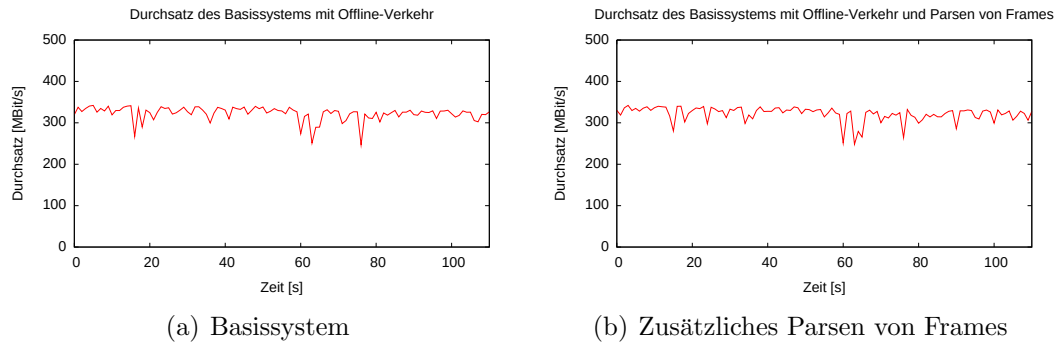


Abbildung 6.7: Durchsatz bei Messung mit Offline-Verkehr

ist eine sehr wichtiger Indikator für Skalierbarkeit und Robustheit. Ein Vermittlungssystem, auf dem das Distack-Framework betrieben wird, darf nicht in seiner eigentlichen Arbeit beeinflusst werden. Dies bedingt eine geringe CPU- und Speicherlast.

CPU- und Speicherlast bei Online-Verkehr

Entsprechend der in Abschnitt 6.1.1.1 beschriebenen Messung mit Online-Verkehr wird hier nun die CPU- und Speicherlast betrachtet, welche durch das Distack-Framework verursacht wird. Diese ist in Abbildung 6.8 einmal für das Basissystem sowie für das zusätzliche Parsen von Frames dargestellt.

Wie der in Abschnitt 6.1.1.1 erreichte Durchsatz der beiden Messungen – nur das Basissystem und zusätzliches Parsen von Frames – ist auch die CPU-Last bei beiden Messungen sehr ähnlich. Die CPU-Last des Basissystems beträgt durchschnittlich 7,5 % und maximal 22 %. Bei zusätzlichem Parsen von Frames wird eine CPU-Last von durchschnittlich 7,6 % und maximal 45 % benötigt. Somit beeinflusst das Distack-Framework nicht das unterliegende System.

Für Frames und Protokoll-Parser werden Speicher-Pools verwendet (siehe Abschnitt 2.5). Somit wird allozierter Speicher der Frame- und Paket-Objekte nicht freigegeben, sondern zur späteren Verwendung gespeichert. Schwankungen im `FrameBuffer` wirken sich somit nicht auf den Speicherverbrauch aus. Auf Grund der sehr hohen Frameraten von 20 293 Frames pro Sekunde bei Verwendung des Basissystems und 20 075 Frames pro Sekunde bei zusätzlichem Parsen von Frames werden die Speicherpools schnell gefüllt. Somit ist keine initiale Steigerung des Speicherverbrauchs zu Beginn der Verarbeitung zu beobachten. Die Speicherlast beträgt sowohl bei Verwendung des Basissystems als auch bei zusätzlichen Parsen von Frames 3 %. Dies entspricht ungefähr 15,4 MByte.

CPU- und Speicherlast bei Offline-Verkehr

Durch die höhere Verarbeitungsgeschwindigkeit bei Verwendung einer Tracedatei ist die CPU-Last höher. Dies ist in Abbildung 6.9 deutlich zu sehen. Vor allem das zusätzliche Parsen von Frames lässt die CPU-Last auf das Doppelte ansteigen. Es ist hier jedoch zu beachten, dass es sich um Offline-Verkehr handelt. Somit findet

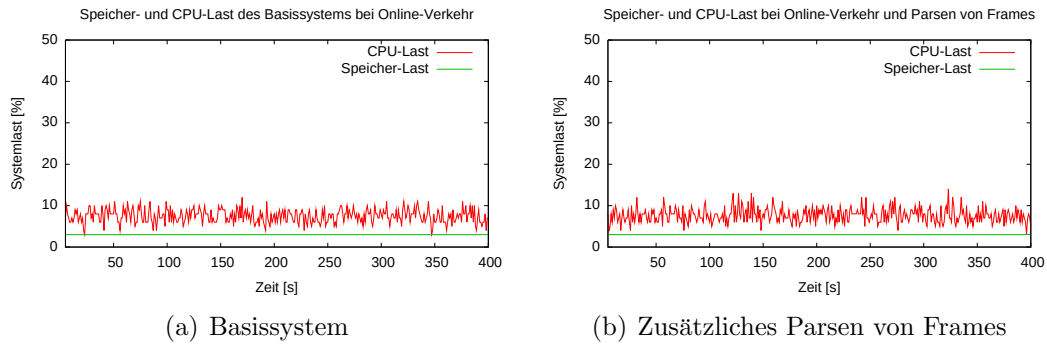


Abbildung 6.8: CPU- und Speicherlast bei Messung mit Online-Verkehr

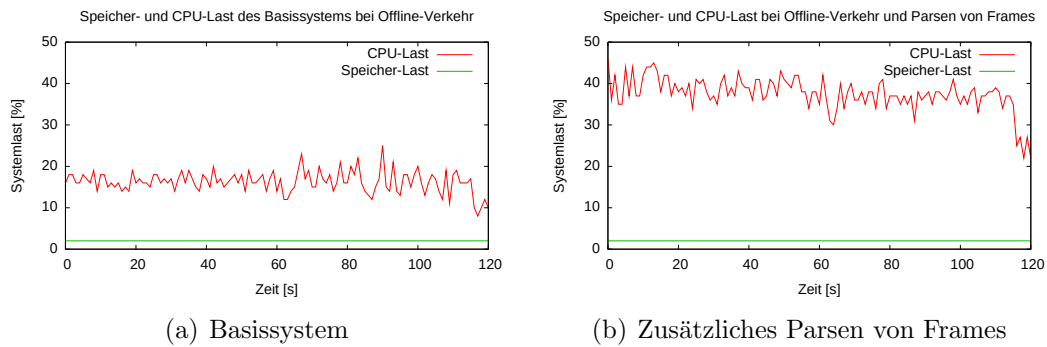


Abbildung 6.9: CPU- und Speicherlast bei Messung mit Offline-Verkehr

die Verarbeitung nicht auf einem System statt welches potentiell in seiner eigentlichen Arbeit beeinflusst werden kann, sondern auf einem dedizierten Analyse-System. Daher kann eine Beeinflussung des verwendeten Systems in Kauf genommen werden.

Wie bei der Verwendung von Online-Verkehr, wird konstante viel Speicher benötigt. Die Speicherlast beträgt in diesem Fall 2 %. Dies entspricht ungefähr 10 MByte. Ein konstanter Speicherverbrauch ist bei Offline-Verkehr, der rein seriell abgearbeitet und nicht zwischengespeichert wird, offensichtlich. Weiterhin ist der Speicherverbrauch geringer, da der **FrameBuffer** nicht benötigt wird und somit keine Frames zwischengespeichert werden. Auch hier unterstützen die Speicherpools das System. Die Allokation und Deallokation von Speicher, welche für das Lesen und Parsen von Frames notwendig ist, ist normalerweise sehr kostspielig. Durch die Verwendung von Speicherpools entfallen diese Kosten, was in einer Schonung der Systemressourcen resultiert.

6.1.2 Verteilte Skalierbarkeit

Die verteilte Skalierbarkeit untersucht die Leistungsfähigkeit und den Ressourcenverbrauch des Nachrichtensystems zur verteilten Kommunikation. Im Gegensatz zum lokalen Nachrichtensystem entstehen hier Kosten durch die Serialisierung und Deserialisierung sowie die Übertragung der serialisierten Daten an entfernte Instanzen. Durch Messung des Durchsatz wird die Leistungsfähigkeit betrachtet. Weiterhin wird die CPU- und Speicherlast, welche durch die verteilte Kommunikation entsteht, gemessen. Zur Durchführung der Messungen werden wieder die Systeme SystemA

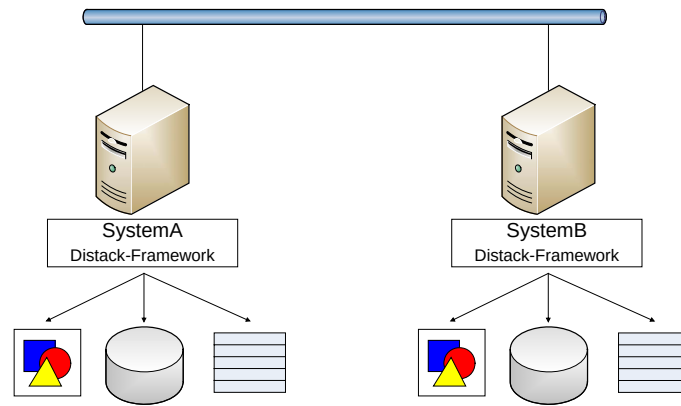


Abbildung 6.10: Systemaufbau zur Evaluierung der verteilten Skalierbarkeit

und SystemB verwendet. Der Systemaufbau ist hier jedoch verändert: beide Systeme betreiben hier das Distack-Framework. SystemE verwendet ein Modul, welches nacheinander Nachrichten an SystemA sendet. Es wird der erreichbare Durchsatz an Nachrichten sowie die CPU- und Speicherlast gemessen. Zur Kommunikation wird die in Abschnitt 5.4.5 beschriebene TCP-Socket-basierte Kommunikationsschicht verwendet.

Der Systemaufbau zur Evaluation der verteilten Skalierbarkeit ist in Abbildung 6.10 gezeigt. Sowohl auf SystemA als auch auf SystemB wird das Distack-Framework betrieben. Beide Systeme sammeln statistische Daten der Übertragung von Nachrichten und Ressourcenlast. SystemB sendet kontinuierlich Nachrichten an SystemA. Dieses nimmt die Nachrichten auf und verteilt sie an alle für diese Nachricht registrierten Module.

Abbildung 6.11(a) zeigt den erreichten Durchsatz an Nachrichten pro Sekunde. Durchschnittlich werden 1 120 Nachrichten pro Sekunde zwischen den beiden Systemen übertragen. Da es sich bei der TCP-Socket-basierten Kommunikationsschicht um eine zuverlässige Übertragung handelt, sind die vom Sender gesendeten Nachrichten pro Sekunde identisch mit den vom Empfänger empfangenen Nachrichten pro Sekunde. Auf Empfängerseite eingehende Nachrichten werden in eine Warteschlange eingereiht und asynchron verarbeitet. Wie zu sehen ist, geschieht die Abarbeitung schnell genug: der maximale Füllstand der Warteschlange beträgt 14 Nachrichten und wird immer nach kurzer Zeit abgebaut. Dass sich teilweise Frames in der Warteschlange ansammeln lässt sich mit Aktivitäten anderer auf dem System gestarteter Prozesse erklären.

Die auf Sender- und Empfängerseite entstehende CPU- und Speicherlast ist in Abbildung 6.11(b) dargestellt. Da SystemA eine geringere Speichergröße besitzt (siehe Tabelle 6.1) ist der Verbrauch prozentual höher als auf SystemB. Die Speicherlast beträgt auf SystemA 7 % und auf SystemA 4 %. Dies entspricht einer absoluten Speicherlast von 18 MByte auf SystemA sowie 20 MByte auf SystemB. Der Unterschied ist auf die Systemkonfiguration der Betriebssysteme zurückzuführen. Trotz der höheren CPU-Ressourcen auf SystemB, ist auf diesem System eine höhere CPU-Last als auf SystemA. Dies lässt darauf schließen, dass die Serialisierung von Nachrichten aufwendiger ist als die Deserialisierung. Auf Senderseite bei SystemB liegt die CPU-

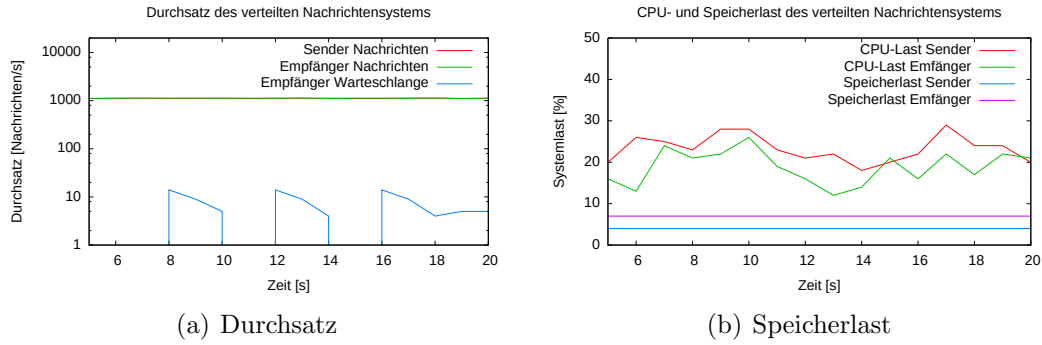


Abbildung 6.11: Ressourcenlast durch das verteilte Nachrichtensystem

Last zur jedem Zeitpunkt unter 30 %, auf Empfängerseite bei SystemA unter 25 %.

Offensichtlich handelt es sich hier um keinen realistischen Test, da nicht mit einem Nachrichtenaufkommen von über 1000 Nachrichten pro Sekunde zu rechnen ist. Bereits 100 Nachrichten pro Sekunde sind wohl in keinem realen Anwendungsszenario zu beobachten. Die Messungen zeigen somit, dass das Nachrichtensystem eine hohe Last verarbeiten kann und die CPU- und Speicherlast das unterliegende Vermittlungssystem nicht beeinflusst. Bei einem Nachrichteaufkommen von 100 Nachrichten pro Sekunde ist keine CPU-Last erkennbar, welche durch den Versand von Nachrichten entstanden wäre.

6.2 Erweiterbarkeit

Die in Abschnitt 3.1.2 geforderte Erweiterbarkeit des Distack-Frameworks wird anhand von zwei Beispielen gezeigt. In Abschnitt 6.2.1 wird die Integration einer bestehenden Angriffserkennung beschrieben. Weiterhin wird in Abschnitt 6.2.2 eine Überwachung des MPLS-Protokolls in das Framework integriert.

6.2.1 Integration einer Angriffserkennung

In [GaSB06] wurde ein System zur Anomalie-basierten Angriffserkennung entwickelt. Es wird hier die Integration dieses Systems in das entwickelte Framework zur Angriffserkennung beschrieben werden. In Abschnitt 6.2.1.1 folgt eine Analyse der Angriffserkennung. Dann wird in Abschnitt 6.2.1.2 beschrieben, wie die Angriffserkennung in das Framework integriert wird. Abschnitt 6.2.1.3 betrachtet Laufzeitmessungen zur Skalierbarkeit. In Abschnitt 6.2.1.4 wird eine Zusammenfassung der Integration der Angriffserkennung gegeben und auf die durch die Integration gewonnenen Vorteile eingegangen.

6.2.1.1 Analyse der Angriffserkennung

Das Angriffserkennung aus [GaSB06] besteht aus drei Stufen, welche einen Granularitäts-basierten Ansatz zur Angriffserkennung ermöglichen. Zur Umsetzung des Systems wurde die Flexinet-Umgebung [FHSZ02] verwendet. Sie ermöglicht dem System Netzwerkframes zu analysieren und dient als Laufzeitumgebung.

In Abbildung 6.12 sind die drei Stufen der Angriffserkennung mit ihren Aufgaben sowie den Daten, welche zwischen den Stufen weitergegeben werden, gezeigt. Stufe 1 überwacht mit einem Schwellenwert-basierten Verfahren verschiedene Protokoll-

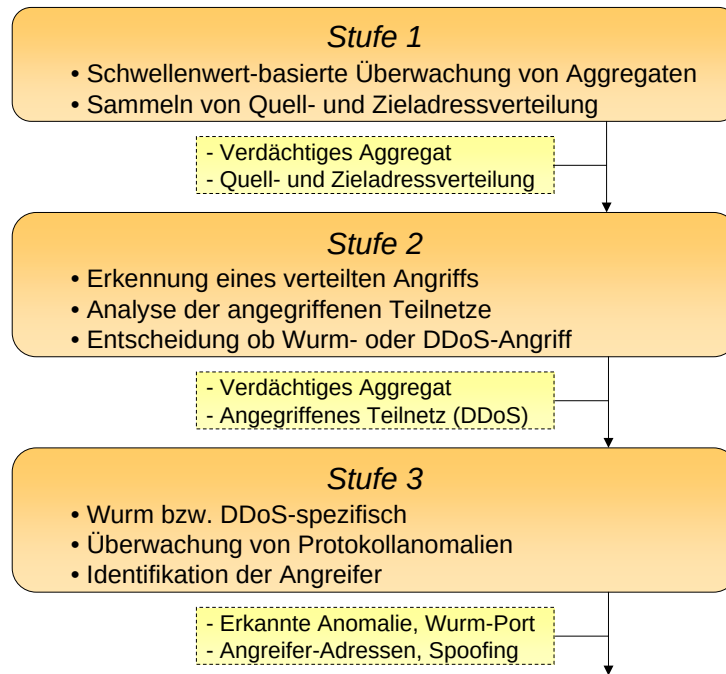


Abbildung 6.12: Aufbau der Angriffserkennung

und Größenaggregate. Weiterhin werden in Stufe 1 Quell- und Zieladressverteilungen der letzten n Intervalle gesammelt. Im Falle einer mehrfachen Überschreitung des dynamisch angepassten Schwellenwertes wird ein Angriff vermutet und die zweite Stufe geladen. Diese analysiert die von der ersten Stufe gesammelten Quell- und Zieladressverteilungen. Ziel der zweiten Stufe ist, zu bestimmen, ob es sich um einen Wurm- oder DDoS-Angriff handelt und im Falle eines DDoS-Angriffs das angegriffene Teilnetz zu bestimmen. Die Information des verdächtigen Aggregats aus der ersten Stufe, die Unterscheidung eines Wurm- oder DDoS-Angriffs sowie – im Falle eines DDoS-Angriffs – das angegriffene Teilnetz werden an die dritte Stufe weitergeleitet. Diese wird geladen, sobald die Analyse der zweiten Stufe abgeschlossen ist. Die dritte Stufe analysiert den Netzverkehr und sucht nach Protokollanomalien. Im Gegensatz zur ersten Stufe wird der analysierte Netzverkehr hier jedoch entsprechend dem verdächtigen Aggregat sowie dem verdächtigen Subnetz eingeschränkt. Wurde eine Anomalie erkannt, so wird dies dem Benutzer mitgeteilt. Weiterhin wird in der dritten Stufe untersucht ob Spoofing-Methoden eingesetzt werden. Ist dies nicht der Fall, so wird versucht die Angreifer zu identifiziert.

6.2.1.2 Integration der Angriffserkennung

Die beschriebene Angriffserkennung ist in drei Stufen aufgeteilt welche einen Granularitäts-basierten Ansatz ermöglichen. Die Aufgaben der drei Stufen sind teilweise nicht klar getrennt, beispielsweise betreibt die erste Stufe neben ihrer eigentlichen Aufgabe – der Überwachung von Protokoll- und Größenaggregaten – eine Sammlung von Quell- und Zieladressverteilung welche für die zweite Stufe benötigt wird. Ziel der Integration der Angriffserkennung ist, dieses in einzelne, in ihren Aufgaben abgeschlossene Bausteine zu zerteilen, so dass eine einfache Integration in Distack möglich ist und somit ein hohes Maß an Erweiterbarkeit und Flexibilität erreicht wird.

Zur Transformation werden die Stufen in ihre Aufgabenbereiche aufgetrennt. Jedes Modul der Angriffserkennung soll wohldefinierte und abgeschlossene Aufgaben übernehmen. Die Weitergabe von Daten ist in der Angriffserkennung explizit realisiert, so dass beim Laden einer Stufe die definierten Daten direkt übergeben werden. Dieses starre Vorgehen wird insofern geändert, als dass jedes Modul selbst dafür verantwortlich ist die für sich notwendigen Daten zu erfahren und zu sammeln.

Die Trennung der Aufgaben ist in Abbildung 6.13 dargestellt. Wie zu sehen ist, wird jede Einzelaufgabe in ein Modul der Angriffserkennung umgesetzt und somit ein hohes Maß an Flexibilität erreicht. Die Angriffserkennung wird in die folgenden Module zerlegt, welche Abbildung 6.13 entsprechen:

- *Sampling*: Für dieses Modul kann eines der in Abschnitt 5.7.2 beschriebenen Sampling-Module verwendet werden.
- *Überwachung von Protokoll- und Größenaggregaten*: Dieses Modul stellt die Grundüberwachung der ersten Stufe der Angriffserkennung dar. Es überwacht Protokoll- und Größenaggregate über ein Schwellenwert-basiertes Verfahren. Die in das Framework integrierte Implementation der Angriffserkennung überwacht automatisiert alle vorhandenen Protokollaggregate. Weiterhin sind die Intervalle der Größenaggregate über die Konfiguration definierbar.
- *Zähler und Schwellenwerte*: In diesem Modul werden die Zähler und Schwellenwerte der letzten x Intervalle gespeichert. Diese Aufgabe ist bei der Angriffserkennung in die erste Stufe integriert. Eine Speicherung der Werte ist jedoch nicht als Aufgabe der ersten Stufe zu betrachten.
- *Sammlung der Ziel- und Quelladressverteilung*: Die Sammlung der Ziel- und Quelladressverteilung wurde aus der ersten Stufe der Angriffserkennung in ein eigenes Modul extrahiert. Hier werden IP-Pakete analysiert und ihre Quell- und Ziel-IP-Adresse verwendet um die Verteilung zu erstellen.
- *Synchronisations-Timer*: Die Funktionalität der ersten Stufe der Angriffserkennung wird über einen Timer realisiert. Dieser zeigt somit den Wechsel eines Intervalls an und wirkt sich auf mehrere Funktionalitäten der ersten Stufe aus. Da die erste Stufe wie beschrieben in verschiedene Module zerteilt wurde, müssen diese Stufen über einen gemeinsamen Timer synchronisiert werden. Dies ist über das in Abschnitt 5.7.1.3 beschriebene Timer-Modul realisiert. Somit können mehrere Module über ein gemeinsames Timer-Modul synchronisiert werden.
- *Analyse Adressverteilung*: Dieses Modul entspricht der zweiten Stufe der Angriffserkennung. Hier muss keine Extraktion vorgenommen werden, da es sich um eine abgeschlossenes Aufgabengebiet handelt.
- *Sampling/Filtering*: Entsprechend dem oben beschriebenen Sampling-Modul können hier Module aus den in Abschnitt 5.7.2 beschriebenen Sampling- und Filtering-Modulen verwendet werden.
- *Analyse Protokollanomalien*: Diese Komponente ist in der Angriffserkennung als eine Menge an Modulen definiert. Jedes Modul überwacht dabei eine Protokollanomalie. Durch die Transformation entsteht nur ein Modul, jedoch mit

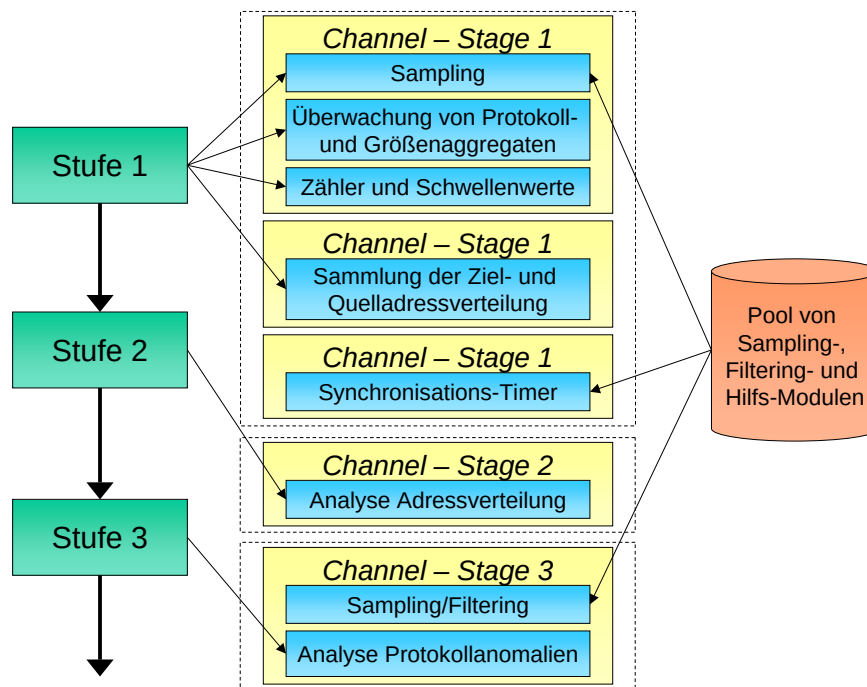


Abbildung 6.13: Integration der Angriffserkennung

flexiblerer Anwendung: Das neue Modul kann eine definierte Menge an Protokollanomalien überwachen, also auch nur eine einzige. Durch mehrfache Verwendung des Moduls ist es nun möglich den Granularitäts-basierten Ansatz zu festigen und die Protokollanomalien wieder in verschiedene Channels oder auch Stages zu definieren. Hiermit ist es auch möglich verschiedene Protokollanomalien mit verschiedenen vorgeschalteten Sampling-Modulen zu betreiben. Dies führt zu hoher Flexibilität und Ressourcenschonung. Weiterhin wird redundante Funktionalität vermieden, welche in den Modulen zur Überwachung von Protokollanomalien bei der bisherigen Angriffserkennung enthalten war.

Alle beschriebenen Module wurden entsprechend ihrer Funktionalität entwickelt und implementiert. Es wurde darauf geachtet, dass die Module nicht nur für die Integration hier verwendeten Angriffserkennung verwendet werden können, sondern allgemein als Bausteine zur Realisierung von Angriffserkennungssystemen im Distack-Framework verwendet werden können.

Wie bereits beschrieben, werden Daten zwischen den Stufen der Angriffserkennung explizit übergeben. Dies bedeutet, dass entsprechende Datenstrukturen definiert sind, über welche die Daten beim Laden der nächsten Stufe übergeben werden. Im Distack-Framework werden Daten über Nachrichten ausgetauscht. Somit existiert eine Kommunikation zwischen den Modulen. Wie in Abschnitt 4.4 beschrieben, wird dabei ein datenzentrierter Ansatz verfolgt. Da bei Integration der Angriffserkennung bekannt ist, welche Module sich für welche Nachrichten registrieren müssen und welche Informationen zur Verfügung stellen können, kann die Kommunikation zwischen den Modulen modelliert werden. Dies ist in Abbildung 6.14 dargestellt. Bei den Nachrichten handelt es sich teilweise um periodisch ausgesandte Informationen, teilweise um explizit angefragte Informationen, welche über Request/Reply-Nachrichten realisiert sind.

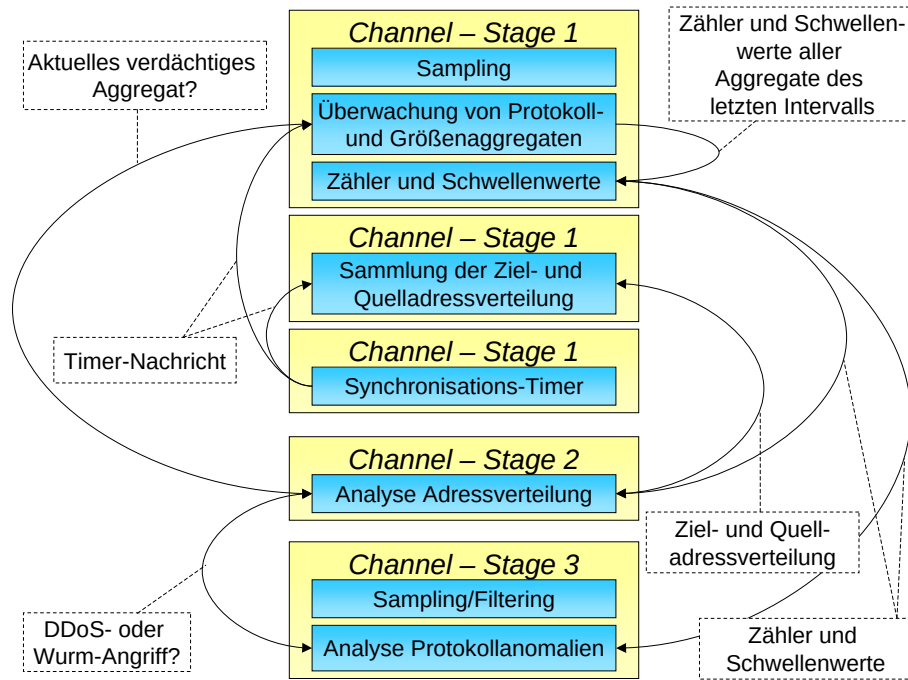


Abbildung 6.14: Nachrichtenkommunikation zwischen Modulen der integrierten Angriffserkennung

Wie in Abbildung 6.14 zu sehen ist, existieren insgesamt sieben Kommunikationsbeziehungen zwischen den Modulen. Diese werden im Folgenden beschrieben:

- *Zähler und Abweichungen aller Aggregate des letzten Intervalls*: Die Überwachung der Protokoll- und Größenaggregate liefert am Ende jedes Intervalls Informationen über die Zähler und Schwellenwerte der Aggregate. Diese werden ins Nachrichtensystem gesendet und vom entsprechenden Modul gespeichert.
- *Timer-Nachrichten*: Zur Synchronisation der Module der ehemals ersten Stufe sowie der ersten Stufe mit der dritten Stufe wird eine Timer-Nachricht verwendet. Dies ermöglicht die Synchronisation des Moduls zur Aggregat-Überwachung sowie des Moduls zur Sammlung der Ziel- und Quelladressverteilung. Weiterhin wird hierdurch die Aggregat-Überwachung der ersten Stufe mit der Überwachung von Protokollanomalien der dritten Stufe synchronisiert.
- *Ziel- und Quelladressverteilung*: Die gesammelte Ziel- und Quelladressverteilung wird auf Anfrage ausgesendet und somit dem Modul zur Analyse der Adressverteilung zur Verfügung gestellt.
- *Zähler und Schwellenwerte*: Ebenso wie die Ziel- und Quelladressverteilung werden Zähler und Schwellenwerte aller Aggregate der letzten x Intervalle abgefragt.
- *Aktuelles verdächtiges Aggregat*: Wie auch die beiden letzten Nachrichten, kann das aktuelle verdächtige Aggregat abgefragt werden und in der Analyse der Ziel- und Quelladressverteilung verwendet werden.
- *DDoS- oder Wurm-Angriff*: Die Instanzen des Moduls zur Analyse der Protokollanomalien benötigen die Information der in der Ziel- und Quelladressver-

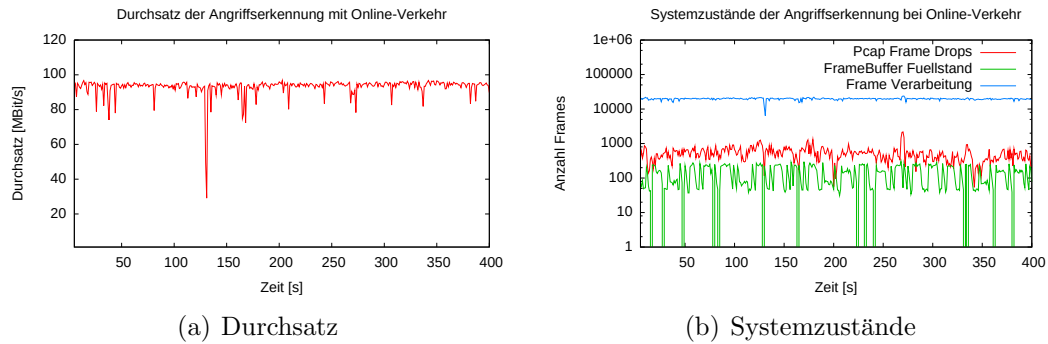


Abbildung 6.15: Laufzeitmessung der integrierten Angriffserkennung

teilung gefundenen Art des Angriffs. Hier wird zwischen DDoS- und Wurm-Angriff unterschieden. Diese Information kann vom Modul zur Analyse der Adressverteilung abgefragt werden.

Es fehlt eine sehr wichtige Nachricht, welche hier nicht eingezeichnet ist, wird nicht zwischen Module ausgetauscht sondern vom ModuleManager verarbeitet. Da ein Granularitäts-basierter Ansatz verfolgt wird, werden im Normalzustand nur die Channels der Stufe 1 betrieben. Das Modul zur Überwachung der Aggregate sendet bei einem erkannten Angriff eine *Alert-Nachricht* aus. Diese wird vom **ModuleManager** des Framework empfangen. Auf diese Weise kann der Channel der Stufe 2 geladen werden. Entsprechend wird auch die dritte Stufe geladen. Bei Empfang der Alert-Nachricht wird die nächste Stage geladen. Dies ist die nächsthöhere Stufe als die, aus der die Nachricht von einem Modul gesendet wurde. Entsprechend funktioniert das Entladen einer Nachricht. Hierzu existiert in der Alert-Nachricht eine entsprechende Option, so dass diese zum Laden – aufklingender Alarm – und Entladen – abklingender Alarm – verwendet werden kann.

Wenn hier von „Abfragen“ die Rede ist, so ist das Aussenden einer Request-Nachricht für bestimmte Daten gemeint, nicht die explizite Anfrage an ein Modul. Da bei der Integration der Angriffserkennung relativ wenige Module beteiligt sind, kann explizit modelliert werden, welche Module letztendlich interagieren. Somit stellt Abbildung 6.14 eine Modellierung der Kommunikation zwischen den Modulen dar. Die Nachrichten werden jedoch nicht direkt zwischen den Modulen ausgetauscht, sondern über das datenzentrierte Nachrichtensystem übertragen.

6.2.1.3 Evaluierung der Skalierbarkeit

Die Skalierbarkeit wird entsprechend der in Abschnitt 6.1.1 durchgeführten Online-Messung für Durchsatz und CPU- und Speicherlast betrachtet.

Abbildung 6.15(a) zeigt den erreichbaren Durchsatz. Dieser liegt knapp unter dem in Abschnitt 6.1.1 gemessenen Durchsatz für die Messung mit nur einem Modul. Daher ergibt sich ein geringer Overhead durch Verwendung der Angriffserkennung. In Abbildung 6.15(b) zeigen sich erhöhte Frame Drops des Pcap Systems. Diese entsprechen ungefähr der in Abschnitt 6.1.1.1 durchgeführten Messung mit Parsen von Frames. Die in Abbildung 6.16 dargestellte CPU- und Speicherlast bei Verwendung der Angriffserkennung ist zufriedenstellend: die CPU-Last liegt bei unter 10%, die

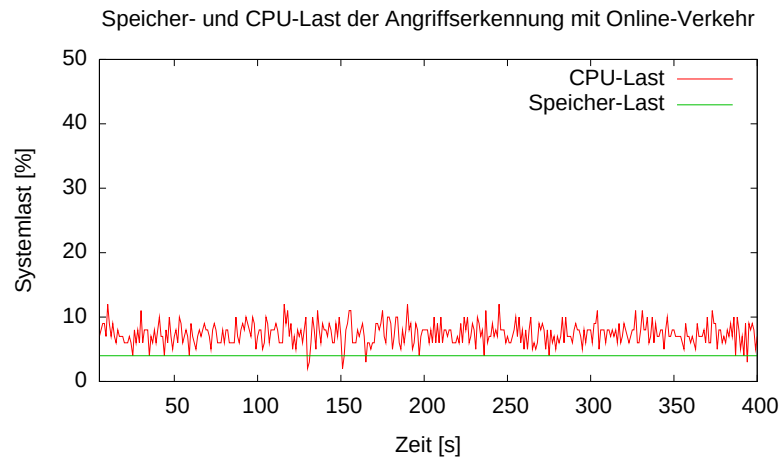


Abbildung 6.16: CPU- und Speicherlast bei Verwendung der Angriffserkennung mit Online-Verkehr

Speicherlast bei 4%.

Es sei hier darauf hingewiesen, dass die Angriffserkennung normalerweise unter Verwendung von Sampling- und Filtering-Methoden betrieben wird. Diese wurden hier explizit nicht verwendet, um den maximal erreichbaren Durchsatz messen zu können. Die Verwendung von Sampling- und Filtering-Methoden reduziert die Systemlast und ermöglicht so einen höheren Durchsatz sowie eine schnellere Verarbeitung.

6.2.1.4 Zusammenfassung

Die Integration der Angriffserkennung in das Distack-Framework wurde durch ein Zerteilen der Aufgaben erreicht. Somit sind Bausteine entstanden, welche auch anderweitig verwendet werden können. Weiterhin ist die Angriffserkennung nun flexibler einsetzbar und profitiert von allen Vorteilen des Distack-Frameworks, wie zum Beispiel dem Betrieb in verschiedenen Laufzeitumgebungen. Weiterhin können bestimmte Bausteine auf entfernte Instanzen ausgelagert werden und somit den Ressourcen des Vermittlungssystems, auf dem sie betrieben werden angepasst werden. Die Skalierbarkeit in Bezug auf Durchsatz und Ressourcenlast ist zufriedenstellend und ermöglicht den Betrieb auf einem Vermittlungssystem.

6.2.2 MPLS-Überwachung

Das Multi Protocol Label Switching Protokoll [RoVC01] ist ein noch recht junges Protokoll, welches im Backbone Bereich verwendet wird, um die verbindungsorientierte Übertragung von Datenpaketen in einem verbindungslosen Netz zu ermöglichen. Hierzu werden Pfade aufgebaut, anhand derer Datenpakete weitergeleitet werden. Dies ermöglicht ein vereinfachtes sowie vordefiniertes Forwarding und erspart somit aufwändiges und nicht-deterministisches Routing.

In [Rey06a, Rey06b, BeMo05, Mars01] wurde auf Sicherheitsprobleme welche durch die Verwendung von MPLS und MPLS-basierten VPNs entstehen können hingewiesen sowie konkrete Angriffsszenarien beschrieben. Daher ist die Integration einer Überwachung von MPLS in das Distack-Framework wünschenswert. Diese wird

im Folgenden beschrieben. Somit wird die Erweiterbarkeit und einfache Integration von neuen Protokollen gezeigt. In Abschnitt 6.2.2.1 wird die Erweiterung der Protokoll-Parser um das MPLS-Paket beschrieben. Die Verwendung eines Moduls zur Erkennung von MPLS-basierten Angriffen wird in Abschnitt 6.2.2.2 dargestellt.

6.2.2.1 MPLS Protokoll-Parser

Es wird hier die Verwendung von MPLS betrachtet um IP-Pakete über Ethernet zu transportieren. Hierzu wird zwischen dem Ethernet- und IP-Paket ein MPLS-Paket eingefügt. Der Aufbau des MPLS-Pakets ist in Abbildung 6.17 dargestellt.

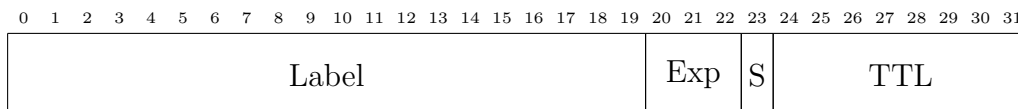


Abbildung 6.17: Bit-basiertes Format eines MPLS-Pakets

Um das MPLS-Paket in die in Abschnitt 2.5 vorgestellten Protokoll-Parser zu integrieren sind folgende Schritte notwendig:

- Erstellung einer neuen Klasse `MplsPacket`, welche von der `Packet`-Klasse abgeleitet wird. Diese muss die Methoden `parsePacket`, `toString` und `getMinProtocolSize` implementieren und Zugriff auf die Protokoll-Attribute zur Verfügung stellen.
- Erweiterung der `Packet`-Klasse um den Protokoll-Typ `PROTO_MPLS`. Dieser dient zur Identifikation des Protokolls in der Protokollkette.
- Erweiterung der `EthernetPacket`-Klasse um den MPLS-Typ MPLS-Unicast (0x8847) und MPLS-Multicast (0x8848) für das `type`-Feld des Ethernet-Pakets.
- Kleinere Anpassungen in den Klassen `PacketFactory` und `ProtocolNameResolver` (siehe hierzu [Maye06]).
- Erweiterung der `IpPacket` Klasse für das `IP-protocol`-Feld um den Wert 0x89. Hiermit wird der Transport von MPLS über IP ermöglicht wird.

6.2.2.2 MPLS-Angriffe und Überwachung

MPLS wird grundsätzlich nur in Provider-Netzen betrieben. Eingehende Frames werden entsprechend durch das MPLS-Paket mit einem Label versehen und durch das Netz weitergegeben. Daher ist normalerweise von starkem Vertrauen innerhalb des Provider-Netzes auszugehen. In [Thom03] wird berichtet, dass über 5000 kompromitierte Router in Provider-Netzen gefunden wurden. Daher muss davon ausgegangen werden, dass Denial-of-Service-Angriffe auch von innerhalb eines Provider-Netzes ausgehen können.

In [BeMo05] wird beschrieben, welche Möglichkeiten bestehen um den MPLS-Core

über DoS-Angriffe anzugreifen. Dabei spielt die frühzeitige Erkennung eines DoS-Angriffs eine entscheidende Rolle. Hierfür kann auf die in Abschnitt 6.2.1 entwickelten Module zurückgegriffen werden, welche für die Integration der Angriffserkennung verwendet wurden. Somit wird deutlich, dass Bausteine, welche für die Verwendung in einer Integration der Angriffserkennung entwickelt wurden, weiterhin verwendet werden können und so ein hohes Maß an Komponierbarkeit gegeben ist.

Das in Abschnitt 6.2.1 entwickelte Modul zur Überwachung von Protokoll- und Größenaggregaten überwacht alle bekannten Protokolle und in der Konfiguration definierte Größenintervalle. Dies bedeutet, dass jeder eingehende Frame geparkt wird und jedes der gekapselten Protokolle sowie der Frame selbst als Aggregat betrachtet wird. Dies hat zur Folge, dass die Aggregat-Überwachung für ein Protokoll dann beginnt, wenn dieses Protokoll das erste Mal gesehen wird. Existiert für ein Protokoll noch keine Überwachung, so wird eine neue erstellt. Durch diesen Mechanismus werden alle Protokolle, für welche Protokoll-Parser zur Verfügung stehen, automatisch überwacht. Kommt ein Protokoll nicht im Netzwerkverkehr vor, so wird auch keine Überwachung vorgenommen. Dies ist sinnvoll, da in einem Netzsegment oft nur bestimmte Protokolle zu sehen sind und somit ein ressourcenschonender Ansatz verfolgt wird, da keine nicht benötigten Datenstrukturen erstellt werden. Die Überwachung eines Protokolls ab dessen ersten Auftreten bedeutet, dass die MPLS-Überwachung sich automatisch integriert, da entsprechende Protokoll-Parser integriert wurden. Somit sind keine Änderungen an dem Modul zur Aggregatüberwachung notwendig: der Baustein kann direkt verwendet werden um das MPLS-Aggregat zu überwachen.

Offensichtlich ist die oben vorgestellte MPLS-Überwachung nicht ausreichend. Weitere Analysemodule müssen entwickelt werden um eine vollwertige MPLS-Überwachung zu erreichen. Sinn der hier durchgeführte Integration von MPLS in das Distack-Framework ist, die einfache Integration von neuen Protokollen in das Distack-Framework zu zeigen.

6.3 Flexibilität

Die Flexibilität des Distack-Frameworks wird einmal in Abschnitt 6.3.1 durch die Portierung in den OMNeT++-Simulator und weiterhin durch die Erstellung von Use-Cases in Abschnitt 6.3.2 gezeigt.

6.3.1 Portierung nach OMNeT++

Die Portierung des Distack-Framework in den OMNeT++-Simulator stellt die Integration in eine andere Laufzeitumgebung dar und ermöglicht beispielsweise die Simulation von Wurm- und DDoS-Angriffen in simulierten Netzen. In Abschnitt 2.7 wurde eine Einführung in den OMNeT++-Simulator gegeben. In diesem Abschnitt wird detailliert erläutert welche Änderungen und Erweiterungen notwendig sind, um ein komplexes System – wie das in dieser Arbeit entwickelte Framework – in die OMNeT++-Simulationsumgebung zu integrieren. Zur Realisierung von IP-basierten Netzen in OMNeT++ wird das INET-Framework [Varg07] verwendet, welches die wichtigsten Pakete und Protokolle zur Verfügung stellt.

6.3.1.1 Kapselung des Framework im einem Simple-Module

Wie in Abschnitt 2.7 erläutert, arbeitet OMNeT++ auf der Basis von verschachtelten Modulen, wobei die niedrigste Modulebene das Simple-Module darstellt. Diese können in C++ programmiert werden. Um das Framework zur Angriffserkennung in OMNeT++ zu integrieren ist es notwendig, das Framework als Simple-Module zu kapseln. Im Gegensatz zu einer Laufzeitumgebung wie Linux oder Windows, wo eine `main`-Funktion die Funktionalität des Programms kapselt, wird ein Simple-Module mit Namen `DistackOmnetModule` definiert und die Funktionalität der `main`-Funktion in dieses gekapselt. Die Initialisierung der Objekte – welche normalerweise in der `main`-Funktion stattfindet – wird nun in der überschriebenen `SimpleModule::initialize`-Funktion vorgenommen. Die `DistackOmnetModule`-Klasse stellt somit ein OMNeT++-Modul dar, welches die Funktionalität des Frameworks kapselt und in einer Simulation verwendet werden kann.

6.3.1.2 Netzwerkabstraktion

In Abschnitt 4.2 wurde die Abstraktion vom unterliegenden Netzwerk durch das Framework vorgestellt. Hierzu wird eine neue Klasse `NetworkInterfaceOmnetpp` erstellt, welche das in Abschnitt 5.1.1 beschriebene `NetworkInterface`-Interface implementiert. Hauptaufgabe des `NetworkInterfaceOmnetpp`-Klasse ist die Konvertierung von OMNeT++-Paketformaten in die von den in Distack verwendeten Protokoll-Parsern verwendeten. OMNeT++ und das INET-Framework verwenden zur Kommunikation zwischen Modulen Message-Objekte. Diese können ineinander gekapselt werden und somit zum Beispiel die Kapselung von Netzwerk-Paketen nachstellen. Die verwendeten Paket-Klassen sind nicht RFC-konform aufgebaut und teilweise nicht vollständig. Beispielsweise werden andere Werte für das ICMP-Typ Feld verwendet als im entsprechenden RFC [Post81a] spezifiziert. Daher müssen die Paket-Klassen in der `NetworkInterfaceOmnetpp`-Klasse in die von Distack verwendeten Paket-Klassen konvertiert werden. Dieser Vorgang ist beispielhaft in Abbildung 6.18 dargestellt: ein `cMessage`-Objekt mit gekapseltem `IPDatagram` und `TCPSegment` wird entsprechend den in Abschnitt 2.5 vorgestellten Protokoll-Parsern konvertiert. Dabei müssen alle Protokoll-Attribute gesetzt werden und die Kapselung entsprechend modelliert werden.

Jede Interaktion mit OMNeT++ geschieht über das oben beschriebene `DistackOmnetModule`-Modul. Somit werden auch alle Netzwerk-Frames über die überschriebene `SimpleModule::handleMessage`-Funktion erhalten. Dies entspricht nicht der Architektur des Frameworks, wo alle Frames über die Netzwerkabstraktion erhalten werden. Um die bisherige Architektur beizubehalten, besitzt das `DistackOmnetModule`-Modul direkten Zugriff auf die beschriebene `NetworkInterfaceOmnetpp`-Klasse. Die `NetworkInterfaceOmnetpp`-Klasse konvertiert erhaltene Frames entsprechend den Protokoll-Klassen und fügt den Frame direkt in die Warteschlange des `FrameBuffer` ein. Ab diesem Zeitpunkt kann ein Frame wie bisher auch verarbeitet werden und es ist keine Spezialbehandlung für die OMNeT++-Umgebung mehr notwendig. Höheren Schichten müssen somit nicht angepasst werden und arbeiten einheitlich in unterschiedlichen Laufzeitumgebungen.

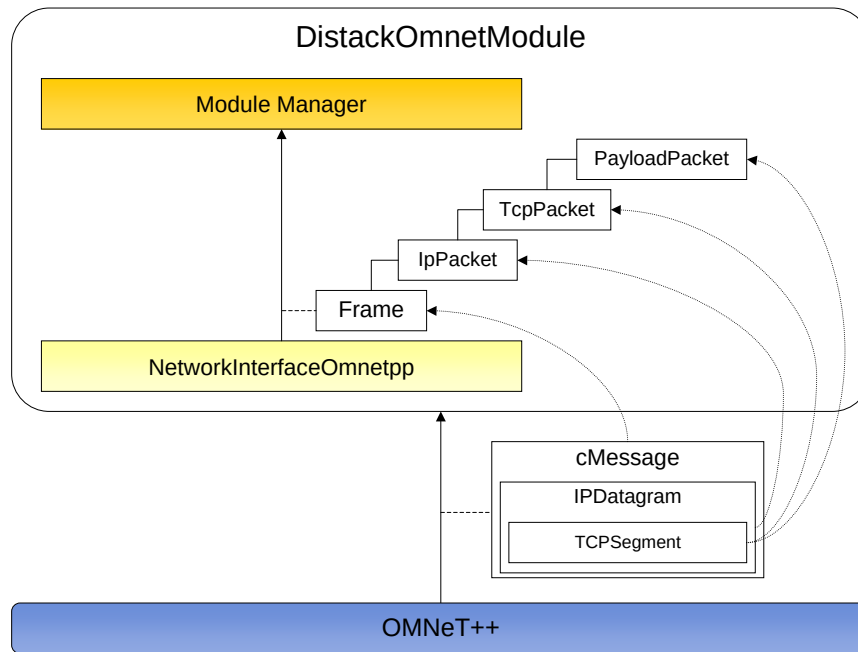


Abbildung 6.18: Konvertierung von OMNeT++ in Distack Protokoll-Klassen

6.3.1.3 Timer- und Zeit-Funktionalität

Die OMNeT++-Umgebung arbeitet zeitdiskret. Dies bedeutet auch, dass sie ihre eigene Zeitdomäne, die Simulationszeit, verwendet. Diese muss im Framework beachtet werden, sonst kommt es zu zeitlichen Verzerrungen. Dies betrifft zum einen Timer, zum anderen Funktionen, welche die aktuelle Zeit für Messungen benötigen.

Timer-Funktionalität

Timer werden in OMNeT++ durch sogenannte *Self-Messages* realisiert. Dies sind Nachrichten, welche ein OMNeT++-Modul an sich selbst sendet. Dabei wird eine Verzögerung angegeben. Diese sagt aus, wann die Simulationsumgebung die Nachricht verarbeiten und somit zustellen soll. Periodische Timer können dadurch realisiert werden, dass eingehende Timer-Nachrichten sofort wieder an sich selbst gesendet werden. Der Versand von Nachrichten benötigt Zugriff auf das OMNeT++-Modul. Das OMNeT++-Modul ist hier durch das `DistackOmnetModule` gegeben. Weiterhin muss zwischen Timer-Nachrichten und Frames unterschieden werden. Dies ist durch die Abfrage möglich, ob es sich bei der Nachricht um eine Self-Message handelt. Das `DistackOmnetModule`-Modul muss für die Unterstützung von Timer-Funktionalität im Framework entsprechend erweitert werden. Hierzu wird die in Abschnitt 5.2.5.5 beschriebene Timer-Klasse angepasst. Sie greift auf das `DistackOmnetModule`-Modul zu und registriert sich dort als Timer. Die Handhabung von Self-Messages wird nun entsprechend im `DistackOmnetModule`-Modul gekapselt. Dies ist notwendig um eine Menge an Timern auf die Self-Messages abzubilden und somit die Zuordnung von Self-Messages zu Timern zu realisieren. Hiermit wird die transparente Verwendung von Timern, welche in der Domäne der Simulationszeit laufen, möglich.

Zeit-Funktionalität

Die in Abschnitt 5.6.4 vorgestellte **Helper**-Klasse bietet die Funktion `getTime`, um die aktuelle Uhrzeit abzufragen. Die Simulationszeit kann über die `simTime`-Funktion der `cModule`-Klasse abgefragt werden. Das `DistackOmnetModule`-Modul wird von `cSimpleModule` abgeleitet und erbt somit auch die Methoden der `cModule`-Klasse. Somit ist der Zugriff auf die `simTime`-Funktion möglich. Hierdurch können Module, welche Zeitinformationen benötigen, diese transparent unter verschiedenen Laufzeitumgebungen entsprechend der korrekten Zeitdomäne abfragen.

6.3.1.4 Threads

Ein Großteil der Funktionalität des Distack-Framework basiert auf Threads. Da die OMNeT++-Simulationsumgebung zeitdiskret arbeitet, steht dies in direktem Widerspruch. Die Zeitmessung von OMNeT++ bezieht sich jedoch nur auf die Dauer der Nachrichtenübertragung: nur Leitungsverzögerungen werden zur Simulationszeit gerechnet. Die Verarbeitungszeit innerhalb eines Moduls wird dabei nicht betrachtet. Daher ist die Verarbeitung von Funktionalität in Threads gegenüber der Zeitdomäne der Simulationsumgebung legitim.

Dessen ungeachtet, birgt die Verwendung von Threads in der OMNeT++-Simulationsumgebung Probleme. Eine laufende Simulation verbraucht meist alle vorhandene Prozessorzeit um schnellstmöglich abzulaufen. Somit ist die CPU-Belastung so hoch, dass nicht sichergestellt ist, dass andere Threads genügend Prozessorzeit zur Verfügung haben um korrekt zu arbeiten. Kurze Threads sind von diesem Problem meist nicht betroffen, da sie innerhalb weniger Scheduling-Zyklen abgearbeitet werden können. Lange Threads, welche zum Beispiel in einer Schleife Elemente einer Warteschlange verarbeiten, arbeiten hier nicht mehr korrekt. Probleme kommen im Distack-Framework an folgenden zwei Stellen auf: der Verarbeitung von Frames durch das `FrameDistribution`-System, sowie die asynchrone Verarbeitung von Nachrichten durch das `MessagingSystem`. In Tests konnten deutliche Verzögerungen bei der Verarbeitung von Frames und Nachrichten erkannt werden, welche beide zu zeitlichen Verzerrungen führen. Um dieses Problem zu lösen, muss die Verarbeitung in diesen Klassen zeitdiskretisiert werden. Beide Klassen basieren auf Warteschlangen welche durch einen Thread verarbeitet werden. Wird die OMNeT++-Laufzeitumgebung verwendet, so werden diese Threads nicht verwendet. Stattdessen werden die Warteschlangen nach jedem Aufruf der `handleMessage`-Funktion durch OMNeT++ explizit verarbeitet: jedes Element der beiden Warteschlangen wird verarbeitet und die Warteschlange damit geleert. Dieses Vorgehen wurde durch Tests validiert: es zeigen sich keinerlei zeitliche Verzerrungen, weder in der Verarbeitung von Frames noch in der Verarbeitung von Nachrichten, welche zwischen den Modulen versandt werden.

Neben den oben beschriebenen Threads, welche zeitdiskretisiert wurden, existieren im Framework kurze Threads. Weiterhin haben Module die Möglichkeit eigene Threads zu erstellen. Diese müssen nicht zeitdiskretisiert werden, da sie innerhalb weniger Scheduling-Zyklen abgearbeitet werden. Da OMNeT++ zeitdiskret arbeitet, ist der Simulationsumgebung zu jedem Zeitpunkt bekannt, welches OMNeT++-Modul aktiven Code ausführt. Dies stellt bei der Verwendung von Threads ein Problem dar. Folgendes Szenario verdeutlicht das Problem: es wird ein Thread verwen-

det durch welchen das Laden eines Moduls ausgelöst wird. Dieses Modul verwendet einen Timer und registriert sich somit beim `DistackOmnetModule`-Modul. Das `DistackOmnetModule`-Modul sendet nun eine Self-Message in die Simulationsumgebung. Hier ist das `DistackOmnetModule`-Modul, welches die Self-Message aussendet, nicht als das aktive Modul in OMNeT++ bekannt. Dies führt zu Problemen innerhalb von OMNeT++ und schließlich zum Absturz der Simulation. Das Problem kann unter Verwendung der `Enter_Method_Silent`-Funktion von OMNeT++ behoben werden. Hiermit wird der aktuelle Kontext von OMNeT++ korrekt gesetzt. Hiermit wird die Verarbeitung innerhalb anderer OMNeT++-Teile angehalten und das `DistackOmnetModule`-Modul als aktives Modul gekennzeichnet.

6.3.1.5 Routing-Tabelle

In Abschnitt 5.6.2 wurde die `GenericRoutingTable`-Klasse zum Zugriff auf die Routing-Tabelle unter Windows und Linux, sowie über Dateien vorgestellt. Unter OMNeT++ besitzt ein Router-Knoten eine eigenen Routing-Tabelle. Die Abfrage der Routing-Tabelle unter OMNeT++ muss daher in der `GenericRoutingTable`-Klasse separat behandelt werden.

Zur Abfrage der Routing-Tabelle muss als erstes das Vater-Modul gefunden werden, welche die Routing-Informationen beinhaltet. Dies kann über die `findModuleSomewhereUp`-Funktion von OMNeT++ realisiert werden. Hierfür ist der Zugriff auf das `DistackOmnetModule` notwendig:

```
cModule* m = findModuleSomewhereUp ("routingTable", (cModule*)module);
RoutingTable* omnetTable = check_and_cast<RoutingTable*> (m);
```

Nun können die Einträge der OMNeT++ Routing-Tabelle durchlaufen werden und entsprechend den Datenstrukturen des Distack-Framework konvertiert werden. Somit ist der Zugriff auf die Routing-Information auch aus der OMNeT++-Laufzeitumgebung transparent möglich.

6.3.1.6 Prozessraum

Sollen Module für die Verwendung im Distack-Framework unter der OMNeT++-Simulationsumgebung entwickelt werden, so ist ein wichtiger Punkt zu beachten: alle Instanzen der Angriffserkennung teilen sich in einer OMNeT++-Simulation den Adressraum. Unter einer normalen Laufzeitumgebung wie Windows oder Linux wird meist nur eine Instanz der Angriffserkennung verwendet. Werden mehrere Instanzen gestartet, so besitzt jede einzelne ihren eigenen Adressraum. Die Verwendung von mehreren Instanzen der Angriffserkennung ist in einer Simulation üblich. In einer OMNeT++-Simulation teilen sich jedoch die Simulationsumgebung und alle Instanzen der Angriffserkennung den Adressraum. Dies hat zur Folge, dass statische Objekte sowie Singletons nur *einmal* existieren und von allen Instanzen der Angriffserkennung gemeinsam verwendet werden. Statische Variablen und Singletons müssen daher mit Vorsicht verwendet werden.

Die gemeinsame Verwendung von Singletons kann aber auch einen Vorteil darstellen und der Speicherverbrauch der Simulation verringert werden. Dies ist beispielsweise

bei den in Abschnitt 2.5 vorgestellten Protokoll-Parsern der Fall. Diese verwenden, unter Hilfe der Boost Pool Library [Clea00], Speicherpools zur schnellen Allokation und Deallokation von Speicher. Wird nun dieser Speicherpool von allen Instanzen der Angriffserkennung in einer Simulation gemeinsam verwendet, so reduziert sich der Gesamtspeicher, welcher durch den Speicherpool benötigt wird.

6.3.1.7 Build-Umgebung und NED-Dateien

Bisher wurde nicht erwähnt, wie das Distack-Framework konkret in OMNeT++ verwendet werden kann. Es wurden in Abschnitt 6.3.1.1 Änderungen beschrieben welche auf Seiten des Framework initial notwendig sind um das Zusammenspiel zu ermöglichen. Hier wird nun erläutert, welche Änderungen an der Build-Umgebung vorzunehmen sind. Es wird von einer Linux-basierten Build-Umgebung ausgegangen. Änderungen, zum Beispiel für die Verwendung unter Windows, sind gering.

Zunächst einmal ist es wichtig zu verstehen, wie eine Simulation in OMNeT++ gebaut wird. Dies geschieht normalerweise über das `opp_makemake`-Skript. Dieses analysiert die eigenen Quelldateien und erstellt ein Makefile, welches die folgenden Komponenten kompiliert :

- NED-Topologie-Beschreibungen: Übersetzt mit dem `nedtool`-Compiler
- Message Definitions Dateien: Übersetzt mit dem `opp_msgc`-Compiler
- C++-Quelldateien: Übersetzt mit einem C++-Compiler

Die kompilierten Komponenten werden dann mit dem Simulationskernel `libsim_std` und der Benutzeroberfläche `libenvir` sowie `tkenv` oder `cmdenv` zusammengelinkt. Die entstehende Datei ist eine ausführbare Anwendung, welche alle notwendigen Komponenten beinhaltet um als eigenständige Simulation lauffähig zu sein. Um eine Simulation auszuführen, ist eine Konfigurationsdatei `omnetpp.ini` notwendig. In ihr werden noch freie Parameter, wie zum Beispiel die Anzahl an Knoten in einem Netz, definiert.

OMNeT++ erlaubt die Nutzung von dynamischen Bibliotheken. Hierzu wird in der Konfigurationsdatei `omnetpp.ini` unter der Gruppe `[General]` der Eintrag `load-libs=libdistack` hinzugefügt. Dies lädt bei Start der Simulation die Bibliothek `libdistack.so` in den Adressraum der Simulation. Um das Distack-Framework als Bibliothek zu erstellen ist die Verwendung der `g++`-Optionen `-fPIC -rdynamic -shared` notwendig.

In einer normalen Laufzeitumgebung wird das Distack-Framework als Anwendung ausgeführt. Über dynamische Bibliotheken werden Module geladen, welche die Angriffserkennung implementieren. Im Falle der OMNeT++-Simulationsumgebung wird das Distack-Framework *selbst* als dynamische Bibliothek von OMNeT++ geladen. Weiterhin lädt das Distack-Framework Module zur Angriffserkennung. Beim Laden einer Bibliothek werden, wie in Abschnitt 2.6 beschrieben, alle nicht aufgelösten Symbole aufgelöst. Symbole, welche sich in der dynamischen Bibliothek befinden, und nicht aufgelöst werden müssen, sind später nicht mehr bekannt. Da die Module zur Angriffserkennung Funktionalität aus dem Framework verwenden,

ist diese nicht zu einem späteren Zeitpunkt verfügbar. Dies ist aber notwendig, da Module zur Angriffserkennung vom Distack-Framework zu einem späteren Zeitpunkt als dynamische Bibliothek geladen werden. Daher ist eine kleine Änderung an der Art des Ladens von dynamischen Bibliotheken in OMNeT++ notwendig. Das `include/platdep` Verzeichnis von OMNeT++ enthält die Datei `loadlib.h`. Hierin ist die `opp_loadlibrary`-Funktion definiert: sie wird zum Laden von dynamischen Bibliotheken verwendet. Wie in Abschnitt 5.2.5.2 beschrieben, wird unter Linux die `dlopen`-Funktion zum Laden von dynamischen Bibliotheken verwendet. Dies ist in der `opp_loadlibrary`-Funktion wie folgt realisiert:

```
inline bool opp_loadlibrary(const char *libname)
{
    std::string libfname(libname);
    libfname += ".so";
    if (!dlopen(libfname.c_str(), RTLD_NOW))
        throw std::runtime_error (...);
    return true;
}
```

Um in dynamischen Bibliotheken vorhandene Symbole öffentlich – und somit für das spätere Laden von dynamischen Bibliotheken verfügbar zu halten – ist neben dem `RTLD_NOW`-Flag das `RTLD_GLOBAL`-Flag notwendig. Somit ändert sich der Aufruf der `dlopen`-Funktion zu:

```
dlopen(libfname.c_str(), RTLD_NOW | RTLD_GLOBAL)
```

Auf diese Weise ist die Verwendung von Funktionalität aus dem Distack-Framework durch dynamisch geladenen Module auch unter OMNeT++ möglich.

Um das Distack-Framework in einer OMNeT++-Simulation zu verwenden, sind neben dem `load-libs`-Eintrag in der `omnetpp.ini`-Datei NED-Dateien notwendig. Diese werden hier beispielhaft vorgestellt. Über die NED-Dateien wird das eigentliche OMNeT++-Modul definiert, welches die Angriffserkennung beinhaltet. Abbildung 6.19 zeigt die notwendigen NED-, ini- und so-Dateien, welche für die Simulation mit dem Distack-Framework und OMNeT++ benötigt werden. Über die Pfeile wird verdeutlicht welche Dateien ineinander eingebunden werden. Die Inhalte der Dateien sind nur lückenhaft dargestellt und enthalten die wichtigsten Informationen für die Verwendung des Distack-Framework in OMNeT++.

- `omnetpp.ini`: Wird von OMNeT++ bzw. vom INET-Framework eingelesen. Sie definiert die zu ladende Bibliothek `libdistack.so` sowie das Netzwerk `distackTestSimulation` welches simuliert werden soll.
- `libdistack.so`: Beinhaltet das Distack-Framework zur Angriffserkennung und eine Klasse `DistackOmnetModule`, welche das Simple-Module darstellt.
- `distackTestSim.ned`: Definiert Netzwerk-Topologie unter Verwendung von Knotentypen sowie den Namen des Netzwerks.

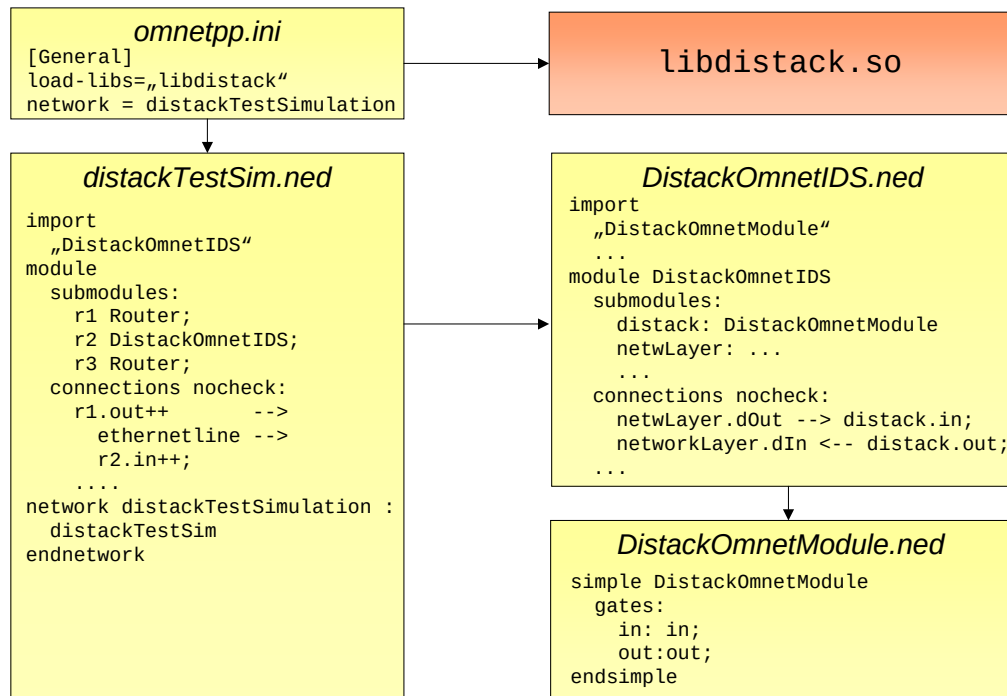


Abbildung 6.19: Konfiguration für die OMNeT++-Simulation mit dem Distack-Framework

- **DistackOmnetIDS.ned:** Definiert einen Knoten im Netzwerk. Dieser beinhaltet einen Protokollstapel. Das Distack-Framework wird dabei über der Vermittlungsschicht betrieben.
- **DistackOmnetModule.ned:** Definiert das Simple-Module, welches als Symbol in der `libdistack.so` Bibliothek vorliegen muss. Dies ist der Name der von `cSimpleModule` abgeleiteten Klasse `DistackOmnetModule`, welche in der `libdistack.so`-Bibliothek integriert ist und das Framework beinhaltet (siehe Abschnitt 6.3.1.1).

6.3.1.8 Simulation der Angriffserkennung

Zur Evaluierung der Flexibilität wurde das Distack-Framework in die OMNeT++-Umgebung portiert. Dies wird nun verwendet um die in Abschnitt 6.2.1 in das Distack-Framework integrierte Angriffserkennung in OMNeT++ zu betreiben. Hierzu wird der von Michael Scharf entwickelte Topologie-Generator [Scha07a] verwendet um eine realitätsnahe Internettopologie zu erstellen. Der Topologie-Generator ermöglicht weiterhin die Erzeugung von realistischem Hintergrundverkehr, der die Bedingung der Selbstähnlichkeit erfüllt.

Zur Angriffserkennung wird eine DDoS-Anwendung verwendet welche, ebenfalls im Rahmen der Arbeit von Scharf entwickelt wurde. Die Struktur der generierten Topologie ist in Abbildung 6.20 dargestellt. Sie besitzt ein Autonomes System, drei Core-Router, 9 Gateway-Router, 113 Edge-Router sowie 3257 Randsysteme. Das Distack-Framework wird dabei auf einem Core-Router im Netzinernen betrieben. In

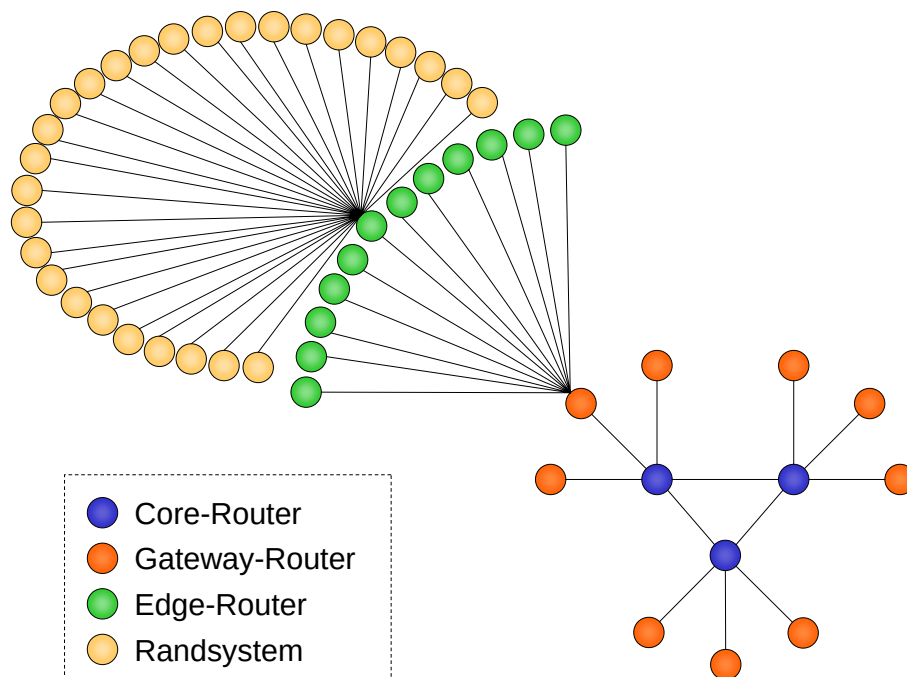


Abbildung 6.20: Topologie zur Simulation der in das Distack-Framework integrierten Angriffserkennung in OMNeT++

der Topologie verteilt befinden sich 150 Zombie-Systeme auf Randsystemen. Diese führen einen verteilten DoS-Angriff auf ein einzelnes Opfersystem aus. Dieses befindet sich – wie auch die Zombie-Systeme, am Rand des Netzes.

Die Darstellung des Zählers und Schwellenwerts für das TCP-Aggregat ist in Abbildung 6.21 gezeigt. Als Grundüberwachung wird nur Stufe 1 betrieben. In Intervall 608 wurde der Schwellenwert mehrmals überschritten. Daher wird die zweite Stufe geladen. Diese nimmt eine Analyse der Subnetzverteilung vor, findet einen verteilten Angriff und veranlasst dann das Laden der dritten Stufe. Diese überwacht den Verkehrsstrom und meldet eine TCP-Count- und TCP-Out-SYN-ACK-Anomalie bezüglich dem Opfersystem.

Das Distack-Framework bietet die Möglichkeit integrierte Angriffserkennungen ohne Änderung in einer Simulation zu betreiben. Hierdurch ergeben sich neue Möglichkeiten zur Simulation von Angriffserkennung in realen Netzen.

6.3.2 Beispielhafte Anwendungsfälle

In den folgenden Abschnitten werden beispielhafte Anwendungsfälle und Konfigurationen beschrieben, in welchen das Distack-Framework betrieben werden kann. Weiterhin werden vielfältige Verwendungsmöglichkeiten beschrieben.

6.3.2.1 Lokale Anwendungsfälle

Die lokalen Anwendungsfälle gliedern sich in Online/Offline sowie Angriffserkennung/Analyse. Für jeden der vier entstehenden Fälle ist in Tabelle 6.3 eine beispielhafte Anwendung gegeben. Diese werden im Folgenden genauer erläutert. Abbildung 6.22 zeigt den Systemaufbau zur lokalen Verwendung des Systems im Falle einer Online- und Offline-Anwendung.

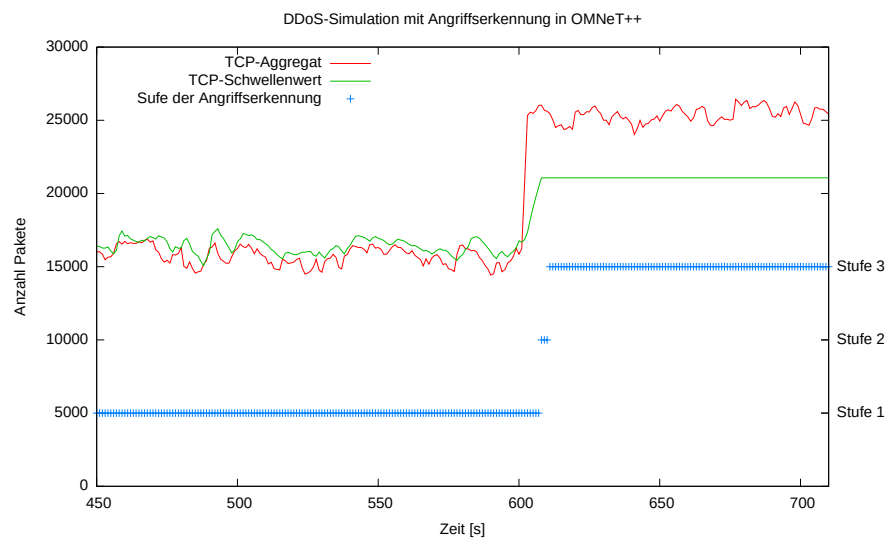


Abbildung 6.21: Überwachung des TCP-Aggregat sowie geladene Stufen der Angriffserkennung

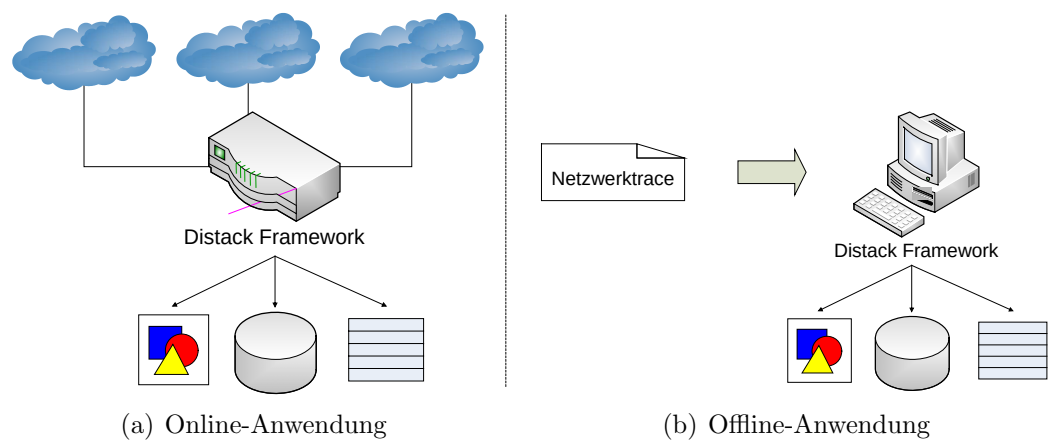


Abbildung 6.22: Systemaufbau für verschiedene Anwendungsfälle

	Online	Offline
Angriffserkennung	Online-Betrieb des Frameworks zur Angriffserkennung in einem realen Netz sowie in einem Simulator.	Post-Mortem Angriffserkennung mit aufgenommenen Netzwerktraces.
Analyse	Analyse und Auswertung von Live-Verkehr.	Analyse und Auswertung von aufgenommenen Netzwerktraces.

Tabelle 6.3: Beispielhafte Anwendungsfälle für den lokalen Betrieb

Online-Betrieb zur Angriffserkennung sowie Simulation

Dieser Anwendungsfall stellt die Hauptaufgabe eines Systems zur Angriffserkennung dar. Über die Verwendung von Bausteinen und den Granularitäts-basierten Ansatz ist es möglich das System zur Angriffserkennung flexibel und ressourcenschonend auf einem Vermittlungssystem zu betreiben.

Weiterhin ist die Verwendung in einem Simulator wie zum Beispiel OMNeT++ (siehe Abschnitt 6.3.1) möglich. Hierdurch ergibt sich die Möglichkeit Angriffe in großen Netzen zu simulieren und dadurch neue Methoden zur verteilten Erkennung von Angriffen zu entwickeln und zu evaluieren.

Post-Mortem Analyse von Angriffen

In der Computer-Forensic stellt die Post-Mortem Analyse von Netzwerkverkehr eine wichtige Rolle dar. Mit dem Distack-Framework ist es möglich (siehe Abschnitt 5.1.1.1) aufgenommene Netzwerktraces in korrekter Zeitdomäne wieder abzuspielen und somit die Originalcharakteristik des Verkehrs zu analysieren und auf Angriffe zu untersuchen. Die Umstellung von Online- zu Offline-Angriffserkennung ist über die Konfigurationsdatei (siehe Abschnitt 5.5.2) möglich und ist so sehr einfach zu betreiben.

Online-Analyse und Auswertung von Live-Verkehr

Über das in Abschnitt 5.7.1.1 beschriebene Plotting-Modul ist es sehr einfach Realverkehr nach Belieben zu visualisieren. Weiterhin ist es einfach möglich eigene Module zu entwickeln, welche statistische Daten über den Netzwerkverkehr sammeln. Wie auch bei der Online-Angriffserkennung ist hier die Möglichkeit gegeben das Framework im OMNeT++-Simulator zu verwenden und den dort anfallenden Verkehr zu analysieren.

Offline-Analyse und Auswertung von aufgenommenem Verkehr

Wie auch bei der Online-Analyse, ist die Auswertung von aufgenommenem Netzwerkverkehr möglich. Dies ermöglicht die Analyse von Netzwerkverkehr, welcher beispielsweise in anderen Netzen aufgenommen oder über Traffic-Archive frei zugänglich gemacht wird [Arch00].

Abbildung 6.23 zeigt zwei beispielhafte Auswertungen einer Netzwerktrace mit Hilfe des Distack-Framework. Durch die integrierten Protokoll-Parser (siehe Abschnitt 2.5) und den Baustein-basierten Aufbau des Framework ist eine Auswertung der Pakete sehr einfach und flexibel zu realisieren. Durch Anpassung des Plotting-Moduls können Daten sofort visualisiert werden. Auch hier ist zu erwähnen, dass die Netzwerkdaten in korrekter Zeitdomäne abgespielt werden, so dass keine zeitlichen Verzerrungen entstehen. Weiterhin ist das beschleunigte Abspielen möglich. Auch hier entstehen keinerlei zeitliche Verzerrungen.

6.3.2.2 Verteilte Anwendungsfälle

Wie auch bei der lokalen Anwendung sind bei der verteilten Anwendung vier Gebiete zu unterscheiden. Diese sind in Tabelle 6.4 dargestellt und werden in den folgenden Abschnitten beispielhaft vorgestellt. Entsprechend den Anwendungsfällen ist

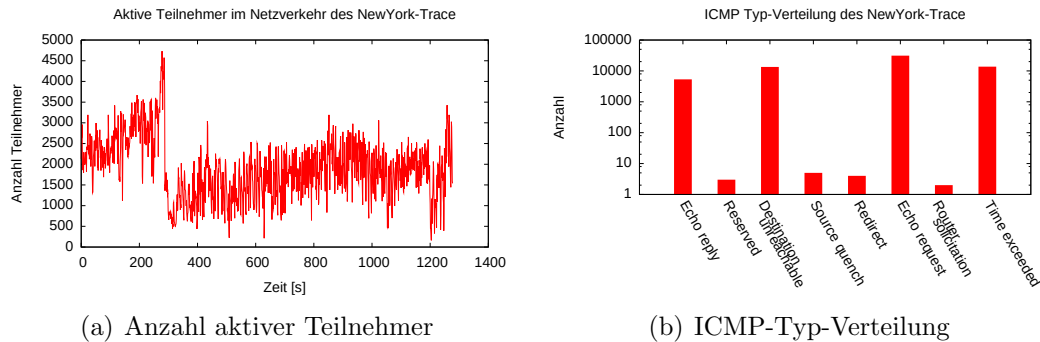


Abbildung 6.23: Mit Hilfe des Distack-Framework erstellte Analysen der NewYork-Trace

in Abbildung 6.24 der Systemaufbau zur verteilten Online-Anwendung dargestellt. Abbildung 6.25 zeigt den Systemaufbau zur verteilten Offline-Anwendung.

Für die verteilte Anwendung sind hier auch Fälle interessant, bei welchen nicht jede Instanz eine vollständige Angriffserkennung bzw. Analyse durchführt. Entsprechend den Ressourcen der Systeme, können auf diesen auch nur einzelne Funktionen in Form von Modulen betrieben werden. Auf diese Weise kann ein System zur Angriffserkennung aufgeteilt und verteilt werden. Auch die Synchronisation von entfernten Instanzen mit Hilfe des in Abschnitt 5.7.1.3 vorgestellten Timer-Moduls kann sehr nützlich sein. Somit ist es möglich mehrere Instanzen zu synchronisieren und auf diese Weise eine bessere Koordination zwischen den Instanzen zu ermöglichen. Offensichtlich ist mit diesen Methoden nur eine ungefähre Synchronisation möglich, da die Übertragung von Daten auf entfernte Instanzen Zeit benötigt. Daher muss bei jedem Anwendungsfall entschieden werden, ob die erreichbare Synchronisation der Instanzen genügt.

Verteilte Angriffserkennung im Online-Betrieb

Dies stellt die Hauptaufgabe des Frameworks zur verteilten Angriffserkennung dar. In einem Netz können an mehreren Punkten Instanzen der Angriffserkennung betrieben werden. Diese kommunizieren über das Nachrichtensystem und können durch den Austausch von Informationen und erkannten Angriffen effektiv arbeiten.

Verteilte Post-Mortem Angriffserkennung

Wurden mehrere Netzwerktraces an verschiedenen Punkten des Netzes aufgenommen, so können diese auf mehreren Instanzen abgespielt werden. Jede Instanz verwendet somit eine andere Netzwerktrace. Die Kommunikation zwischen den Instanzen spiegelt somit das wahre Verhalten wieder, wie es auch im Online-Betrieb stattfinden würde. Eine Post-Mortem Analyse kann somit realistisch durchgeführt und Angriffe erkannt werden.

Verteilte Analyse von Online-Verkehr

An verschiedenen Punkten im Netz ist unterschiedlicher Verkehr zu beobachten. Eine Online-Analyse, zum Beispiel über eine Visualisierung, ermöglicht es Administratoren frühzeitig Probleme zu identifizieren. Durch Betrieb mehrerer Instanzen des

	Online	Offline
Angriffserkennung	Online-Betrieb des Framework mit mehreren verteilten Instanzen die über das Nachrichtensystem kommunizieren.	Zur Post-Mortem Angriffserkennung mit großen Datenbeständen können mehrere Instanzen der Angriffserkennung auf verschiedenen Systemen betrieben werden. Die Ergebnisse der Angriffserkennung werden über das Nachrichtensystem zu einer zentralen Instanz gesendet und dort zusammengeführt.
Analyse	Live-Analyse von Netzwerkverkehr über mehrere Instanzen welche an unterschiedlichen Punkten im Netz betrieben werden. Die Daten können an eine zentrale Management-Instanz gesendet und dort zusammengeführt bzw. zusätzlich visualisiert werden.	Zur Analyse von Datenbeständen über mehrere Instanzen mit zentraler Management-Instanz. Im Falle einer sehr großen Tracedatei kann diese aufgeteilt werden und von mehreren Instanzen analysiert werden. Die Ergebnisse der Analysen werden an die Management-Instanz gesendet und dort zusammengeführt.

Tabelle 6.4: Beispielhafte Anwendungsfälle für den verteilten Betrieb

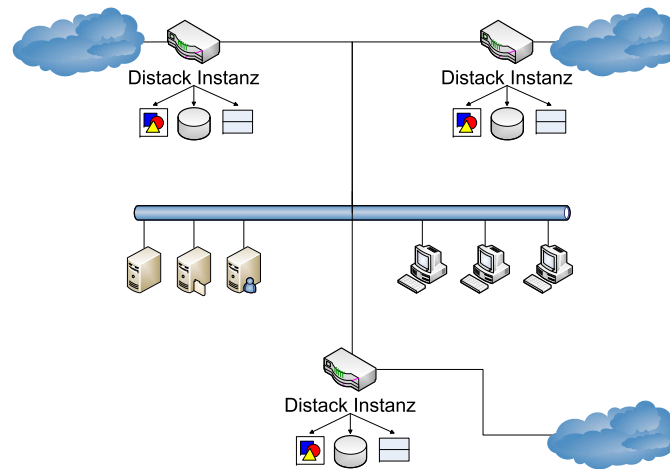


Abbildung 6.24: Systemaufbau zur verteilten Online-Anwendung

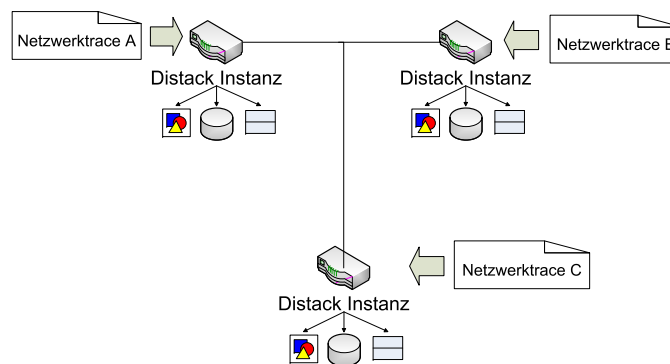


Abbildung 6.25: Systemaufbau zur verteilten Offline-Anwendung

Frameworks an verschiedenen Punkten des Netzes ist es möglich, dass alle Instanzen ihre Informationen zu einer zentralen Instanz senden. Diese kann die Daten der Instanzen visualisieren und somit einen Gesamtblick auf das Netzwerk ermöglichen.

Verteilte Analyse von aufgenommenen Netzwerktraces

Bei großen Datenbeständen muss auf die verteilte Analyse durch mehrere Systeme zurückgegriffen werden. Durch das Distack-Framework ist es möglich, mehrere Instanzen auf entfernten Systemen zur Analyse eines Teils der Daten zu verwenden. Die von jeder Instanz gewonnenen Informationen können dann an eine zentrale Management-Instanz gesendet und zusammengeführt werden.

6.4 Robustheit

Die Robustheit des Distack-Frameworks wird vor allem durch zwei Faktoren beeinflusst: zum einen müssen die Operationen im System korrekt arbeiten und nicht durch korrupte Netzwerkpakete zu Fehlfunktionen leitbar sein. Weiterhin darf das

System nicht durch hohe Verkehrsraten oder eine große Menge an interner Kommunikation überlastet werden. Diese beiden Fälle stellen Angriffspunkte auf das Framework selbst dar und werden in den folgenden Abschnitten 6.4.1 und 6.4.2 untersucht, um die Robustheit des Distack-Frameworks zu zeigen.

6.4.1 Korrupte Netzwerkpakete

Durch korrupte Attribute in den Paketen des Netzwerkverkehrs können die Protokoll-Parser des Systems angegriffen werden. Dass die Verarbeitung von Paketen nicht trivial und fehleranfällig ist, zeigt die hohe Anzahl an veröffentlichten Sicherheitslücken in diesem Bereich. Beispielsweise wurden Anfang des Jahres 2007 zwei Probleme im Angriffserkennungssystem Snort (siehe auch Abschnitt 3.3.1) gefunden, welche durch korrupte ICMP- bzw. TCP-Pakete das System zum Absturz bringen können [Secu07c, Secu07b]. Der WinNuke-Angriff [Fink07], welcher Windows Systeme durch ein einziges Paket zum Absturz brachte, ist ein sehr prominentes Beispiel für fehlerhafte Handhabung von Netzwerkverkehr. Er benutzt den selten verwendeten Urgent Pointer des TCP-Pakets. Dieser wurde vom System nicht korrekt gehandhabt. Auch wenn der WinNuke-Angriff aus dem Jahre 1997 stammt, so ist Anfang des Jahres 2007 ein ähnliches Problem in aktuellen Version von Windows bekannt geworden [Secu07a, Micr07]. Dies zeigt, dass die Handhabung von Netzwerkverkehr sehr komplex ist und die Robustheit sehr beeinflusst.

Wir bereits in Abschnitt 2.5 beschrieben, wurden die hier verwendeten Protokoll-Parser mit dem ISIC-Testprogramm [Fran04] validiert. Dadurch wurde die Robustheit der derzeit implementierten Protokoll-Parser gezeigt. Es ist hier sehr wichtig neue Protokoll-Parser sehr sorgfältig zu testen und Sicherheitsprobleme anderer Systeme – wie zum Beispiel Snort – zu verfolgen.

6.4.2 Überlast

Die Robustheit des Distack-Systems gegenüber Überlast wird in zwei Bereichen gezeigt. Einmal wird gezeigt, dass das System durch sehr hohe Verkehrsraten nicht überlastet werden kann. Zum anderen wird gezeigt, dass das Nachrichtensystem nicht durch eine hohe Zahl an Nachrichten – von lokalen Modulen oder Modulen entfernter Instanzen – überlastet werden kann.

Beide Mechanismen zum Schutz vor Überlastsituationen arbeiten auf ähnliche Weise. Eingehende Frames bzw. Nachrichten werden in eine Warteschlange eingereiht. Die Zustellung von Frames wie auch Nachrichten basiert auf einer asynchronen Verarbeitung. Offensichtlich korreliert die CPU-Last mit dem durchschnittlichen Füllstand der Warteschlangen: Erreichen diese nie einen geleerten Zustand, so arbeitet das System dauerhaft auf hoher CPU-Last. Die Speicherlast lässt sich durch die Füllstände der Warteschlangen beschreiben: jeder im `FrameBuffer` sowie im `MessagingSystem` gespeicherte Frame bzw. jede Nachricht belegt Speicherplatz im System.

Um die CPU- und Speicher-Last durch eine obere Grenze zu beschränken, werden maximale Größen für die Warteschlangen definiert. Hiermit wird sichergestellt, dass die Speicher-Last begrenzt ist. Die CPU-Last wird mit diesem Ansatz nicht begrenzt. Jedoch ist die Wahrscheinlichkeit, dass die Warteschlange einen Leerzustand erreicht sehr viel höher als bei unbegrenzter Warteschlange.

Im Falle einer begrenzten Warteschlange muss entschieden werden, was passieren soll, wenn die Warteschlange vollständig gefüllt ist. Hier können mehrere Ansätze betrachtet werden:

1. Verwerfen des neuen Elements und Beibehalten der vollen Warteschlange
2. Leeren der Warteschlange und Einfügen des neuen Elements
3. Löschen des ältesten Elements der Warteschlange und Einfügen des neuen Elements
4. Löschen eines Elements an einer zufälligen Position der Warteschlange und Einfügen des neuen Elements am Ende der Warteschlange

Die Wahl der verwendeten Methode hängt unter anderem von dem Informationsgehalt der Elemente der Warteschlange ab. Im Falle von Frames kann von einem geringen Informationsgehalt ausgegangen werden. Die zwischen Modulen versendeten Nachrichten haben für das System einen weit höheren Informationsgehalt.

Für die Verarbeitung von Frames wird somit der Ansatz verfolgt, alle Elemente der Warteschlange zu löschen und das neue Element einzufügen (2). Für die Verarbeitung von Nachrichten wird die Strategie gewählt, die älteste Nachricht der Warteschlange zu löschen und die neue einzufügen.

Durch diese Mechanismen kann das System vor Überlastsituationen geschützt werden. Im Gegenzug geht jedoch Information verloren. Die Wahrscheinlichkeit einer Überlastsituation bei der Verarbeitung von Frames kann durch die Anwendung von Sampling- und Filtering-Mechanismen minimiert werden. In diesem Fall werden Frames sehr schnell aus der Warteschlange ausgelesen und entsprechend verarbeitet. Eine Überlastsituation bei der Verarbeitung von Nachrichten ist somit als gering einzustufen. Die von Modulen überwachten Intervalle befinden sich meist im Sekundenbereich. Da Nachrichten in den meisten Fällen durch abgelaufene Intervalle ausgelöst werden, ist die Wahrscheinlichkeit für eine Überlast durch Nachrichten gering.

7. Zusammenfassung und Ausblick

Der *Worldwide Infrastructure Security Report 2007* [Arbo07] nennt DDoS- und Wurmangriffe als die größten Gefahren, welche das Internet derzeit bedrohen. Die zunehmende Heterogenität von Zugangsnetzen und Endsystemen wird dieses Problem in Zukunft verstärken und macht daher ein Umdenken im Bereich der Angriffserkennung notwendig.

Eine kooperative Detektion von Angriffen durch verteilte Instanzen ermöglicht neue Ansätze zur frühzeitigen Identifikation und Bekämpfung von DDoS- und Wurmangriffen. Das in dieser Arbeit entwickelte Framework zur verteilten Anomalie-basierten Angriffserkennung bietet daher ideale Möglichkeiten Methoden zur lokalen und verteilten Angriffserkennung schnell und einfach zu integrieren. Durch die hohe Komponierbarkeit von Modul-Bausteinen und die einfache Konfigurierbarkeit ist es möglich, das Framework flexibel und effektiv zu betreiben. Die Abstraktion der unterliegenden Hard- und Softwareumgebung ermöglicht es Methoden zur lokalen und verteilten Angriffserkennung transparent in unterschiedlichen Laufzeitumgebungen, wie z. B. einem Router oder einem Simulator, zu betreiben. Dies eröffnet neue Möglichkeiten für Forschung und Entwicklung auf dem Gebiet der Angriffserkennung.

Die Integration neuer Methoden zur lokalen und verteilten Angriffserkennung wird durch das Framework bestmöglich unterstützt. Das Framework bietet zusätzlich ein breites Spektrum an fertigen Modulen und Hilfsfunktionalität, wie z. B. von der Laufzeitumgebung unabhängige Timer und Protokoll-Parser. Hiermit können einheitliche, von der Laufzeitumgebung unabhängige Module zur Angriffserkennung entwickelt und evaluiert werden. Dies ermöglicht die Konzentration auf die eigentliche Entwicklung neuer Methoden zur Angriffserkennung.

Durch die beispielhafte Integration einer Angriffserkennung aus [Game05] konnte gezeigt werden, dass das Framework zur einfachen Integration von Methoden zur Detektion von Angriffen geeignet ist. Die Möglichkeit eine bereits integrierte Angriffserkennung in einer anderen Laufzeitumgebung, wie z. B. einem Simulator, zu betreiben ermöglicht die Evaluation der Angriffserkennung in großen Netzen.

Im Rahmen der Arbeit durchgeführte Laufzeitmessungen zeigten, dass das Framework bezüglich der Datenrate des Netzverkehrs skaliert und das unterliegende System nicht beeinflusst. Interne Mechanismen sorgen außerdem für einen effektiven Schutz vor durch Überlast induzierten Angriffen und erhöhen so die Robustheit des Frameworks.

Die Portierung des Frameworks in weitere Laufzeitumgebungen eröffnet neue Möglichkeiten. Sehr interessant sind hier Router bzw. Netzwerkkarten mit speziellen Prozessoren - sogenannte *Netzwerkprozessoren*. Sie ermöglichen die Ausführung des Frameworks direkt auf der Netzwerkkarte unter Verwendung von hochspezialisierten Bausteinen, welche die Verarbeitung von Frames und Paketen beschleunigen. Unter anderem sind hier die Netzwerkprozessoren Toaster 2 von Cisco, PowerNP von IBM und IXP von Intel zu nennen.

Mit der Integration in die OMNeT++-Umgebung ist es möglich, Methoden zur lokalen und verteilten Angriffserkennung in großen Netzen zu evaluieren. Dabei ist es wichtig, realistische Internet-ähnliche Topologien sowie realistischen Hintergrundverkehr zu verwenden. Diese Thematik wird derzeit im Rahmen einer Diplomarbeit am Institut für Telematik der Universität Karlsruhe (TH) bearbeitet [Scha07a]. Erste Simulationen von verteilten DoS-Angriffen, welche mit den dort erstellten Topologien unter Verwendung des Distack-Frameworks durchgeführt wurden, waren sehr vielversprechend. Hier müssen weiterführende Simulationen betrieben werden um Angriffserkennung in großen Netzen zu evaluieren.

Als letzter Punkt für zukünftige Entwicklungen seien Methoden zur lokalen und verteilten Angriffserkennung sowie die Erweiterung des Frameworks durch neue Protokolle und Kommunikationsschichten zur verteilten Kommunikation zu nennen. Hier bietet das Distack-Framework ideale Möglichkeiten zur einfachen Integration. Im Rahmen einer weiteren Diplomarbeit [Brei07] am Institut für Telematik der Universität Karlsruhe (TH) werden beispielsweise Module für das Distack-Framework entwickelt, welche es ermöglichen, Angriffe auf Basis erkannter Anomalien zu identifizieren und zu beschreiben. Dies stellt eine wichtige Voraussetzung für die Kommunikation zwischen verteilten Instanzen sowie für die Einleitung reaktiver Maßnahmen dar.

Literatur

- [Abba94] Janet Ellen Abbate. *From ARPANET to Internet: A history of ARPA-sponsored computer networks, 1966-1988*. Dissertation, University of Pennsylvania, 1994.
- [Arbo07] Arbor. Worldwide Infrastructure Security Report. <http://www.arbornetworks.com/report>, 2007.
- [Arch00] Internet Traffic Archive. The Internet Traffic Archive. <http://ita.ee.lbl.gov>, 2000.
- [BeMo05] Michael H. Behringer und Monique J. Morrow. *MPLS VPN Security*. Cisco Press. 2005.
- [Brei07] Jannis Breitwieser. Anomalie-basierte Identifikation von Angriffen im Internet. Diplomarbeit, Institut für Telematik, Universität Karlsruhe (TH), 2007. To appear.
- [BrMü06] Lothar Braun und Gerhard Münz. Netzbasierte Angriffs- und Anomalieerkennung mit TOPAS. In *1. GI FG SIDAR Graduierten-Workshop über Reaktive Sicherheit (SPRING)*, 2006, S. 8.
- [CHKM⁺06] Buck Clay, Daniel Harrison, Avi Kelman, Suzanne Matthews und Eric Smith. Random Graph Generation, 2006.
- [Cisc02] Cisco-Systems. Evolution of the Firewall Industry, 2002.
- [Clea00] Stephen Cleary. Boost Pool Library, 2000.
- [Cons07] Diadem Firewall Consortium. Diadem Firewall Projekt. <http://www.diadem-firewall.org>, 2007.
- [CPBD⁺05] Clark, Partridge, Braden, Davie, Floyd, Jacobson, Katabi, Minshall, Ramakrishnan, Roscoe, Stoica, Wroclawski und Zhang. Making the World (of Communications) a Different Place. In *ACM SIGCOMM Computer Communication Review*, 2005, S. 91–96.
- [Degi00] Loris Degioanni. Development of an Architecture for Packet Capture and Network Traffic Analysis. Diplomarbeit, Politecnico di Torino, Facoltà di Ingegneria, 2000.
- [Exch07] Amsterdam Internet Exchange. AMS-IX Traffic Statistics. <https://www.ams-ix.net/technical/stats/>, 2007.

- [Falk04] Georg Carle Falko Dressler, Gerhard Münz. CATS: Cooperating Autonomous Detection Systems. In *International Workshop on Autonomic Communication*, 2004. Poster Session.
- [FHSZ02] Fuhrmann, Harbuam, Schöller und Zitterbart. AMnet 2.0: An Improved Architecture for Programmable Networks. In *Proceedings of the 4th International Working Conference on Active Networks*, 2002, S. 162–176.
- [Fink07] Dan Finkelstein. The WinNuke Relief Page. <http://www.users.nac.net/splat/winnuke/>, 2007.
- [Fran04] Mike Frantzen. ISIC - IP Stack Integrity Checker. <http://www.packetfactory.net/projects/ISIC/>, 2004.
- [FrWi01] D. Frincke und E. Wilhite. Distributed Network Defense. In *Proc. 2001 IEEE Workshop on Information Assurance and Security West Point, NY.*, 2001, S. 236–238.
- [FuLi06] V. Fuller und T. Li. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632 (Best Current Practice), August 2006. IETF.
- [Game05] Thomas Gamer. A System for in-Network Anomaly Detection. In *Kommunikation in Verteilten Systemen*. Springer, 2005.
- [GaSB06] Thomas Gamer, Marcus Schöller und Roland Bless. A Granularity-adaptive System for in-Network Attack Detection. In *Proceedings of the IEEE / IST Workshop on Monitoring, Attack Detection and Mitigation 2006*, Sep 2006, S. 47–50.
- [GaSS07] Thomas Gamer, Michael Scharf und Marcus Schöller. Collaborative Anomaly-based Attack Detection. In *IWSOS*. Springer, 2007, S. 280–287.
- [GoSp01] Rajeev Gopalakrishna und Eugene H. Spafford. A framework for distributed intrusion detection using interest driven cooperating agents. <http://www.cs.memphis.edu/classweb/-comp7990/fall2002/agents/slides/intrusion.ppt>, 2001.
- [Greg01] Douglas Gregor. Boost Signals Library, 2001.
- [Gru05] Radu Gruian. Patricia Trie Template Class. <http://www.codeproject.com/string/PatriciaTrieTemplateClass.asp>, 2005.
- [Gutt06] Zvi Gutterman. Open to Attack: Vulnerabilities of the Linux Random Number Generator. Blackhat USA, 2006. Presentation.
- [HKLB05] R. Hancock, G. Karagiannis, J. Loughney und S. Van den Bosch. Next Steps in Signaling (NSIS): Framework. RFC 4080 (Informational), Juni 2005. IETF.

- [HoHo02] R. Housley und S. Hollenbeck. EtherIP: Tunneling Ethernet Frames in IP Datagrams. RFC 3378 (Informational), September 2002. IETF.
- [InFo02] Kenneth Ingham und Stephanie Forrest. A History and Survey of Network Firewalls. Technischer Bericht, University of New Mexico, 2002.
- [Isem00] David Iseminger. *Microsoft Win32 Developer's Reference Library*. Microsoft Press Corp. 2000.
- [JaWZ03] R. Janakiraman, M. Waldvogel und Q. Zhang. Indra: A peer-to-peer approach to network intrusion detection and prevention. In *IEEE WETICE*, 2003, S. 226–231.
- [Katz97] D. Katz. IP Router Alert Option. RFC 2113 (Proposed Standard), Februar 1997. IETF.
- [Kohl03] Christopher Kohlhoff. Boost Asio Library, 2003.
- [KuRo04] James F. Kurose und Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Pearson Benjamin Cummings. 2004.
- [Lord05] Mark Lord. hdparm. <http://sourceforge.net/projects/hdparm>, 2005.
- [MaNi98] Makoto Matsumoto und Takuji Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation: Special Issue on Uniform Random Number Generation*, Band 8, 1998, S. 3–30.
- [Mars01] Carolyn Duffy Marsan. Experts call MPLS bad for 'Net. <http://www.networkworld.com/news/2001/0806mpls.html>, 2001.
- [Maur00] Jens Maurer. Boost Random Number Library, 2000.
- [Maye06] Christoph Mayer. Datenschutzkonforme Anonymisierung von Datenverkehr auf einem Vermittlungssystem. Studienarbeit, Institut für Telematik, Universität Karlsruhe (TH), 2006.
- [McJa93] Steven McCanne und Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter*, 1993, S. 259–270.
- [Micr07] Microsoft. MS02-045: Unchecked Buffer in Network Share Provider May Lead to Denial-of-Service. <http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q326830&>, 2007.
- [MoRA87] J. Mogul, R. Rashid und M. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, Band 21, 1987, S. 39–51.

- [Morr68] Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4), 1968, S. 514–534.
- [Ness07] Nessus. Nessus Vulnerability Scanner. <http://www.nessus.org>, 2007.
- [Netw07] Top Layer Networks. SPAN Expander. <http://www.toplayer.com>, 2007.
- [PaMi88] Park und Miller. Random Number Generators: Good ones are Hard to Find. *Communications of the ACM*, Band 31, 1988, S. 1192–1201.
- [PLMH⁺03] Christos Papadopoulos, Robert Lindell, John Mehringer, Alefiya Husain und Ramesh Govindan. COSSACK: Coordinated Suppression of Simultaneous Attacks. In *DARPA Information Survivability Conference and Exposition*, 2003, S. 94–96.
- [PoNe97] Phillip Porras und Peter Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *National Information Systems Security Conference*, Oct 1997, S. 353–365.
- [Post81a] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. IETF.
- [Post81b] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. IETF.
- [Post06] Washington Post. The Botnet Trackers, February 2006.
- [Rame02] Robert Ramey. Boost Serialization Library, 2002.
- [Rey06a] Enno Rey. MPLS and VPLS Security. Blackhat Europe, Presentation, 2006.
- [Rey06b] Enno Rey. MPLS-Technik und Angriffsszenarien. *LANline*, Band 5, 2006, S. 52–55.
- [RFC-06] RFC-Editor. Number of RFC documents per year, 2006.
- [Roes01] Martin Roesch. Snort. <http://www.snort.org>, 2001.
- [RoVC01] E. Rosen, A. Viswanathan und R. Callon. Multiprotocol Label Switching Architecture. RFC 3031 (Proposed Standard), Januar 2001. IETF.
- [Samh07] Samhain. Samhain File Integrity. <http://la-samhna.de/samhain/>, 2007.
- [Scha07a] Michael Scharf. Effektivität einer koordinierten Angriffserkennung in simulierten IP-Netzen. Diplomarbeit, Institut für Telematik, Universität Karlsruhe (TH), 2007. To appear.
- [ScHa07b] Schulzrinne und Hancock. GIST: General Internet Signaling Transport. Internet-Draft, IETF, 2007. Work in Progress.

- [Secu07a] Securityfocus. Microsoft Network Share Provider SMB Request Buffer Overflow Vulnerability. <http://www.securityfocus.com/bid/5556/>, 2007.
- [Secu07b] Securityfocus. Snort GRE Packet Decoding Integer Underflow Vulnerability. <http://www.securityfocus.com/bid/22004/>, 2007.
- [Secu07c] Securityfocus. Snort ICMP Denial of Service Vulnerability. <http://www.securityfocus.com/bid/3849/>, 2007.
- [SHSD⁺01] D. Schnackengerg, H. Holliday, R. Smith, Kelly Djahandari und Dan Sterne. Cooperative Intrusion Traceback and Response Architecture (CITRA). In *DISCEX*, Band 1, 2001, S. 56–68.
- [Simp95] W. Simpson. IP in IP Tunneling. RFC 1853 (Informational), Oktober 1995. IETF.
- [Somm03] Robin Sommer. Bro: An Open Source Network Intrusion Detection System. In *Proceedings of the 17. DFN-Arbeitstagung über Kommunikationsnetze*, 2003, S. 273–288.
- [Sour07] Sourcefire. Sourcefire Management Console. <http://www.sourcefire.com>, 2007.
- [Spaf04] Eugene Spafford. The Internet Worm Program: An Analysis. Technischer Bericht, Purdue University, 2004.
- [Tcpd00] Tcpdump. Tcpdump Pcap. <http://www.tcpdump.org>, 2000.
- [Tech07] Applied Watch Technologies. Applied Watch Console. <http://www.appliedwatch.com>, 2007.
- [ThMW97] Thompson, Miller und Wilder. Wide-area Internet traffic patterns and characteristics. *Network, IEEE*, Band 11, 1997, S. 10–23.
- [Thom03] Rob Thomas. ISP Security BOF III. In *NANOG*, 2003.
- [Trip07] Tripwire. Tripwire Software. <http://www.tripwire.com>, 2007.
- [Turn00] Aaron Turner. Tcpreplay, 2000.
- [Turn02] Aaron Turner. Tcprewrite, 2002.
- [UnPN02] Jeffrey Undercoffer, Filip Perich und Charles Nicholas. SHOMAR: An Open Architecture for Distributed Intrusion Detection Services. Technischer Bericht, University of Maryland, 2002.
- [Varg01] Andras Varga. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference*, 2001, S. 319–324.
- [Varg07] Varga. INET Framework for OMNeT++/OMNEST. <http://www.omnetpp.org/doc/INET>, 2007.

- [ViVK03] Giovanni Vigna, Fredrik Valeur und Richard A. Kemmerer. Designing and implementing a family of intrusion detection systems. In *9th European software engineering conference*, 2003, S. 88–97.
- [WiKe86] Thomas Williams und Colin Kelley. Gnuplot. <http://www.gnuplot.info>, 1986.
- [Zara03] Krzysztof Zaraska. Prelude IDS: current state and development perspectives. <http://www.prelude-ids.org>, 2003.
- [ZMDN⁺07] Zseby, Molina, Duffield, Niccolini und Raspall. Sampling and Filtering Techniques for IP Packet Selection. Internet-Draft, IETF, 2007. Work in Progress.