Hochschule für Technik, Wirtschaft und Kultur Leipzig
Fakultät Informatik, Mathematik und Naturwissenschaften

Bachelorarbeit

Anomalieerkennung und Vorhersage kritischer System-Zustände anhand von Nagios-Daten

Vorgelegt von
Daniel Borkmann
Leipzig, Oktober 2009

Studiengang:	Technische Informatik (B. Sc.)
Betreuender Professor:	Prof. Dr.-Ing. Dietmar Reimann
Betreut durch das Max-Planck-Institut für Kognitions- und Neurowissenschaften Leipzig:	Dr. Helmut Hayd

Diese Arbeit befasst sich mit der nichtlinearen Extrapolation von Messdaten sowie Methoden der Erkennung von Verhaltensanomalien einer Messreihe. Der Schwerpunkt der hierbei exemplarisch betrachteten Messdaten bezieht sich auf Netzwerkpaketstatistiken des LANs vom Max-Planck-Institut für Kognitions- und Neurowissenschaften Leipzig. Auf Basis des Nagios-Frameworks erfolgt eine Integration dieser Statistiken zur weiteren Verarbeitung für die Analysen. Nach einer Einführung in die Grundlagen werden zunächst Annahmen über die in Nagios integrierten Prozesse getroffen und anschließend auf zwei der möglichen Verfahren für eine Anomalieanalyse eingegangen. Im Anschluss erfolgt die Vorstellung der Methode zur Messdatenvorhersage und von möglichen anzuwendenden Modellen. Ebenfalls eingegangen wird auf die Arbeitsweise und Integration der im Rahmen dieser Arbeit entwickelten Software zur effizienten Ermittlung von Netzwerkpaketstatistiken sowie der in dieser Arbeit entwickelten Prototypen für Extrapolation und Anomalieerkennung.

Inhaltsverzeichnis

Abbildungsverzeichnis **I**

Tabellenverzeichnis **II**

1 Einleitung **1**

1.1 Motivation 1

1.2 Zielsetzung 3

2 Grundlagen **5**

2.1 Nagios 5

2.1.1 Einführung 5

2.1.2 Plugin API 8

2.1.3 Check-Performance-Daten 9

2.1.4 Arten von Service-Checks 10

2.2 Round Robin Datenbank 15

2.3 NagiosGrapher 17

2.4 Linux Kernel Paketanalyse 19

2.4.1 Allgemeines 19

2.4.2 PF_PACKET 19

2.4.3 Promiscuous Mode 21

2.4.4 Berkeley Packet Filter 22

2.4.5 RX_RING 26

3 Algorithmische Realisierung **28**

3.1 Zeitreihen und Stochastische Prozesse 28

3.2 Erkennung von Anomalien 35

3.2.1 Exponentielle Glättung 36

3.2.2 Clusteranalyse zur Intervallschätzung 45

3.3 Extrapolation von Daten 49

3.3.1 Nichtlineare Ausgleichsrechnung 49

3.3.2 Mögliche Approximationsfunktionen 54

4 Implementierung **62**

4.1 Systemvoraussetzungen und eingesetzte Werkzeuge 62

4.1.1 Router mit netsniff-ng 62

4.1.2 Nagios-Server 63

4.2 Funktionsbibliotheken 63

4.2.1 FTX 63
4.2.2 Baseline::Algorithm::CalcRRD 67
4.2.3 Baseline::Util::Routines 68
4.3 Nagios-Plugin „netsniff-ng“ 69
4.3.1 netsniff-ng 69
4.3.2 check_packets 76
4.3.3 BPF Routinen 76
4.4 Vorhersage-Framework „rea“ 78
4.4.1 generate.pl 79
4.4.2 predict.cgi 79
4.5 Anomalie-Modulübersicht sowie Schnittstelle zu NagiosGrapher . . . 80
4.6 Integration in die Nagios-Weboberfläche 81

5 Schluss **86**
5.1 Zusammenfassung und Diskussion 86
5.2 Danksagung 89

A Inhalt der CD **III**

B Resultate aus Vorhersage und Anomalieanalyse **IV**

C Paketfilter für den Zimbra-Mailserver **V**

Literatur **VI**

Eidesstattliche Erklärung **VII**

Abbildungsverzeichnis

1 Netzwerkraum der EDV, in dem der Router zum Internet und einer der beiden Core-Switche steht . . . 2
2 Nagios Weboberfläche mit zu überwachenden Diensten . . . 6
3 Syntax der Plugin-Performance-Daten . . . 9
4 Beispiel einer Pluginausgabe . . . 9
5 Verschiedene Arten der Durchführung von Service-Checks . . . 12
6 Daten-Fluss innerhalb einer RRD . . . 15
7 Aufbau einer RRD . . . 16
8 NagiosGrapher Weboberfläche . . . 18
9 Übersicht Linux Netzwerk-Stack . . . 20
10 Linux Packet Filter mit BPF Maschinencode . . . 23
11 Linux Packet Filter aus Kernelsicht . . . 23
12 Beispiel-Filter für Pakete mit der Bedingung „ip or arp“ . . . 26
13 RX_RING Aufbau . . . 27
14 Netzwerktraffic am Institutsrouter, Ein- und Zwei-Tagesansicht . . . 30
15 Dekomposition der Zeitserie vom HTTP-Datendurchsatz am MPI-Router . . . 31
16 Zwei unterschiedliche, unabhängige Realisationen des Simple Random Walk . . . 33
17 Holt-Winters Forecasting vom HTTP-Datendurchsatz am MPI-Router 40
18 Kumulative Poisson-Verteilungsfunktion vom HTTP-Datendurchsatz am MPI-Router . . . 43
19 Holt-Winters Forecasting mit Kalman-Filter . . . 45
20 Prinzip Clusteranalyse zur Intervallschätzung . . . 45
21 Extrapolation vom Datendurchsatz am MPI-Router . . . 50
22 Vergleich zweier Strings mittels strcmp(), Quelle: Linux Kernel (lib/string.c) 65
23 Zusammenspiel der FTX-Komponenten . . . 65
24 Gespeicherte Informationen innerhalb einer Node . . . 65
25 Systemarchitektur von netsniff-ng . . . 71
26 strace Auszug von „tcpdump -i eth0 -n arp“ . . . 72
27 strace Auszug von „netsniff-ng -d eth0 -f /etc/netsniff-ng/rules/arp.bpf -C“ . . . 73
28 Performancetest von netsniff-ng mittels Ping-Flooding über Gigabit-Ethernet . . . 74
29 Netzplan der DMZ mit Produktiveinsatz von netsniff-ng . . . 75
30 Interaktion von netsniff-ng mit Nagios . . . 77

31 Aufbau und Funktionsweise vom Vorhersage-Framework 78
32 Modulübersicht und Interaktion mit NagiosGrapher 80
33 Nagios Webinterface - Einstiegspunkt 82
34 Nagios „Service Detail“-Seite mit Anomalieanzeige 82
35 NagiosGrapher-Seite mit *health*-Anzeige 83
36 Prototypische Weboberfläche vom Vorhersage-Framework 84
37 Prototypische Weboberfläche vom Vorhersage-Framework, Konfiguration 85

Tabellenverzeichnis

1 Mögliche Nagios-Plugin Rückgabewerte 8
2 Eigenschaften der einzelnen Round Robin Archive, die durch Nagios-Grapher standardmäßig erstellt wurden 17
3 BPF Instruction Set 24
4 BPF Addressing Modes 25
5 BPF Operationscode-Format 25
6 Benchmark einiger XML-Parser aus Perl (zum Teil mit *C*-Bindings), sowie der eigenen FTX-Lib 66

1 Einleitung

1.1 Motivation

In der heutigen Zeit wächst die Zahl der Server in Unternehmen oder Forschungsinstituten ständig an, sodass der Einsatz von Monitoringsystemen ein notwendiges Instrument ist, um den laufenden Betrieb vor Ausfällen zu bewahren. Ein solches System warnt beispielsweise die IT-Administratoren bei Ausfällen von Rechnern, vollen Festplatten, kritischen Akkuzuständen der zentralen USV, Überschreitung von Temperaturgrenzen in Serverräumen oder vielen weiteren wichtigen Prozessen. Dadurch kann bereits im Vorfeld angemessen agiert werden und den Benutzern bleibt ein Anruf bei der IT-Hotline erspart.
Nagios ist beispielsweise ein solches Überwachungssystem, das auch am Max-Planck-Institut für Kognitions- und Neurowissenschaften Leipzig (im Weiteren auch als MPI bezeichnet) seit geraumer Zeit im Einsatz ist und wegen seiner Flexibilität und Erweiterbarkeit sehr geschätzt wird.
Zwar können für die genannten Prozesse kritische Schwellwerte festgelegt werden, sodass bei Überschreiten eine Alarmierung erfolgt, jedoch sind diese innerhalb *Nagios* lediglich statischer Natur.
Oft sind allerdings in der Realität fest definierte Schwellwerte unangemessen, da diese Herangehensweise zu primitiv ist und somit gewisse Ereignisse übersehen werden können. Prozesse oder Systeme verhalten sich im Allgemeinen dynamisch und ändern ihr Verhalten im Laufe der Zeit. Über statische Zustandsgrenzen kann jedoch kein „Normalverhalten“ von nicht-normalen Verhalten (im Weiteren auch Anomalie) unterschieden werden. Als einfaches Beispiel sei daher der Datendurchsatz zum Internet genannt. Eine statische Grenze zum „kritischen Zustand“ kann hier schlecht beschrieben werden, da zu beliebigen Zeitpunkten die Bandbreite (am MPI: $100MBit/s$) voll ausgenutzt werden kann. Ist man jedoch in der Lage, das Verhalten systematisch zu analysieren, so erkennt man beispielsweise bereits innerhalb eines Tages eine unterschiedliche Ausnutzung. Weiterhin fallen Unterschiede innerhalb der Wochentage auf und über einen längeren Zeitraum ist mit großer Wahrscheinlichkeit ein ansteigender Trend bei den Durchsatzraten zu erkennen. All diese Informationen bleiben bei den statisch festgelegten Zustandsgrenzen verborgen, sodass beispielsweise bei Netzwerkeinbrüchen eine Verhaltensanomalie oft unentdeckt bleibt. Eröffnet eine unautorisierte Person über Nacht plötzlich eine Tauschbörse mit illegaler Software, sodass Netzwerkdurchsatzraten ansteigen, würde dies eine Anomalie im Normalverhalten darstellen und kann über eine dynamische Herangehensweise erkannt werden.

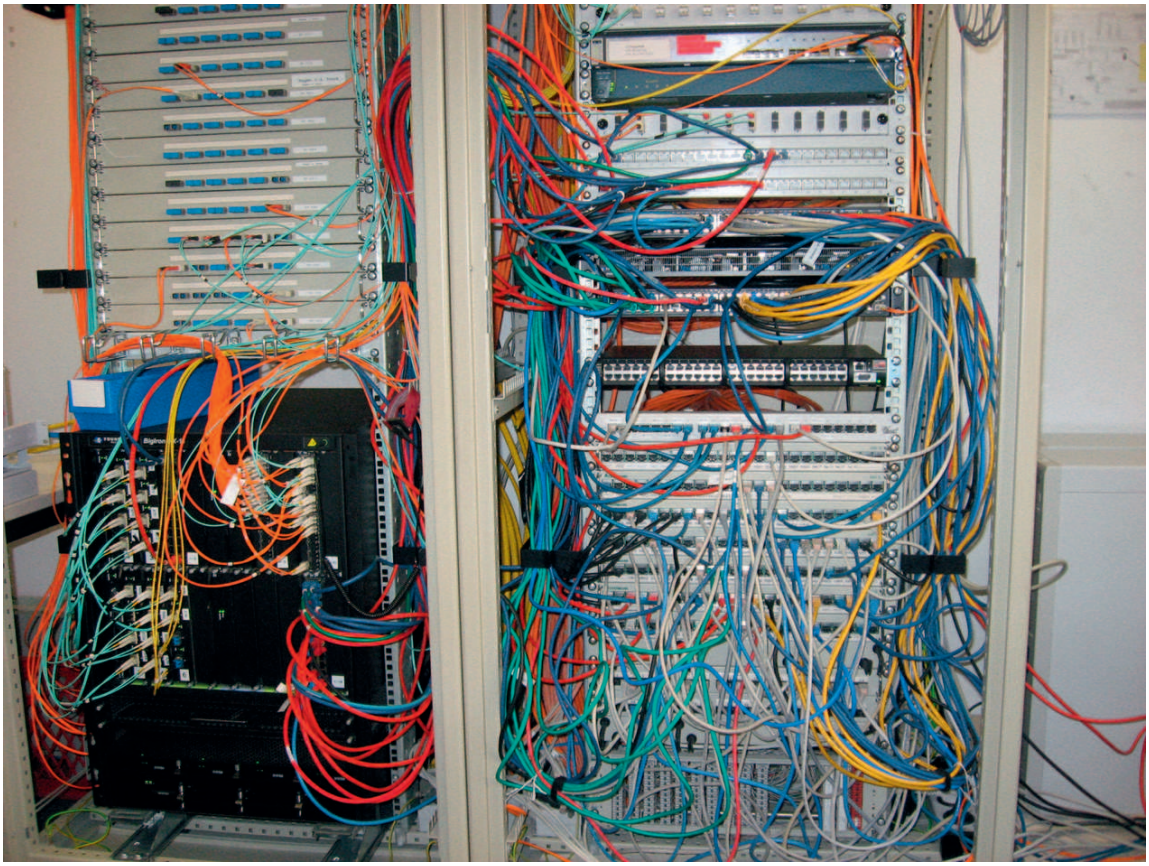


Abbildung 1: Netzwerkraum der EDV, in dem der Router zum Internet und einer der beiden Core-Switche steht. *Mit den über 80 Servern und einer ebenso nicht geringen Zahl an Netzwerkkomponenten wird der Einsatz eines Monitoringsystems fast schon zur Pflicht, um die IT-Landschaft am Leben zu erhalten. Ideal wäre es jedoch, Fehler oder Anzeichen für Systemausfälle in einem Frühstadium zu erkennen und zu beheben, bevor der Benutzer dies überhaupt bemerkt. Das Erkennen solcher Zustände bedarf allerdings mehr als nur statischer Zustandsgrenzen, da sich das Systemverhalten im Laufe der Zeit ändert.*

„Prognosen sind schwierig, besonders wenn sie die Zukunft betreffen.“
- Mark Twain

Ein weiteres, wichtiges Thema ist es, Informationen über ein mögliches Verhalten der zukünftigen Messdaten zu erhalten. Wie bereits das etwas spöttische Bonmot von Mark Twain aussagt, ist es jedoch keineswegs eine triviale Angelegenheit, etwas vorhersagen zu wollen.
Anhand mathematischer Methoden kann bei geeigneter Modellwahl durchaus ein brauchbarer Trend herausgelesen werden, der gerade für die Planung der IT-Infrastruktur oder bei Neuanschaffungen von Softwarelizenzen von großer Bedeutung wäre.
Verschiedene Anwendungen sind hierbei denkbar.
So könnten Prognosen erstellt werden, zu welchen Zeitpunkt das vorhandene Datenvolumen vom Internetzugang des Gästenetzes aufgebraucht ist. Da eine Überschreitung der erlaubten Datenmenge Nachzahlungen in nicht unerheblicher Höhe zur Folge haben, kann entsprechend vorher reagiert werden, indem beispielsweise die Bandbreite gedrosselt wird. Ein anderes Szenario betrifft den Erwerb von Matlab-Lizenzen. So könnte rechtzeitig geplant werden, wann neue Lizenzen im Voraus erworben werden müssen. Andererseits kann auch eine Aussage darüber getroffen werden, ob es zu gegebenem Zeitpunkt überhaupt notwendig ist, Lizenzen anzuschaffen.
Eine solche Möglichkeit der Vorhersage existiert allerdings nicht in Nagios, sodass oben genannte Szenarien aktuell nicht umsetzbar sind.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es daher, zum einen die Entwicklung eines Software-Plugins für Nagios, welches Statistiken über Netzwerkpaketdaten erfasst, die im Weiteren graphisch visualisiert werden können und zum anderen eine Erweiterung, die diese Daten anhand diverser Algorithmen auf Anomalien hin untersucht sowie mittels verschiedener Modelle Vorhersagen erzeugt.

Die Netzwerkpaketdaten sollen dabei vom internen Router, der die Pakete über das Netz der Universität Leipzig ins Internet leitet, abgegriffen und Statistiken für Nagios erzeugt werden.

Zur Anomalieanalyse sollen zwei verschiedene Ansätze getestet werden. Dies ist zum einen das *Holt-Winters-Forecasting*, welches auf Basis der *exponentiellen Glättung* aufsetzt, und zum anderen eine *Clusteranalyse*, die Messdaten gruppiert, um damit mögliche Intervalle für ein Normalverhalten abzuschätzen.

Für die Vorhersage soll sich der *nichtlinearen Ausgleichsrechnung* bedient und anhand des robusten, in der Praxis am häufigsten verwendeten Verfahrens namens *Levenberg-Marquardt* verschiedene mathematische Modelle evaluiert werden.

Für die Algorithmen der Anomalieanalyse und Vorhersage sei hervorzuheben, dass diese generischer Art sein sollen und somit die Umgebung des Max-Planck-Instituts lediglich als *Beispiel* eines LANs und die Analyse der Netzwerkpaketdaten lediglich als *Beispiel* für numerische Werte anzusehen sind.

Beide Erweiterungen sowie das Plugin sollen in die Nagios-Umgebung integriert werden.

2 Grundlagen

In diesem Abschnitt werden grundlegende Werkzeuge und Arbeitsumgebungen vorgestellt, in welche die Anomalienanalyse und Datenvorhersage integriert werden sollen. Zunächst wird dabei auf das *Nagios-Framework* eingegangen, welches als Basis zur Beobachtung der verschiedenen Prozesse dient. Im Anschluss folgt eine spezielle Nagios-Erweiterung zur Datenvisualisierung namens *NagiosGrapher*, das auf den im gleichen Kapitel vorgestellten *Round Robin Datenbanken* basiert.
Weitere Grundlagen für ein in dieser Arbeit entwickeltes Nagios-Plugin zum Sammeln von Daten, welches auf dem MPI-internen Router läuft, der zur Kommunikation nach außen dient und Netzwerkpakete analysiert, werden nachfolgend ebenfalls beschrieben.

2.1 Nagios

2.1.1 Einführung

Nagios ist ein Open Source Monitoring System für Linux und Unix, welches von Ethan Galstad in der Programmiersprache C entwickelt wurde. Es dient zur Überwachung von Hosts und deren Diensten (im Folgenden auch als Service bezeichnet). Das Projekt Nagios (ehemals NetSaint) existiert seit 1999. Der Name selbst setzt sich aus den Begriffen „network“ und „hagios“ (griech. hagios = heilig) zusammen [Schr 04].
Mit Hilfe von Nagios ist es möglich, Prozesse, die in irgendeiner Art und Weise numerische oder boolesche Werte (Beispiel: Ping funktioniert oder funktioniert nicht) produzieren, zentral zu überwachen. Überwacht werden neben Servern oder Workstations auch Netzwerk-Switche, Temperatur- und Feuchtigkeitssensoren, Drucker oder weitere, mit einer Netzwerkschnittstelle vorhandene Hardware. Diese müssen für Nagios definiert und zugehörige Services festgelegt werden. Nagios übernimmt die Überwachung anschließend von selbst und alarmiert den Systemadministrator bei Überschreitung kritischer Zustandsgrenzen. Zu diesem Zweck wurde ein leistungsfähiges und flexibles Benachrichtigungssystem in Nagios implementiert. Eine Alarmierung kann neben E-Mail-Benachrichtigung auch über Instant Messenger, SMS oder ähnlichem erfolgen. Daten werden in Nagios persistent gehalten, sodass auch auf frühere Systemzustände zurückgegriffen werden kann [Bart 08b].
Im Wesentlichen kann man folgende Vorteile hervorheben: Im Fehlerfall muss der Administrator nicht erst nach der Quelle suchen, denn diese wird bereits von Nagios angezeigt. Auf kritische Zustände kann bereits im Frühstadium durch eine Warnung an den Administrator aufmerksam gemacht werden. Da die Überwachung außerdem unbeaufsichtigt erfolgt, kann sich so das Administratoren-Team wichtigeren Tätigkeiten

widmen. Alle wichtigen Informationen über Hosts und deren Dienste-Überprüfung (im Weiteren auch als Service-Checks bezeichnet) liegen zudem, wie in Abbildung 2 [S. 6], zu sehen an zentraler Stelle. Weiterhin skaliert Nagios gut und Dank einer großen öffentlichen Community sind bereits viele Plugins vorhanden. Da Nagios ein offenes Framework ist, können auch selbst Tests leicht implementiert werden.

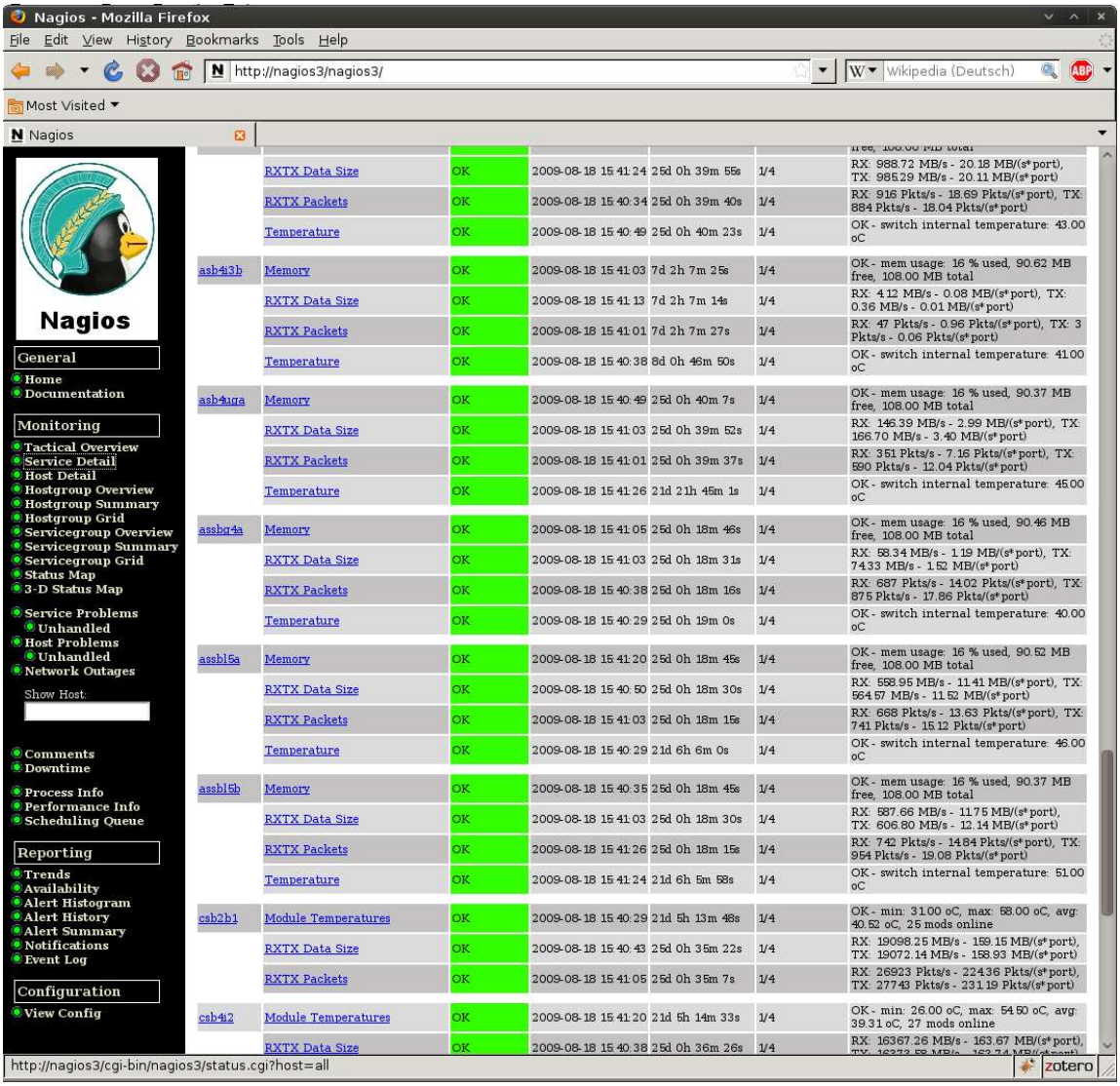


Abbildung 2: Nagios Weboberfläche mit zu überwachenden Diensten: *Zu sehen ist der im Rahmen dieser Arbeit aufgesetze institutsinterne Nagios-Test-Server, welcher unter anderem wichtige Betriebsdaten aller sich im Institut befindenden Netzwerkswitche über SNMP abfragt. Die Daten beziehen sich beispielsweise auf die interne Switch-Temperatur, den freien Arbeitsspeicher oder aber TX- und RX-Transferraten. Werden definierte Schwellwerte überschritten, so erscheint zunächst eine Warnung und im Anschluss der Alarmzustand. Administratoren können per E-Mail, Jabber-Nachricht oder sogar SMS benachrichtigt werden.*

Nagios selbst ist ein *Framework*, dessen Funktionalität durch *Plugins* definiert wird. Das Framework selbst enthält keine Plugins. Diese werden in weiteren Softwarepaketen mitgeliefert oder für spezielle Bedürfnisse selbst implementiert.
Die Hauptobjekt-Typen des *Frameworks* setzen sich wie folgt zusammen [Kocj 08]:

Kommandos definieren, wie Nagios die Service-Checks durchführen soll. Diese stellen eine Abstraktionsschicht oberhalb der eigentlichen Plugins dar und ermöglichen ähnliche Plugin-Typen zu gruppieren. Ein Kommando gibt hauptsächlich den Pfad zum ausführbaren Plugin an.

Zeitperioden sind wiederkehrende Zeitfenster, in denen Operationen ausgeführt oder unterbunden werden sollen. Zeitfenster können zum Beispiel Werktage oder Feiertage sein.

Kontakte und Kontaktgruppen sind Personen, die bei Überschreitung definierter Schwellwerte über das Benachrichtigungssystem alarmiert werden. Kontakte können mit Zeitperioden gekoppelt sein, sodass Mitarbeiter während ihrer Urlaubszeit keine Alarmmeldungen erhalten. Mehrere Kontakte mit gleichen Eigenschaften können auch zu Kontaktgruppen zusammengefasst werden.

Hosts stellen physische Maschinen dar, die mit Kontakten und Services gekoppelt sind. IP-Adresse und Hostname gehören ebenfalls zu dessen Daten. Anhand des Templatesystems können Hosts auch in verschiedene Abstraktionsschichten untergliedert werden.

Services abstrahieren Kommandos und sind für einen speziellen Host definiert (1:n-Beziehung). Services stehen mit Kontakten und Zeitperioden eng in Verbindung. Services können auch zu Service-Gruppen zusammengefasst werden.

Host- und Service-Eskalation definieren Zeitintervalle, für die nach deren Ablauf weitere Personen (EDV-Leiter, Team-Vertretungen, ...) zur Problemlösung kontaktiert werden sollen.

Templates dienen zur Verallgemeinerung beziehungsweise zur sukzessiven Abstraktion von ähnlichen Objekten. Templates sind in diesem Sinne keine Objekttypen. Da jedoch Nagios ein sehr ausgeprägtes Template-System besitzt und Objekte von diesen erben, seien sie hier mit erwähnt.

Die zentrale Instanz, der Nagios-Daemon, durchläuft im Betrieb die nachfolgend beschriebenen Aktionen [Bart 08b]:

Zunächst prüft der Daemon in regelmäßigen Zeitabständen in mehreren Schritten die Zustände des Systems. Ein einfacher Ping kontrolliert, ob der Host überhaupt erreichbar ist. Bei Erfolg wird anschließend mit den zugehörigen Services fortgefahren. Die Plugins werden nacheinander ausgeführt und die Ergebnisse der lokalen oder extern gelieferten Daten verarbeitet. Die benötigten Gesamtzustandsinformationen für die Ansicht im Webfrontend werden nach Abarbeitung zusammengetragen. Bei anormalen Zuständen versendet der Daemon eine Nachricht an die definierten Kontakte.

2.1.2 Plugin API

Die Nagios-Plugin-API, welche im Kapitel 4.3 [S. 69] verwendet wird, hat wenige, sehr einfache Konventionen, die man bei der Entwicklung beachten muss. Ein Plugin ist lediglich ein ausführbares Programm und kann in beliebiger, unter Unix zur Verfügung stehender Programmier- oder Skriptsprache geschrieben sein [Bart 08b]. Häufig anzutreffen sind Sprachen wie C, Perl, Python oder Shell-Skripte.
Um mit Nagios interagieren zu können, sind die folgenden Rückgabe-Werte bei Programmende erlaubt [Team 09]:

Exit-Wert	**Bedeutung**
0	OK
1	Warning
2	Critical
3	Unknown

Tabelle 1: Mögliche Nagios-Plugin Rückgabewerte

Diese werden während des Programmablaufs je nach Prozesszustand vom Plugin ermittelt und entsprechend bei Programmende gesetzt.
Des Weiteren darf lediglich nur eine Zeile Text nach *STDOUT* geschrieben werden (Letztes Zeichen: \n). Diese erscheint, wie in Abbildung 2 [S. 6] zu sehen, in der rechten Spalte. Neben dem Text dürfen optional auch sogenannte Plugin-Performance-Daten [Gals 04] stehen, die vom normalen Text mittels eines *Pipe*-Symbols separiert werden (siehe Abbildung 4 [S. 9]). Plugin-Performance-Daten geben zusätzliche Informationen über die Schwellwerte des Services an. Diese können wiederum von Addons, welche die eintreffenden Daten graphisch als Zeitreihe darstellen, per regulärem Ausdruck ausgewertet werden. Folgendes, in Abbildung 3 [S. 9] dargestelltes Format wurde dafür definiert.

```
'label'=[value];[warn];[crit];[min];[max]
```

Abbildung 3: Syntax der Plugin-Performance-Daten

Nachfolgend sei ein Beispiel einer vollständigen Plugin-Ausgabe anhand Abbildung 4 [S. 9] dargestellt.

```
OK - UPS temperature: 21.30 oC | temp=21.30;23.00;27.00;5.00; \n
```

Abbildung 4: Beispiel einer Pluginausgabe: *Als „guten Stil“ wird die Ausgabe zuerst mit dem Status, gefolgt von einem Bindestrich mit anschließender kurzen Service-Beschreibung und nachfolgenden numerischen Messwerten formatiert. Optional können Plugin-Performance-Daten folgen, die den eigentlichen Wert, die Schwellwerte für Zustand „Warning“ und „Critical“, sowie minimal und maximal möglichen Messwert beinhalten. Als Abschluss folgt der Newline-Delimiter.*

2.1.3 Check-Performance-Daten

Da sich Nagios das Ziel gesetzt hat, dem Administrator eine Informationsflut zu ersparen, werden die wenigen Zustände (OK, Warning, Critical, Unknown) farbig, ähnlich wie bei einer „Verkehrsampel“ (Abbildung 2 [S. 6]), dargestellt. Allerdings können über diesen Ansatz keine Messwerte, die über einen längerfristigen Zeitraum aufgenommen wurden, visuell dargestellt werden. Es wird lediglich immer nur der „Jetzt-Zustand“ angezeigt.
Nagios löst dieses Problem insofern, dass es für die Messwerte eine Schnittstelle zur Weiterverarbeitung liefert [Bart 08b]. Mit Hilfe von diversen Nagios-Addons, wie beispielsweise dem NagiosGrapher, können diese Messwerte als Graph über der Zeit dargestellt werden. Im Nagios-Jargon werden diese nach außen geleiteten Daten Check-Performance-Daten genannt und sind nicht zu verwechseln mit den Plugin-Performance-Daten aus der Plugin-API, welche optional nach dem Pipe-Symbol auftreten können [Gals 04].

Check-Performance-Daten können über zwei verschiedene Methoden durch externe Programme weiterverarbeitet werden. Zum einen kann die Kommunikation über Files stattfinden und zum anderen über Unix-Pipes [Gals 09]. Zunächst jedoch treffen die STDOUT-Daten der Plugins (Abbildung 4 [S. 9]) im Nagios-Framework ein. Im Webfrontend wird lediglich der Ausgabetext vor dem Pipe-Symbol angezeigt.
Wurde entsprechend in Nagios die Benutzung von Check-Performance-Daten aktiviert, so wird der komplette STDOUT-String auf ein sogenanntes *Template* ab-

gebildet. Ein solches Template definiert sich für gewöhnlich aus folgendem *Daten-Tupel* [Gals 09]:

1. [SERVICEPERFDATA] (String)
2. Unix-Timestamp
3. Hostname
4. Service-Beschreibung
5. Service-Ausführungszeit
6. Service-Latenzzeit
7. Service-Ausgabe (vor Pipe-Symbol)
8. Service-Performance-Daten (nach Pipe-Symbol, auch: Plugin-Performance-Daten)

Dieses 8-Tupel wird als ganzer String (Elemente werden separiert durch Tabulatoren) in das *File-* beziehungsweise *Pipe-Interface* geschrieben. Wird ein File zum Datentransfer verwendet, so werden diese Strings im Append-Modus angefügt. Um das Dateisystem vor einem Überlauf zu bewahren, wird diese Datei nach jedem Auslesen der Daten gelöscht. Diese Interface-Methode ist prinzipiell einfacher zu verwenden, hat jedoch den Nachteil, dass Daten auf Festplatte geschrieben werden müssen[1]. Ist die externe Software einmal nicht angebunden, so werden unabhängig von ihr die Daten weiterhin gesammelt, sodass diese auch zu einem späteren Zeitpunkt ausgelesen werden können. Im Gegensatz dazu existiert die *Pipe* als Schnittstelle. Da der Transfer über ein *Special File* abgewickelt wird, müssen Daten nicht auf Festplatte geschrieben werden. Dies resultiert in einer besseren Performance und in geringerem Speicherbedarf. Zu Datenverlusten kann es jedoch dann kommen, wenn die externe Software nicht mit Nagios verbunden ist. Nagios versucht über einen gewissen Zeitraum Daten in die Pipe zu schreiben. Werden diese jedoch nicht ausgelesen, so bricht Nagios nach Ablauf einer Zeitüberschreitung ab und schließt die Pipe.

2.1.4 Arten von Service-Checks

Die Mehrheit der Nagios-Plugins ist dafür ausgelegt, Daten lokal aus der Maschine zu extrahieren. Dies bedeutet wiederum, dass im Allgemeinen Plugins auch auf den zu überwachenden Computern lokal installiert werden müssen. Da es kein allgemeingültiges Netzwerk-Protokoll gibt, welches beispielsweise die CPU-Load eines

[1] Dies kann allerdings auch bei Wahl eines entsprechenden Filesystems umgangen werden (Beispiel: *ramfs* oder *tmpfs*).

Rechners auslesen kann, bedarf es daher sicherer Verfahren, um Daten von den zu überwachenden Maschinen zum Nagios-Server transportieren zu können.
Zunächst soll die Abbildung 5 [S. 12] einen allgemeinen Überblick über die Möglichkeiten geben, wie Service-Checks ausgeführt werden können. Diese fünf Varianten werden in „aktive“ (Client 1-4) und „passive“ (Client 5) Checks klassifiziert [Bart 08b]. Bei aktiven Checks triggert Nagios die Pluginabfrage, wobei im Falle der passiven Checks ein sogenannter *Nagios Service Check Acceptor* (NSCA) auf Daten vom entfernten Client wartet und Nagios über Ereignisse informiert. Dabei unternimmt Nagios selbst keine Aktion, sondern wartet lediglich auf den NSCA-Daemon [Bart 08b]. Die aktiven Checks können wiederum untergliedert werden in „direkte“ (Client 1, Client 4) und „indirekte“ (Client 2-3) Service-Checks. Bei den indirekten Checks existiert zur Pluginausführung und anschließender Datenübertragung noch eine Mittelschicht an Software.

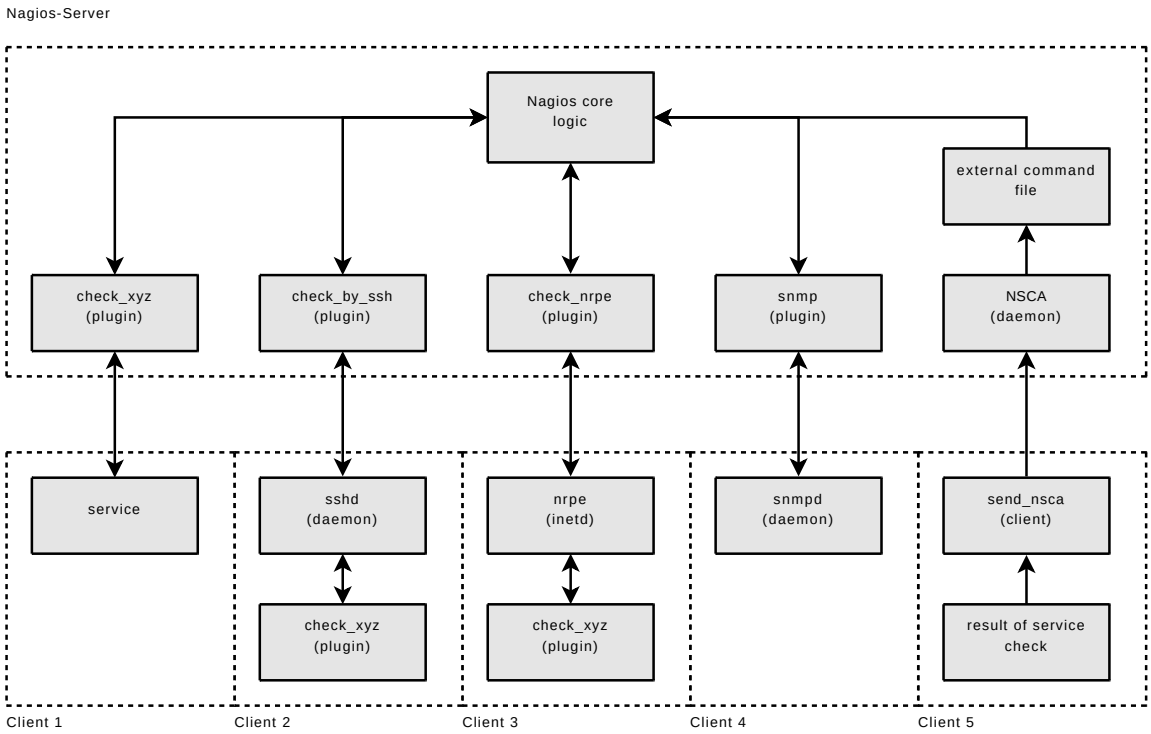


Abbildung 5: Verschiedene Arten der Durchführung von Service-Checks [Bart 08b]: *Um einen Netzwerk-Dienst auf dem ersten Client zu beobachten, startet der Nagios-Server sein eigenes Plugin check_ xyz. Zusammen mit dem entsprechenden Service auf der Client-Seite könnte dies durch ein Client-Server-Paar umgesetzt werden. Eine andere Variante könnte ein sogenanntes Probing von Services sein. Denkbar wäre beispielsweise bei Mailservern, dass sich das Plugin mit dem entsprechendem Dienst verbindet, ein HELLO empfängt und somit erkennt, dass der Dienst verfügbar ist. Anschließend wird die Verbindung wieder getrennt.*
Im Beispiel des zweiten und dritten Clients startet Nagios ein „Zwischen"-Plugin, check_ by_ ssh beziehungsweise check_ nrpe, um auf der gegenüberliegenden Seite das eigentliche Plugin zu starten. Beim vierten Client wird eine SNMP-Abfrage gestartet, wobei der Client einen verfügbaren SNMP-Agent besitzen muss. Häufig entsprechen solche Clients Embedded Devices wie USV-Controller, Temperatursensoren oder aber Netzwerk-Switche. Diese vier erwähnten Methoden entsprechen sogenannten „aktiven" Checks.
Das letzte Beispiel ist im Gegenzug dazu ein „passiver" Check. Nagios triggert das Plugin nicht von selbst, sondern wartet, bis das Plugin von sich aus Daten liefert. Dies geschieht über das Programm send_ nsca. Auf dem Nagios-Server läuft der sogenannte Nagios Service Check Acceptor (NSCA) als Daemon, der die Daten von send_ nsca empfängt und weiterleitet.

Im Folgenden soll kurz der Vollständigkeit halber auf den aktiven, direkten Check, SNMP, und im Anschluss auf die beiden aktiven, indirekten Checks, check_by_ssh und NRPE, eingegangen werden. Die letzten beiden indirekten Checks gilt es vor allem für das Kapitel 4.3 [S. 69] zu evaluieren.

Die Informationsabfrage per *Simple Network Management Protocol* (SNMP) wird meistens bei Hosts wie Routern, Switchen, Druckern oder Embedded Devices verwendet, seltener jedoch für Workstations oder Server (es existieren beispielsweise für Unix SNMP-Daemonen).
Obwohl SNMP ein Protokoll ist, steht es ferner für ein *Internet Standard Management Framework* [Bart 08b]. Dieses *Framework* besteht aus folgenden Komponenten [Maur 05]:

- Mindestens einem Agenten
- Mindestens einem Manager
- Einem Protokoll zur Kommunikation zwischen Agent und Manager
- Genau einer Informationsbasis (Management Information Base)

Als Agent kann eine SNMP-Engine bezeichnet werden, die auf einem netzwerkfähigen Host läuft. Aus Sicht von Nagios sind dies Hosts, welche die für uns interessanten Messdaten liefern. Im Gegensatz dazu existiert der Manager, welcher die gegenüberliegende Seite zum Agenten bildet. Dies wäre das entsprechende Nagios-Plugin, welches auf dem Nagios-Server läuft und in bestimmten Zeitintervallen die Agenten abfragt. Zum Austausch von Daten zwischen Agent und Manager steht das Simple Network Management Protocol. Damit Informationen ausgetauscht werden können, muss allerdings auch exakt definiert sein, welche Informationen überhaupt existieren. Diese Informationsmenge ist in der sogenannten *Management Information Base* (MIB) definiert [Maur 05]. Anhand dieser Schnittstelle kann der User die SNMP-Anfragen für den Agenten bilden. Diese Zeichenketten (SNMP-String) sind im Allgemeinen herstellerspezifisch. Das Einbeziehen eigener Plugins auf den Geräten oder neuer Abfragedaten ist daher nicht möglich. Im Zuge dieser Arbeit wurden auch eigene Plugins für Nagios entwickelt, welche die Netzwerk-Switche des Instituts per SNMP nach freien Arbeitsspeicher, Temperatur, Netzwerkdatenraten und Weiterem abfragen.

Im Gegensatz zu SNMP existieren *check_by_ssh* und *NRPE*. Wie in Abbildung 5 [S. 12] zu sehen, stellen beide Plugins lediglich eine Mittelschicht zur eigentlichen

Plugin-Ausführung dar.

Das Nagios-Plugin *check_by_ssh* setzt auf OpenSSH auf. Zur Computeradministration ist auf den meisten Servern für gewöhnlich bereits der OpenSSH-Daemon installiert und Port 22 freigeschaltet. Diesen Vorteil nutzt *check_by_ssh* aus, indem es eine Client-Server-Verbindung über Secure Shell vom Nagios-Server zum entfernten Client aufbaut, das auszuführende, lokale Plugin triggert und dessen Ausgabedaten zum Nagios-Server zurück transportiert. Es muss keine weitere Software auf dem entfernten Rechner installiert werden und da zur Kommunikation das verbreitetere SSH verwendet wird, ist die Datenübertragung immer verschlüsselt [Flac 07]. Auf dem Nagios-Server muss lediglich ein *Private/Public-Keypaar* erzeugt und der *Public-Key* auf den Client abgelegt werden, damit eine passwortlose Authentifizierung beim OpenSSH-Server durchgeführt werden kann. Im Gegensatz zu NRPE ist dieses Verfahren nicht-proprietär. Der wesentliche Vorteil liegt vor allem darin, dass für Service-Checks innerhalb der DMZ (Demilitarized Zone) kein weiterer Port auf der Firewall freigeschaltet werden muss.

Die Alternative zu check_by_ssh ist der *Nagios Remote Plugin Executor* (NRPE). Im Gegensatz zu check_by_ssh bringt NRPE seine eigene Software-Kollektion mit, wobei ein Teil auf dem entfernten Client als System-Daemon installiert werden muss (NRPE-Daemon) und der andere Teil auf dem Nagios-Server als Plugin (check_nrpe) eingerichtet wird. Eine verschlüsselte Verbindung über OpenSSL wird nur dann hergestellt, wenn diese einkompiliert wird. Standardmäßig wird dies jedoch aus Kompatibilitätsgründen deaktiviert [Flac 07]. Auf der Clientseite muss der NRPE-Daemon über den *inetd* in das System eingebunden und ein extra Port (5666/TCP) freigeschaltet werden. Nachteilig ist bei dieser Variante, dass zur Pluginausführung innerhalb Hosts der DMZ bei der Firewall ein weiterer Port geöffnet werden muss, was in den meisten Fällen eher unerwünscht ist. Der Source-Code vom NRPE ist zwar frei zugänglich, jedoch steht dieser nicht unter einer freien Lizenz wie der GPL, sondern ist proprietär [Flac 07].

2.2 Round Robin Datenbank

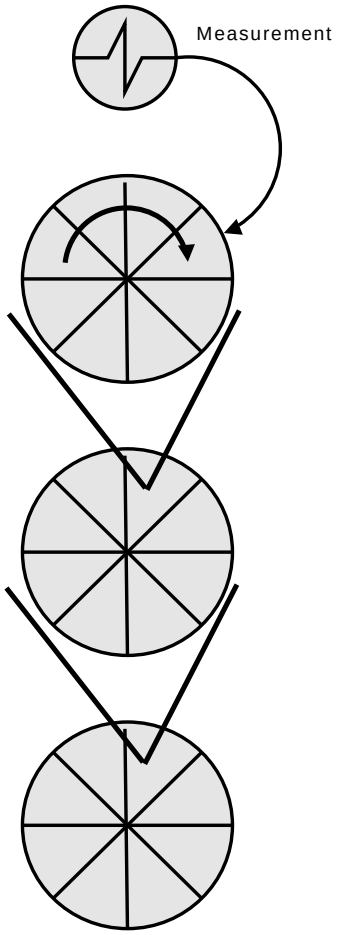


Abbildung 6: Daten-Fluss innerhalb einer RRD: *Eintreffende Messdaten kommen beispielsweise in ein minütlich-auflösendes Round Robin Archiv. Da die Anzahl der Datensätze konstant bleibt, werden nach jedem Eintreffen neuer Daten die alten Daten um einen Datensatz im Ring verschoben. Um Daten über Jahre hinweg sammeln zu können, werden zeitlich höher-auflösende Datensätze zusammengefasst und in zeitlich niedriger-auflösende Round Robin Archive gespeichert und somit Messdaten verlustbehaftet komprimiert.*

Round Robin Datenbanken (RRD) sind Dateien, die numerische Werte als Zeitreihe speichern können. Dies sind beispielsweise Daten vom Netzwerk-Traffic, Druckaufkommen oder von Temperatur-Sensoren. Round Robin bedeutet, dass die Daten in einem Ringpuffer abgelegt werden. Das heißt, dass die Größe der Datenbank selbst über die Zeit konstant bleibt und ältere Werte mit Neuen überschrieben werden [Oeti 02]. Um Werte über Jahre hinweg zu speichern, müssen Daten zusammengefasst und komprimiert werden, damit die Datenbank nicht zu viel Ressourcen in Anspruch nimmt. So besitzt eine Round Robin Datenbank mehrere sogenannte Round Robin Archive [Oeti 02]. Ein solches Archiv besteht jeweils aus einem Ringpuffer [Bart 08a]. Es existieren für gewöhnlich Archive zur minütlichen, stündlichen, täglichen, wöchentlichen, monatlichen und jährlichen Darstellung. Wie in Abbildung 6 [S. 15] zu sehen ist, werden dabei in bestimmten Intervallen von zeitlich höher auflösende Daten Durchschnitte gebildet und in ein Round Robin Archiv von zeitlich niedriger auflösenden Daten gespeichert. Details gehen so über die Mittelung verloren.

Diese Art von numerischer Datenbank findet in vielen Anwendungen Interesse. So werden diese unter anderem auch in Webfrontends von Routern zur Darstellung der Netzwerkauslastung verwendet (*The Multi Router Traffic Grapher* [Oeti 98] oder beispielsweise auch für beliebige Prozesse *Cacti* und *NagiosGrapher*).

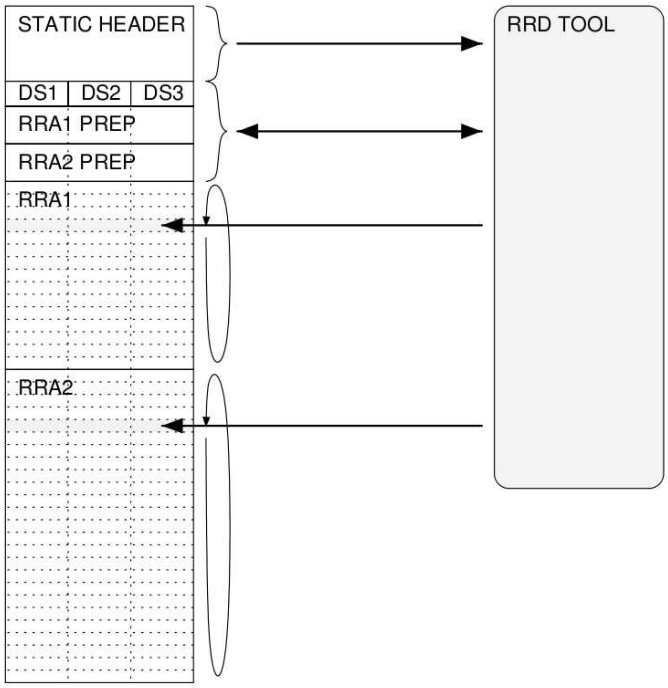


Abbildung 7: Aufbau einer RRD, [Oeti 98]

Auf eine Round Robin Datenbank kann lediglich indirekt über das von Tobias Oetiker in C entwickelte Programm *rrdtool* zugegriffen werden (Abbildung 7 [S. 16]). Das RRDTool dient dem Erstellen, Aktualisieren oder Ändern von RRDs sowie dem Ausleiten von Datensätzen und dem Erstellen von Graphiken [Oeti 02].
Mittels eines XML-Dumps einer RRD kann die interne Struktur sichtbar gemacht werden [Rock 04]. So existiert ein Header mit RRD-Versionsinformationen, dem Intervall in Sekunden, in welchen Daten eintreffen, und einem Unix-Timestamp vom letzten Update. Des Weiteren gibt es verschiedene Datenquellen (DSn), welche unter anderem Namen, Typ und letztes Datum beinhalten. Zu den Round Robin Archiven (RRAn), in denen sich die Datensätze befinden, kommen zusätzlich Metadaten (RRAn PREP) hinzu, die die Komprimierungsfunktion (Consolidation Functions: Minimum, Maximum und Average), einen Zeit-Multiplikator (pdp_per_row), über den die verschiedenen Archive identifiziert werden können, die letzten Datenpunkte und weitere Informationen beinhalten [Oeti 02].

NagiosGrapher erstellt standardmäßig RRDs mit einem Datenintervall von 300s, die folgende Parametern besitzen:

Archiv	Diagramm-Intervall	Auflösung	Eindeutige ID	# Datensätze
RRA1	Letzte 12h	5min	1	300
RRA2	Letzte 2d	10min	2	1010
RRA3	Letzte 2Wo	1h	12	770
RRA4	Letzte 2Mo	9.5h	72	1488
RRA5	Letzte 2a	24h	288	744

Tabelle 2: Eigenschaften der einzelnen Round Robin Archive, die durch NagiosGrapher standardmäßig erstellt wurden: *Die Anzahl der Datensätze kann erhöht werden, indem die zeitlichen Service-Check-Intervalle von Nagios verringert werden. In Tests stellte sich heraus, dass ein Verringern des Intervalls von fünf auf eine Minute eine Erhöhung der RRA1-Datensatz-Menge von 300 auf 700 Datensätze nach sich zog. Für mathematische Analysen ist eine größere Menge an Messdaten immer von Vorteil, jedoch überfordert ein Service-Check-Intervall von einer Minute bereits die momentane Teststellung mit 39 Hosts und 147 zu überprüfenden Services. Lediglich durchschnittlich 20% der Services konnten innerhalb einer Minute überprüft werden.*

2.3 NagiosGrapher

Die in dieser Arbeit entwickelte Schnittstelle zum Evaluieren diverser Algorithmen erweitert das freie, in Perl geschriebene Nagios-Addon *NagiosGrapher* der Firma Netways. NagiosGrapher ermöglicht es, Check-Performance-Daten über die Zeit graphisch abzubilden. Die von Nagios erfassten Daten werden dabei durch NagiosGrapher in Round Robin Datenbanken gespeichert [Bart 08b]. NagiosGrapher kann somit als Schnittstelle zwischen Nagios und den Round Robin Datenbank-Backend betrachtet werden.
Sobald eintreffende Daten erkannt wurden, erstellt NagiosGrapher die zugehörigen RRDs vollautomatisch. Die Integration in die Nagios Weboberfläche verläuft ebenfalls ohne händische Eingriffe. Die zu erstellenden Graphen werden pro Graph konfiguriert, sodass gleiche Services auf unterschiedlichen Hosts lediglich einmalig eingerichtet werden müssen [Bart 08b]. Daten werden nahezu in „Echtzeit“ erfasst und als Zeitreihe visualisiert (Abbildung 8 [S. 18]).

Auf dem eingerichteten Nagios-Testserver werden die Check-Performance-Daten durch Nagios über das File-Interface an NagiosGrapher durchgereicht. Aller fünf Minuten werden durch Nagios die Pluginabfragen getriggert und die Ausgaben in das File-Interface geschrieben, wobei NagiosGrapher aller 30 Sekunden nach Daten abfragt. Neue Einträge werden über *Reguläre Ausdrücke* mit konfigurierten NagiosGrapher-Services verglichen und nach zu extrahierenden numerischen Werten durchsucht [Bart 08b]. Die RRDs werden daraufhin mit den gefundenen Daten aktualisiert. In der Praxis kann es bei ähnlichen Service-Namen oft zu Konflikten kommen, sodass Service-Checks umbenannt werden müssen. Laut [Ehm 08] kann es bereits

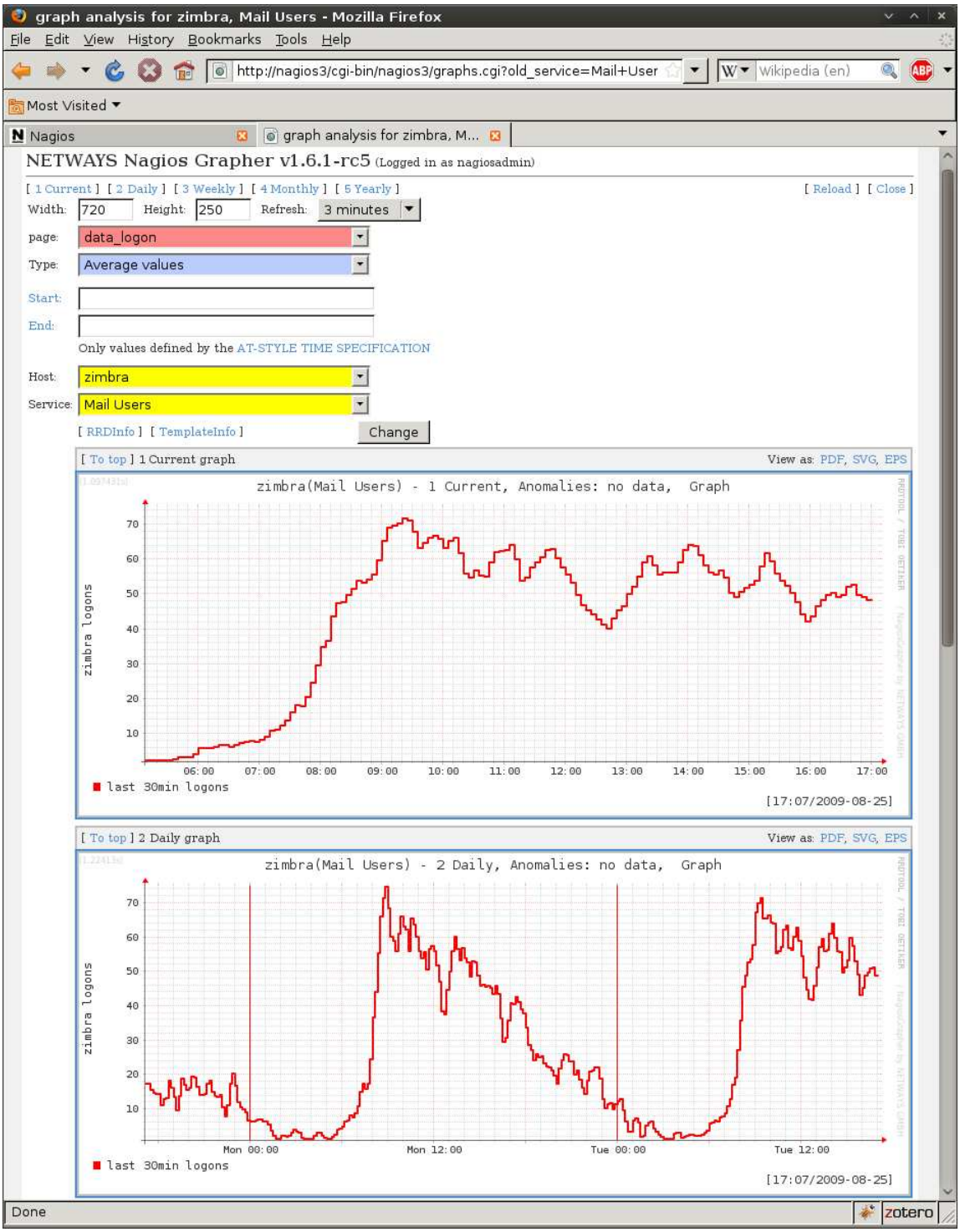


Abbildung 8: NagiosGrapher Weboberfläche: *Zu sehen ist der Graph aller aktuellen Zimbra-Mail-Benutzer (Mitarbeiter) des Instituts während eines Arbeitstages. Die Bedienoberfläche vom NagiosGrapher ist einfach und intuitiv gehalten. Die Auswahl von Host und Service ist jeweils gelb unterlegt. Grafiken können als Vektorgrafiken exportiert werden.*

bei 7.000 beobachteten Services zu enormen Performance-Problemen kommen. In Tests am CERN wurden Update-Zeiten für RRDs von bis zu 20 Minuten beobachtet [Ehm 08]. Bei derart vielen Services wäre allerdings ein Raw-Processing der Check-Performance-Daten mit einer eigenentwickelten performanteren C-Schnittstelle zu den RRDs die bessere Lösung. In unseren Tests werden circa 150 Services überprüft.

2.4 Linux Kernel Paketanalyse

2.4.1 Allgemeines

Im Zuge dieser Arbeit wurden Plugins für Nagios entwickelt, um überhaupt für uns interessante Daten sammeln zu können. So entstand beispielsweise ein Perl-Plugin, welches über SNMP Informationen über Arbeitsspeicher, interne Temperatur, sowie empfangene und versendete Pakete und entsprechende Transferraten in Megabyte von allen Foundry-Networks Switchen im Hause abgreift. Ein anderes entwickeltes Plugin wiederum fragt die aktuelle Benutzerstatistik vom Zimbra-Mail-Server via LDAP ab. Diese und weitere kleine Plugins seien jedoch nur am Rande erwähnt.
Im weiteren Verlauf der Arbeit wird der Schwerpunkt auf ein etwas umfangreicheres Nagios-Plugin gesetzt, welches in C geschrieben ist und eng mit dem Linux Kernel interagiert. Dieses ist auf dem Institutsrouter, über den alle Verbindungen zum Internet gehen, im Einsatz und fängt dabei Netzwerkpakete zur weiteren Analyse ab.

Dieser Abschnitt soll daher die Grundlagen für ein spezielles Verfahren zur passiven Netzwerkpaketanalyse näher erläutern, um die später vorgestellte Arbeitsweise verstehen zu können. Das im Abschnitt 4.3 [S. 69] entwickelte Programm nutzt diese hier näher vorgestellten Methoden, um so Statistiken für Nagios erzeugen zu können.

2.4.2 PF_PACKET

Um eine Netzwerkpaketanalyse betreiben zu können, muss man sich diversen Problemen stellen:

Ethernet-Adressierung: Pakete, welche nicht der Ethernet-Adresse des eigenen Netzwerk-Interfaces entsprechen, werden vom Netzwerktreiber verworfen.

Protokoll-Stack: Um Paketverwerfungen zu verhindern, würde ein Listening-Socket je Ethernet-Protokoll-Typ (insgesamt 58 im Linux Kernel) und je Port (insgesamt 65535) benötigt

Netzwerk-Stack: Informationsverluste von Paketen nehmen mit höheren Schichten immer mehr zu (Wegfall von Headern)

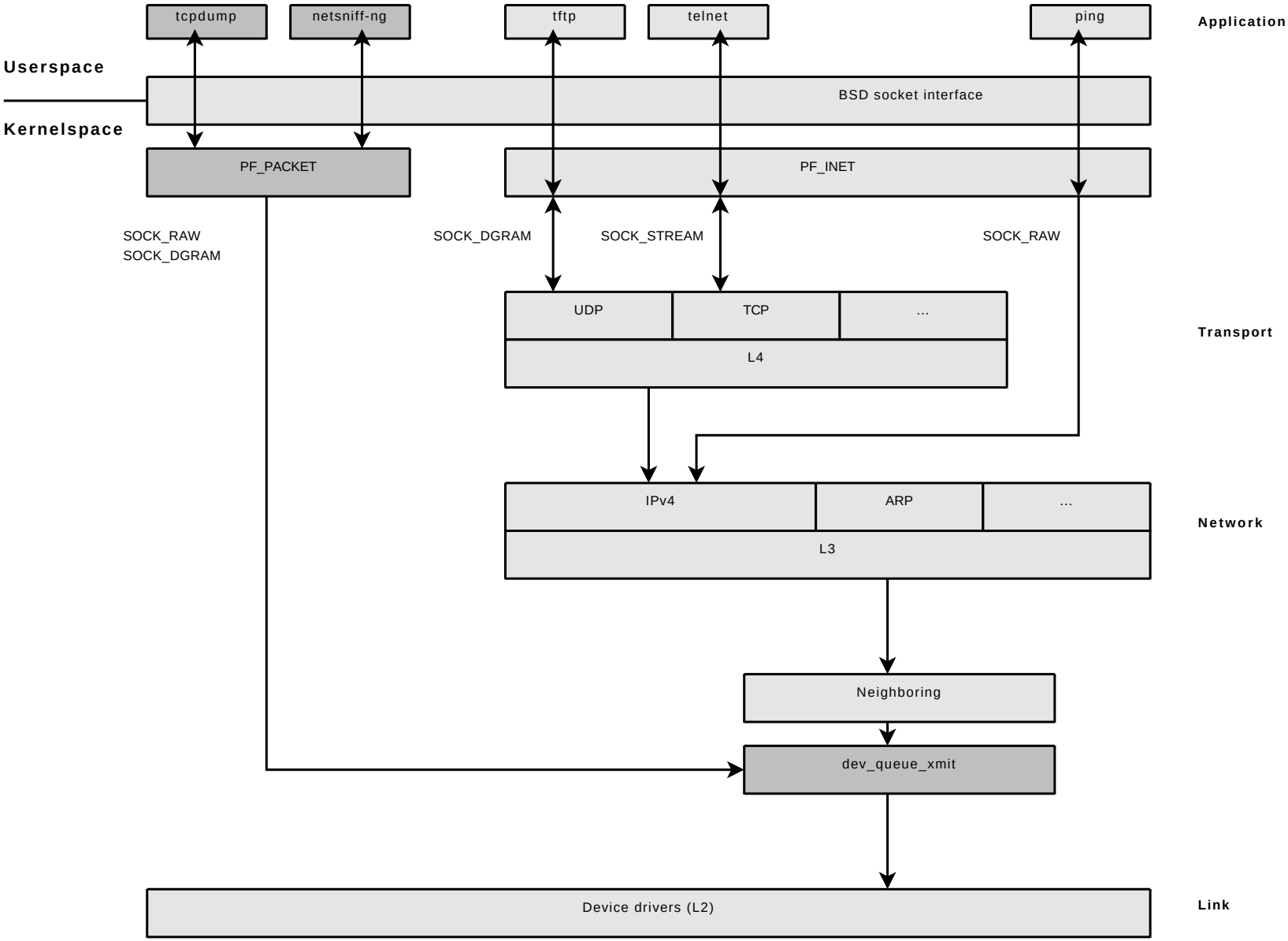


Abbildung 9: Übersicht Linux Netzwerk-Stack [Benv 05]

Letzteres Problem kann gelöst werden, indem man einen speziellen Typ an Socketfamilie verwendet. Wie in Abbildung 9 [S. 20] ersichtlich, existieren neben gewöhnlichen Socketfamilien wie *PF_INET* daher auch eher selten verwendete, dafür aber für Spezialfälle ausgelegte Familien wie *PF_PACKET*.
Das *PF_PACKET*-Protokoll soll im Folgenden näher betrachtet werden.

PF_PACKET ermöglicht Userspace-Anwendungen das Empfangen und Versenden von Paketen direkt über den Netzwerkkarten-Treiber und umgeht somit den kompletten Netzwerk-Stack [Inso 02a] [Inso 02b]. Dies bedeutet, dass Pakete von der Applikation direkt an den Gerätetreiber des Ethernet-Interfaces durchgereicht werden. Die Methode *dev_queue_xmit()* ist dabei für den Versand des Paketes zuständig [Manw 03].
Zwei unterschiedliche Sockettypen werden von *PF_PACKET* unterstützt. Dies sind zum einen *SOCK_DGRAM* und zum anderen *SOCK_RAW*. Erstere überlässt dem Kernel noch die Aufgabe, Ethernet-Header zu verwalten, wobei letztere der Userspace-Anwendung die komplette Kontrolle überlässt [Inso 02a] [Inso 02b].
Neben Sockettyp ist auch die Angabe des Ethernet Protokoll-Identifiers notwendig, welche unter *include/linux/if_ether.h* definiert sind. Der wohl am häufigsten verwendete Identifier ist *ETH_P_IP*. Da jedoch bei Verwendung dieses Identifiers das Mitschneiden von vielen weiteren interessanten Paketen wie beispielsweise ARP von vornherein ausgeschlossen wird, sollten für die Implementierung eher alle Protokoll-Identifier mit der Option *ETH_P_ALL* zugelassen werden.
Allerdings ist dies mit Vorsicht zu behandeln, da bei Messungen von komplett ungefiltertem Netzwerkverkehr im sogenannten *Promiscuous Mode* innerhalb weniger Sekunden sehr große Datenmengen an den Userspace durchgeleitet werden können. Empfehlenswert ist daher die Verwendung von *ETH_P_ALL* in Verbindung mit einer *Berkeley Packet Filter*-Routine.

2.4.3 Promiscuous Mode

Die *PF_PACKET* Protokollfamilie erlaubt den Zugriff auf Frames, die am Interface eintreffen und deren Ziel-MAC-Adresse der Adresse der eigenen Netzwerkschnittstelle entspricht. Um jedoch Pakete mitschneiden zu können, sollten möglichst auch jene Unicast-Frames weitergereicht werden, welche andere MAC-Adressen als Ziel haben. Für gewöhnlich werden diese Pakete von der Netzwerkkarte bereits auf Hardwareebene verworfen. Diese Arbeitsweise wird auch *Nonpromiscuous Mode* genannt [Reus 04]. Ausnahmen sind allerdings eintreffende Broadcasts, Multicasts, welche in der Multicast-Liste der Netzwerkkarte registriert sind und All-Multicast-Pakete, bei welchen das

group-bit gesetzt ist, werden von jedem Interface empfangen und ausgewertet [Reus 04]. Des weiteren gibt es noch eine letzte Ausnahme, auf die kurz eingegangen wird: Der *Promiscuous Mode*.
In diesem Mode werden im Gegensatz zu den bisher erwähnten die Ziel-Adressen nicht per Bitmaske überprüft, sodass alle Frames am Interface eintreffen und fremde Frames nicht mehr verworfen werden [Eric 08]. Die Freischaltung dieses Modus ist eine wichtige Voraussetzung für eine Netzwerkpaketanalyse.
Es sollte allerdings zur Kenntnis genommen werden, dass heutzutage bereits Switche der unteren Preissegmentklasse so intelligent sind und Pakete nicht an alle Ports weiterleiten. Intern besitzen diese eine sogenannte *Source-Address-Table*, welche MAC-Adressen mit zugehörigen physikalischen Port abspeichert. Diese Tabelle wird bereits durch Abfangen der ARP-Pakete ausreichend gefüllt [Eric 08]. Sobald jedoch eine Zieladresse keinem Port zugeordnet werden kann (Lernphase), flutet der Switch sämtliche Ports mit diesem bestimmten Frame und speichert bei einer Rückantwort den Quellport. Durch sogenanntes *aktives Sniffing* kann dies allerdings auch mit Hilfe von Techniken wie *ARP cache poisoning* im Allgemeinen umgangen werden [Eric 08]. Im Normalfall macht allerdings ein Einsatz von Netzwerksniffern nur an zentralen Datentransferstellen wie beispielsweise Routern Sinn.

2.4.4 Berkeley Packet Filter

Wie bereits erwähnt, macht es aufgrund der Datenflut wenig Sinn, im *Promiscuous Mode* über *PF_PACKET*-Sockets sämtlichen Verkehr zu analysieren. Es ist demzufolge ein Filter notwendig. Der einfachste Ansatz wäre ein Paketfilter im Userspace, welcher mit einfachen *if*-Bedingungen auf Paketkonditionen prüft. Dies wäre allerdings sehr ineffizient, da jedes Paket innerhalb des Kernels und vom Kernel in den Userspace-Adressbereich kopiert werden muss, damit eine Auswertung im Userspace stattfinden kann. Eine bessere Lösung wäre es, die Pakete gleich im Kernel so früh wie möglich zu filtern.
Dieser Ansatz wurde als *Linux Packet Filter* in den Kernel integriert, sodass dieser direkt mit dem *PF_PACKET*-Protokoll zusammenarbeiten kann [Inso 02a] [Inso 02b]. Kurz nach der lower-half Empfangsroutine des Netzwerk-Treibers wird anschließend der Filter angesetzt, der entscheidet, ob Frames zur Userspace-Applikation durchgereicht oder verworfen werden [Benv 05].

Um dem Benutzer Flexibilität zu gewährleisten, wurde die *Paket-Filter-Engine* als Zustandsautomat in den Kernel integriert. Diese kann vom Nutzer in einer pseudo Maschinensprache, welche BPF (Berkeley Packet Filter) genannt wird, definierte Pro-

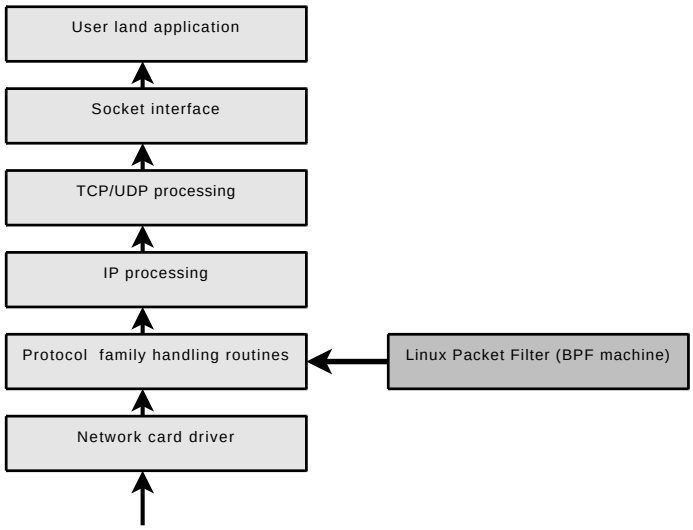


Abbildung 10: Linux Packet Filter mit BPF Maschinencode [Inso 01]

gramme interpretieren und entsprechende Entscheidungen für ankommende Frames treffen [McCa 92]. Da diese im Kernel direkt im Anschluss an die Netzwerk-Treiber-Schicht und vor dem Netzwerk-Stack integriert wurde, kann effizient entschieden werden, welche Pakete zu den überliegenden Socket-Handlern durchgereicht und welche verworfen werden. Die Befehle werden direkt auf ein Ethernet-Frame angewendet, sodass auf jedes einzelne Byte zugegriffen werden kann.

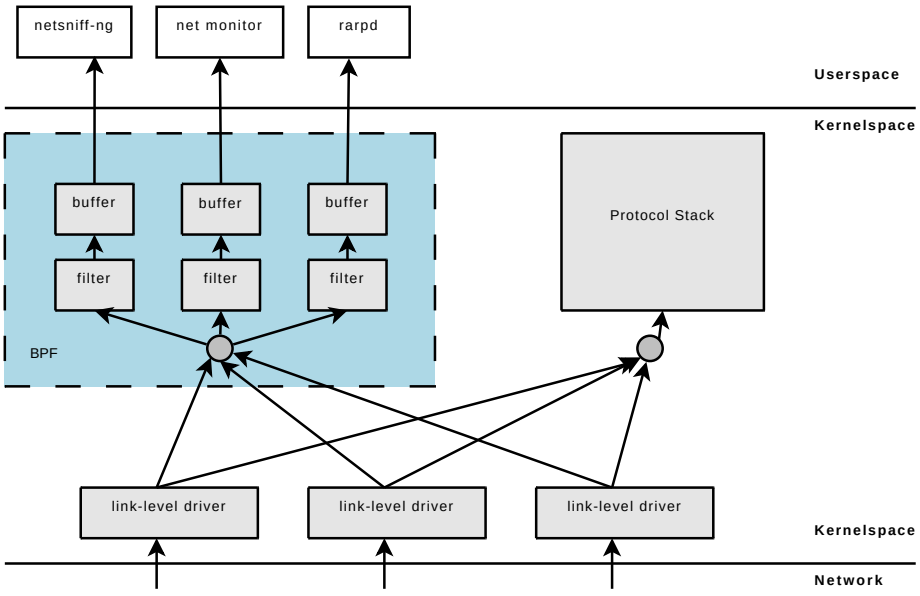


Abbildung 11: Linux Packet Filter aus Kernelsicht [McCa 92]

Opcode	Addressing Mode	Description
	Load Instructions	
ldb	[k] or [x + k]	load unsigned byte into accu
ldh	[k] or [x + k]	load unsigned half word into accu
ld	#k or #len or M[k] or [k] or [x + k]	load into accu
ldx	#k or #len or M[k] or 4*([k]&0xf)	load into index register
	Store Instructions	
st	M[k]	copy accu into scratch memory store
stx	M[k]	copy index register into scratch memory store
	Branch Instructions	
jmp	L	jump
jeq	#k, Lt, Lf	jump if equal
jgt	#k, Lt, Lf	jump if greater than
jge	#k, Lt, Lf	jump if greater than or equal
jset	#k, Lt, Lf	jump if bitwise and
	ALU Instructions	
add	#k or x	addition
sub	#k or x	subtraction
mul	#k or x	multiplication
div	#k or x	division
and	#k or x	bitwise and
or	#k or x	bitwise or
lsh	#k or x	left shift
rsh	#k or x	right shift
	Return Instructions	
ret	#k or a	return
	Miscellaneous Instructions	
tax		copy accu to index register
txa		copy index register to accu

Tabelle 3: BPF Instruction Set [McCa 92]

Mode	Description
#k	the literal value stored in k
#len	the length of the packet
M[k]	the word at offset k in the scratch memory store
[k]	the byte, halfword, or word at byte offset k in the packet
[x + k]	the byte, halfword, or word at the offset $x+k$ in the packet
L	an offset from the current instruction to L
#k, Lt, Lf	the offset to Lt if the predicate is true, otherwise the offset to Lf
x	the index register
4*([k]&0xf)	four times the value of the low four bits of the byte at offset k in the packet

Tabelle 4: BPF Addressing Modes [McCa 92]

Das eigentliche BPF-Programm, welches mit oben erwähnten Befehlen programmiert werden kann, muss in einem speziellen Format an den Socket übergeben werden. Dies bedeutet, dass nach Programmentwurf Zeile für Zeile in Opcodes mit entsprechenden Argumenten umgewandelt werden muss [McCa 92]. Diese haben folgendes festes Format:

opcode:16	**jt:8**	**jf:8**
k:32		

Tabelle 5: BPF Operationscode-Format [McCa 92]

jt und jf geben dabei die Sprung-Offsets für die nächste Anweisung an, wobei Null die zum aktuellen Befehl nachfolgende Anweisung darstellt. Ein Offset von Eins würde einen Sprung zur übernächsten Anweisung zur Folge haben.

Das folgende einfache Beispiel aus Abbildung 12 [S. 26] soll dabei zum Verständnis beitragen. Später im Anhang C [S. V] folgt ein zweites Beispiel, welches neben weiteren Filtern im Rahmen dieser Arbeit entwickelt wurde. Zu sehen ist dabei, wie ein typischer Filter in das genannte Operationscode-Format umgewandelt werden muss. Da die Opcodes direkt im Kontext des Ethernet-Frames arbeiten und sich auf dessen Offsets beziehen, wird Kenntnis im Aufbau eines Ethernet-Frames mit IP-Datagramm und TCP-, sowie UDP-Segmenten vorausgesetzt, welche unter anderem ausführlich in [Pete 07] beschrieben sind.

Nachteilig am Berkeley Packet Filter ist, dass pro Socket lediglich ein Filter-Code verwendet werden kann. Sollen mehrere Filter auf Frames angewendet werden, so

```
          ldh    [12]
          jeq    #ETHERTYPE_IP, L1, L2
    L2:   jeq    #ETHERTYPE_ARP, L1, L3
    L1:   ret    #TRUE
    L3:   ret    #0
```

Abbildung 12: Beispiel-Filter für Pakete mit der Bedingung „ip or arp“: *Die ersten beiden Bytes (unsigned halfword), die nach Destination- und Source-MAC-Address folgen und den Ethernet-Typ (IP, ARP, RARP, usw.) bezeichnen, werden in den Akkumulator geladen. Falls der Wert vom Typ IP (#0x800) ist, so wird zum Label L1 gesprungen und mit dem Return-Wert true das Frame zum Socket-Handler weitergeleitet. Das Frame passiert den Filter. Falls jedoch der Wert nicht vom Typ IP ist, so wird zum Label L2 gesprungen und auf den Wert ARP (#0x806) getestet. Ist das Typfeld vom Wert ARP, so wird wiederum zum Label L3 gesprungen und das Frame passiert den Filter. Mit dem Return-Wert false wird das BPF-Programm beendet, falls auch diese Bedingung fehlschlägt. Das Frame wird nicht zum Socket-Handler durchgereicht.*

müssten entsprechende Threads oder Programminstanzen gestartet werden, die jeweils einen Socket mit Filter erstellen (sogenanntes Socket-Clustering [Deri 09b]). Wendet man dies auf den RX_RING an, so kommt man unter Umständen schnell an die Speichergrenzen des Kernelspace.
Eine Alternative zum BPF ist die Möglichkeit des *Dynamic Bloom Filterings*, welche unter [Deri 09b] näher beschrieben ist.

2.4.5 RX_RING

Der Linux Kernel unterteilt den physischen Adressbereich in einen virtuellen, der wiederum in sogenannte Pages (für gewöhnlich 4096 Byte je Page für 32-Bit Architekturen) eingeteilt ist. Dadurch ist es möglich, den Speicher unabhängig von der physischen Grenze des RAMs zu verwalten und damit auch langsameren Festplattenspeicher bei Bedarf mit einzubeziehen (Swap-Space). Die Gesamtheit des virtuellen Speichers wird in zwei Teile untergliedert: in lowmem und in highmem [Corb 04]. Bei einer 32-Bit Architektur steht dies im Verhältnis 1 zu 3, d.h. 4 GB RAM werden in 1 GB lowmem und 3 GB highmem eingeteilt, wobei für den Kernel die hinteren Adressbereiche vorgesehen sind [Corb 04]. Dieser hintere Teil wird nicht in Pages unterteilt [Love 05]. Dabei steht dem Kernelspace der *lowmem* Bereich und dem Userspace mit all seinen Prozessen der *highmem* Bereich zur Verfügung. Durch diese Untergliederung ist es auch nicht ohne weiteres möglich, vom lowmem in den highmem Bereich zuzugreifen oder umgekehrt. Dies bedeutet, dass Speicher bei Systemcalls von dem einen in den anderen Bereich kopiert werden muss (siehe *copy_to_user()* bzw. *copy_from_user()*) [Wats 07]. Oft tritt dieser Fall bei Gerätetreiberroutinen [Venk 08] oder Systemcalls ein, sodass bei Datentransfers mit einem

zusätzlichen Kopiervorgang gerechnet werden muss. Bei zeitkritischen Anwendungen wie der Netzwerkpaketanalyse bringt dies zusätzlichen, unnötigen Aufwand mit sich. Pro Paket wird im Normalfall ein Systemcall (beispielsweise *recvfrom()*) gerufen, welcher das Byte-Array vom Frame im Kernel mittels *copy_to_user()*-Aufruf vom lowmem in den highmem kopiert, damit der rufende Prozess auf diesen Daten arbeiten kann.
Für Anwendungen wie Netzwerksniffer kann dies zur Folge haben, dass aufgrund dieser Vorgänge Pakete verloren gehen. Um dies zu umgehen, wurde im Linux Kernel für Netzwerk-Pakete ein Ringpuffer implementiert, welcher eintreffende Pakete direkt in diese Speicherstruktur hineinschreibt [Cama 09]. Mittels speziellem Memory Mapping auf Socketstrukturen kann daraufhin vom Userspace ohne Systemcall und ohne extra Kopiervorgang in den allokierten lowmem Bereich via Pointer zugegriffen werden. Der vorher allokierte Ring ist spezieller Kernelspeicher, welcher nicht in den Swap-Space verschoben werden kann [Cama 09].
Die Funktionalität des Ringpuffers gibt es für eintreffende Frames, RX_RING genannt, und für abgehende Frames, TX_RING genannt [Deri 09b]. Ist dabei der allokierte Ring mit Frames komplett gefüllt, so werden die ältesten wieder mit neu eintreffenden überschrieben.
Der RX_RING ist untergliedert in Blöcke (Abbildung 13 [S. 27]), sodass mitgeschnittene Frames entsprechend gruppiert werden. Jeder Block ist ein nicht fragmentierter Speicherbereich im RAM. Zur Speicherallokierung ist die Angabe der Blockgröße, Framegröße und die Anzahl an Blöcken notwendig [Cama 09]. Die Framegröße sollte immer kleiner als die Blockgröße sein. Für die Blockgröße ist es sinnvoll, ein Vielfaches der Page-Größe zu verwenden [Cama 09].

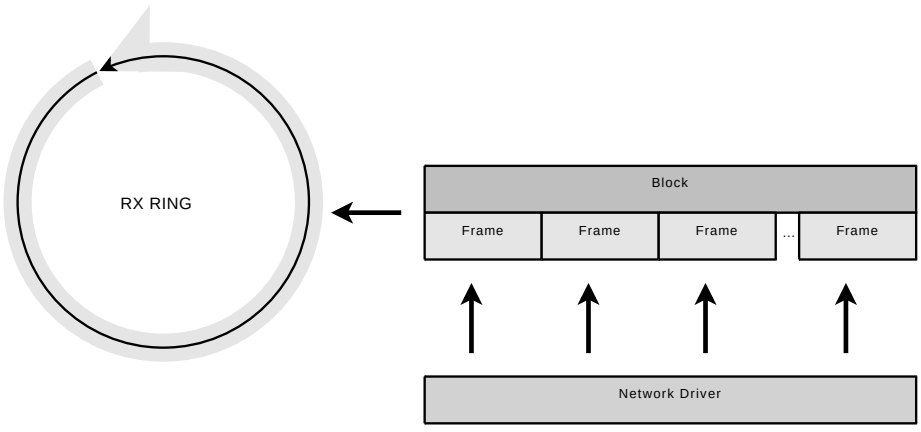


Abbildung 13: RX_RING Aufbau: *Ein RX_RING besteht aus mehreren nicht-fragmentierbaren Blöcken, die wiederum in Frames gleicher Länge untergliedert sind. Eintreffende Pakete von der Netzwerkschnittstelle werden durch den Netzwerktreiber in den Ring hineingeschrieben.*

3 Algorithmische Realisierung

In diesem Abschnitt werden Algorithmen und Ideen zur Anomalieerkennung (Abschnitt *Methoden der Erkennung von Anomalien*), sowie zur Extrapolation (Abschnitt *Methoden der Extrapolation von Daten*) von Nagios-Messdaten vorgestellt. Zunächst werden einführend Annahmen über das Modell der eintreffenden Daten getroffen (Abschnitt *Zeitreihen und Stochastische Prozesse*) und im Anschluss einige der möglichen algorithmischen Herangehensweisen gezeigt.

3.1 Zeitreihen und Stochastische Prozesse

Unser Ziel ist es, Plugin-Daten von Nagios über der Zeit auf Verhaltensanomalien hin zu untersuchen, sowie längerfristige Trends vorherzusagen. Da die eintreffenden Daten durch *NagiosGrapher* in Round Robin Datenbanken als zeitabhängige Folge von numerischen Datenpunkten gespeichert werden, erhalten wir somit zu analysierende Zeitreihen. Die Datenpunkte sind dabei aufgrund der Beschaffenheit der RRDs zeitdiskret und äquidistant. Zur Analyse werden die jeweiligen Datenpunkte als univariate Zeitreihe betrachtet.
Für Zeitreihen ist es typisch, dass sie durch ein Zusammenwirken von regelhaften und zufälligen Ursachen entstehen. So konnten beispielsweise bei institutsinternen Netzwerkpaketanalysen (Abschnitt 4.3 [S. 69]) vom Router *saphir* regelmäßig besonders herausstechende Maximalpunkte während der Vormittagszeit beobachtet werden (Abbildung 14 [S. 30]), die allerdings je nach zufälligen Ursachen in Amplitude, Häufigkeit und Uhrzeit schwanken. Für die Amplituden spielen unter anderem deterministische Ursachen wie die Wochentage eine Rolle. So fällt die Amplitude der Maximalpunkte an Vormittagen von Wochenenden deutlich geringer als an Arbeitstagen aus, da sich schlussfolgernd auch weniger Benutzer im Institutsnetz befinden. Ursachen für die unterschiedlichen Amplituden innerhalb der Arbeitstage (zu gleichen Uhrzeiten) wiederum können nicht ohne weitere Informationen über den Prozess gedeutet werden. Allerdings fallen auf den ersten Blick Ähnlichkeiten in den Tagesverläufen auf, die man sich zunutze machen kann. Diese Ähnlichkeiten, welche innerhalb einer Zeitreihe beobachtet werden, können durch die sogenannte Autokovarianz gemessen werden. Die Autokovarianz ist definiert durch

$$\gamma(t_1, t_2) = E[(Y_{t1} - \mu_{t1})(Y_{t2} - \mu_{t2})]; \qquad \gamma(t_1, t_2) \in \mathbb{R} \tag{1}$$

und gibt den Zusammenhang zwischen zwei verschiedenen Werten (Y_{t1}, Y_{t2}) der gleichen Zufallsvariablen an. Ist die Autokovarianz normiert, so spricht man von der Autokorrelation, definiert durch

$$\rho\left(t_1, t_2\right) = \frac{\gamma\left(t_1, t_2\right)}{\sigma_{t1}\sigma_{t2}} \qquad \text{mit} -1 \leq \rho(t_1, t_2) \leq +1 \tag{2}$$

wobei ein Wert von $\rho\left(t_1, t_2\right) = 1$ perfekte Korrelation und ein Wert von $\rho\left(t_1, t_2\right) = -1$ perfekte Anti-Korrelation beschreibt. Dieser Ansatz der Autokorrelation kann bei Zeitreihen genutzt werden, um Periodizitäten zu finden. Die Periodizität gilt dabei als eine der Komponenten, aus denen eine Zeitreihe modelliert wird. Würde beispielsweise der besprochene Maximalpunkt am Vormittag plötzlich wegfallen, so deutet dies auf ein anormales Verhalten hin. Ebenso könnte in unserem Beispiel auf ein anormales Verhalten interpretiert werden, falls Nachts plötzlich der Datendurchsatz ansteigen würde.
Da das Modell der Zeitreihe für gewöhnlich in Komponenten zerlegt wird (Abbildung 15 [S. 31]), gibt es neben der Periodizität beziehungsweise Saisonalität S, die kurzfristige Schwankungen beschreibt, den Trend T, der eine allgemeine Grundrichtung der Zeitreihe angibt, die zyklische Komponente Z für langfristige Schwankungen, sowie die Restkomponente beziehungsweise irreguläre Komponente I. Zu letzteren gehören Strukturbrüche und Ausreißer, welche die für uns interessanten Anomalien beinhalten können. Meist wird auch die Trend-Komponente mit der zyklischen Komponente zur glatten Komponente zusammengefasst. Oft wird die Bezeichnung Trend beibehalten. Die einzelnen Komponenten sind nicht direkt beobachtbar. Das Modell einer Zeitreihe ist wie folgt definiert:

$$Y = f\left(T, Z, S, I\right) \tag{3}$$

Die Verknüpfung der Komponenten kann dabei *additiv*

$$Y = T + Z + S + I \tag{4}$$

oder *multiplikativ*

$$Y = T * Z * S * I \tag{5}$$

oder additiv *und* multiplikativ sein. Methoden zur Bestimmung der Komponenten sind jeweils von der Art der Verknüpfung abhängig.

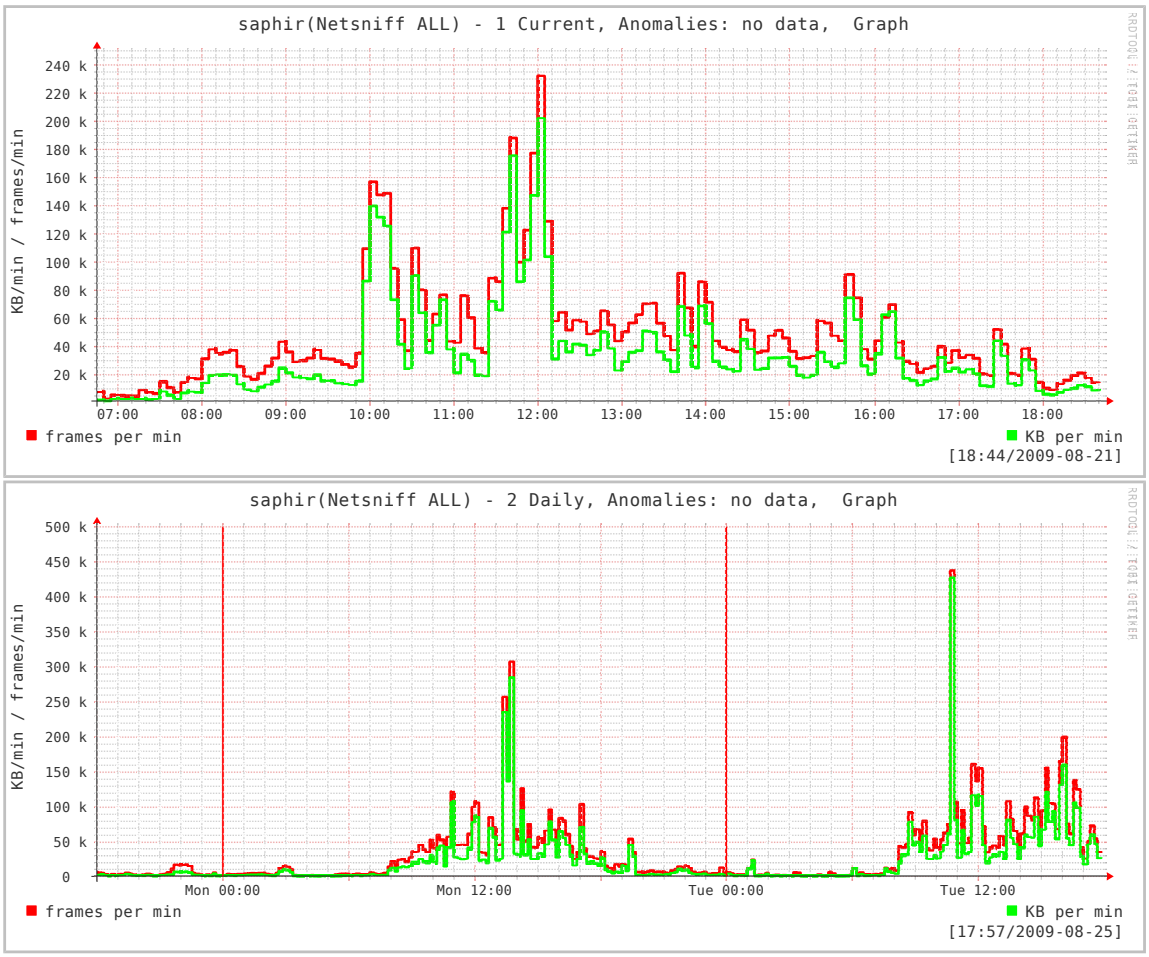


Abbildung 14: Netzwerktraffic am Institutsrouter, Ein- und Zwei-Tagesansicht: *Erfasst wurde diese Statistik mit Hilfe des im Abschnitt 4.3 [S. 69] entwickelten Netzwerksniffers „netsniff-ng“, welcher ins Nagios-System als Plugin integriert wurde.*

Zeitreihen können stark stationär, schwach stationär oder nicht-stationär sein. Dies bedeutet, dass beim schwach stationären Prozess Erwartungswert, Varianz und die Struktur der Autokorrelation invariant für alle $t \in \mathbb{T}$ sind [NIST 06]. Starke Stationarität würde bedeuten, dass die Verteilung $(Y_{s+t})_{t \in \mathbb{T}}$ nicht vom Index $s \in \mathbb{T}$ abhängig ist. Die über Nagios beobachteten Prozesse werden in unserem Fall als nicht-stationär angenommen.

Im weiteren Verlauf sollen die uns vorliegenden Zeitreihen als endliche Realisation eines stochastischen Prozesses eines bestimmten Typs interpretiert werden. Zufallsschwankungen werden hierbei nicht als strukturneutral betrachtet, sondern spielen

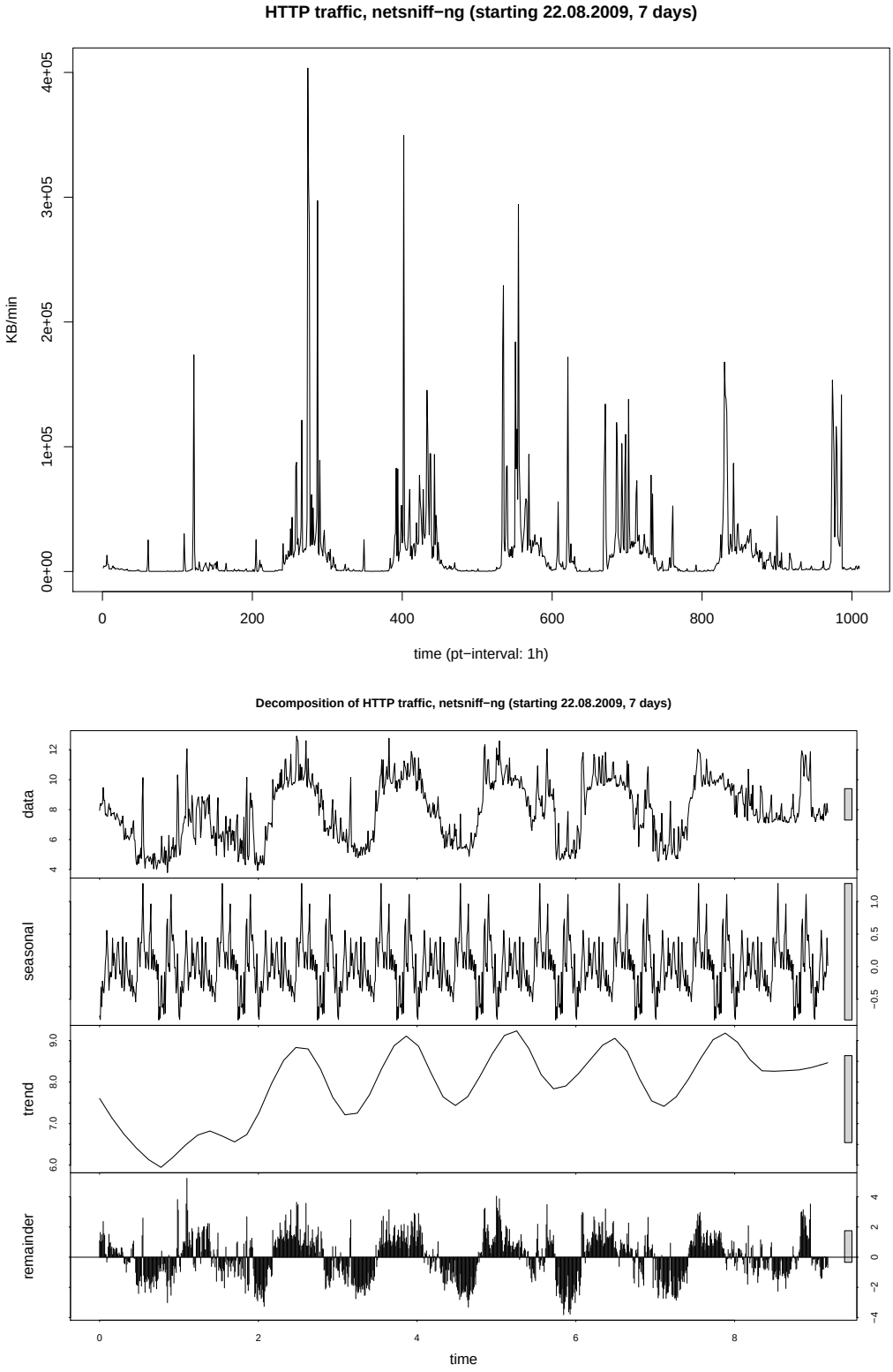


Abbildung 15: Dekomposition der Zeitserie vom HTTP-Datendurchsatz am MPI-Router mittels GNU R: *Die über sieben Tage (beginnend mit dem Übergang von Samstag zum Sonntag) aufgenommene Zeitserie vom HTTP-Datendurchsatz über das entwickelte Plugin netsniff-ng wird über ein GNU R Programm in seine einzelnen Teilkomponenten (Trend, Saisonalität, Rest) des Modells der Zeitreihe zerlegt. Zu erkennen ist in diesem Beispiel weiterhin, dass die Messdaten nicht immer angemessen zerlegt werden können.*

eine wichtige Rolle bei der Komponentenmodellierung.
Der Begriff des stochastischen Prozesses soll in diesem Zusammenhang etwas näher erläutert werden. Einführend wird dies am Beispiel eines der einfachsten stochastischen Prozesse, dem *Simple Random Walk*, geschehen.

Beim Simple Random Walk (Abbildung 16 [S. 33]), wie in [Cox 65] beschrieben, gibt die Zufallsvariable X_n die Position eines sich bewegenden Teilchens zum Zeitpunkt $n \in \mathbb{N}_{\geq 0}$ an. Als Initialwert X_0 gilt der Koordinatenursprung. Im Zeitpunkt $n = 1$ gibt es entweder einen Sprung um $+1$ nach oben mit der Wahrscheinlichkeit $\frac{1}{2}$ oder aber einen Sprung um -1 nach unten mit der gleichen Wahrscheinlichkeit von $\frac{1}{2}$. Im Zeitpunkt $n = 2$ gibt es einen weiteren Sprung entsprechend nach unten oder oben bei gleichen Wahrscheinlichkeiten. Die Art des Sprunges („unten“, „oben“) ist unabhängig voneinander in allen Zeitpunkten n. Allgemein wird der Simple Random Walk wie folgt beschrieben

$$X_n = X_{n-1} + Z_n \tag{6}$$

wobei Z_n der Sprung zum Zeitpunkt n ist, sodass die Zufallsvariablen $\{Z_1, Z_2, \ldots\}$ unabhängig voneinander sind und alle die Verteilung

$$P(Z_n = 1) = P(Z_n = -1) = \frac{1}{2}; \qquad n \in \mathbb{N}_{>0} \tag{7}$$

besitzen. Bei Initialwert $X_0 = 0$ gilt

$$X_n = Z_1 + Z_2 + \ldots + Z_n. \tag{8}$$

Die Verteilung für gegebene n kann auch als Binomialverteilung betrachtet werden, bei der für n Versuche j Sprünge nach „unten“ und $n-j$ Sprünge nach „oben“ gemacht werden, sodass die Wahrscheinlichkeit ausgerechnet werden kann, dass genau zum Zeitpunkt n das Teilchen sich an Stelle X_n befindet

$$P(X_n = n - 2j) = \binom{n}{j} 2^{-n} \tag{9}$$

Für gewöhnlich wird eine solche Stelle als Zustand bezeichnet. Die Gesamtheit aller Zustände ist dabei der Zustandsraum. Weiter verallgemeinert kann beispielsweise

$$P(Z_n = 1) = p \qquad P(Z_n = -1) = q \qquad P(Z_n = 0) = 1 - q - p; \qquad n \in \mathbb{N}_{>0} \quad (10)$$

festgehalten werden, welches auch als „eigentlicher“ Simple Random Walk bezeichnet wird. Das gezeigte Beispiel, bei welchen lediglich Sprünge nach „unten“ und „oben“ möglich sind, ist ein vereinfachter Spezialfall des Simple Random Walks. Als *Random Walk* wiederum bezeichnet man einen stochastischen Prozess der Form (6 [S. 32]) mit Z_n, welches weitere mögliche, voneinander unabhängige, gleichverteilte Werte als nur $\{-1; 0; 1\}$ besitzt. Beispiele wie in Abbildung 16 [S. 33] werden auch als Realisation beziehungsweise Sample Path bezeichnet. Die einzelnen Realisationen sind jeweils unabhängig voneinander.

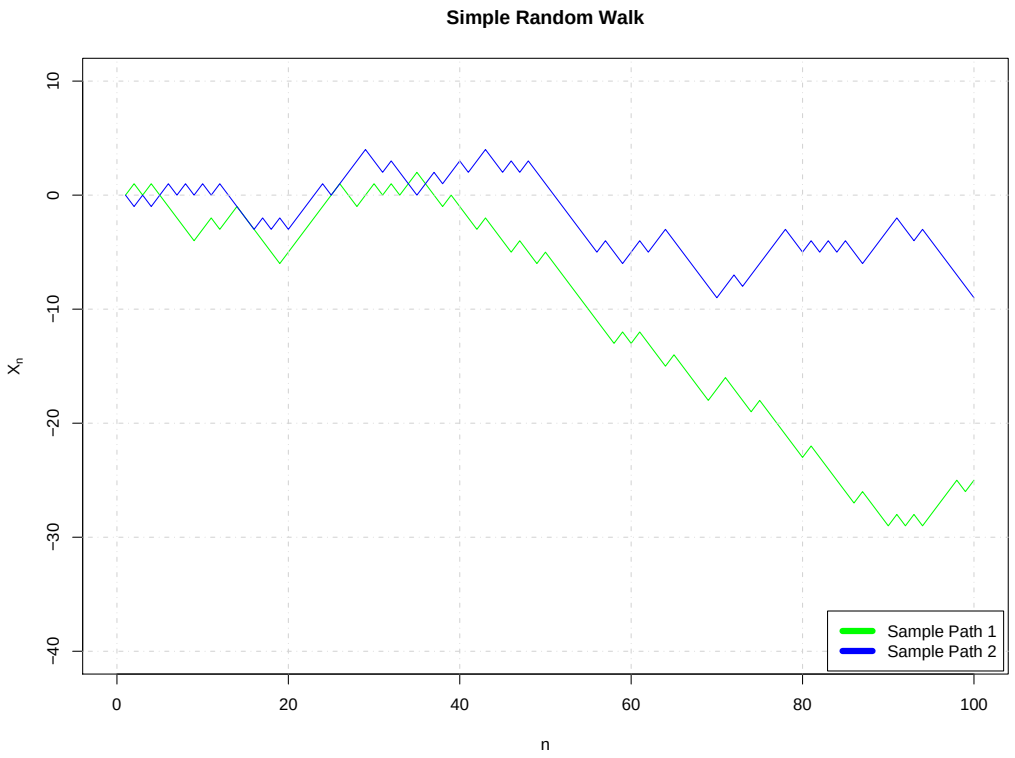


Abbildung 16: Zwei unterschiedliche, unabhängige Realisationen des Simple Random Walk (*GNU R*): *Der Simple Random Walk ist eines der einfachsten stochastischen Prozesse.*

Nachfolgend soll nun der stochastische Prozesses im Allgemeinen beschrieben werden. Zunächst muss dabei die Menge $\mathbb{I}$ der gegebenen Zeitpunkte erwähnt werden, in denen der Prozess definiert ist. Dies können diskrete ($n = 0, 1, 2, \ldots$ bzw $n =$

$0, -1, -2, \ldots; n \in \mathbb{Z}$) oder aber kontinuierliche ($0 \leq n$ oder $-\infty < n < \infty; n \in \mathbb{R}$) Zeitpunkte sein. Für gewöhnlich existiert beim stochastischen Prozess eine Menge von Zufallsvariablen $\{X_n\}$ für jeden beliebigen Zeitpunkt n, sodass $X_n : n \in \mathbb{I}$ bei gegebenen Wahrscheinlichkeitsraum $(\Omega, \mathcal{F}, P)$ definiert ist. Der Zustandsraum des Prozesses wiederum ist selbst eine Menge von Zufallsvariablen X_n und kann ein- oder mehrdimensional sein (für einfache Prozesse eindimensional). Der Zustandsraum ist diskret, falls die Punktmenge endlich oder abzählbar unendlich ist, beziehungsweise kontinuierlich, falls überabzählbar unendlich.
Folgende Möglichkeiten existieren beispielsweise für einen eindimensionalen Prozess: diskrete Zeit & diskreter Zustandsraum, diskrete Zeit & kontinuierlicher Zustandsraum, kontinuierliche Zeit & diskreter Zustandsraum, sowie kontinuierliche Zeit & kontinuierlicher Zustandsraum.
Des weiteren wird $n \mapsto X_n(\omega)$ als Realisation oder Sample Path des stochastischen Prozesses bezeichnet. Ein bestimmter stochastischer Prozesses kann beliebig viele Realisationen haben.

Ein spezieller stochastischer Prozess, der im weiterem Verlauf eine wichtige Rolle spielen wird, ist der *Poisson Prozess*, welcher in vielen Anwendungen zum Tragen kommt. Das wohl bekannteste Beispiel sind die zufällig eintreffende Anzahl von Telefonanrufen pro Zeiteinheit in einem Callcenter [Tayl 07]. Weiterhin, etwas mehr aus Sicht der Informatik, zu nennen sind auch die Zeitpunkte, in denen Anforderungen (wie Tasks, Client-Requests oder Hardware-Interrupts) bei einem Bediener (wie Prozess-Scheduler, Server oder CPU/Interrupt-Controller) eingehen, oder, wie in dieser Arbeit näher beobachtet, Netzwerkpakete, die eine bestimmte Stelle innerhalb des Netzes wie beispielsweise Router passieren.
Vom Massachusetts Institute of Technology wurde sogar ein Versuch unternommen, die Abfolge von Selbstmorden am MIT als Poisson-Prozess zu modellieren [Chew 00].
Allgemein betrachtet sollen Häufigkeiten *seltener* Ereignisse bestimmt werden. Meist besitzen die mit einem Poisson-Prozess beschriebenen seltenen Ereignisse in der Praxis ein hohes Risiko. Dabei wird der Poisson Prozess in die Klasse der kontinuierlichen Zeit und eines diskreten Zustandsraums eingeordnet.

Nachfolgend wird der Poisson Prozess anhand [Cox 65] näher erklärt. Sei eine Konstante ρ ($\rho > 0$) als eine Frequenz definiert, welche als *Eintrittshäufigkeit* (im Sinne „Ereignis tritt ein“) bezeichnet wird, sowie $N(t, t + \Delta t)$ als Anzahl der Ereignisse im Zeit-Intervall $(t, t + \Delta t]$, mit $\Delta t \to 0_+$, so wird angenommen, dass

$$P\{N(t, t + \Delta t) = 0\} = 1 - \rho \Delta t + o(\Delta t) \tag{11}$$

und

$$P\{N(t, t+\Delta t) = 1\} = \rho\Delta t + o(\Delta t) \tag{12}$$

entsprechend

$$P\{N(t, t+\Delta t) > 1\} = o(\Delta t) \tag{13}$$

ergibt. Die Funktion $o(\Delta t)$ strebt dabei schneller nach 0 als Δt. Desweiteren soll angenommen werden, dass $N(t, t+\Delta t)$ unabhängig von Ereignissen aus $(0, t]$ ist. Ein solcher stochastischer Prozess von diskreten Zuständen mit aus Gleichung (11 [S. 34]) und (13 [S. 35]) erfüllten Bedingungen wird *Poisson Prozess* der Häufigkeit ρ genannt.
Letztere Bedingung bedeutet, dass Ereignisse zufällig eintreten. Bedingung aus Formel (13 [S. 35]) besagt, dass Ereignisse einzeln auftreten und die Eigenschaft der Konstanz in ρ aus Gleichung (12 [S. 35]) bedeutet, dass es weder zeitliche Trends oder andere systematische Änderungen bei der Eintrittshäufigkeit gibt.
Die Zufallsvariable X_t ist dabei wie folgt *poissonverteilt*,

$$P\{N(t) = i\} \equiv p_i(t) = \mathrm{e}^{-\rho t}\frac{(\rho t)^i}{i!}; \qquad i \in \mathbb{N}_{\geq 0} \tag{14}$$

wobei Erwartungswert und Varianz jeweils ρt entsprechen.
Wie anfangs erwähnt, sind die in RRDs eintreffende Daten zeitdiskret. Zwar ist die Definition zwischen zeitdiskret und zeitkontinuierlich strikt mathematisch getrennt, jedoch darf für praktische Anwendungen des Poisson Prozesses eine zeitdiskrete Approximierung für ein zeitkontinuierliches Phänomen (und umgekehrt) verwendet werden [Cox 65]. Im weiteren Verlauf wird angenommen, dass über Nagios eintreffende Daten eines stochastischen Prozesses vom Typ Poisson Prozess sind. Die als *selten* eintretenden Ereignisse seien als Verhaltensanomalien gegenüber dem approximierten Normalverhalten (im weiteren Verlauf auch als *Baseline* bezeichnet) des beobachteten Systems definiert.

3.2 Erkennung von Anomalien

Auf die über Nagios beobachteten Poisson Prozesse sollen nun über verschiedene Ansätze Algorithmen angewendet werden, die mögliche Verhaltensanomalien anzeigen. Von den vielen möglichen Verfahren wurden zwei verschiedene Ansätze näher analysiert und in der Praxis getestet. Da Verhaltensanomalien möglichst früh erkannt

werden sollen, sind vor allem aus Sicht der RRD Archive mit zeitlich hoher Auflösung von Interesse. Dies betrifft vor allem Archive, welche die Datenhaltung über Stunden, Tage oder Wochen übernehmen.

3.2.1 Exponentielle Glättung

Das Forschungsgebiet der *exponentielle Glättung* findet vor allem zur Erstellung kurzfristiger Prognosen für Zeitreihen Anwendung. Der Vorteil in diesem Verfahren liegt darin, dass Zeitreihenwerte, die dem Prognosezeitpunkt am nächsten liegen, mit höherem Gewicht in die Berechnung eingehen, als weiter zurückliegende. Damit wird der Nachteil aufgehoben, dass Zeitreihenwerte wie im Falle eines einfachen *gleitenden Durchschnitts* mit gleichem Gewicht in die Berechnung eingehen. In der Praxis ist es jedoch erwünscht, jüngeren Entwicklungen mehr Gewicht zu geben, als länger vergangenen Werten [Scha 04].
Bei der exponentiellen Glättung liegt somit ein gewichtetes Mittel über alle Zeitpunkte der Zeitreihe vor, wobei das Gewicht der älteren Werte mit der Zeit exponentiell kleiner wird. Genutzt wird dieses Verfahren vor allem, um die glatte Komponente der Zeitreihe nachzubilden. Der jeweilige Schätzwert $\hat{y}_t$ baut sich rekursiv auf, sodass der nächste Schätzwert sich aus dem aktuellem Schätzwert und dem aktuell beobachtetem „echten“ Wert zusammensetzt.

$$\hat{y}_{t+1} = \alpha y_t + (1 - \alpha)\hat{y}_t \tag{15}$$

Eine Auflösung der Rekursivität ergibt:

$$\hat{y}_{t+1} = \sum_{i=0}^{t} \varsigma(i)(1 - \alpha)^{t-i} y_i \tag{16}$$

$$\text{mit} \qquad \varsigma(i) = \begin{cases} \alpha & \text{falls } i > 0, \\ 1 & \text{sonst.} \end{cases}$$

Der Modell-Parameter α liegt zwischen $0 < \alpha < 1$ und gibt dabei den Grad des „Verfalls“ älterer Messwerte mit $(1 - \alpha)$ an, sowie das Gewicht des aktuell beobachteten Wertes mit α. Dem Prinzip der einfachen exponentiellen Glättung werden allerdings lediglich stationäre Daten mit fehlendem Trend und fehlender Saisonalität gerecht [Loom 04b], jedoch können darauf aufbauend komplexere Modelle wie dem des *Holt-Winters Forecastings* erstellt werden (exponentielle Glättung höherer Ord-

nungen).

Von Jake D. Brutlag wurde zuletzt erwähntes Modell innerhalb der RRD-Architektur und des Monitoring-Systems *Cricket* [Alle 00] mit dem Ziel implementiert, anormales Verhalten der Zeitserien zu erkennen. Die ihm vorliegenden Zeitreihen beziehen sich vor allem auf Datendurchsatzraten des Netzwerkes auf Internet-Service-Provider-Ebene [Brut 00]. Dabei sollten die Messdaten *automatisch* Verhalten jenseits der Baseline erkennen.
Diese RRD-Erweiterung wird jedoch nicht von NagiosGrapher unterstützt. Für die in der Arbeit entwickelte Schnittstelle zu NagiosGrapher sollen daher über den bereits implementierten Algorithmus hinaus weitere Verbesserungen, beschrieben in wissenschaftlichen Veröffentlichungen, vorgestellt und getestet werden.
Im weiteren Verlauf wird anhand [Brut 00] der Holt-Winters-Algorithmus, sowie im Anschluss weitere Modell-Änderungen für robustere Vorhersagen anhand [Gelp 07] und [Mill 07] näher erklärt.

Brutlags entwickeltes Modell untergliedert die Anomalieerkennung in drei Bestandteile:

- Ein Algorithmus, der anhand der gegebenen Zeitreihe den als nächsten eintreffenden Wert zum Zeitpunkt $t + 1$ vorhersagt
- Ein Maß zur Bestimmung der Abweichung des beobachteten und vorhergesagten Wertes
- Ein Algorithmus, der entscheidet, ab wann beobachteter und vorhergesagter Wert zu weit voneinander divergieren

Dabei erweitern die beiden letzten Punkte das eigentliche *Holt-Winters Forecasting*. Die Wahl für Holt-Winters begründet Brutlag unter anderem mit folgendem Zitat des Wissenschaftlers Richard Lawton:

> „The Holt-Winters method is one of the best known forecasting techniques wich allows the seasonal pattern to adapt over time... When compared with other methods the technique has been found to perform relativley well and it has the merit of being understood by users who lack a statistical background without sacrificing the ability to adapt to changing patterns in data.“ [Lawt 98]

Des Weiteren wurde in einer wissenschaftlichen Veröffentlichung [Tayl 06] für höherfrequente Zeitreihen (Saisonalität innerhalb eines Tages und zusätzlich innerhalb der

Wochen) das *Holt-Winters Forecasting* mit doppelter, multiplikativer Saisonalität erweitert und anderen Verfahren wie beispielsweise *künstlich neuronalen Netzen* oder *ARIMA*-Modellen zur kurzfristigen Zeitreihenvorhersage gegenübergestellt. Das Ergebnis der Studie fiel unter anderem zu Gunsten der exponentiellen Glättung aus:

> „A highlight of this study was the success of the exponential smoothing method. In addition to its forecasting performance, it is important to note that, of the four sophisticated methods that we considered, this method is comfortably the simplest and quickest to implement. On the disappointing side, the neural network did not perform well.“ [Tayl 06]

Holt-Winters Forecasting ist eine *dreifache exponentielle Glättung* und besteht daher auch aus drei Modell-Komponenten. Diese sind definiert durch den linearen Trend („glatte Komponente“), der Saisonalität und der irregulären Komponente. Würde die Saisonalität in diesem Modell fehlen, so entspräche dies einer *zweifachen exponentielle Glättung*, welche im Falle Holt-Winters bei Zeitreihen ohne erkennbarer Periode angewendet werden kann (Abbildung 15 [S. 31]).

Die Vorhersage $\hat{y}_{t+1}$ ergibt sich beim additiven Modell aus der Summe der drei Komponenten

$$\hat{y}_{t+1} = a_t + b_t + c_{t+1-m} \tag{17}$$

wobei m die Länge einer „Saison“ angibt und a_t, b_t, c_t sich jeweils durch exponentielle Glättung aus

$$\begin{aligned}
a_t &= \alpha(y_t - c_{t-m}) + (1-\alpha)(a_{t-1} + b_{t-1}) && \text{(irreguläre Komponente)}, && (18)\\
b_t &= \beta(a_t - a_{t-1}) + (1-\beta)b_{t-1} && \text{(linearer Trend)}, && (19)\\
c_t &= \gamma(y_t - a_t) + (1-\gamma)c_{t-m} && \text{(Saisonalität)} && (20)
\end{aligned}$$

ergeben. Die Modell-Parameter sind α, β, γ und liegen jeweils im Intervall $\mathbb{I}(0,1)$. Wie auch bei der einfachen exponentiellen Glättung bedeuten größere Modell-Parameter eine schnellere Anpassung an neueste Messwerte und kleinere Parameter eine höhere Gewichtung weiter zurückliegender Werte.

Neben dem *Holt-Winters Forecasting* muss nun noch ein Intervall bestimmt werden, in welches der nächste, eintreffende Messpunkt hineinfallen darf, sodass dieser nicht

als Anomalie zählt. Dieses Intervall wird *Confidence Band* genannt und modelliert saisonale Variabilität. Die Bestimmung des Intervalls geschieht über einen weiteren Koeffizienten d_t, der das gewichtete Mittel der absoluten Abweichung über exponentielle Glättung erfasst:

$$d_t = \gamma|y_t - \hat{y}_t| + (1-\gamma)d_{t-m} \tag{21}$$

d_t ist dabei definiert als vorhergesagte Abweichung zum Zeitpunkt t und nutzt den selben Modell-Parameter wie c_t. Das Intervall für den jeweiligen Zeitpunkt y_t wird nun wie folgt definiert:

$$\mathbb{I}(\hat{y}_t - \delta_- * d_{t-m}, \hat{y}_t + \delta_+ * d_{t-m}) \qquad 2 < \delta_-, \delta_+ < 3 \tag{22}$$

δ_- und δ_+ sind jeweils Skalierungskonstanten für die Intervallbreite. Anomalien könnten nun im einfachsten Fall interpretiert werden, indem der neue, eintreffende Messwert sich außerhalb des erlaubten Intervalls befindet. Dies kann unter Umständen zu einer hohen Rate an Falschvorhersagen (*false positives*) führen. Oft wird daher eine Art *gleitendes Fenster* über eine feste Anzahl an Werten (beispielsweise der letzten zwölf Messwerte) gelegt und überprüft, wie oft die jeweiligen „erlaubten“ Intervalle verfehlt wurden. Wird dabei ein Schwellwert an erlaubten Überschreitungen übertroffen, so wird auf eine Anomalie im Verhalten gedeutet.

Für die Wahl der bereits erwähnten Modell-Parameter α, β, γ gibt es einige Richtlinien, an denen man sich orientieren kann. So ergibt sich für die jeweilige Parameterwahl die Formel

$$\alpha = 1 - exp\Big(\frac{\ln(1-\phi)}{n}\Big) \qquad 0 < \phi < 1, n \in \mathbb{N}_{>0} \tag{23}$$

wobei n die Anzahl der Zeitpunkte und ϕ die summierten Wahrscheinlichkeiten der Gewichte für die n Zeitpunkte (Wahrscheinlichkeit aller Gewichte ist 1) angeben. Diese Formel wurde anhand der Summe der letzten n Gewichte umgeformt:

$$\phi = 1 - (1-\alpha)^n \tag{24}$$

α sollte dabei sehr viel größer als β gewählt werden, um Anpassungen innerhalb kurzer Zeit bei der irregulären Komponente zu erlauben. Im Falle von $\gamma = 0$ ergibt

das Modell eine *zweifache exponentielle Glättung*.

Nachfolgend sei ein Beispiel des *Holt-Winters Forecastings* auf den HTTP-Durchsatz des Instituts über den Zeitraum von einer Woche angewandt (Abbildung 15 [S. 31]). Die über GNU R automatisch geschätzten Modell-Parameter ergaben eine Vorhersage mit linearem Trend. Dies bedeutet, dass innerhalb dieses kurzen Zeitraumes keine Saisonalität festgestellt werden konnte und daher dem Modell nur zwei Komponenten zugeordnet wurden (*Holt-Winters mit zweifacher exponentielle Glättung*).

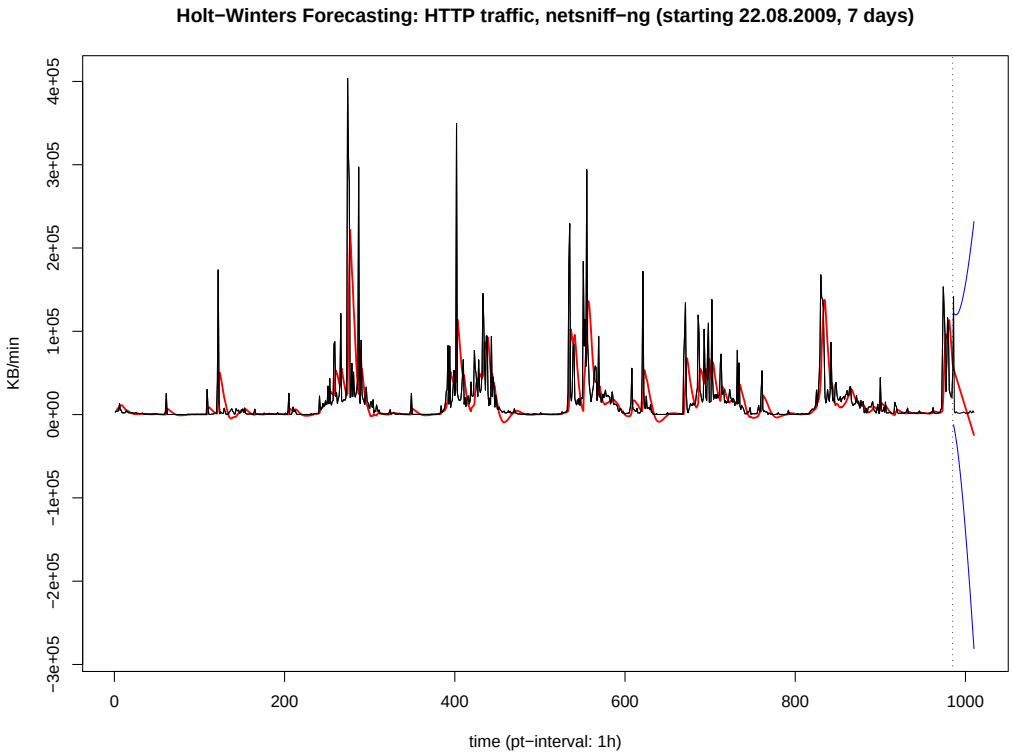


Abbildung 17: Holt-Winters Forecasting vom HTTP-Datendurchsatz am MPI-Router mittels GNU R: *Die über sieben Tage (beginnend mit dem Übergang von Samstag zum Sonntag) aufgenommene Zeitserie vom HTTP-Datendurchsatz über das entwickelte Plugin netsniff-ng wird über ein GNU R Programm mittels Holt-Winters Forecasting (roter Graph, hier: fehlende Saisonkomponente) vorhergesagt. An der gestrichelten Linie wird ein Abwärtstrend vorhergesagt, welcher in den Messwerten ebenfalls zu erkennen ist.*

Im weiteren Verlauf soll neben Brutlags Implementierung nun die durch Miller beschriebenen Verbesserungen [Mill 07] innerhalb der Anomalie-Analyse diskutiert werden.

Da Brutlags Modell hauptsächlich für Netzwerkverkehr mit hohen Datendurchsatzraten gedacht ist, kann es bei zu beobachtenden Prozessen mit geringen Datenraten wie

im Beispiel eintreffender Tickets eines Ticket-Systems oft zu *false positives* kommen. Brutlags Methode würde bei einer (theoretisch) unendlich langen Nullfolge nicht-anormales Verhalten anzeigen, wobei jedoch erwünscht wäre, dass lediglich kurze Zeitintervalle mit Nullfolgen erlaubt und längerfristig, aufeinanderfolgende Nullfolgen eine Anomalie darstellen sollen.
Seine Erweiterung für den Teil der Anomalieanalyse innerhalb Brutlags Modell, welche sowohl für Prozesse mit großen als auch mit geringen Datenraten modelliert ist, soll im Weiteren vorgestellt und diskutiert werden. Eintreffende Daten werden dabei als unabhängig voneinander betrachtet.

Die zwei wichtigen Werte für einen Zeitpunkt stellen die Vorhersage $\hat{y}_t$ und die Vorhersageabweichung d_t dar. Bei letzterer wird die statistische Annahme getroffen, dass das Residuum $(y_t - \hat{y}_t)$ des Messwertes zum vorhergesagten Wert einer Standardabweichung der Normalverteilung (unter Normalverhalten) um den Wert 0 entspricht. Daraus formuliert Miller, mittels Offset k innerhalb einer Saison, über N erfasste Saisons (dies heißt, dass Parameter t in k und N zerlegt wird), die Annäherung für Brutlags d_t-Parameter

$$d_{k,N} \approx \sigma_{k,N} \tag{25}$$

mit

$$\sigma_{k,N} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_{k,i} - \hat{y}_{k,i})^2}. \tag{26}$$

Dies bedeutet, dass der in Brutlag beschriebene Koeffizient d_t, der zur Intervallbestimmung vom *Confidence Band* dient, mit der Standardabweichung der Normalverteilung abgeschätzt wird.

Das als Poisson-Prozess betrachtete Modell besitzt einen unbestimmten Poisson-Parameter λ ($\lambda = \rho t$, siehe Poisson-Prozess). Dieser Parameter λ ist bei der Poisson-Verteilung gleich der Varianz σ^2 und da Varianz (bei der Poisson-Verteilung) dem Erwartungswert gleicht, kann diese durch $\hat{y}$ angenähert werden, da $\hat{y}$ dem erwarteten, vorhergesagten Wert entspricht.

$$\lambda = \sigma^2 \approx \hat{y} \tag{27}$$

$$d_{k,N} \approx \sigma_{k,N} \approx \sqrt{\hat{y}} \tag{28}$$

Zur Anomalieermittlung verwendet Miller eine sogenannte *health*, die als „Gesundheitszustand“ vom zu untersuchenden Graphen interpretiert werden soll. Das Modell betrachtet dabei die Vorhersage als Menge von möglichen Vorhersagen, die durch die Standardabweichung beschrieben ist. Untersucht wird die Wahrscheinlichkeit p, die besagt, dass ein Messwert $\leq y$ eintrifft:

$$p_{\hat{y},d^2}(y) = \int_{-\infty}^{\infty} f(y;\lambda)\varphi_{\hat{y},d^2}(\lambda)d\lambda \tag{29}$$

Die Funktion $f(y;\lambda)$ ist die kumulierte Wahrscheinlichkeit der Poisson-Verteilung (Beispiel siehe Abbildung 18 [S. 43]), welche besagt, zu welcher Wahrscheinlichkeit der Wert $\leq y$ eintritt bei gegebenem Erwartungswert beziehungsweise Varianz.
$\varphi_{\hat{y},d^2}(\lambda)$ ergibt die Wahrscheinlichkeitsdichtefunktion einer Normalverteilung. Diese wird dazu verwendet, die Dichte der einzelnen Vorhersageabweichungen (Residuen $y_t - \hat{y}_t$), die als normalverteilt angenommen werden, zu beschreiben.

Die einzelnen Funktionen ergeben sich aus:

$$f(y;\lambda) = \sum_{i=0}^{y} \frac{e^{-\lambda}\lambda^i}{i!} \tag{30}$$

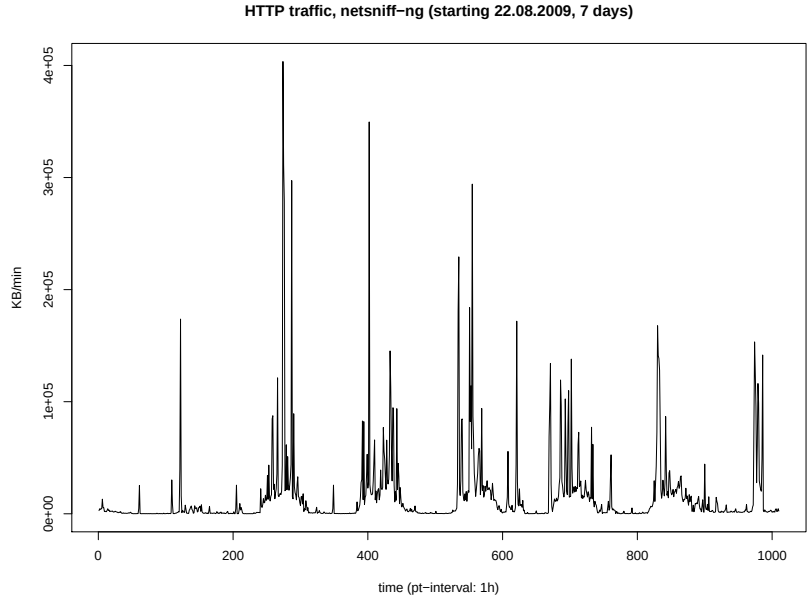
$$\varphi_{\hat{y},d^2}(\lambda) = \frac{1}{d\sqrt{2\pi}} exp\Big(-\frac{(y-\lambda)^2}{2d^2}\Big) \tag{31}$$

Normalisiert und mit unterer Integralgrenze $\lambda = 0$ (da $\lambda \in (0,\infty)$ bei Poisson-Verteilung) ergibt dies nach Miller

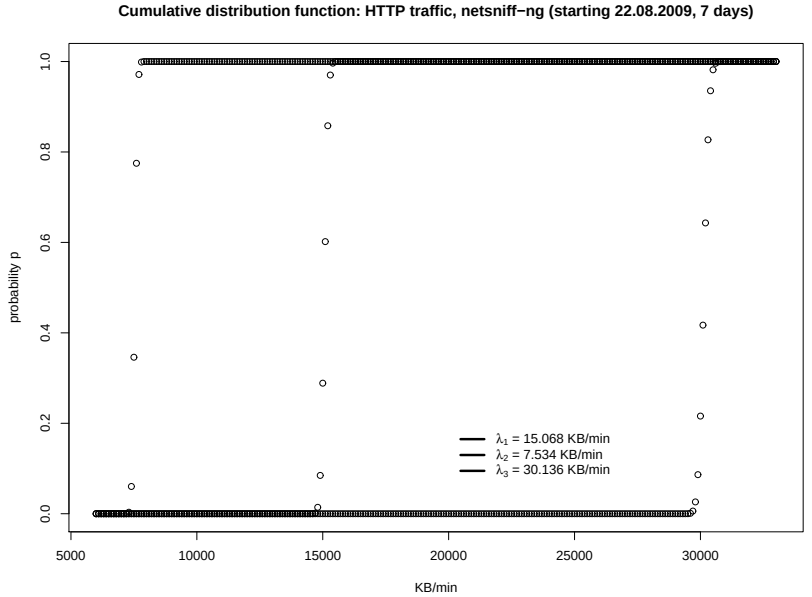
$$p_{\hat{y},d^2}(y) = \frac{\int_0^{\infty} f(y,\lambda)\varphi_{\hat{y},d^2}(\lambda)d\lambda}{\int_0^{\infty} \varphi_{\hat{y},d^2}(\lambda)d\lambda} \tag{32}$$

Der schon erwähnte *health*-Wert ist dabei die kleinste Wahrscheinlichkeit p aus den rückblickend berechneten p's. Wie weit zurückgeblickt werden soll, gibt Parameter T an. Dies bedeutet, dass bei schrittweiser Vergrößerung der Intervalle p neu berechnet wird. Das kleinste p ist der gesuchte *health*-Wert. Für Offset k innerhalb Saison i definiert sich dieser Wert aus

$$h_{k,i}(T) = \min_{t\in(1,\dots,T)} \left(p_{\sum_{j=1}^{t}(\hat{y}_{k-j,i}),\sum_{j=1}^{t}(d^2_{k-j,i})}(\sum_{j=1}^{T} y_{k-j,i}) \right) \tag{33}$$



(a) HTTP-Datendurchsatz am MPI-Router



(b) Kumulative Poisson-Verteilungsfunktion vom HTTP-Datendurchsatz am MPI-Router

Abbildung 18: Kumulative Verteilungsfunktion vom HTTP-Datendurchsatz am MPI-Router mittels GNU R: *Die über sieben Tage (beginnend mit dem Übergang von Samstag zum Sonntag) aufgenommene Zeitserie vom HTTP-Datendurchsatz über das entwickelte Plugin netsniff-ng wird über ein GNU R Programm auf die kumulative Poisson-Verteilungsfunktion untersucht. Als Erwartungswert wurde einfach der Durchschnitt über den oben gezeigten Datendurchsatz der gesamten Zeitreihe von einer Woche genommen (*$15.068 KB/min$ *an HTTP Verkehr). Nun soll über die kumulative Poisson-Wahrscheinlichkeitsfunktion gezeigt werden, zu welcher Wahrscheinlichkeitsverteilung die Zufallsvariable X (definiert durch HTTP-Datendurchsatz in KB/min am Institutsrouter) mit Werten von* $0 KB/min$ *bis hin zum (theoretisch) maximal möglichen Durchsatz von* $768.000 KB/min$ *auftritt, bei unterschiedlichen* λ*:* $\lambda_1 = 15.068 KB/min, \lambda_2 = 0.5\lambda_1, \lambda_3 = 2.0\lambda_1$.
Beispielsweise besagt die stückweise definierte Verteilungsfunktion mit λ_1 *für den Fall das maximal* $16.000 KB/min$ *an Datendurchsatz eintreten eine Wahrscheinlichkeit von nahezu 1.0*

Als Beispiel sollen zwei Zeitintervalle mit jeweils $d = 0, y = 0, \hat{y} = \lambda$ betrachtet werden. Bei Untersuchung beider Intervalle ergibt sich ein kleinerer *health*-Wert als bei Untersuchung von lediglich einem Intervall:

$$\begin{aligned} h_k(1) &= p_{\hat{y},d^2}(y) = p_{\lambda,0}(0) = f(0,\lambda) = e^{-\lambda} \\ h_k(2) &= \min(p_{\hat{y},d^2}(y), p_{2\hat{y},(2d)^2}(2y)) = \min(p_{\lambda,0}(0), p_{2\lambda,0}(0)) \\ &= \min(f(0;\lambda), f(0;2\lambda)) = \min(e^{-\lambda}, e^{-2\lambda}) \\ &= e^{-2\lambda} < h_k(1) \end{aligned}$$

Durch Setzen eines Schwellwertes S mit $h_k(1) > S > h_k(2)$ würde bei Betrachten von nur einem Intervall keine Anomalie eintreten, jedoch bei Untersuchung beider Intervalle schon. In der Praxis wird empfohlen, die Schwellwerte für Stufe „Warnung“ auf 10^{-4} und für Stufe „kritisch“ auf 10^{-5} zu setzen.
Die zu errechnenden Integrale werden in der Implementierung durch *Riemann-Integrale* approximiert.

Von der *Katholischen Universität Löwen* stammt eine weitere wissenschaftliche Veröffentlichung zum *Holt-Winters Forecasting* [Gelp 07], welche beschreibt, wie der Holt-Winters Algorithmus robuster gegenüber *false positives* gemacht werden kann. Ausblickend, könnte diese Modell-Erweiterung mit in Brutlags Modell integriert werden, um bessere Vorhersageergebnisse erzielen zu können.

Die Modellparameter α, β, γ für die Glättung, sowie die Gleichungen werden robuster modelliert. Geglättete Werte werden beim *Holt-Winters Forecasting* dahingehend beeinträchtigt, dass diese von aktuell eintreffenden und vergangenen Werten, inklusive deren Anomalien (vor allem bei der saisonalen Komponente), abhängen. Die Modellparameter wiederum regulieren den Grad der Glättung und beeinflussen somit den Einfluss der Ausreißer.
Gelper et al. nutzen eine spezielle Version des *Kalman-Filters* [Gelp 07], welcher in der Signalverarbeitung häufig dafür eingesetzt wird, Störungen zu entfernen. Beim *Holt-Winters Forecasting* wird dadurch die Glättung stabilisiert. Die Verwendung eines solchen Filters bringt weitere Richtlinien, wie in [Gelp 07] beschrieben, zur Parameterwahl mit sich. Abbildung 19 [S. 45] zeigt, wie stabil diese Variante im Gegensatz zum klassischen Holt-Winters ist.

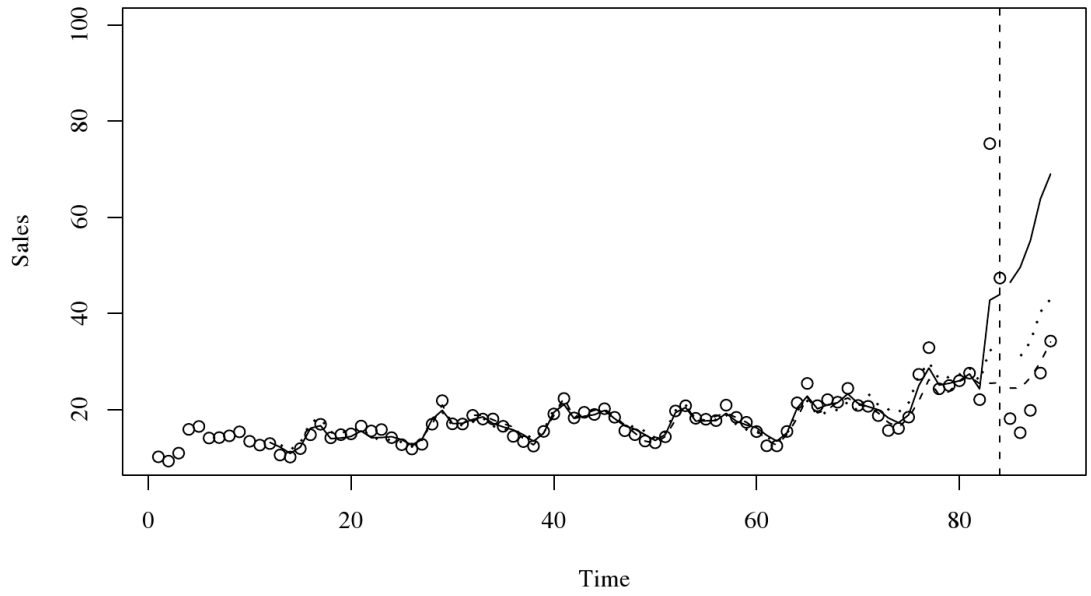


Abbildung 19: Holt-Winters Forecasting mit Kalman-Filter: *Die durchgehende Linie zeigt den durch Holt-Winters geglätteten Verlauf im klassischen Sinne und die sich darunter befindliche gestrichelte Linie das Holt-Winters-Modell mit Kalman-Filter. Zu sehen ist ein deutlich stabileres Verhalten gegenüber Ausreißern. [Gelp 07]*

3.2.2 Clusteranalyse zur Intervallschätzung

Eine völlig andere, einfachere Variante zur Untersuchung der Zeitreihen auf Anomalien ist die *Clusteranalyse zur Intervallschätzung*. Hierbei geht es nicht um ein gewichtetes Mittel, sondern die im Rahmen dieser Arbeit entwickelte Idee basiert auf einer Gruppierung von nah aneinander liegenden Werten zu schlussendlich möglichen Intervallgrenzen in die der nächste eintreffende Messwert hineinfallen darf (Abbildung 20 [S. 45]).

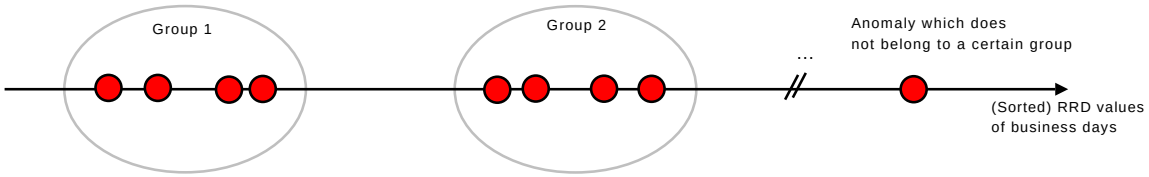


Abbildung 20: Prinzip Clusteranalyse zur Intervallschätzung: *Idee: RRD-Werte (beispielsweise aller Montage bis Freitage) werden eindimensional auf einer Achse sortiert abgetragen und zu Gruppen zusammengefasst. Da die meisten Werte sich innerhalb dieser Gruppen befinden, wird davon ausgegangen, dass neue eintreffende Messwerte im Normalfall ebenfalls hier hineinfallen. Dies bedeutet, dass die gefundenen Gruppen mögliche Intervalle für ein Normalverhalten darstellen.*

Zusammengefasst werden die Differenzen des jeweils nächsten Nachbarn gebildet und sukzessive von der kleinsten Differenz ausgegangen, mögliche obere und untere Intervallgrenze der Gruppe gebildet. Ob ein Nachbarelement zur Gruppe gehört, wird anhand der Standardabweichung der Poisson-Verteilung als Schwellwert überprüft. Eine zusammengehörige Gruppe (von Messpunkten) wird als solche akzeptiert, falls diese mindestens ein Viertel aller vorhandenen Messdaten beinhaltet und somit eine Art „Zentrum“ der Punkte bildet, die bei Normalverhalten dort „hineinfallen“. Ausreißer zählen somit nicht zu den jeweiligen Gruppen.
Bei zu großer Streuung des Wertebereichs kann es durchaus auch vorkommen, dass gar keine Gruppierungen von Punkten entstehen können.
Das größte, falls vorhandene, Intervall wird ermittelt. Bei mehreren Intervallen mit einer gleichen Anzahl an Punkten wird die Dichte berechnet und jenes Intervall mit größerer Dichte als „Bestes“ angesehen.
Im Anschluss wird, ähnlich wie bei Miller [Mill 07], ein *health*-Wert ermittelt, der jedoch auf andere Weise berechnet wird und einen Wertebereich von $\mathbb{I}[0.0, 1.0]$ besitzt. Im Falle, dass es zu keiner Gruppenbildung kam, wird der Median berechnet und die prozentuale Abweichung des Wertes zum Median vom *health* abgezogen.
Weiterhin werden sukzessive sogenannte *penalty* Werte (als „Strafpunkte“ zu interpretieren) vom *health* abgezogen, wenn der Wert außerhalb der Intervalle liegt (*Intervall-Probing*).

Als Vorverarbeitungsschritt werden hierbei aus dem Wochen-RRA alle Werte entnommen, sowie Werktage und Feiertage entsprechend voneinander getrennt. Die Ursache für diese Trennung liegt in der generischen Annahme, dass bei den zu beobachteten Prozessen ein unterschiedliches Verhalten von Werktagen gegenüber Feiertagen vorliegt. Eine genauere Zeitauflösung wird anschließend durch eine aufsteigende Sortierung der beiden Werte-Listen verworfen. Letztendlich bedeutet dies, dass Informationen zur Saisonalität der Zeitreihe verloren gehen. Innerhalb der Liste existiert lediglich nur noch eine Dimension.

Die Standardabweichung für die Poisson-Verteilung für die letzten T Werte errechnet sich durch:

$$\hat{\sigma} = \sqrt{\frac{1}{T} \sum_{i=1}^{T} y_i} \tag{34}$$

Die Differenz zum nächsten Nachbar wird berechnet und die kleinste Differenz festgehalten:

$$z_i = \min_{t \in (1,\ldots,T-1)} |y_t - y_{t+1}| \tag{35}$$

Die Intervallgrenzen für das Intervall $\mathbb{I}_i$ wird durch eine Suche über die jeweils beiden Nachbarn ausgedehnt. Das Intervall kann in die entsprechende Richtung erweitert werden, falls $|y_t - y_{t+1}| < \nu * \hat{\sigma}, \quad \nu = const., \in (2.0; 3.0)$. Wenn das Intervall mindestens ein Viertel der Punkte aus den Listen beinhaltet, so werden diese Punkte durch das gefundene (und gespeicherte) Intervall gruppiert.

$$\mathbb{I}_i(y_{t_i} - \delta_{i|-}, y_{t+1_i} + \delta_{i|+}) \qquad \delta_{i|-}, \delta_{i|+} \in \mathbb{R}_{\geq 0} \tag{36}$$

Diese Prozedur wird für die restlichen Werte der Liste wiederholt, bis keine Intervalle mehr gebildet werden können.

In algorithmischer Schreibweise ergibt sich folgendes:

Algorithm Clusteranalyse zur Intervallschätzung

```
ν ← 2.65
stddev ← 0
data ← sort data
for all val ∈ data do
  stddev ← stddev + val
end for
stddev ← ν * √(stddev/elements data)
Δdata ← diff data
elem_count ← elements Δdata
while elem_count > 0 do
  min_index ← getminpos Δdata
  upper_thres ← min_index
  lower_thres ← min_index
  for i ← 1, i <= min_index, i ← i + 1 do
    if Δdata[min_index - i] < stddev then
      lower_thres ← lower_thres - 1
    else
      last
    end if
  end for
```

```
    for i ← 1, i < elements Δdata - min_index, i ← i + 1 do
      if Δdata[min_index + i] < stddev then
        upper_thres ← upper_thres + 1
      else
        last
      end if
    end for
    upper_thres ← upper_thres + 1
    if upper_thres - lower_thres + 1 > 0.25 * elements data then
      push intervals, [lower_thres, upper_thres]
    end if
    for i ← lower_thres, i < upper_thres, i ← i + 1 do
      Δdata[i] ← inf
      elem_count ← elem_count - 1
    end for
    if elements getmaxinterval intervals > 1 then
      best_interval ← getmaxdensity getmaxinterval intervals
    else
      best_interval ← getmaxinterval intervals
    end if
    health ← 1.0
    if elements intervals = 0 then
      health ← normalize abs(abs(y_current - median data) / median data)
    else if not(y_current ∈ best_interval) then
      for all interval ∈ intervals do
        if y_current ∈ interval then
          last
        else
          health ← normalize health - length interval / elements data
        end if
      end for
    end if
    return health
  end while
```

3.3 Extrapolation von Daten

Bei der Extrapolation von Daten wird das Ziel verfolgt, einen Trend der Zeitreihe zu ermitteln, um mögliche, längerfristige Vorhersagen über das Verhalten der Zeitreihe in der Zukunft zu treffen. Aus Sicht der RRD sind somit nicht die Round Robin Archive mit zeitlich hoher Auflösung von Interesse, sondern jene, die die Datenhaltung über Monate oder Jahre übernehmen.

3.3.1 Nichtlineare Ausgleichsrechnung

Häufig werden in der Praxis vorhandene Messdaten auf ein Modell abgebildet, um weitere Analysen durchführen zu können. Da ein Modell lediglich eine Abstraktion des Beobachteten darstellt, können nicht alle Aspekte aus der „realen Welt“ berücksichtigt werden. Oftmals wird aber ein solches Modell benötigt, um Prognosen erstellen zu können (bei Zeitreihen: *Extrapolation*).
Dabei wird ein sogenannter *Fit* anhand der Messdaten der Zeitreihe gebildet, der je nach gewähltem Modell die zugehörigen Modellparameter ermittelt, die einen minimalen Fehler (*Residuum*) zum eigentlichen Messwert besitzen. Dieses Modell mit den errechneten Modellparametern stellt somit eine Annäherung der echten Daten und damit ein Modell des jeweiligen Prozesses dar.
Die einfachste Herangehensweise an ein solches Problem ist die lineare Ausgleichsrechnung mit dem Modell $b(t) = x_1 t + x_2$ und den zugehörigen Modellparametern x_1, x_2. Diese Gerade wird durch die vorhandenen Messpunkte so gelegt, dass die Abstände zur eigentlichen (unbekannten) Funktion $y(t)$ minimal werden.
Das gewählte Modell soll dabei annähernd das Verhalten der Messwerte aus der „echten“ Welt nachbilden und wird als „perfekt“ angenommen, sodass Rückschlüsse gezogen werden können, wie sich $y(t)$ in Zukunft (beziehungsweise allgemeiner: wie sich $y(t)$ außerhalb der vorhandenen Messwerte) verhalten könnte.
Da nicht immer ein polynomialer Trend zu erkennen sein muss, werden die Modellparameter über den allgemeineren Fall der Nichtlinearität und somit mit Hilfe der *nichtlineare Ausgleichsrechnung* ermittelt. Statt der Lösung eines linearen Fehlergleichungssystems werden die Parameter iterativ, mit Lösen eines Fehlergleichungssystems pro Iteration ermittelt. Dabei müssen die Parameter allerdings nicht immer konvergieren.

Folgendes Zitat aus dem Tool *FUDGIT*, welches zum Fitten von Messdaten dient, schildert das Problem der Ausgleichsrechnung:

> Nonlinear fitting is an art! [Merr 07]

In Abbildung 21 [S. 50] werden die letzten 300 Messwerte vom Datendurchsatz aller Pakete über den Router *saphir* (aufgenommen über *netsniff-ng*) herangezogen und über ein Polynom fünften Grades gefittet. Der erhaltene, rot dargestellte Graph gibt das als „perfekt“ angenommene Modell dieser Daten wieder und kann somit auch zur Anzeige von zukünftigen, geschätzten Messwerten benutzt werden. Allerdings sind diese vorhergesagten Daten immer mit Vorsicht zu genießen und je nach dem gewählten Modell qualitativ zu werten.

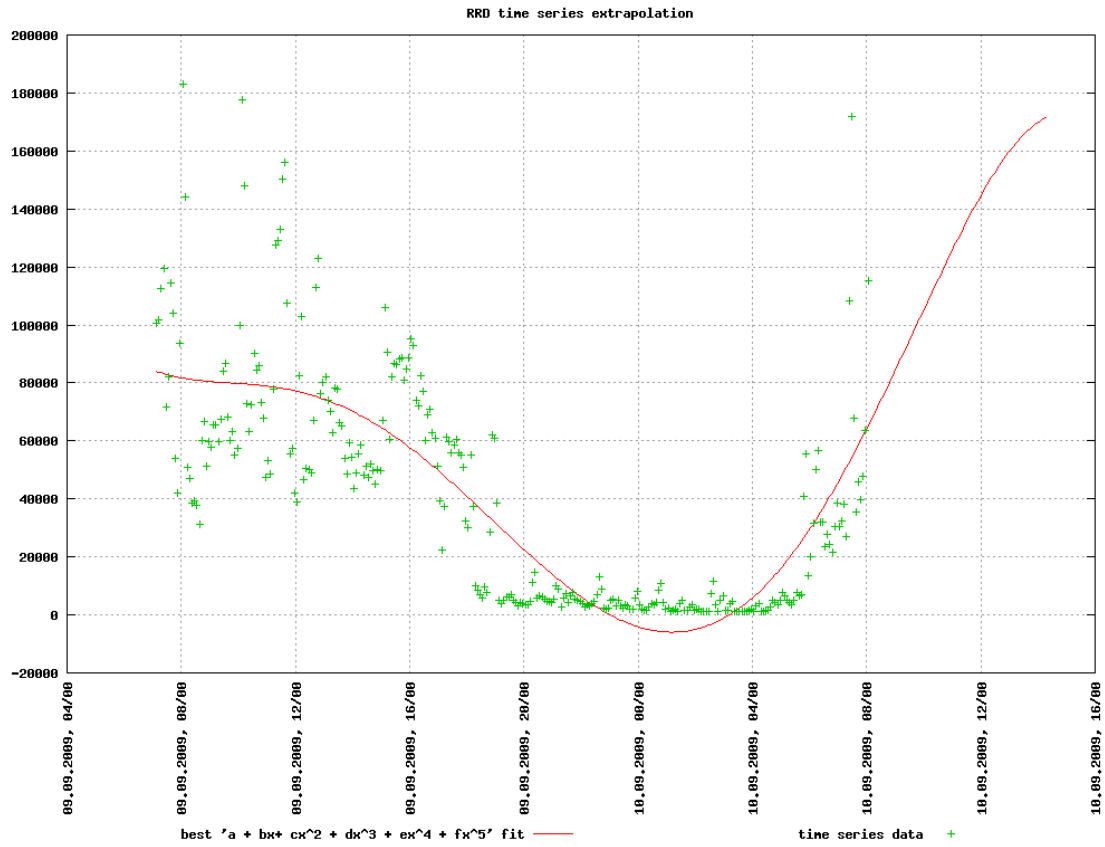


Abbildung 21: Extrapolation vom Datendurchsatz am MPI-Router: *In diesem Beispiel werden die durch netsniff-ng eingesammelten Daten in KB/min über ein Polynom fünften Grades mittels Levenberg-Marquardt-Algorithmus extrapoliert. In der roten Linie wird der Graph mit den besten Modellparametern angezeigt. Die „besten Modellparameter“ sind jene, bei der das Polynom so gewählt wird, dass die Abstände zu den eigentlichen Punkten minimiert werden. Ein solches Verfahren wird auch allgemein als Fit beziehungsweise Fitting bezeichnet.*

In diesem Abschnitt soll anhand von [Dahm 08] zunächst das *Gauss-Newton-Verfahren* vorgestellt und im Anschluss auf die robustere Weiterentwicklung, dem *Levenberg-Marquardt-Verfahren*, eingegangen werden, welches zur Messdatenvorhersage verwendet wird.

Für die Problemstellung sei zunächst eine Messreihe mit $b_i, i = 1, \ldots, m$ mit $m > n$ Messdaten gegeben, die auf eine von unbekannten Parametern abhängige Funktion

$$b(t) = y(t; x_1, \ldots, x_n) \tag{37}$$

zurückgeführt werden soll. Diese Funktion stellt das mathematische Modell der Messreihe dar, wobei die Parameter $x_1, \ldots, x_n$ *optimal* zu finden sind. *Optimal* dahingehend, dass $b_i \approx b(t_i),$ für $i = 1, \ldots, m$. Dies bedeutet, dass jene Parameter $x_1^*, \ldots, x_n^*$ bestimmt werden sollen, für die gilt:

$$\sum_{i=1}^{m} (y(t_i; x_1^*, \ldots, x_n^*) - b_i)^2 = \min_{x \in \mathbb{R}^n} \sum_{i=1}^{m} (y(t_i; x_1, \ldots, x_n) - b_i)^2 \tag{38}$$

Allgemein wird das nichtlineare Ausgleichsproblem wie folgt formuliert:
Bestimme $x^* \in \mathbb{R}^n$, so dass

$$||F(x^*)||_2 = \min_{x \in \mathbb{R}^n} ||F(x)||_2 \tag{39}$$

mit

$$F : \mathbb{R}^n \to \mathbb{R}^m, \qquad F_i(x) := y(t_i; x) - b_i, \qquad i = 1, \ldots, m \tag{40}$$

Beim *Gauss-Newton-Verfahren* wird nun versucht das *nichtlineare* Problem iterativ über eine Reihe geeigneter *linearer* Probleme anzunähern. Diese lineare Annäherung findet über eine *Taylorentwicklung* statt, sodass $F(x)$ durch

$$F(x) = F(x^k) + F'(x^k)(x - x^k) + O(||x - x^k||_2^2) \tag{41}$$

ersetzt wird, wobei $x^k \in \mathbb{R}^n$ eine bekannte Annäherung des nichtlinearen Ausgleichproblems sei. F' ist dabei die *Jacobi-Matrix*, welche sämtliche erste partielle Ableitungen beinhaltet.
Umformuliert führt dies zum *linearen Ausgleichproblem*

$$||F'(x^k)s^k + F(x^k)||_2 = \min_{s \in \mathbb{R}^n} ||F'(x^k)s + F(x^k)||_2 \tag{42}$$

wobei $s^k \in \mathbb{R}^n$ (mit minimaler 2-Norm[2]) gesucht sei. Dabei wird

[2] 2-Norm $\hat{=}$ Euklidische Norm

$$x^{k+1} = x^k + s^k \tag{43}$$

gesetzt.

Häufig wird s^k mit einem Schrittweitenparameter λ gedämpft, um eine Verringerung der Fehlerquadratsummen zu erzwingen, da nun die Funktion für verschiedene λ ausgewertet wird und um gleichzeitig ein mögliches Abdriften vom Minimum zu verhindern.

$$x^{k+1} = x^k + \lambda s^k, \qquad 0 < \lambda < 1 \tag{44}$$

Algorithmisch festgehalten ergibt sich nun folgendes:

Algorithm 1 Gauss-Newton-Verfahren

$z \leftarrow$ max. number of iterations
$\epsilon \leftarrow$ tolerance
$x^0 \leftarrow$ starting values
for all $k = 0, 1, 2, \ldots, z$ **do**
  calculate $F(x^k), F'(x^k)$
  solve linear curve fitting: $||F'(x^k)s^k + F(x^k)||_2 = min$
    e.g. via Cholesky decomposition $F'(x^k)^T F'(x^k)s^k = -F'(x^k)^T F(x^k)$ or
      via QR decomposition
  if $||F'(x^k)^T F(x^k)||_2 \leq \epsilon$ **then**
    break
  end if
  set new stepping:
    $\varphi(\lambda) = ||F(x^k + \lambda s^k)||_2$
    choose largest $\lambda \in \{1, \frac{1}{2}, \frac{1}{4}, \ldots\}$ so that $\varphi(\lambda) < \varphi(0)$
    $x^{k+1} \leftarrow x^k + s^k$
end for

Eine Alternative zu dieser Dämpfungsstrategie, die laut [Dahm 08] in der Praxis viel häufiger eingesetzt wird, ist die Technik nach Levenberg und Marquardt. Diese soll auch bei der Implementierung für die Vorhersage verwendet werden.

Die Minimierungsaufgabe zur Korrektur s^k wird beim *Levenberg-Marquardt-Verfahren* etwas anders formuliert als beim Gauss-Newton-Verfahren:

$$||F'(x^k)s^k + F(x^k)||_2^2 + \mu^2||s^k||_2^2 = min; \qquad \mu > 0 \tag{45}$$

μ ist ein festzulegender Parameter. Die neue Annäherung lautet wie beim ungedämpften Gauss-Newton-Verfahren:

$$x^{k+1} = x^k + s^k \tag{46}$$

Eine äquivalente Formulierung des Problems lautet

$$\left|\left| \begin{pmatrix} F'(x^k) \\ \mu E \end{pmatrix} s^k + \begin{pmatrix} F(x^k) \\ O \end{pmatrix} \right|\right|_2 = min, \tag{47}$$

wobei E die Einheitsmatrix ist und für $\mu > 0$ die Matrix

$$\begin{pmatrix} F'(x^k) \\ \mu I \end{pmatrix} \tag{48}$$

immer den vollen Rang hat und damit s^k eindeutig lösbar ist. Das lineare Gleichungssystem kann dann wie auch beim Gauss-Newton-Verfahren über eine *Cholesky-Zerlegung* oder über eine *QR-Zerlegung* gelöst werden.

Durch

$$||s^k||_2 \leq \frac{||F(x^k)||_2}{\mu} \tag{49}$$

kann mittels einer geeigneten Wahl von μ eine zu starke Korrektur von s^k verhindert (μ dient zur Dämpfung der Korrektur) und eine Verbesserung in Richtung Minimum bewirkt werden. Auch bei schlechten Startbedingungen konvergiert das Verfahren besser als *Gauss-Newton*, welches analog zum *Newton-Algorithmus* keine Konvergenz der Startwerte gewährleistet. Um die Konvergenz jedoch unabhängig von den Startwerten x^0 zu sichern bzw. zu verbessern, wird daher beim *Levenberg-Marquardt-Verfahren* über die Skalierung mit μ eine Dämpfung beziehungsweise Abstiegsbedingung erzwungen [Schr 09].

Der Algorithmus hierbei lautet:

Algorithm 2 Levenberg-Marquardt-Verfahren

$z \leftarrow$ max. number of iterations

$\epsilon \leftarrow$ tolerance

$x^0 \leftarrow$ starting values

for all $k = 0, 1, 2, \ldots, z$ **do**

calculate $F(x^k), F'(x^k)$

solve linear curve fitting: $\left|\left| \begin{pmatrix} F'(x^k) \\ \mu I \end{pmatrix} s^k + \begin{pmatrix} F(x^k) \\ O \end{pmatrix} \right|\right|_2 = min$

e.g. via Cholesky decomposition or QR decomposition

if $||s^k||_2 > \dfrac{||F(x^k)||_2}{\mu}$ **then**

adaptation of μ, solve linear curve fitting again (above)

end if

set new stepping:

$x^{k+1} \leftarrow x^k + s^k$

end for

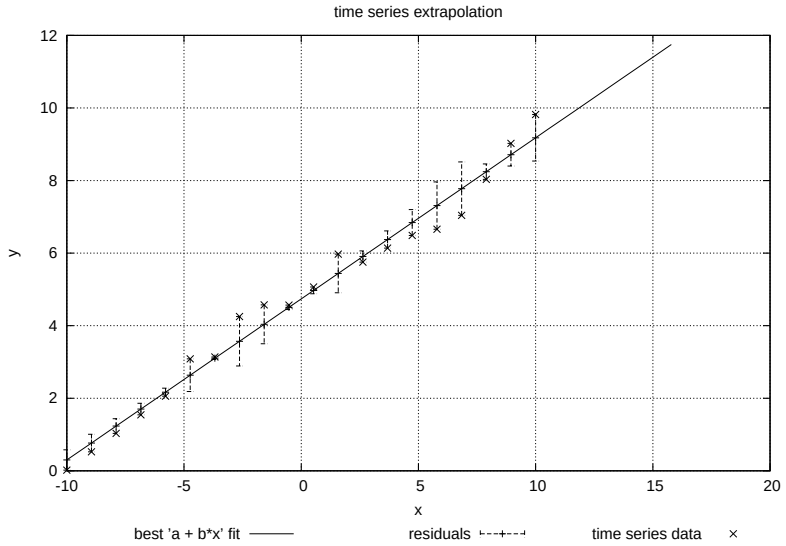
3.3.2 Mögliche Approximationsfunktionen

Die wohl schwierigste Aufgabe ist es, eine angemessene Approximationsfunktion zu finden, deren Parameter durch die vorher vorgestellten Verfahren geschätzt werden sollen.

Im Rahmen dieser Arbeit wurden einige mögliche Modelle zum weiteren Testen entwickelt, welche im Folgenden mit Abbildungen erwähnt sind. Neben den Messpunkten ist jeweils die gefittete Funktion inklusive derer Residuen abgebildet. Die Beispielmessdaten sind willkürlich gewählt und dienen lediglich zur Veranschaulichung der möglichen, über *Levenberg-Marquardt* approximierten Graphen.
Je mehr man über das etwaige Verhalten des Prozesses weiß, umso besser kann im Allgemeinen dieser auch modelliert werden.
Die hierbei aufgeführten Modelle sind lediglich einige wichtige Vertreter ihrer Klasse. Zu beachten sei außerdem, dass Modelle aus verschiedenen Klassen auch miteinander kombiniert werden können.
Für gewöhnlich ist die Gruppe der *polynomialen Fits* oder der *monomialen Fits* die erste Wahl, falls keine näheren Informationen zum Modell zur Verfügung stehen.

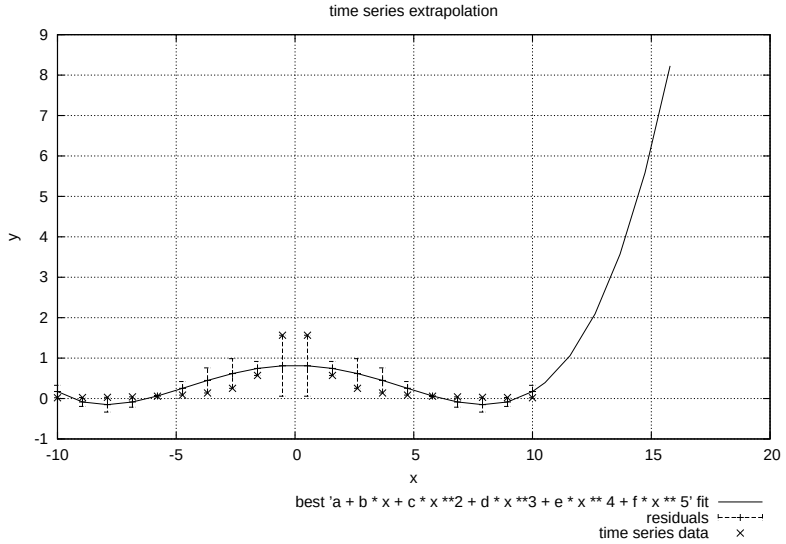
Lineare Fits Die einfachste Variante, allerdings womöglich auch sehr ungenaue ist es, einen linearen Trend vorherzusagen. Beispielsweise bei Prozessen mit großer irregulären Komponente und schlecht zu deutendem Verhalten kann der lineare Fit

durchaus eine brauchbare Verhaltensgrundrichtung vorhersagen. Ein solches Modell sieht typischerweise wie folgt aus:



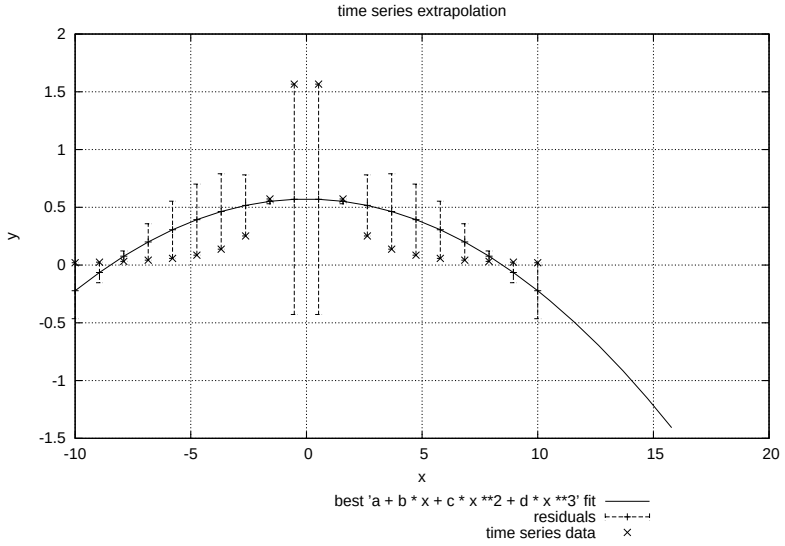
$$\textbf{Modell:} \qquad f(x) = a + bx, \qquad a, b \in \mathbb{R} \tag{50}$$

Polynomiale Fits Bei Unkenntnis über das Modell sind zunächst polynomiale Fits eine gute Wahl. Allerdings wie auch in den Beispieldaten zu sehen, haben sie den Nachteil, dass diese nach $+/-$ unendlich streben und bei zu groß gewähltem „Vorhersagefenster“ die Modellwerte nicht zu gebrauchen sind.



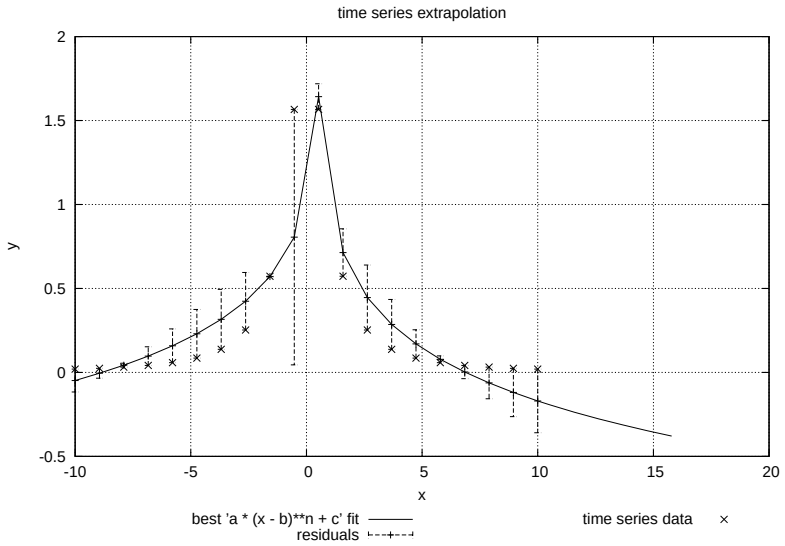
$$\textbf{Modell:} \qquad f(x) = a + bx + cx^2 + dx^3 + ex^4 + fx^5, \qquad a, b, c, d, e, f \in \mathbb{R} \tag{51}$$

Wie man weiterhin auch sehen kann, können bei unterschiedlichen Polynommodellen (oben: Polynom vom Grad 5, unten: Polynom vom Grad 3) auch sehr verschiedene Vorhersagen entstehen:



Ein Vergleich der Residuen wäre angebracht beziehungsweise Hinzunahme von weiteren Messwerten und, falls vorhanden, Kenntnis zur Verhaltensweise, um entscheiden zu können, welches Modell das Bessere ist.
Ein Hinzufügen weiterer Modellparameter und somit auch Polynomgrade kann den Fit durchaus auch verschlechtern (Oszillationen) - das Motto „viel hilft viel“ ist hierbei unangebracht.

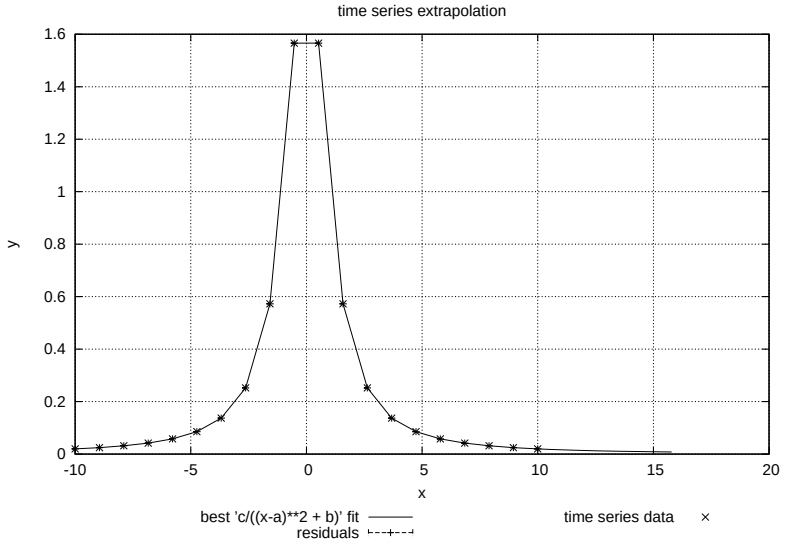
Monomiale Fits Monome sind Polynome, welche aus nur einem Glied bestehen. Neben den polynomialen Fits können diese ebenfalls bei wenig Vorwissen über das zugrundeliegende Modell verwendet werden.



$$\textbf{Modell:} \qquad f(x) = a * (x-b)^n + c, \qquad a, b, c, n \in \mathbb{R} \tag{52}$$

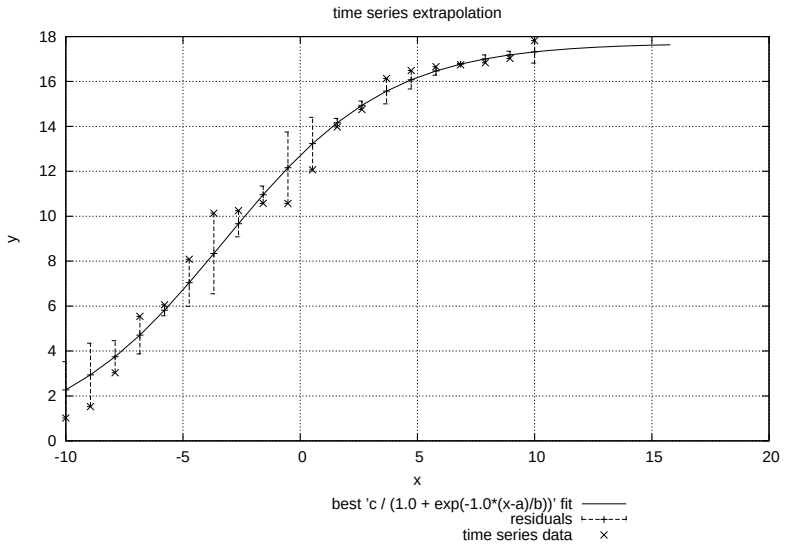
Der Parameter n kann entweder vorher fest definiert (Beispiel: $n \in \frac{1}{2}, \frac{3}{2}, -\frac{5}{2}, \ldots$) oder geschätzt werden.

Als weiteres Beispiel sei die Lorentzkurve erwähnt, welche bei Anwendung der selben Messdaten einen besseren Fit liefert.



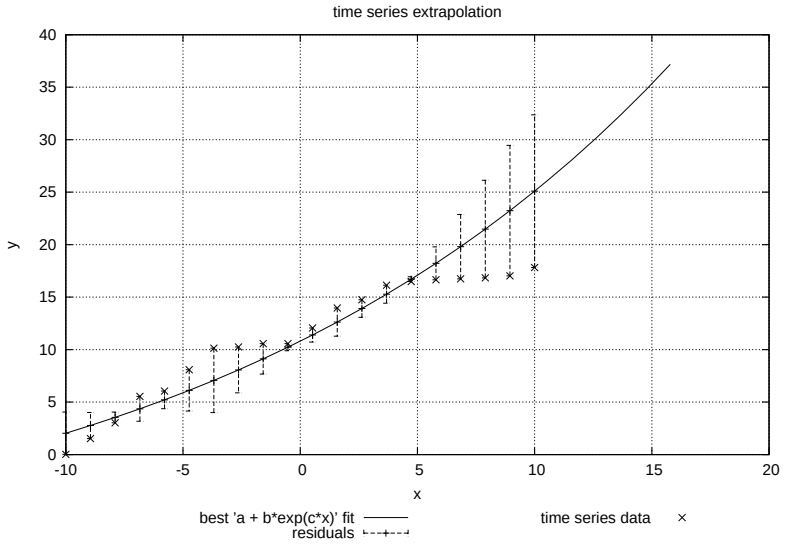
$$\textbf{Modell:} \qquad f(x) = \frac{c}{(x-a)^2 + b} \tag{53}$$

Exponentielle Fits Falls Daten für große $|x|$ „saturieren“, könnte sich beispielsweise folgendes Modell anbieten:



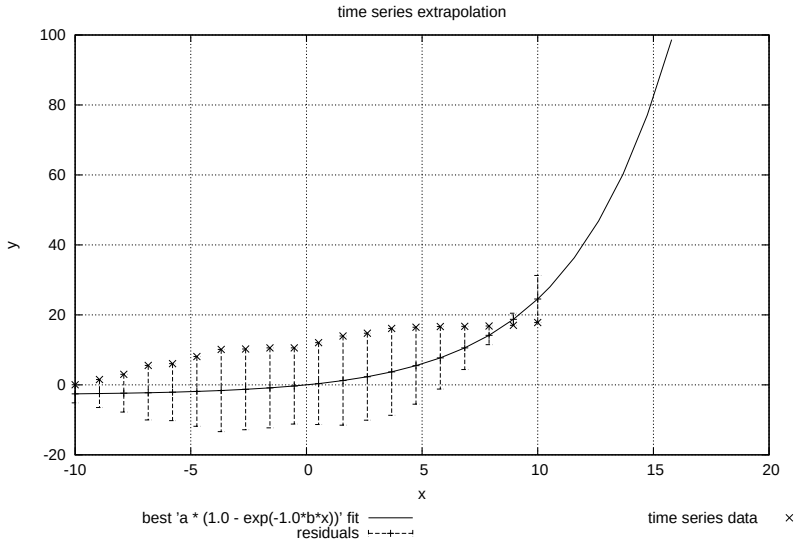
$$\textbf{Modell:} \qquad f(x) = \frac{c}{1 + e^{-\frac{x-a}{b}}}, \qquad a, b, c \in \mathbb{R} \tag{54}$$

Bei Anwendung der Daten auf den klassischen Vertreter der exponentiellen Modelle ergibt sich folgendes:



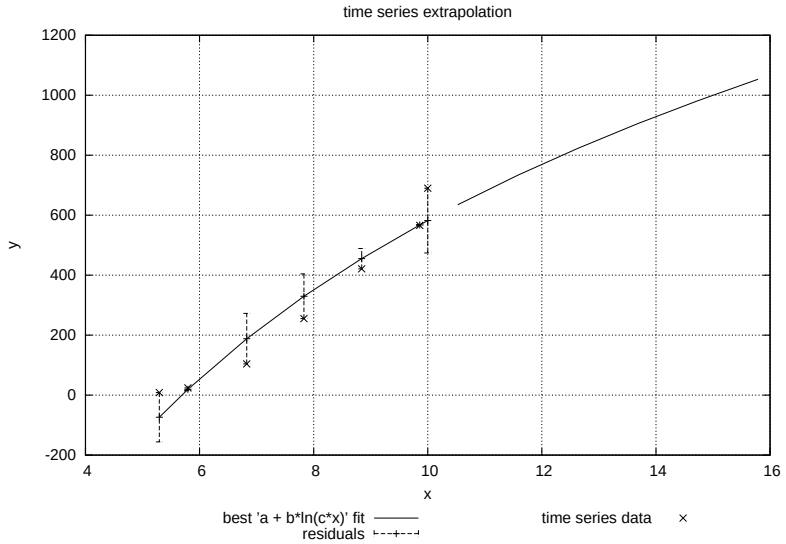
$$\textbf{Modell:} \qquad f(x) = a + be^{cx}, \qquad a, b, c \in \mathbb{R} \tag{55}$$

Beziehungsweise das restriktivere Modell (im Beispiel mit größerem Residuum), welches sich aus Vorherigem ebenfalls ergeben könnte:



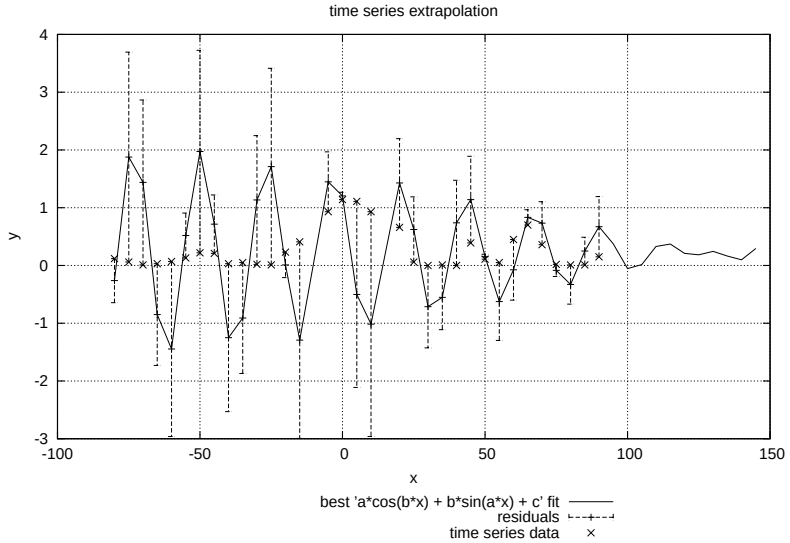
$$\textbf{Modell:} \qquad f(x) = a(1 - e^{-bx}), \qquad a, b \in \mathbb{R} \tag{56}$$

Logarithmische Fits Bei „Saturierung“ von Daten im Intervall $x > 0$ oder $x < 0$ könnte sich beispielsweise folgendes Modell anbieten:



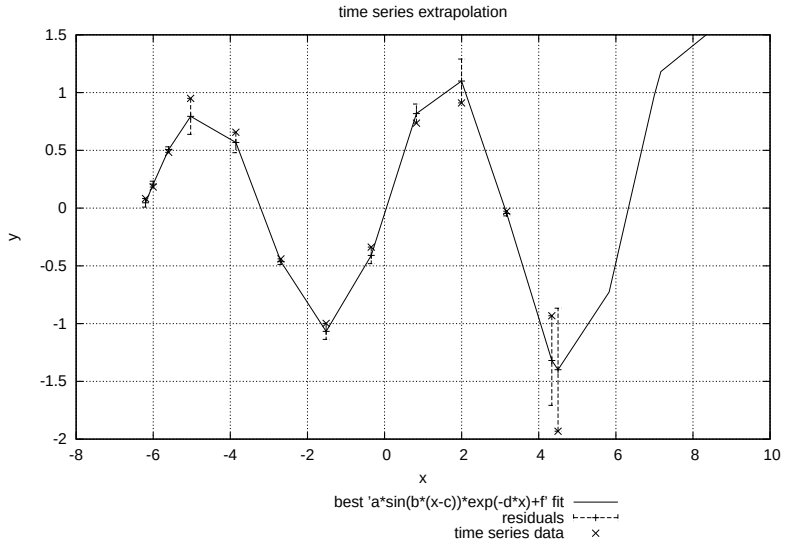
$$\textbf{Modell:} \qquad f(x) = a + b \ln(cx), \qquad a, b, c \in \mathbb{R} \tag{57}$$

Trigonometrische Fits Für oszillierende Funktionen, beispielsweise mit täglichen und wöchentlichen Saisonalitäten im Verhalten, können trigonometrische Modelle durchaus Mittel der Wahl sein. Eine Verknüpfung mit polynomialen Komponenten oder eintretender Sättigung mit monomialen oder logarithmischen Gliedern wäre denkbar.



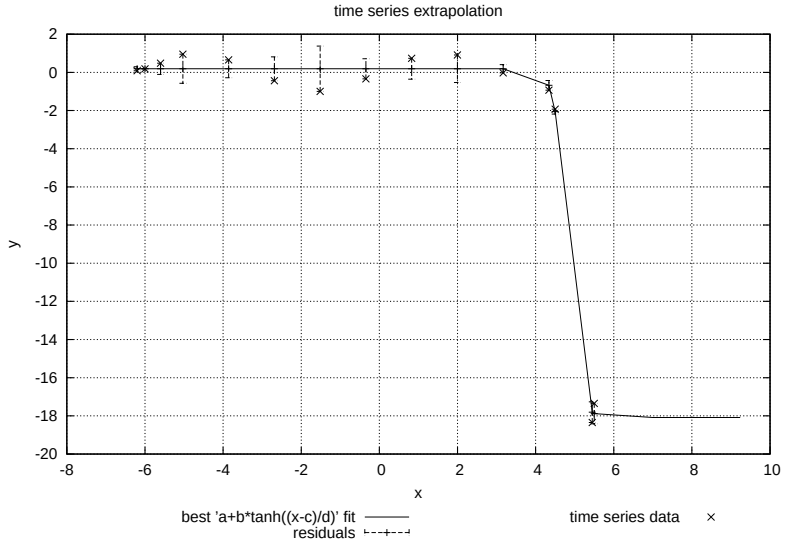
$$\textbf{Modell:} \qquad f(x) = a \cos(bx) + b \sin(ax) + c, \qquad a, b, c \in \mathbb{R} \tag{58}$$

Verknüpft mit einer nicht-konvergierenden Komponente sieht ein solches Modell beispielsweise wie folgt aus:



$$\textbf{Modell:} \qquad f(x) = a\sin\left(b(x-c)\right)e^{-dx} + f, \qquad a, b, c, d, f \in \mathbb{R} \tag{59}$$

Verhaltensweisen mit zu erwartenden Sprüngen könnten zum Beispiel über *tangens hyperbolicus* modelliert werden:

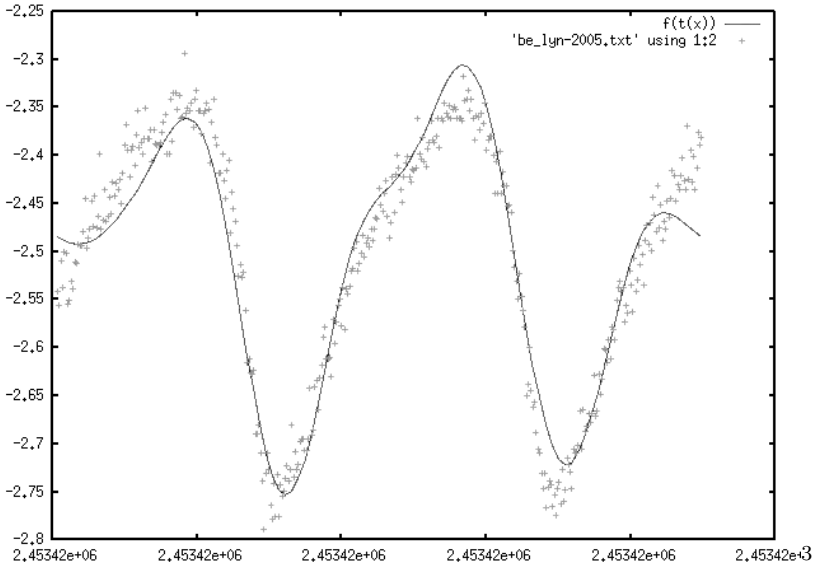


$$\textbf{Modell:} \qquad f(x) = a + b\tanh\left(\frac{x-c}{d}\right), \qquad a, b, c, d \in \mathbb{R} \tag{60}$$

Eine weitere Möglichkeit zum *Fitting* bieten auch Fourierreihen. In allgemeiner Form sehen diese wie folgt aus:

$$f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty}(a_k \cdot \cos(k\omega t) + b_k \cdot \sin(k\omega t)) \qquad , \omega = 2\pi/T \tag{61}$$

Die Kreisfrequenz ω dient dabei zur Skalierung der Periode 2π auf die Periode T. Für das zu modellierende Modell wird für gewöhnlich nach endlich vielen Reihengliedern (n) abgebrochen, sodass $f_n(t)$ eine akzeptable Annäherung für $f(t)$ ergibt.



Modell:

$$\begin{aligned} f(t) = c &+ a_0\cos(t) + a_1\sin(t) + a_2\cos(2t) + a_3\sin(2t) \\ &+ a_4\cos(3t) + a_5\sin(3t), \qquad c, a_0, a_1, ..., a_5 \in \mathbb{R} \end{aligned} \tag{62}$$

[3]http://freespace.googlepages.com/be_lyn-0013.18.03.2006.png/be_lyn-0013.18.03.2006-full.png, 22.09.2009

4 Implementierung

In diesem Abschnitt soll die Implementierung des entwickelten Nagios-Plugins, der Erweiterung von NagiosGrapher zur Erkennung von Anomalien, sowie das Framework für die Extrapolation der Messdaten beschrieben werden. Auf konkrete Implementierungsdetails kann dabei nicht eingegangen werden, da dies den Rahmen dieser Arbeit sprengen würde, jedoch sollen Schemata die Beschreibung von Aufbau und Interaktion der einzelnen Komponenten unterstützen.

4.1 Systemvoraussetzungen und eingesetzte Werkzeuge

Die Implementierung der in dieser Arbeit vorgestellten Programme und Algorithmen wurde unter Einsatz folgender Werkzeuge realisiert:

- **Betriebssystem:** Debian Lenny, z.T. Ubuntu 8.04
- **Programmiersprachen:** C, Perl, R (*z.T. für Prototyping*)
- **Programmierumgebung:** VIM - Vi IMproved 7.1
- **Sonstiges:** gnuplot 4.2 patchlevel 2, Perl XS (*h2xs*)

Nachfolgend genannt seien getestete Systemeinstellungen für den Router, auf welchem das im Rahmen dieser Arbeit entwickelte Plugin läuft, sowie für den Nagios-Server, welcher mit der hier erläuterten Erweiterung versehen ist.

4.1.1 Router mit netsniff-ng

Für den Router ist ein handelsüblicher Rechner im Einsatz. Dieser kann aufgrund der konstant niedrigen Last (*load*) ein älteres Modell mit Einkernprozessor sein.

Verwendete Hardware:

- **CPU:** AMD Athlon XP2400+, 1.5GHz
- **RAM:** 1.5GB
- **NICs:** 2 x Intel Corporation 82557/8/9/0/1 Ethernet Pro 100, 2 x 3Com Corporation 3c905C-TX/TX-M

Verwendete Software:

- **Betriebssystem:** Debian Lenny

- **Kernel:** 2.6.26-2-686, mit Option: *CONFIG_PACKET_MMAP=y*
- **Compiler:** gcc version 4.3.2

Für den Betrieb von netsniff-ng muss zwingend die Kerneloption *CONFIG_PACKET_MMAP* aktiviert sein, da ansonsten kein Memory-mapped RX_RING aufgebaut werden kann. Bei einer Standardinstallation von Debian ist diese bereits aktiviert, bei den institutsspezifischen Kerneln jedoch nicht.

4.1.2 Nagios-Server

Als Nagios-Server genügt eine handelsübliche Workstation, wobei ein Multicore-Prozessor zur besseren Skalierung empfohlen wird.

Verwendete Hardware:

- **CPU:** Intel Core 2 Duo, E8500, 2 x 3.16GHz
- **RAM:** 4GB

Verwendete Software:

- **Betriebssystem:** Debian Lenny
- **Kernel:** 2.6.26-1-686
- **Compiler:** gcc version 4.3.2
- **Sonstiges:** Nagios 3.0.6, RRDtool 1.3.1, NagiosGrapher v1.6.1-rc5, Perl v5.10.0, gnuplot 4.2 patchlevel 2

4.2 Funktionsbibliotheken

In diesem Abschnitt seien die für die Anomalieanalyse entwickelten Perl-Module mit deren wichtigsten Methoden näher erklärt.

4.2.1 FTX

Da von den RRDs die XML-Schnittstelle genutzt wird, um auf Daten zuzugreifen beziehungsweise um Daten zurückzuschreiben, wurde in dieser Arbeit ein performanter XML-Parser in *C* entwickelt, welcher sowohl für nachfolgend beschriebene Perl-Module der Anomalieanalyse im Einsatz ist, als auch für die Datenextrahierung im Vorhersage-Framework.
Die Schnittstelle von *C* nach *Perl* wurde mit *Perl XS* realisiert und das entstandene

Module im Späteren *FTX* genannt.

Ursprünglich wurde zum Parsen der RRD XML-Dumps ein Standard-XML-Parser (Verarbeitungsprinzip *DOM*) aus Perl verwendet, bei welchem sich jedoch nach einiger Zeit herausstellte, dass bei steigenden Datenmengen die Last nicht mehr zu bewältigen war. Da während der Anomalieanalyse Daten sehr schnell verarbeitet werden müssen (die Analyse wird an das NagiosGrapher-Skript *collect2.pl* kaskadiert, welches Daten aller 300s in RRDs schreibt), ist es von Vorteil, möglichst geringe Latenzzeiten für die Datenberechnung zu haben. Ein möglicher Flaschenhals soll nicht bereits bei der Datentransformation entstehen.

Ein typischer XML-Dump einer Round Robin Datenbank hat in etwa 13.400 Zeilen einfaches XML. Mit einfachem XML sei gemeint, dass keine Attribute oder dergleichen verwendet werden.

Unter der Annahme, dass das XML, welches vom *RRDtool* erzeugt wird, korrekt ist und nicht weiter auf Korrektheit überprüft werden müsste (wie dies jedoch viele Parser durchführen), kann an dieser Stelle bereits Zeit eingespart werden.

Die Idee basiert daher darauf, einen DOM-ähnlichen Parser zu entwickeln, der schneller ist, als die vorhandenen Standardparser. Dazu wurde von den üblichen DOM-Prinzipien abgewichen und ein eigenes Baumformat namens *Fast Token XML* entwickelt. Die verarbeiteten *XML-Tags* werden als *Nodes* im Baum gespeichert. Eine solche Node stellt ein *Bitfeld* mit wichtigen Informationen dar.
Da häufig die Funktion *strcmp()* (bzw. die sichere Variante *strncmp()*) zum Vergleich von *Tags* aus dem erstellten *DOM-Baum* mit dem gesuchten *Tag* verwendet wird, muss die CPU ein *String-Matching* (siehe Abbildung 22 [S. 65]) betreiben. Einfacher wäre es jedoch, nur einen einzigen Vergleich pro *Tag* durchführen zu müssen.

Grundlegend besteht *Fast Token XML* aus den Elementen *struct cache*, *struct levelcache* und *struct node*. Der komplette Baum wird dabei durch *struct cache* repräsentiert. Der *struct levelcache* repräsentiert sämtliche Child-Knoten einer *struct node*, wobei letztere wiederum Informationen zum eigentlichem XML-Tag enthält (siehe Abbildung 24 [S. 65]) und einen weiteren *struct levelcache* beinhalten kann.
Ein kleines Beispiel soll das Zusammenspiel der Komponenten verdeutlichen (Abbildung 23 [S. 65]).

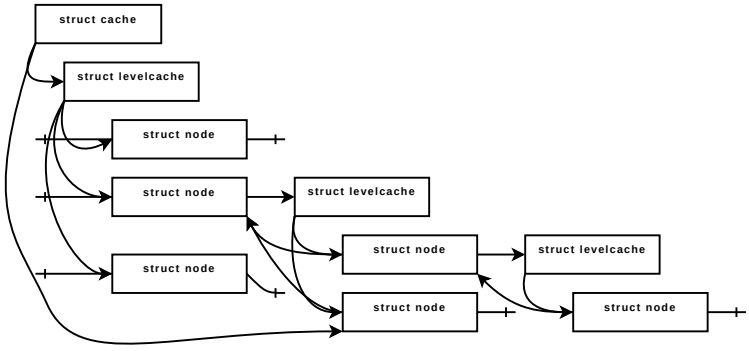
```
int strcmp(const char *cs, const char *ct)
{
        signed char __res;

        while (1) {
                if ((__res = *cs - *ct++) != 0 || !*cs++)
                        break;
        }
        return __res;
}
EXPORT_SYMBOL(strcmp);
```

Abbildung 22: Vergleich zweier Strings mittels strcmp(), Quelle: Linux Kernel (lib/string.c): *Zu sehen ist, dass pro Zeichen ein Vergleich im Akkumulator stattfindet. Im schlechtesten Fall entspricht die Anzahl der Vergleiche der Länge des kürzeren Strings. Effizienter wäre dabei nur ein einziger Vergleich mittels Hashwert.*

XML-Struktur

```
<root>
   <a>foo</a>
   <b>
       <b1>
           <b11>bar</b11>
       </b1>
       <b2>oof</b2>
   </b>
   <c>rab</c>
</root>
```



Interne FTX-Struktur

Abbildung 23: Zusammenspiel der FTX-Komponenten: *Oben angeführte XML-Struktur wird zu darunter folgenden FTX-Baumstruktur transformiert.*

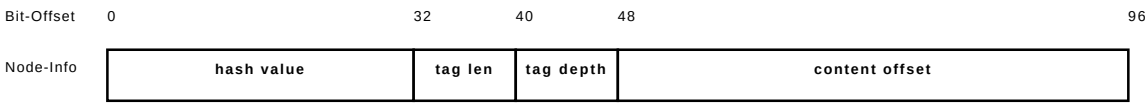


Abbildung 24: Gespeicherte Informationen innerhalb einer Node

Als Hashfunktion wurde eine Lookup-Tabelle mit CRC-32-Werten im Speicher gehalten. Das Traversieren der XML-Struktur brachte keine Kollisionen hervor.

Trotz unterschiedlich vielen Funktionalitäten soll ein Benchmark die wichtigsten Perl-Module von DOM-ähnlichen XML-Parsern untersuchen und FTX gegenüberstellen. Einige dieser Module sind in reinem *Perl* geschrieben, einige andere wiederum in *C* mit Bindings nach *Perl*. Als Eingabe dient - wie auch im Realfall - ein herkömmlicher XML-Dump einer Round Robin Datenbank. Das einzigste, für uns interessante Kriterium, unbeachtet aller Extras, die der jeweilige Parser bietet, ist die Zeit, in welcher die interne Baumstruktur aufgebaut wird, sowie die Abfrage des Inhalts von allen vorkommenden */rrd/rra/pdp_per_row*-Elementen. Dies wird zehntausend mal hintereinander ausgeführt.
Der Test läuft dabei auf einer Intel Core Duo T2600 Maschine mit 2.16GHz und 4GB RAM. Die Ergebnisse sind in Tabelle 6 [S. 66] abzulesen.

Perl-Modul	**10.000-faches Parsen eines XML-RRD-Dumps**		
	Real-Zeit	CPU-Zeit	Kernelmode-Zeit
XML::Trivial	16h13m44.240s	16h01m02.482s	5m20.320s
XML::DOM	11h46m52.122s	11h37m15.754s	5m01.241s
XML::Simple	10h02m34.994s	9h55m29.394s	3m45.528s
XML::TreePP	9h52m03.231s	9h48m43.230s	2m37.572s
XML::Twig	4h56m11.451s	4h53m23.939s	1m56.816s
XML::LibXML	28m18.839s	18m39.258s	2m08.844s
XML::Xerces	26m28.786s	25m23.915s	1m03.980s
FTX	6m42.074s	6m26.956s	0m14.953s

Tabelle 6: Benchmark einiger XML-Parser aus Perl (zum Teil mit *C*-Bindings), sowie der eigenen FTX-Lib: *Viele der XML-Parser sind relativ langsam. Die XML::Xerces vom Apache-Webserver, sowie die XML::LibXML vom GNOME-Projekt konnten als einzige gegenüber den anderen Standardparsern überzeugen. Sie sind beide jedoch in etwa 4.3fach langsamer als die FTX-Bibliothek. Zwar bietet FTX weniger Möglichkeiten als XML::Libxml beziehungsweise XML::Xerces, jedoch erfüllt diese die gewünschten Anforderungen, einen RRD-Dump in sehr kurzer Zeit zu parsen.*
Einige andere, vielversprechende XML-Parser aus dem CPAN-Verzeichnis (unter anderem XML::Bare), welche viel Wert auf Geschwindigkeit legen, brachen jedoch zum Teil bereits bei der Baumerstellung mit einem „Segmentation fault“ ab.

Nach Umschreiben der Baseline-Library auf den neuen XML-Parser (FTX) war eine signifikante Zeiteinsparung zu erkennen. Zu beachten ist allerdings, dass dieser XML-Parser lediglich für spezielle Zwecke entwickelt wurde und nur bedingt für andere Anwendungen in Frage käme.

init() Diese Funktion, welche als Übergabeparameter das zu parsende XML-File benötigt, erstellt die interne Baumstruktur, die später durch ein *find()* traversiert wird.

find() Die Funktion *find()* dient zum Suchen von Knoten und gibt deren Inhalt zurück. Unterstützt werden zwei sehr einfache *XPath*-Ausdrücke. Zum einen kann nach allen vorkommenden Elementen gesucht werden (Beispiel: */rrd/rra/pdp_per_row*) und zum anderen nach allen vorkommenden Elementen, die eine Bedingung erfüllen (Beispiel: */rrd/rra/database/row/v[/rrd/rra/pdp_per_row=72]*). Dies reicht aus, um die notwendigen Daten der verschiedenen Zeitreihen gezielt zu extrahieren. Der aktuelle „Suchpointer“ wird auf die Baumwurzel gesetzt und der Suchausdruck wird geparst.

findnext() Mittels *findnext()* wird vom zuletzt gefundenen Knoten aus der Baum weiter durchsucht. Die durch *find()* erstellten Strukturen vom Suchausdruck werden weiterhin benutzt.

release() Der komplette, allokierte Speicher wird wieder freigegeben.

4.2.2 Baseline::Algorithm::CalcRRD

Dieses in dieser Arbeit entwickelte Modul beinhaltet den Algorithmus zur Anomalieanalyse. Die Idee einer solchen Kapselung ist dazu gedacht, dass verschiedene *Baseline::Algorithm::CalcRRD*-Module entwickelt werden (hier: eines für Holt-Winters und ein anderes wiederum für die Clusteranalyse) und dann zur Laufzeit ausgetauscht werden können, ohne dass von den RRDs der Messdaten Informationen verloren gehen. So können Algorithmen besser getestet und Neue schneller entwickelt werden. Lediglich die zwei nachfolgend aufgeführten Funktionen müssen exportiert werden. Implementiert wurde jeweils ein Modul für die in Abschnitt 3 [S. 28] vorgestellten Algorithmen zur Anomalieanalyse.

calc_rrd() Diese Funktion beinhaltet die spezifische algorithmische Realisierung zur Anomalieanalyse. Um den Entwickler möglichst viele Freiheitsgrade zu lassen, werden lediglich durch das nachfolgend erwähnte Modul *Baseline::Util::Routines* Hilfsmethoden für die Arbeit mit RRDs zur Verfügung gestellt. Hauptaufgabe von *calc_rrd()* ist es, die Nutzdaten aus der RRD zu laden, Berechnungen zur Anomalieanalyse durchzuführen und die Ergebnisse in eine eigene Datenstruktur zu speichern.

check_baseline() Mit Hilfe von *check_baseline()* wird dem Nagios-Nutzer angezeigt, ob sich in der angegebenen RRD der Messdaten eine Verhaltensanomalie befindet. NagiosGrapher wird dabei dahingehend modifiziert, dass im Webinterface von Nagios, direkt zu den Services, Informationen über Anomalien erwähnt werden. Da das Erkennen einer Anomalie je nach Algorithmus unterschiedlich ist, gilt es neben *calc_rrd()* auch diese Funktion zu implementieren und zu exportieren.

4.2.3 Baseline::Util::Routines

Das im Rahmen dieser Arbeit entwickelte *Baseline::Util::Routines*-Modul beinhaltet Hilfsroutinen, die die Implementierung der Algorithmen in *Baseline::Algorithm::CalcRRD* unterstützen sollen. Hauptsächlich sind dies (*I/O*-)Funktionen, die zur Extrahierung und zum Zurückschreiben von RRD-Daten dienen. Lediglich die wichtigsten seien hier erwähnt. Zum Arbeiten auf den XML-Datensätzen wird das Modul *FTX* herangezogen.

fetch_stepping(), fetch_lastupdate() Die Methode *fetch_stepping()* dient zum Auslesen des Update-Intervalls der RRD (in Sekunden). Zusammen mit *fetch_lastupdate()*, welches den letzten Aktualisierungszeitpunkt als Unix-Timestamp ausliest, können so die Zeitpunkte zurückberechnet werden, in denen die Datensätze eingetroffen sind.

fetch_pdplist(), fetch_table() Die erstere Funktion liest alle PDP-Konstanten aus. Diese Konstanten sind Faktoren für die Rückrechnung der Zeitpunkte, in denen Daten eingetroffen sind. Diese werden mit dem Update-Intervall multipliziert. Weiterhin dienen die PDPs zur eindeutigen Identifikation der RRA-Datensätze. Anhand der PDP-Liste kann über einen bedingten *XPath*-Ausdruck mit Hilfe von *FTX* die zugehörigen Datensätze ausgelesen werden. Dies erledigt die Funktion *fetch_table()*, welche auch optional die jeweiligen Unix-Timestamps berechnet und in die Datensatztabelle anhängt.

shiftleft(), shiftright() *shiftleft()* und *shiftright()* sind jeweils Operationen auf Datensatztabellen und dienen zum Einfügen von neuen berechneten Werten. Dabei werden die bereits vorhandenen Tabellenwerte um einen Eintrag verschoben (links- oder rechtsseitig) und das Resultat der Berechnung an die entsprechend neue freie Stelle eingefügt. Dies wird benötigt, um die Ergebnisse der *calc_rrd()* Berechnung (den *health*-Wert) in die dafür reservierte RRD zu speichern.

dump_rrd(), restore_rrd() Diese Funktionen dienen zum Exportieren der RRD in einen XML-Dump (*dump_rrd()*) beziehungsweise zum Importieren der neuen

XML-Datei in die RRD ((*restore_rrd()*)). Bei letzterer Funktion gehen die alten Daten der RRD verloren und werden mit den neuen XML-Daten überschrieben.

write_rrd(), merge_rrd() Die Funktion *write_rrd()* dient zum Umwandeln der Datensatztabelle in einen XML-Dump, der RRD-konform ist und somit über *restore_rrd()* zurückgeschrieben werden kann. *merge_rrd()* wird dazu verwendet, zwei RRD-Datenbanken zu einer einzigen verschmelzen zu lassen. Da sowohl Messdatensätze als auch die jeweiligen *health*-Datensätze über NagiosGrapher angezeigt werden sollen, müssen diese zur Anzeige in eine RRD kombiniert werden. Durch eine Anpassung der NagiosGrapher-Module ist es nun möglich, eine RRD mit Messdaten unabhängig vom Analysealgorithmus zu aktualisieren. Des Weiteren wird eine RRD für die *health*-Daten reserviert. Zur graphischen Darstellung in der Weboberfläche werden beide RRDs zusammengeführt.

4.3 Nagios-Plugin „netsniff-ng“

4.3.1 netsniff-ng

Das Programm „netsniff-ng“ ist ein passiver, in *C* entwickelter Netzwerk-Sniffer, mit dem Ziel, Daten für Nagios zu sammeln, die anschließend auf Anomalien hin überprüft werden sollen und ist Rahmen dieser Arbeit entstanden.

Dabei ist ein Einsatz als Vordergrund-Prozess oder aber als System-Daemon möglich. Beide Varianten haben jedoch folgende Arbeitsschritte zum Einsammeln der Pakete gemeinsam:

```
 1. Socket der PF_PACKET-Familie allokieren
 2. Netzwerkschnittstelle in Promiscuous Mode schalten
 3. Berkeley Packet Filter für Socket parsen und laden
 4. RX RING erstellen
 5. Netzwerkschnittstelle an Socket binden (mit Protokoll ETH_P_ALL)
 6. Memory-Map vom RX RING bilden
 7. Traversieren des Ringes nach Frames != TP_STATUS_KERNEL
 8. Verarbeiten dieser Frames und Rücksetzen zu TP_STATUS_KERNEL
 9. Auf Ereignisse pollen (falls alle Frames == TP_STATUS_KERNEL)
10. Gehe zu Punkt 7
```

Die Ringgröße beläuft sich auf 128 MB Kernelspeicher. Somit können bei einer Blockgröße von 16.384 Byte und einer statischen Framegröße von 2.048 Byte (aligned)

8.192 Blöcke bzw. 65.536 Frames im Ringpuffer eingefangen werden. Dieser Puffer ist in den meisten Fällen groß genug, um mit Gigabit-Ethernet umgehen zu können. Sollten in kurzer Zeit mehr als 65.536 Frames eintreffen und alle Speicherblöcke des Puffers belegt sein, so werden die eintreffenden, nicht mehr zu verarbeitenden Frames für den PF_PACKET-Socket verworfen.
In Abbildung 25 [S. 71] wird der grundlegende Aufbau von „netsniff-ng“ dargestellt. Das Programm besteht aus mehreren Komponenten und kann auch als eigenständige Lösung ohne Nagios verwendet werden.
Läuft *netsniff-ng* im Daemon-Modus, dann sind Statistiken zum einen über das Logfile einsehbar (das Signal SIGUSR1 muss an den netsniff-ng-Prozess gesendet werden) und zum anderen über Unix Domain Sockets abrufbar.
Während des Startens als System-Daemon wird dabei ein Thread für den Unix Domain Socket Server eingerichtet. Sobald eine Verbindung mit einem Client zustande kommt, wird ein weiterer Thread zum Abarbeiten und Versenden der Statistik erzeugt.

Ein *strace*-Vergleich (Abbildung 26 [S. 72] und 27 [S. 73]) zeigt, dass *netsniff-ng* effizienter ist als das auf *libpcap* basierende *tcpdump*[4]. Während *tcpdump* für jedes eintreffende Paket einen Systemcall macht (*recvfrom()*), wird bei *netsniff-ng* kein einziger aufgerufen, sondern die eintreffenden Pakete werden einfach über einen *Pointer* auf eine Struktur abgerufen. Der dafür bestimmte Kernelspeicherbereich wird über eine *shared* Memory-Map mit dem Userspace geteilt. Bei *tcpdump* wird hingegen bei jedem *recvfrom()*-Aufruf der zu übergebende Speicher vom Userspace in den Kernelspace kopiert (*copy_from_user()*) und nach Abarbeiten der Routinen wieder zurück vom Kernelspace in den Userspace (*copy_to_user()*). Da diese Kopieraktionen sehr zeitintensiv sind, kann es zu Paketverlusten (und somit auch ungenauen Statistiken) kommen, falls sehr viele eintreffende Pakete abgearbeitet werden müssen.

Zwar wurde *netsniff-ng* mit dem Ziel konzipiert, *Fast-Ethernet*-Pakete einzusammeln, ein Performance-Test zeigt allerdings, dass selbst mit *Gigabit*-Ethernet gut umgegangen werden kann. Anhand eines *Ping-Floodings* wurde die Bandbreitengrenze von circa 114 MB pro Sekunde ausgereizt, lediglich mit einer CPU-Auslastung von 0 bis 3 Prozent (Abbildung 28 [S. 74]). *netsniff-ng* konnte so innerhalb der Testzeit von 4,75 Minuten mehr als 27 Gigabyte an Netzwerkverkehr ohne Paketverluste analysieren.

[4] tcpdump (3.9.8-4) aus dem Debian Lenny *stable*-Zweig

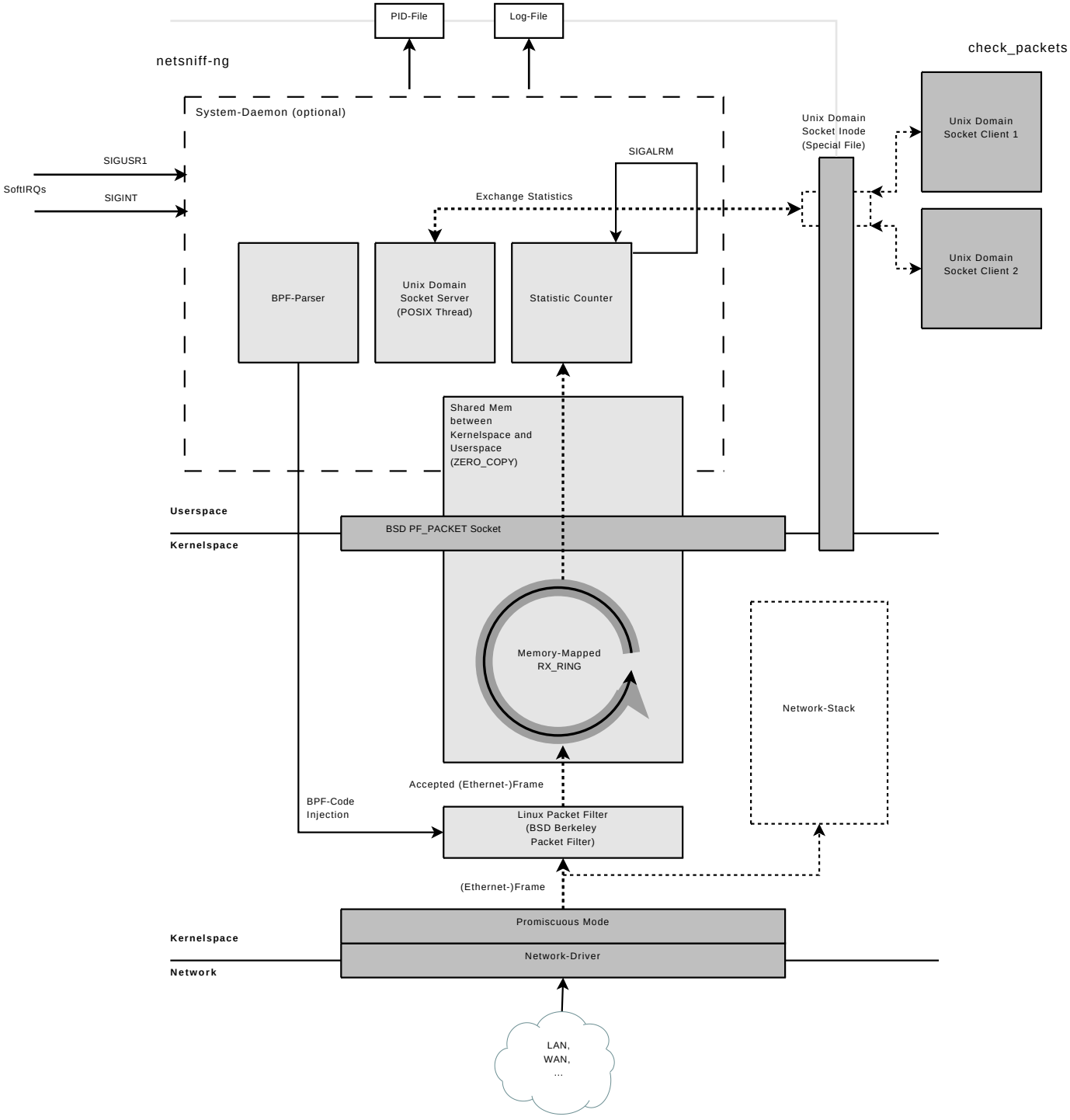


Abbildung 25: Systemarchitektur von netsniff-ng

```
recvfrom(3, "\377\377\377\377\377\377\0\32ME|\211\10\6\0\1\10\0\6\4"..., 96, MSG_TRUNC,
  {sa_family=AF_PACKET, proto=0x806, if50, pkttype=PACKET_BROADCAST, addr(6)={1, 001a4d457c89}, [18]) = 60
ioctl(3, SIOCGSTAMP, 0xbfb43e70)        = 0
write(1, "16:14:50.538813 arp who-has 10.0"..., 5516:14:50.538813 arp who-has 10.0.53.26
  tell 10.0.63.10) = 55
recvfrom(3, "\377\377\377\377\377\377\0\4v\243\330\242\10\6\0\1\10\0"..., 96, MSG_TRUNC,
  {sa_family=AF_PACKET, proto=0x806, if50, pkttype=PACKET_BROADCAST, addr(6)={1, 000476a3d8a2}, [18]) = 60
ioctl(3, SIOCGSTAMP, 0xbfb43e70)        = 0
write(1, "16:14:50.624868 arp who-has 10.0"..., 5616:14:50.624868 arp who-has 10.0.63.53
  tell 10.0.54.184) = 56
recvfrom(3, "\377\377\377\377\377\377\0\4v\243\330\242\10\6\0\1\10\0"..., 96, MSG_TRUNC,
  {sa_family=AF_PACKET, proto=0x806, if50, pkttype=PACKET_BROADCAST, addr(6)={1, 000476a3d8a2}, [18]) = 60
ioctl(3, SIOCGSTAMP, 0xbfb43e70)        = 0
write(1, "16:14:50.624989 arp who-has 10.0"..., 5616:14:50.624989 arp who-has 10.0.63.55
  tell 10.0.54.184) = 56
```

Abbildung 26: strace Auszug von **„tcpdump -i eth0 -n arp“**: *tcpdump verwendet im Gegensatz zu netsniff-ng kein RX RING, sodass pro eintreffendem Paket ein Systemcall und damit verbunden eine Kopie des Frames in den Userspace durchgeführt werden muss. Bei hoher Netzwerklast hätte dies aufgrund der Zeitkosten Paket-Verwerfungen zur Folge. Vorteilhaft am recvfrom-Aufruf ist jedoch, dass auch ausgehende Pakete vom eigenen Host mitgeschnitten werden können (pkttype=PACKET_ HOST). Da netsniff-ng allerdings auf dem Router läuft, sind diese eigen-produzierten Daten im Gegensatz zu den Transferdaten weniger von Interesse.*

```
rt_sigprocmask(SIG_BLOCK, [USR1 ALRM], NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [USR1 ALRM], NULL, 8) = 0
poll([{fd=3, events=POLLIN|POLLERR, revents=POLLIN}], 1, -1) = 1
write(2, "I: ", 3I: )                       = 3
write(2, "60 bytes from 00:1a:4d:45:7c:89 "..., 5360 bytes from 00:1a:4d:45:7c:89 to ff:ff:ff:ff:ff:ff) = 53
rt_sigprocmask(SIG_BLOCK, [USR1 ALRM], NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [USR1 ALRM], NULL, 8) = 0
poll([{fd=3, events=POLLIN|POLLERR, revents=POLLIN}], 1, -1) = 1
write(2, "I: ", 3I: )                       = 3
write(2, "60 bytes from 00:10:5a:d8:9a:a4 "..., 5360 bytes from 00:10:5a:d8:9a:a4 to ff:ff:ff:ff:ff:ff) = 53
rt_sigprocmask(SIG_BLOCK, [USR1 ALRM], NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [USR1 ALRM], NULL, 8) = 0
poll([{fd=3, events=POLLIN|POLLERR, revents=POLLIN}], 1, -1) = 1
write(2, "I: ", 3I: )                       = 3
write(2, "60 bytes from 00:30:48:c6:15:d0 "..., 5360 bytes from 00:30:48:c6:15:d0 to ff:ff:ff:ff:ff:ff) = 53
```

Abbildung 27: strace Auszug von **„netsniff-ng -d eth0 -f /etc/netsniff-ng/rules/arp.bpf -C“**: *netsniff-ng verwendet im Gegensatz zu tcpdump einen RX_RING. Damit fallen Systemcalls wie recvfrom weg und es muss von der Kernelseite aus kein copy_to_user() aufgerufen werden. Dies hat eine bessere Performance zur Folge. Lediglich poll() muss gerufen werden, falls keine Daten mehr im Ringpuffer liegen. Dies ist dann der Fall, wenn wenige Daten anliegen, sodass die Frames aus dem Puffer schneller verarbeitet werden als diese überhaupt eintreffen können und somit wieder in den Status TP_STATUS_KERNEL zurückkehren.*

```
...
I: stats summary:
I: ------------------------------------------------------------------------------------
I: elapsed time: 0 d, 0 h, 4 min, 45 s
I: -----------+--------------------------+--------------------------+--------------------------
I:            |  per sec                 |  per min                 |  total
I: -----------+--------------------------+--------------------------+--------------------------
I:   frames   |                    80201 |                  4696273 |                  20149411
I: -----------+--------------------------+--------------------------+--------------------------
I:   in B     |                119622775 |               7003546464 |               30048067864
I:   in KB    |                   116819 |                  6839400 |                  29343816
I:   in MB    |                      114 |                     6679 |                     28656
I:   in GB    |                        0 |                        6 |                        27
I: -----------+--------------------------+--------------------------+--------------------------
...
I: 23724545 frames incoming
I: 23724545 frames passed filter
I: 0 frames failed filter (due to out of space)
I: captured frames: 23724545, captured bytes: 35377999772 [34548827 KB, 33739 MB, 32 GB]
```

Abbildung 28: Performancetest von netsniff-ng mittels Ping-Flooding über Gigabit-Ethernet: *Aufruf von „netsniff-ng -d eth0 -f /etc/netsniff-ng/rules/icmp.bpf“ mit dem Ergebnis, dass unter Gigabit-Volllast lediglich 0-3% CPU-Leistung in Anspruch genommen wird. netsniff-ng lief dabei auf einer Intel Core(TM)2 Duo CPU, E8500 Maschine mit 4 GB RAM und einem 2.6.26-1-686 #1 SMP Debian/Lenny-Kernel. Der Onboard-Ethernet-Controller ist von Realtek mit einem RTL8111/8168B Chipsatz.*

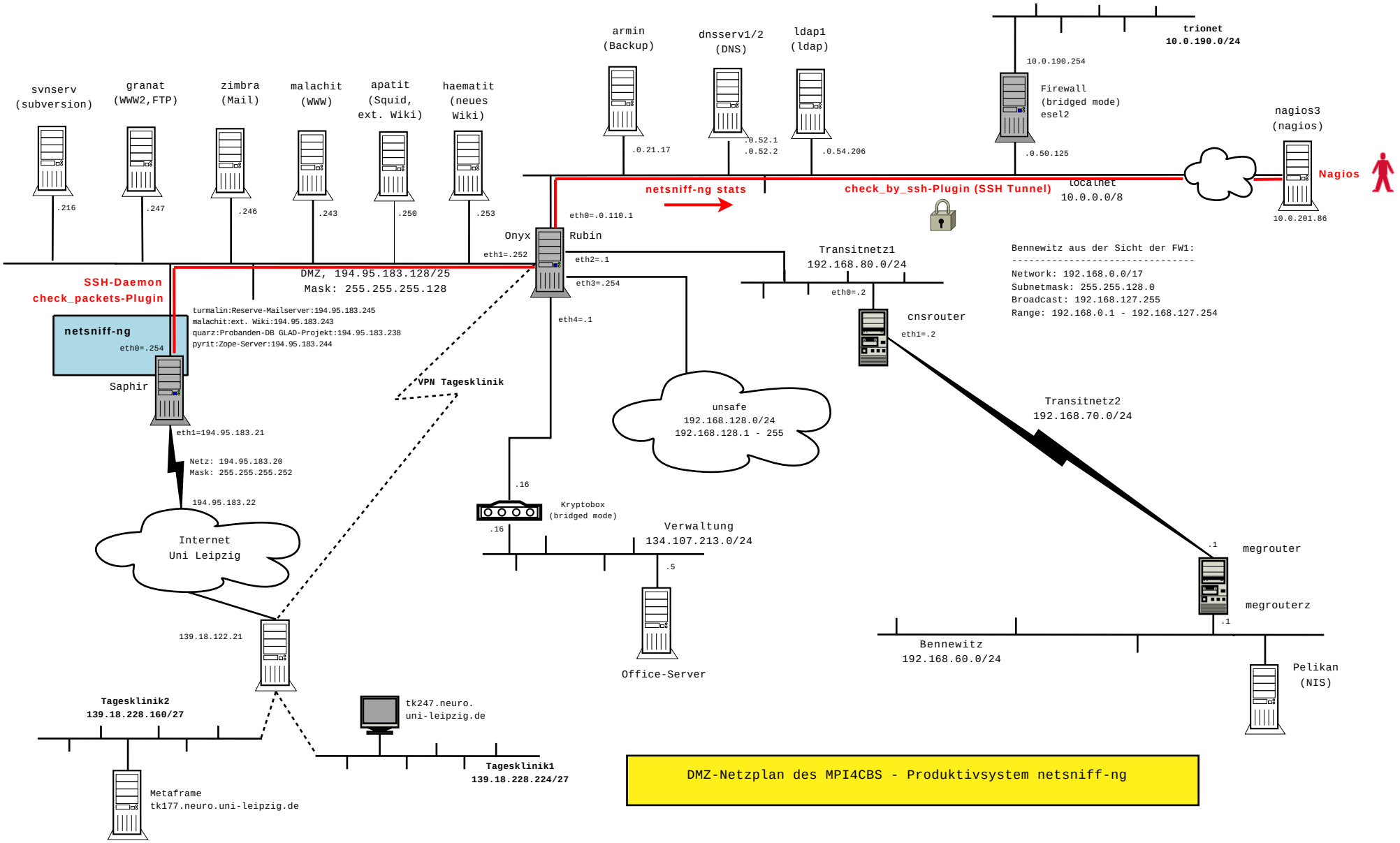


Abbildung 29: Netzplan der DMZ mit Produktiveinsatz von netsniff-ng

Zum Produktiveinsatz kommt *netsniff-ng* auf dem Institutsrouter, der den gesamten internen Institutsverkehr ins Internet leitet. Ein Netzplan in Abbildung 29 [S. 75] zeigt die Konstellation der DMZ mit dem am äußeren Ende befindlichen Router *saphir*, auf dem das Programm läuft. Dieser ist über das Netz der *Universität Leipzig* an das *Deutsche Forschungsnetz* (DFN) angebunden.

4.3.2 check_packets

netsniff-ng wird mit einem weiteren Programm namens check_packets geliefert. Dies ist ein Unix Domain Socket Client, der, wie in Abbildung 30 [S. 77] zu sehen, zur indirekten Interaktion mit Nagios gedacht ist und im Rahmen dieser Arbeit entwickelt wurde.

Da der Sniffer auf dem Institutsrouter läuft, wurde auf einen eigenständigen TCP/IP-Server innerhalb *netsniff-ng* aus Sicherheitsgründen verzichtet.
Die Statistiken müssen allerdings über geeignete Mittel noch zum Nagios-Server transportiert werden. Da sich aber der Nagios-Server im internen sicheren Netz befindet, muss die Firewall *onyx* am gegenüberliegenden Ende des Routers, die ins interne Netz zeigt, überwunden werden.

Nagios selbst bietet zum Datentransport vom Remotehost zum Server die Erweiterung NRPE und das Plugin check_by_ssh an. Da jedoch für NRPE ein weiterer Port an der Firewall freigeschalten werden müsste (und somit neue Sicherheitslücken entstehen könnten), fällt die Wahl auf check_by_ssh, das den am Router verfügbaren Port 22 verwendet.

check_packets kann sowohl eine ausführliche Statistik über empfangene Pakete dem Nutzer anzeigen, als auch eine für Nagios integrierbare (Option -*N*) als Einzeiler.

4.3.3 BPF Routinen

Die nachfolgend aufgelisteten *Berkeley Packet Filter* Programme wurden dabei für weitere Untersuchungen innerhalb dieser Arbeit entwickelt und mitgeliefert. Die jeweiligen Routinen werden an netsniff-ng übergeben und an dessen Socket gebunden. Die mit † versehene Programme werden zur Zeit auf dem Produktivsystem zur Analyse verwendet.

- **all_traffic.bpf**† Abfangen des gesamten Netzwerkverkehrs
- **arp.bpf** Abfangen des ARP-Verkehrs

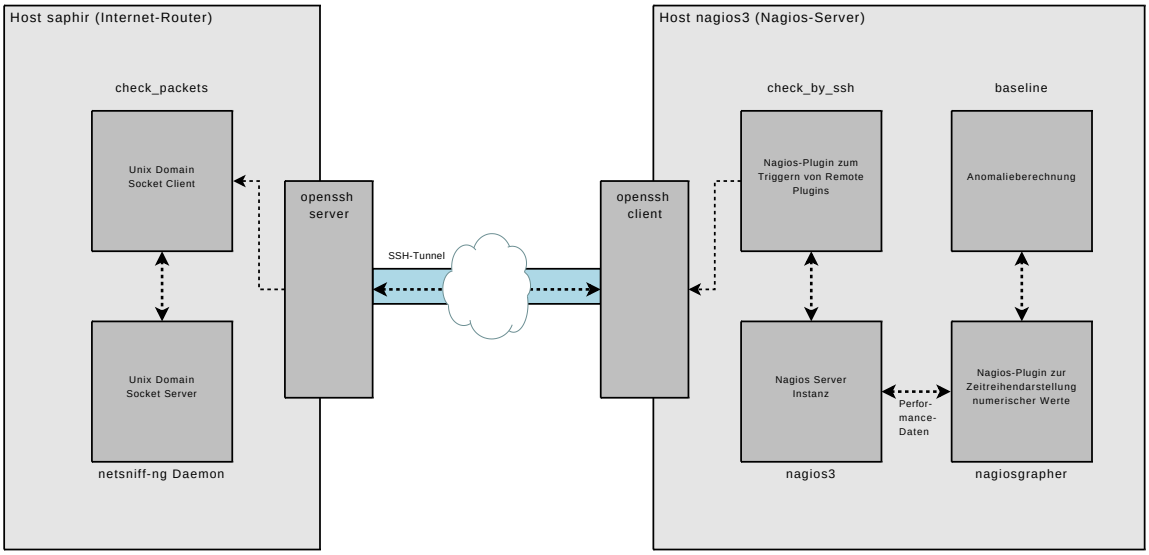


Abbildung 30: Interaktion von netsniff-ng mit Nagios

- **atalk.bpf** Abfangen des AppleTalk-Verkehrs
- **broadcast.bpf**† Abfangen aller Broadcasts
- **http.bpf**† Abfangen des gesamten HTTP-Verkehrs
- **icmp.bpf** Abfangen des gesamten ICMP-Verkehrs
- **imap.bpf** Abfangen des gesamten IMAP-Verkehrs
- **ip_broadcast.bpf** Abfangen von Broadcasts auf IP-Ebene
- **multicast.bpf** Abfangen aller Multicasts
- **pop3.bpf** Abfangen des gesamten POP3-Verkehrs
- **rsync.bpf** Abfangen von Rsync-Verbindungen
- **skype_pre.bpf** Test, Skype UDP-Probings zu erkennen
- **smtp.bpf** Abfangen des gesamten SMTP-Verkehrs
- **ssh.bpf** Abfangen des gesamten SSH-Verkehrs
- **svnserv.bpf** Abfangen des Verkehrs zum Subversion-Server
- **vlan1000.bpf** Abfangen des Verkehrs zum VLAN mit der ID 1000
- **www.bpf**† Abfangen des Verkehrs zum öffentlichen Webserver-Server
- **zimbra.bpf** Abfangen des gesamten Verkehrs zum öffentlichen Mailserver

- **zimbra2.bpf**† Abfangen der E-Mail-Protokolle zum öffentlichen Mailserver (Filter als Beispiel im Anhang V zu finden)

4.4 Vorhersage-Framework „rea“

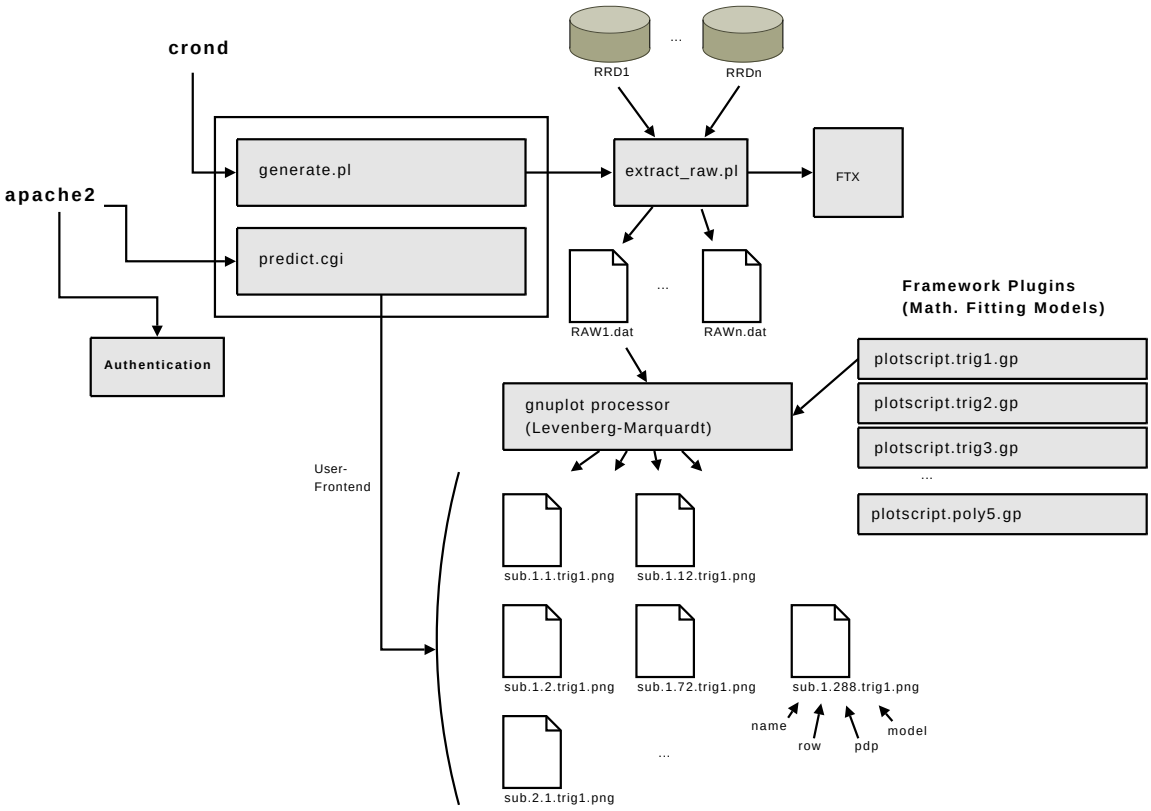


Abbildung 31: Aufbau und Funktionsweise vom Vorhersage-Framework

Das Framework zur Vorhersage namens *rea* (*RRD extrapolation and analysis*) setzt nicht auf NagiosGrapher auf, sondern ist eine Sammlung unabhängiger Perl-Skripte, die in dieser Arbeit entwickelt wurden. Beliebige andere RRDs können somit ebenfalls integriert werden. Die Messdaten, sowie der zugehörige vorhergesagte Graph wird durch das Programm *gnuplot* visualisiert. Zur Approximierung der Daten wird das *Levenberg-Marquardt*-Verfahren angewandt.
Da dies bereits in der Funktion *fit* in *gnuplot* integriert ist, müssen lediglich *gnuplot*-Skripte entwickelt werden, die ein *Fitting* nach dem jeweiligen vorgegebenen Modell betreiben, sowie die Resultate in angemessener Form als Grafik ausgeben. Somit ist das Framework leicht mit weiteren *gnuplot*-Skripten erweiterbar. Die beiden Programme *generate.pl*, sowie *predict.cgi* stellen den Hauptteil des Frameworks dar. Diese unterteilen die Architektur zum einen in das Generieren der Grafiken und zum anderen in einen Zugriff der Daten über ein Webfrontend.

Die Konfiguration beider Dateien erfolgt über eine einzige Konfigurationsdatei. Die Syntax entspricht der von den Konfigurationsdateien des *Source Code Management*-Systems *git* und wird anhand es Perl-Moduls *Config::GitLike* geparst. Der Vorteil hierbei liegt darin, dass mittels diesem Moduls auch Konfigurationsdateien erzeugt oder verändert werden können, sodass das Einrichten (oder eine Migration) unter Umständen auch anhand von Programmen erleichtert werden kann.
Weiterhin ist die aktuell verwendete Konfiguration auch über das Webfrontent einsichtbar (Abbildung 37 [S. 85]), sodass Veränderungen gleich auf Korrektheit kontrolliert werden können.

4.4.1 generate.pl

Mit Hilfe von *generate.pl* wird das angegebene Wurzelverzeichnis nach allen vorkommenden RRDs durchsucht. Über das FTX-Modul werden alle RRA-Datensätze extrahiert und mit Angabe des Zeitparameters in ein für *gnuplot* lesbares Format konvertiert. Dabei wird als Definitionsbereich des jeweiligen Modells nicht die Menge der Unix-Timestamps verwendet, sondern es wird ein zweiter, normalisierter Definitionsbereich in die Rohdatendatei hinzugefügt, sodass keine zu großen Werte zur Approximierung verwendet werden müssen. Dieser „versteckte“ Definitionsbereich ist in der späteren *gnuplot*-Ausgabe nicht sichtbar. Lediglich die umformatierten Timestamps werden als x-Achse dargestellt. Optional können auch die Residuen mit in die Grafikdatei gedruckt werden.

Über einen nächtlichen *CRON-Job* kann das *generate.pl*-Skript alle Grafiken aktualisieren. Die erzeugten Grafiken werden anschließend in den jeweiligen Ordner für die Webschnittstelle transferiert.

Die in Abschnitt 3.3.2 [S. 54] erwähnten Vorhersagemodelle wurden jeweils als *gnuplot*-Skript (*rea-Plugins*) entwickelt.

4.4.2 predict.cgi

Das *predict.cgi* ist ein *Perl-CGI*-Skript, welches die Weboberfläche und damit die Benutzerschnittstelle darstellt. Die durch *generate.pl* erzeugten Grafiken können hierbei angezeigt werden. Auswahlmöglichkeiten bestehen für den jeweiligen Host mit Service, sowie für das zugehörige Modell. Alle damit verbundenen Resultate werden anschließend angezeigt (Abbildung 36 [S. 84]).

4.5 Anomalie-Modulübersicht sowie Schnittstelle zu NagiosGrapher

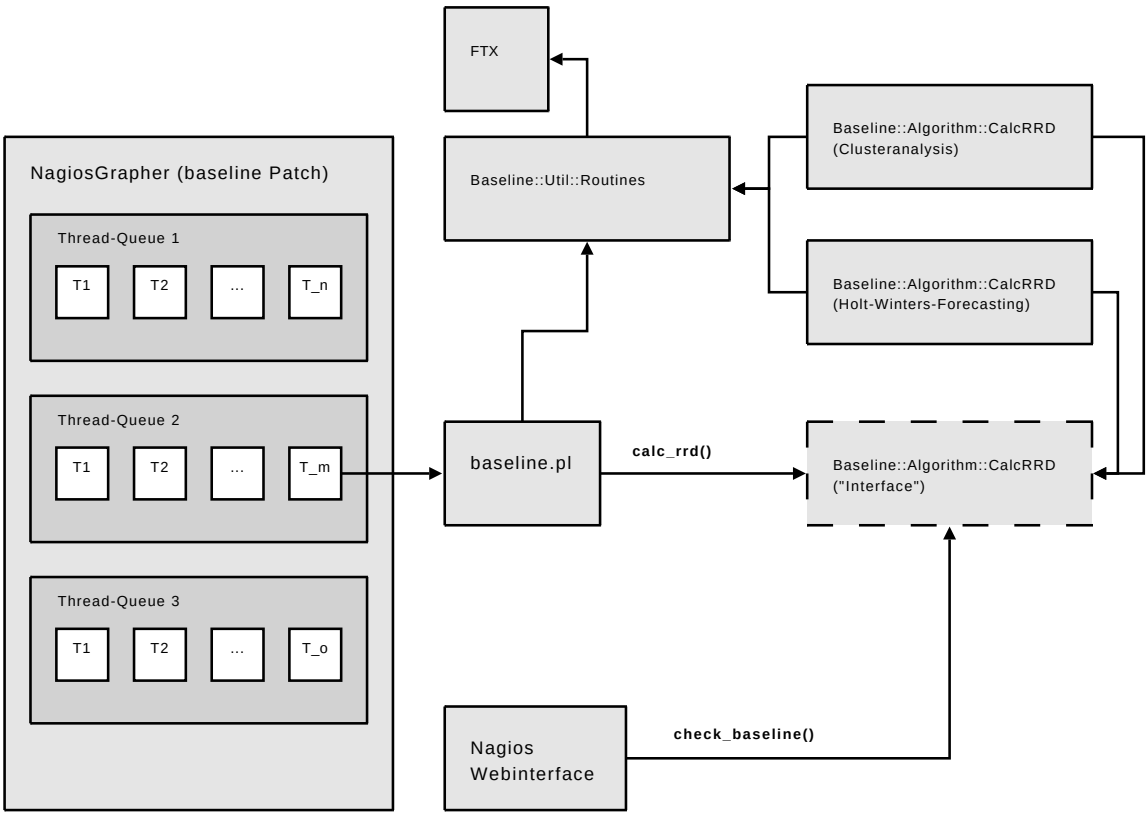


Abbildung 32: Modulübersicht und Interaktion mit NagiosGrapher

Wie in Abbildung 32 [S. 80] zu sehen, wurden innerhalb dieser Arbeit die NagiosGrapher-Module dahingehend gepatched, dass nach jedem RRD-Update ein neuer Thread (*detached*) erzeugt wird, der das ebenfalls in dieser Arbeit entwickelte Skript *baseline.pl* ausführt, welches wiederum die *calc_rrd()*-Routine vom aktuellen *Baseline::Algorithm::CalcRRD*-Modul aufruft.

Für die Threadbehandlung wurden drei Thread-Queues angelegt, die jeweils für sich abzuarbeitende Threads beinhalten. Im *Round-Robin*-Verfahren wird je RRD-Update ein neuer Thread in der aktuellen Queue erstellt. So kann bei einer Mehrkernmaschine eine bessere Skalierung für die Anomalieanalyse erreicht werden, vor allem auch wegen der Unabhängigkeit der einzelnen RRDs.

Das Skript *baseline.pl* ist eher als Vermittler zwischen NagiosGrapher und der jeweiligen Implementierung der Anomalieanalyse zu sehen. Nach dem *calc_rrd()*-Aufruf wurde die reservierte RRD des Algorithmus aktualisiert, sodass ein anschließender

Merge die RRD der Messdaten mit der RRD des Algorithmus für die grafische Anzeige zusammengeführt werden kann. Dies wird über die Routine *merge_rrd()* aus *Baseline::Util::Routines* abgehandelt.

Die beiden in der *Algorithmischen Realisierung* beschriebenen Verfahren zur Anomalieanalyse wurden jeweils als *Baseline::Algorithm::CalcRRD-Module* implementiert.

4.6 Integration in die Nagios-Weboberfläche

Nachfolgende Abbildungen zeigen, wie die implementierten Erweiterungen in die Weboberfläche integriert wurden.

Abbildung 33 [S. 82] stellt den Startbildschirm der Weboberfläche dar. Der Benutzer kann auswählen, ob er zur Nagios-Weboberfläche oder zum Vorhersage-Framework navigieren möchte. Beide Teilsysteme wurden mit Hilfe des *Apache*-Webservers dahingehend konfiguriert, dass ein autorisierter Zugriff nur über *htaccess* (*/etc/apache2/conf.d/*) möglich ist. Es besteht daher auch die Möglichkeit, Benutzer beziehungsweise Benutzergruppen gegen *LDAP* oder *Kerberos* zu authentifizieren.

Auf der „Service Detail"-Übersichtsseite von Nagios (Abbildung 34 [S. 82]) werden automatisch Informationen über erkannte Verhaltensanomalien zusammengetragen und je nach Zustand mit entsprechenden „Ampelfarben" hinterlegt. Ohne sich jede einzelne Grafik der NagiosGrapher-Benutzerschnittstelle ansehen zu müssen, kann so der Administrator über mögliches Fehlverhalten sofort informiert werden. Ausblickend bestünde auch die Möglichkeit, ein kleines Skript zu entwickeln, welches durch eine Kurznachricht über *Jabber* (*XMPP*) den Administrator bei Zustand „Kritisch" informiert.
Die Methode check_baseline() aus Baseline::Algorithm::CalcRRD sorgt für das Zusammentragen der Anomalieinformationen und muss je nach Algorithmus unterschiedlich implementiert werden.

Die *health*-Graphen der Anomaliealgorithmen wurden, wie in Abbildung 35 [S. 83] zu sehen, in die NagiosGrapher-Weboberfläche integriert. Um dies zu ermöglichen, musste NagiosGrapher an einigen Modulen gepatched werden. Je Service existieren somit Anzeigeseiten mit den Messdaten (unter *page* → *data*), sowie weitere Anzeigeseiten mit dem Verhaltenszustand der Messdaten (unter *page* → *health*). Grenzwerte für die Zustände „Warnung" und „Zustand kritisch" werden durch zwei statische Ge-

raden eingezeichnet. Eine Überschreitung von „Zustand kritisch“ stellt folglich eine Verhaltensanomalie innerhalb der Messdaten dar.

Abbildung 36 [S. 84] zeigt die prototypische Weboberfläche des Vorhersage-Frameworks. Über zwei Auswahlboxen werden jeweils Host (*subject*) und Art der Anomalieanalyse (*analysis type*) gewählt und mittels der Schaltfläche *go* zugehörige Graphen angezeigt.
Die durch *gnuplot* erzeugten Diagramme beinhalten zum einen Messdaten aus den RRDs, sowie einen rot gekennzeichneten *Fit* der Messdaten. Für eine Vorhersage wurde entsprechend der Definitionsbereich der Grafik erweitert. Zu einem bestimmten Host werden alle Services angezeigt.

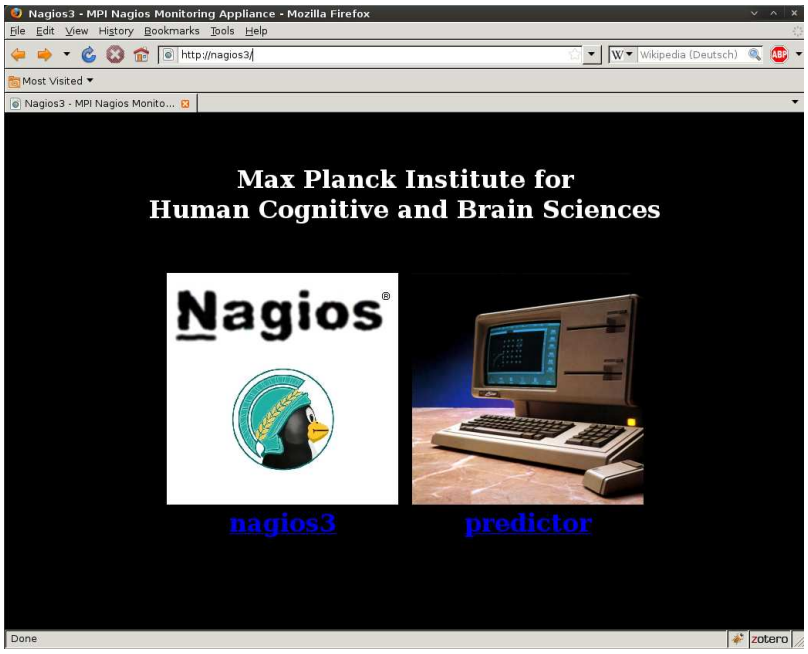


Abbildung 33: Nagios Webinterface - Einstiegspunkt: *Der Nutzer hat hier die Wahl zwischen der Nagios-Weboberfläche (nagios3) und der Schnittstelle zum Vorhersage-Framework (predictor).*

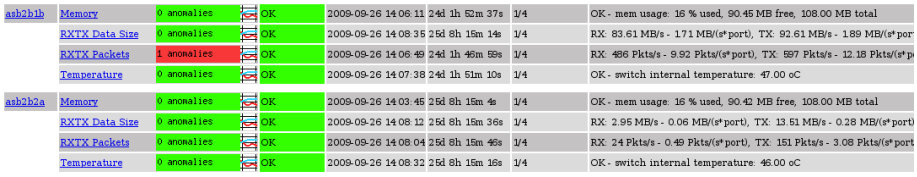


Abbildung 34: Nagios „Service Detail“-Seite mit Anomalieanzeige: *Auf der Übersichtsseite der beobachteten Nagios-Services werden automatisch erkannte Anomalien zum Service anzeigt. Die Hintergrundfarbe des generierten Bildes ist entweder rot, gelb oder grün - je nach Anomaliezustand. Entsprechend muss zu diesem Zweck die Methode check_ baseline() aus Baseline::Algorithm::CalcRRD algorithmenspezifisch implementiert werden.*

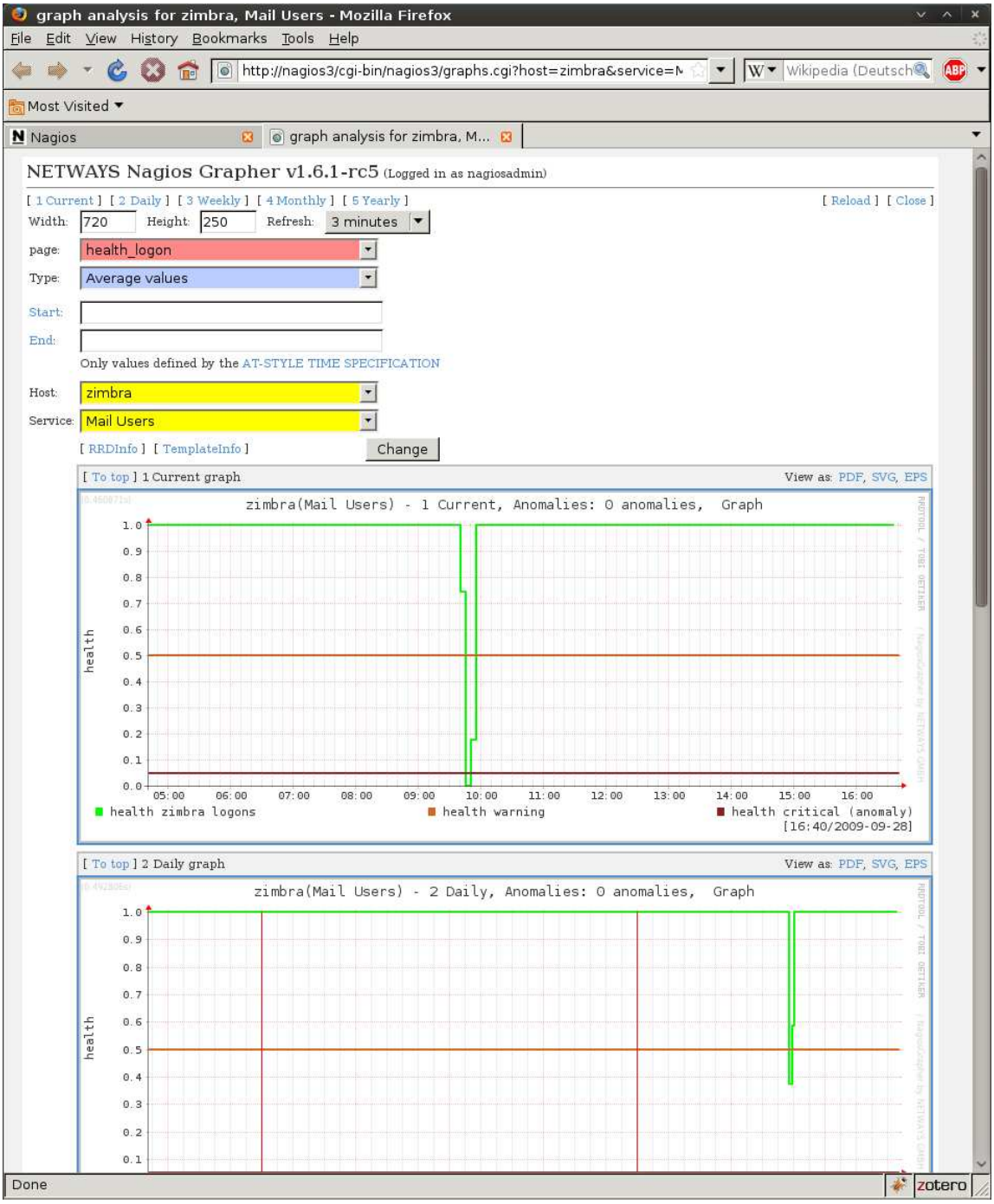


Abbildung 35: NagiosGrapher-Seite mit *health*-Anzeige: *Die durch Baseline::Algorithm::CalcRRD errechneten health-Daten sind von der jeweiligen Service-Seite eines Hosts über die Auswahlbox „page“ abrufbar. Zwei statisch eingezeichnete Geraden geben die Grenze für „Warnung“ sowie „Kritischer Zustand“ an.*

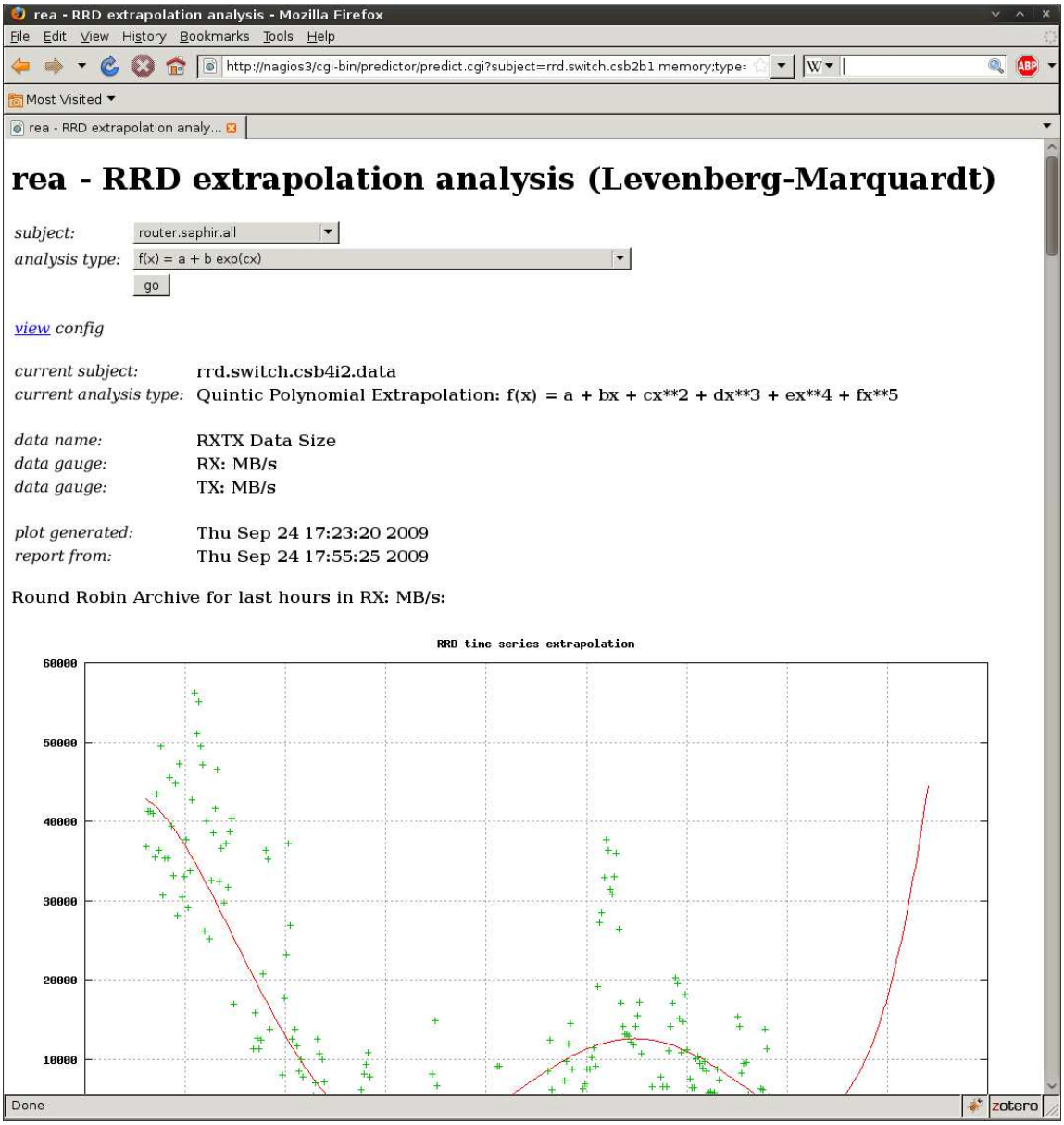


Abbildung 36: Prototypische Weboberfläche vom Vorhersage-Framework: *Der Benutzer kann aus einem „subject“ (hier: Hostname mit jeweiligem Service), sowie einem „analysis type“ (Mathematisches Modell) wählen und mit Bestätigung durch „go“ sich die zugehörigen Graphen anzeigen lassen.*

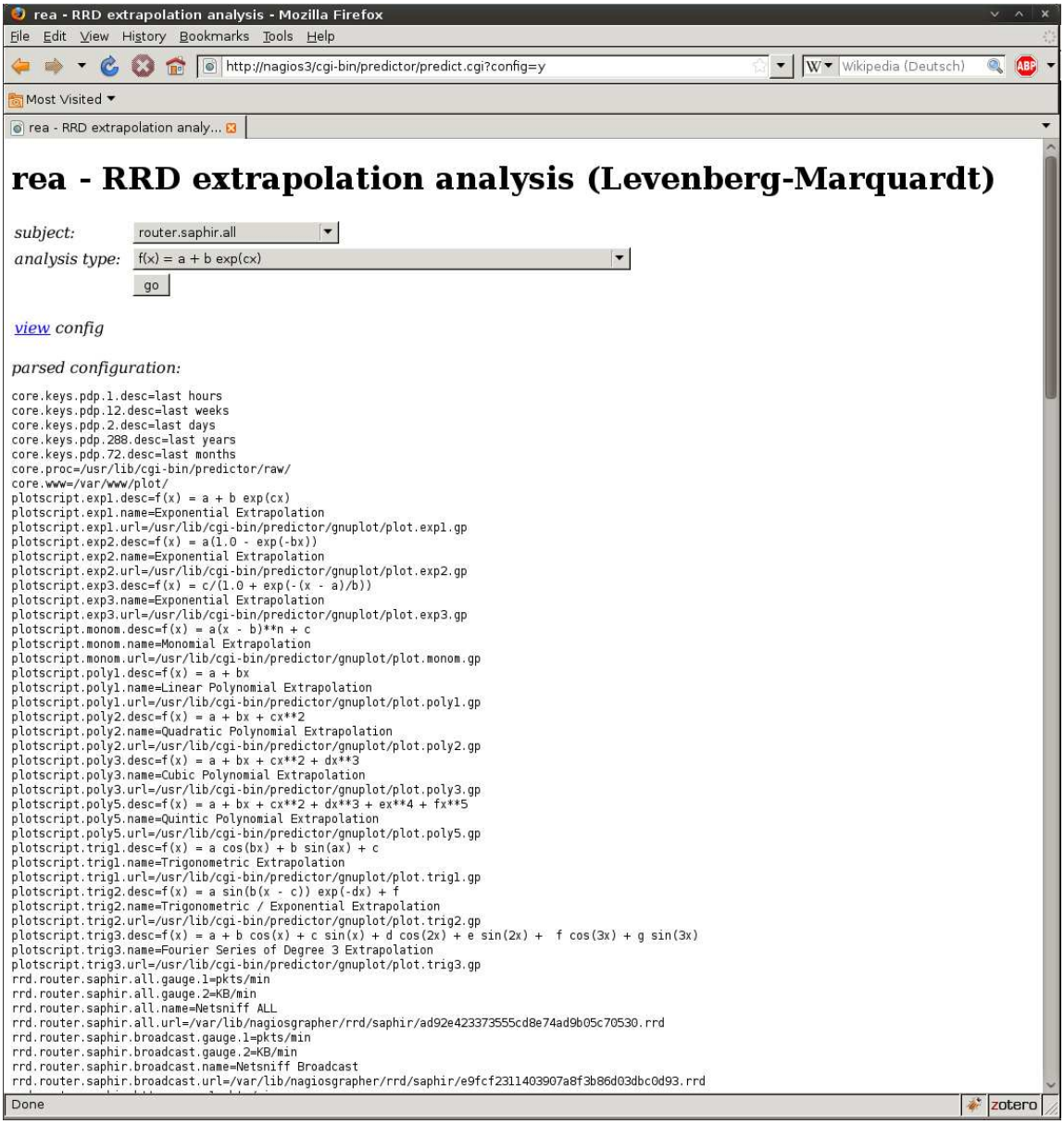


Abbildung 37: Prototypische Weboberfläche vom Vorhersage-Framework, Konfiguration: *Nach der Konfiguration mittels Textdatei kann der Benutzer anhand der Weboberfläche überprüfen, ob seine Eingaben korrekt erkannt wurden. Die Konfiguration erfolgt in der Syntax wie beim „Souce Code Management“-System git. Anhand des Perl-Moduls Config::GitLike wird die Konfiguration geparst. Eine programmtechnische Generierung einer Konfigurationsdatei ist daher ebenfalls möglich.*

5 Schluss

5.1 Zusammenfassung und Diskussion

Anliegen dieser Arbeit war es, anhand von Netzwerkpaketstatistiken des Instituts Algorithmen zur Anomalieanalyse und Zeitreihenvorhersage zu entwickeln und zu evaluieren.

Um die Algorithmen im praktischen Einsatz testen zu können, wurde zuerst das Plugin zum Sammeln und zur Integration der Daten für das Nagios-Framework entwickelt. Dieses läuft auf dem Institutsrouter und greift sehr effizient eintreffende Netzwerkpakete auf Kernelebene ab. Statistiken zur späteren Weiterverarbeitung wurden für HTTP- und SSH-Verkehr sowie Netzwerkverkehr zum Webserver und E-Mail-Server, als auch für den gesamten Verkehr mit Hilfe dieses entwickelten Plugins erzeugt.

Da es im *Monitoringsystem* Nagios und in der Erweiterung NagiosGrapher keine Möglichkeit gab, Daten auf Anomalien zu prüfen oder vorherzusagen, wurde daraufhin zur Anomalieanalyse eine Schnittstelle und außerdem zur Messdatenvorhersage ein Framework entwickelt, die es überhaupt erst ermöglichen, verschiedene Algorithmen beziehungsweise Modelle in eine solche Architektur zu integrieren.

Ausgangspunkt für die Anomalieerkennung war das innerhalb der RRDs eingesetzte Holt-Winters-Forecasting. Da diese Option nicht innerhalb NagiosGrapher genutzt werden kann, wurde deshalb das Verfahren anhand der Schnittstelle zusammen mit der Erweiterung nach Miller sowie einer eigenen Idee, einem Spezialfall der Clusteranalyse zur Intervallschätzung, implementiert.

Zur Messdatenvorhersage wurde das robuste *Levenberg-Marquardt*-Verfahren zur nichtlinearen Ausgleichsrechnung mit verschiedenen Modellen (lineare, polynomiale, monomiale, exponentielle und trigonometrische Fits mit unter anderem auch Fourierreihen) betrachtet und innerhalb des enstandenen Frameworks umgesetzt.

Eine Gegenüberstellung des Holt-Winters-Forecastings mit der Clusteranalyse zur Intervallschätzung ergab, dass zuerst genanntes Verfahren eine bessere Vorhersage der Anomalien liefert.
Während das Holt-Winters-Forecasting die Messdaten als Zeitreihe interpretiert und sie in seine Komponenten zerlegt, wird bei der Clusteranalyse die Strategie verfolgt,

Messdaten zunächst in zwei Klassen zu unterteilen (Daten von Montag bis Freitag, Daten von Samstag und Sonntag) und innerhalb einer solchen Klasse Daten eindimensional abzutragen. Diese Daten werden zu Gruppen zusammengefasst. Dementsprechend ist diese Vereinfachung auch der Schwachpunkt dieses Verfahrens, da hier an speziellen Offsets einer Saison Anomalien unter Umständen nicht erkannt werden können und diese innerhalb der zulässigen Intervalle fallen.
Holt-Winters-Forecasting hingegen betrachtet und gewichtet auch Messpunkte der vorherigen Saison (bei gleichem Saisonoffset), sodass starke Abweichungen dieser Punkte dahingehend ins Gewicht fallen, dass ein anormales Verhalten erkannt wird. Schlussfolgernd sollte daher das Holt-Winters-Forecasting (mit Verbesserung nach Miller) verwendet und ausblickend mit der robusteren Variante unter Hinzunahme des Kalman-Filters (beschrieben in [Gelp 07]) ergänzt werden.

Dank der entwickelten Schnittstelle können weitere Verfahren zur Anomalieanalyse in der Praxis evaluiert werden. Der in [Gent 04] beschriebene Ansatz, Anomalien mit Hilfe von *Kerneldichteschätzungen* zu erkennen, wird beispielsweise bei auf Statistik basierenden *Intrusion Detection Systemen* angewandt. In Ausblick zu stellen wäre es, sowohl den Ansatz der *Kerneldichteschätzung* als auch eine weitere Methode, nämlich der Untersuchung anhand *Künstlich Neuronaler Netze*, dem Holt-Winters-Forecasting gegenüberzustellen.

Die Modellwahl des Vorhersage-Frameworks bezieht sich in dieser Arbeit auf grundlegende Modellklassen wie lineare, polynomiale oder exponentielle Fits. Einige Vertreter dieser Klassen wurden herangezogen und auf die Netzwerkpaket-Messdaten angewandt. Die Schwierigkeit der nichtlinearen Ausgleichsrechnung ist dabei ein Modell zu finden, welches annähernd das Verhalten der echten Daten widerspiegelt. In der Praxis hat sich herausgestellt, dass die Klassen der Polynome und Monome oft eine brauchbare Prognose liefern. Versuche mit exponentiellen Fits hingegen ergaben bei den verwendeten Netzwerkpaketstatistiken im Allgmeinen eher schlechte Ergebnisse, da ein entsprechendes annähernd-exponentielles Verhalten bei den Messdaten nicht existiert. Für einen angemessenen trigonometrischen Fit sind die Daten für die untersuchten Modelle zu stochastisch, sodass die Modellparameter eher einem linearen Modells (ohne Anstieg) gleichen und sehr schwache Oszillationen besitzen. Aufgrund zu hoher Modellresiduen können keine Saisonalitäten oder ähnliche Grundmuster in das trigonometrische Modell abgebildet werden. Im Speziellen könnte anhand einer *Fast Fourier Transformation* die Frequenzspektren genauer untersucht und angemessene Startwerte an *gnuplot* übergeben werden, sodass das *Levenberg-Marquardt*-Verfahren bei der Anwendung einer Fourierreihe als Modell konvergieren kann.

Im Allgemeinen sind daher bei wenig Kenntnis über den Prozess Modelle der Monome und Polynome zu favorisieren, wobei zu beachten ist, dass der Grad des Polynoms aufgrund möglicher Oszillationen nicht zu hoch gewählt werden darf. Bei vorhandener Kenntnis über ein mögliches Verhalten sollte ein entsprechendes Modell entwickelt werden, um geringere Residuen zu erhalten.
Im Speziellen, bezogen auf die Netzwerkpaketdaten, ergaben das Modell der Monome

$$f(x) = a(x-b)^n + c, \qquad a, b, c, n \in \mathbb{R}$$

und ein spezielles exponentielles Modell

$$f(x) = \frac{c}{1 + e^{-\frac{x-a}{b}}}, \qquad a, b, c \in \mathbb{R}$$

gute Ergebnisse.

Künftig sollten die Prozesse längerfristig beobachtet und Kombinationen aus verschiedenen Modellklassen entwickelt und getestet werden, um Vorhersagefehler weiter zu minimieren.

Denkbar wäre auch die Entwicklung eines *Evolutionären Algorithmus*, welcher für eine angemessen große Zahl an zeitdiskreten Messdaten (mehr als in den RRDs vorhanden) selbstständig für einen konkreten Prozess ein mathematisches Modell erstellt. Dieses Optimierungsproblem könnte auf die Summe der Residuen angewendet werden. Innerhalb der verschiedenen Generationen werden einzelne Komponente eines Modells zufällig hinzugefügt und Generationen mit schlechten Residuen(-summen) nicht weiter verfolgt. Das entstandene Modell könnte anschließend ins das Framework integriert werden.

5.2 Danksagung

Mein besonderer Dank gilt an dieser Stelle Herrn Prof. Dr.-Ing. Dietmar Reimann von der Hochschule für Technik, Wirtschaft und Kultur Leipzig sowie Herrn Dr. Helmut Hayd vom Max-Planck-Institut für Kognitions- und Neurowissenschaften Leipzig, die mir das Thema dieser Arbeit ermöglicht haben und mir bei der Betreuung zur Seite standen.

Weiterhin möchte ich mich bei Herrn Dr. Karsten Müller vom Max-Planck-Institut für Kognitions- und Neurowissenschaften Leipzig, bei Herrn Dr. Marek W. Gutowski von der Polnischen Akademie der Wissenschaften Warschau sowie bei meinen Kommilitonen für Gedankenaustausch und konstruktive Anregungen bedanken.

A Inhalt der CD

- **Bachelorarbeit**, enthält diese Arbeit im PostScript-Format und als LaTeX-Quellcode
- **Anwendung**, alle im Rahmen dieser Arbeit entwickelten Programme mit zugehörigen Quelltexten
- **Literatur**, ein Teil der verwendeten Literatur

B Resultate aus Vorhersage und Anomalieanalyse

Nachfolgend sind einige Resultate aus Datenvorhersage und Anomalieerkennung zu finden. Die Messdaten wurden mit dem in dieser Arbeit entwickelten Plugin *netsniff-ng* erstellt.

Abbildung 38: Resultate der Clusteranalyse zur Intervallschätzung: netsniff-ng, SSH-Verkehr: *Gegenübergestellt sind jeweils Messwerte mit health-Graphen. Aufgrund des geringen Verkehrs werden bereits kleine Änderungen als Warnung gedeutet. Eine echte Anomalie wurde zwischen acht und neun Uhr korrekt erkannt. Zu vermuten ist ein Dateitransfer vom Institutsnetz nach aussen per scp. Ob von einer echten Gefahr ausgegangen werden kann, liegt im Ermessen des Betrachters und kann nur durch Erfahrungswerte beurteilt werden. Würde beispielsweise eine Tauschbörse im internen Netz oder in der DMZ durch einen kompromitierten Server eröffnet werden, so kann dies hierüber in den meisten Fällen erkannt werden.*

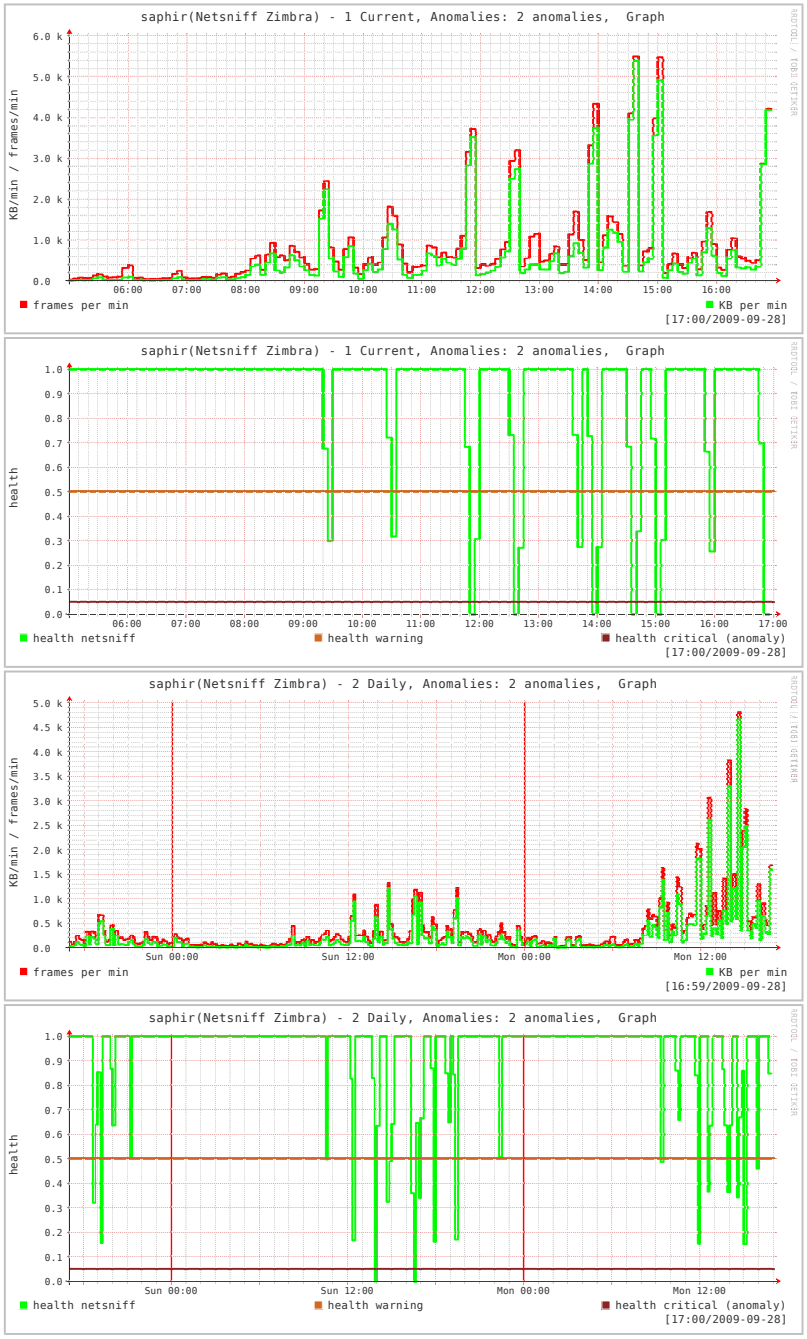


Abbildung 39: Resultate der Clusteranalyse zur Intervallschätzung: netsniff-ng, Verkehr zum Zimbra-Mailserver: *Gegenübergestellt sind jeweils Messwerte mit health-Graphen. In der oberen Ansicht existieren einige Maximas, welche als Anomalien gedeutet werden. Denkbar wäre allerdings, dass zu diesen Zeitpunkten E-Mail Anhänge vom Server herunter- oder hochgeladen wurden, sodass diese Mehrzahl an Frames/min nicht unbedingt Anomalien darstellen müssen. In der unteren Anischt werden beispielsweise Sonntags Anomalien erkannt. Da bei der Clusteranalyse Daten von Wochentagen separiert und somit Samstage und Sonntage zu einer Datengruppe zusammengefasst werden, sind diese Messdaten innerhalb der Gruppe mit einem verhältnismäßig hohem Datendurchsatz als Anomalie korrekt zu deuten. Eine Erklärung könnte darin zu finden sein, dass beispielsweise an diesem Sonntag eine größere Zahl an Mitarbeitern im Institut tätig war.*

Abbildung 40: Resultate der Clusteranalyse zur Intervallschätzung: netsniff-ng, Broadcast-Verkehr: *Gegenübergestellt sind jeweils Messwerte mit health-Graphen. Als Warnung, jedoch noch nicht als Anomalie werden die beiden Ausreißer im oberen Bild erkannt, da kurzzeitig in etwa 30% mehr Broadcast-Frames am Netzwerkinterface eingetroffen sind, als dies sonst der Fall war. Aufgrund der Datenverdichtung ist zu erkennen, dass am Sonntag vereinzelt auf anormales Broadcastverhalten interpretiert wurde. Dieser Umstand kann wiederum mit den Anomalien beim Zimbra-Server korrelieren. Weiterhin könnten aufgrund der Datenseparierung beim Clusterverfahren diese Maxima oder Minima bisher nicht während Samstagen oder Sonntagen eingetreten sein.*

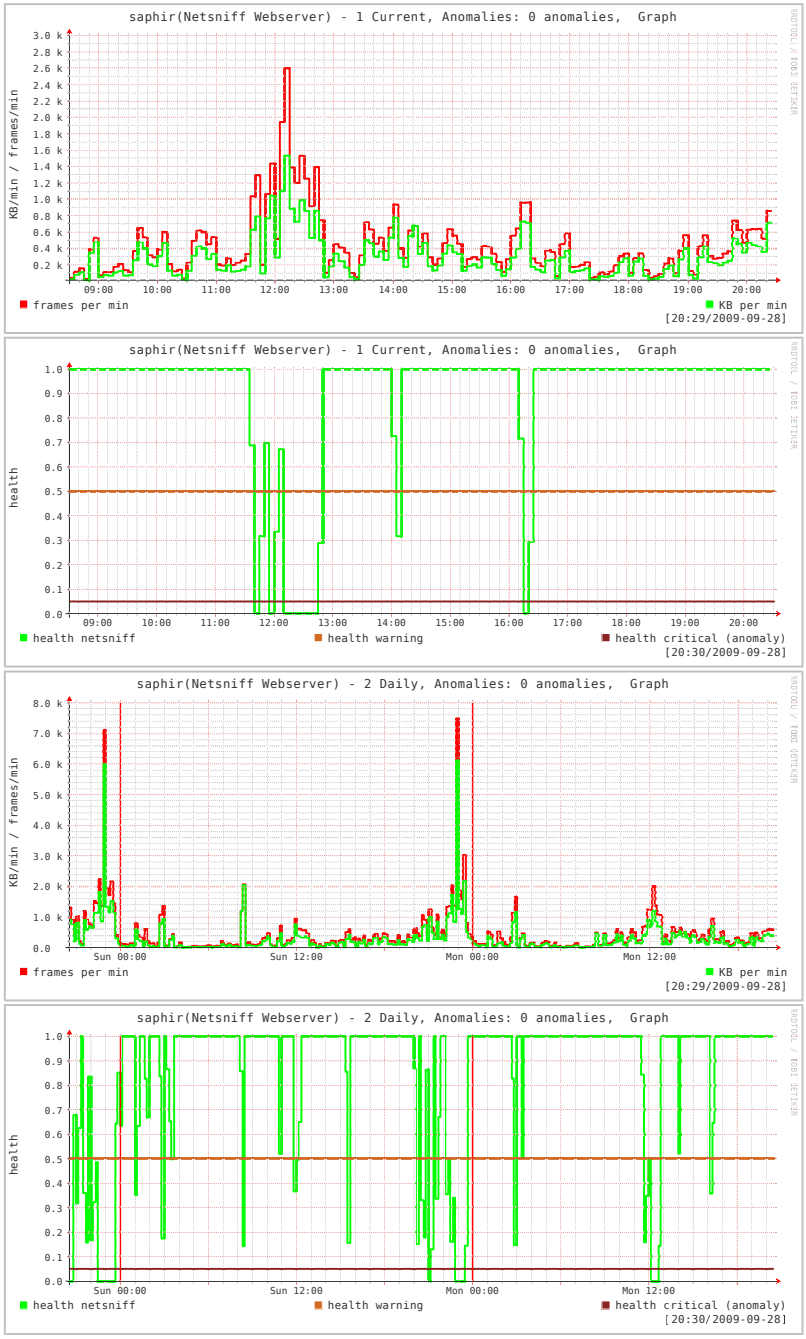


Abbildung 41: Resultate der Clusteranalyse zur Intervallschätzung: netsniff-ng, Verkehr zum WWW-Server: *Gegenübergestellt sind jeweils Messwerte mit health-Graphen. Im oberen Graphen wurden bei den Frames anormales Verhalten entdeckt, da vor allem zur Mittagszeit vergleichsweise hoher Datendurchsatz stattfand. Die Datenmengen sind allerdings nicht als kritisch zu betrachten. Viel interessanter hingegen sind die im unteren Bild auftretenden Anomalien nachtsüber von Samstag zu Sonntag sowie von Sonntag zu Montag. Maximas (fast) gleicher Quantität sind zu identischen Uhrzeiten zu finden und wurden korrekt als Verhaltensanomalien erkannt.*

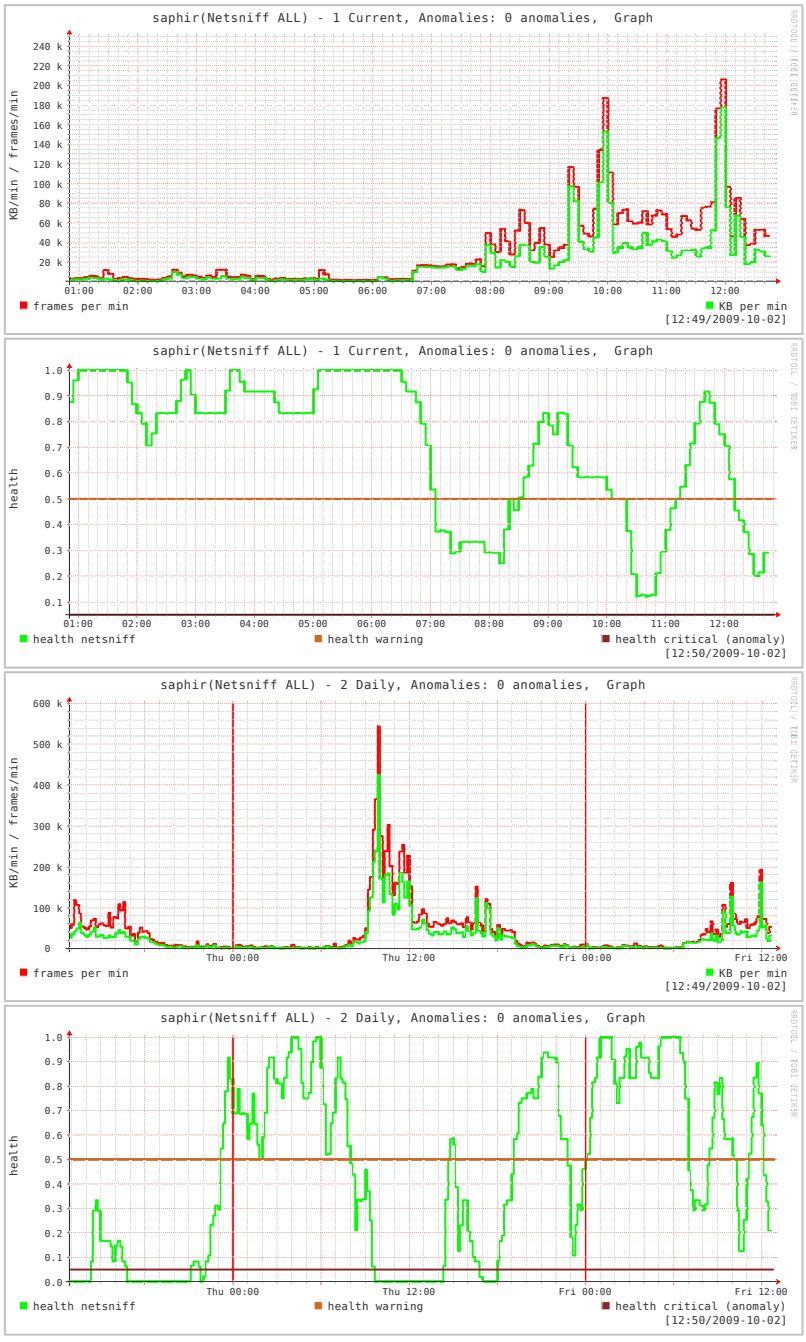


Abbildung 42: Resultate des Holt-Winters-Forecasting: netsniff-ng, Gesamter Verkehr: *Gegenübergestellt sind jeweils Messwerte mit health-Graphen. Da im Gegensatz zu den Vortagen am Donnerstag um den Zeitraum von 12 Uhr verhältnismäßig viel Datendurchsatz (KB/min) über den Router ging, wurde der Schwellwert von zwölf aufeinanderfolgenden Anomalien erreicht, sodass der health-Wert auf null sank. Im Laufe des Freitags hat sich das Verhalten wieder etwas normalisiert.*

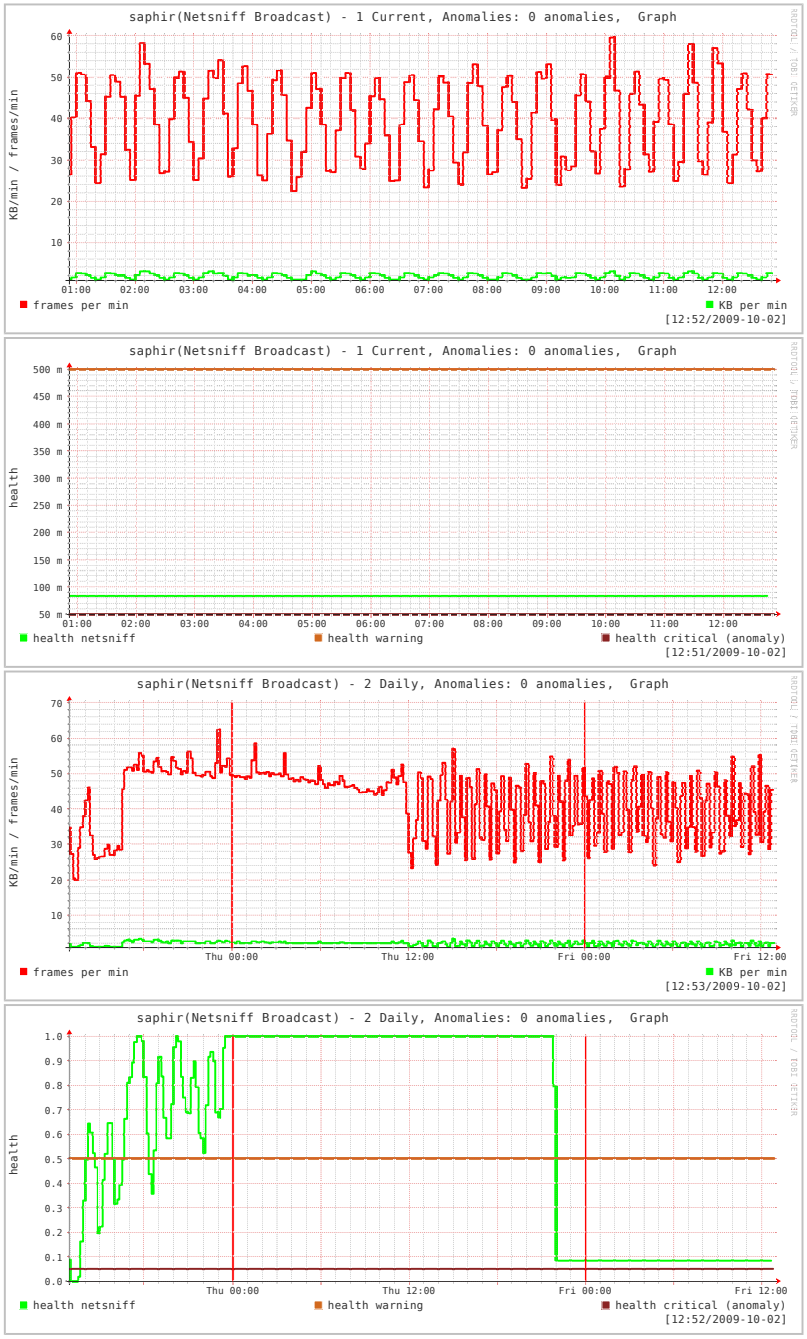


Abbildung 43: Resultate des Holt-Winters-Forecasting: netsniff-ng, Broadcast-Verkehr: *Gegenübergestellt sind jeweils Messwerte mit health-Graphen. Zu sehen ist jeweils die health der KB/min. Dieses oszillierende Verhalten im oberen Bild hat eine dauerhaft schlechte health-Bewertung (jedoch noch keine Anomalie). Ursache dafür ist im unteren Bild zu erkennen. Der Donnerstag Vormittag zeigt ein interessantes Broadcasting-Verhalten der Frames auf, welches in den bisher gemessenen Daten sogar ein globales Maximum (bei Datenverdichtung anhand des Durchschnitts) darstellt. Dies zog somit auch die KB/min-Rate innerhalb dieses Zeitraumes auf ein fast konstant höheres Niveau. Bei einer Saisonalität von einem Tag hat dies zur Folge, dass bei einem kleinen Confidence-Band häufig Anomalien erkannt werden. Da allerdings ein Schwellwert gesetzt ist, treffen die lokalen Maximas im oberen Bild auf ein Normalverhalten, sodass dies insgesamt in einer „Warnung“ resultiert.*

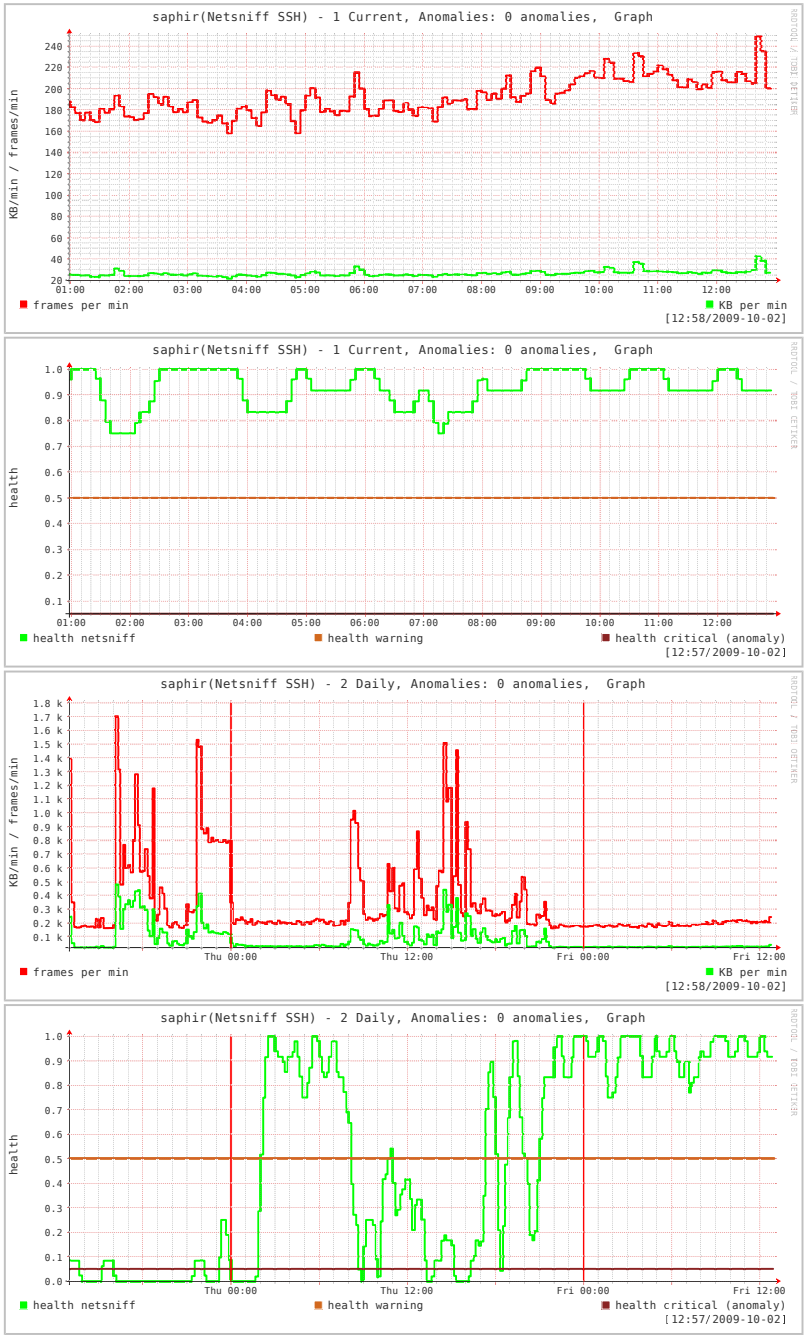


Abbildung 44: Resultate des Holt-Winters-Forecasting: netsniff-ng, SSH-Verkehr: *Gegenübergestellt sind jeweils Messwerte mit health-Graphen. Wie in den health-Graphen zu erkennen wird in etwa ein konstantes Niveau von circa 200 Frames/min erwartet. Die Ausreisser von Donnerstag Nacht und im Laufe des Tages werden daher mit einer schlechten health beurteilt. Bei längefristigem Betrachten der Datenraten ist zu erkennen, dass nur gelegentlich SSH-Verkehr nach aussen geht und sonst ein fast konstant niedriges Level vorherrscht. Dies bedeutet wiederum, dass bereits eine Dateiübertragung nach aussen folglich zurecht in einer Anomalie enden kann.*

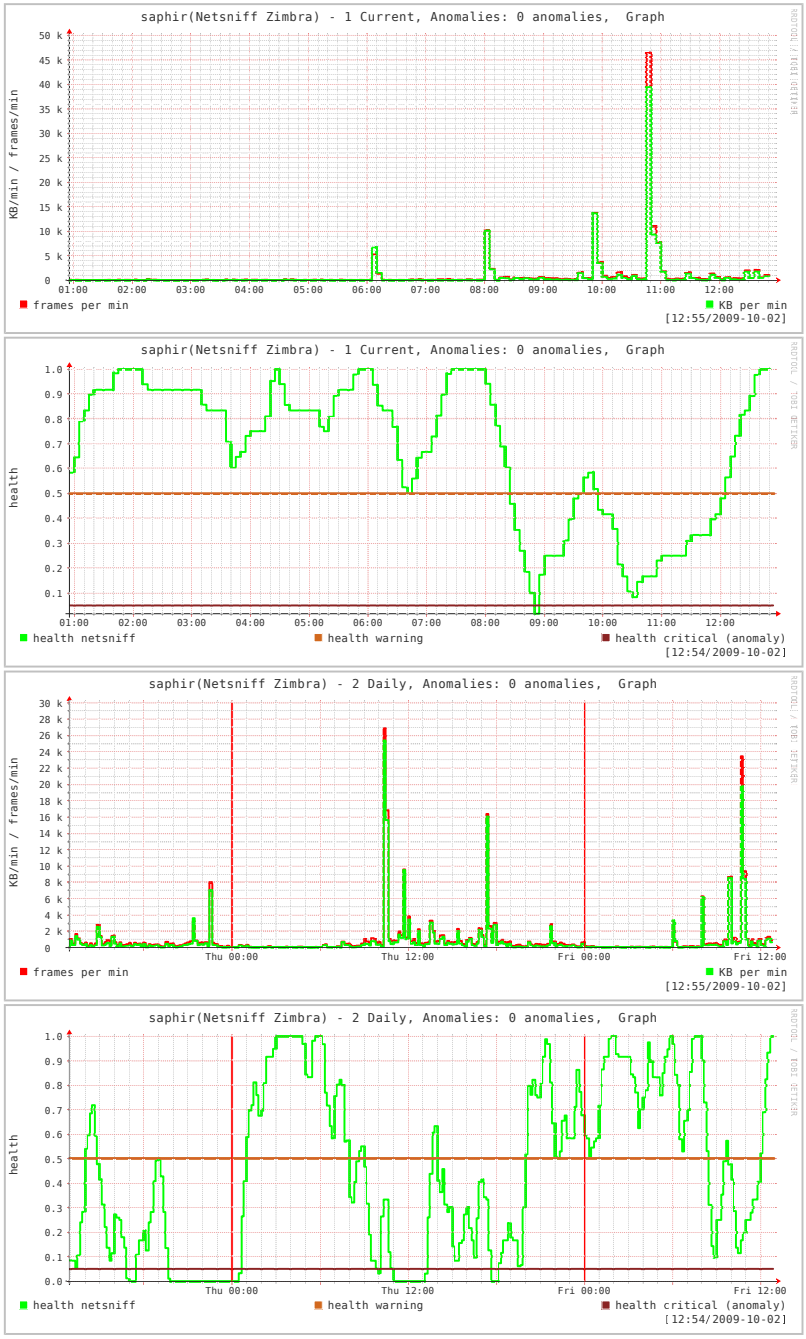


Abbildung 45: Resultate des Holt-Winters-Forecasting: netsniff-ng, Verkehr zum Zimbra-Mailserver: *Gegenübergestellt sind jeweils Messwerte mit health-Graphen. Im Laufe des Freitags sinkt die health-Bewertung der Frames/min zu einem kritischen Level, da ein verhältnismäßig zu geringer Datenverkehr zum Zimbra-Mailserver vorherrscht. Weiterhin interessant ist die Benutzung des Mailservers gegen Mittwoch Nacht. Da vermutlich Mitarbeiter aus gegebenen Anlass spät Abends E-Mails versendet oder auf neue eintreffende geprüft haben, ist hierdurch eine Anomalie zustande gekommen, da dies in den Vortagen nicht der Fall war.*

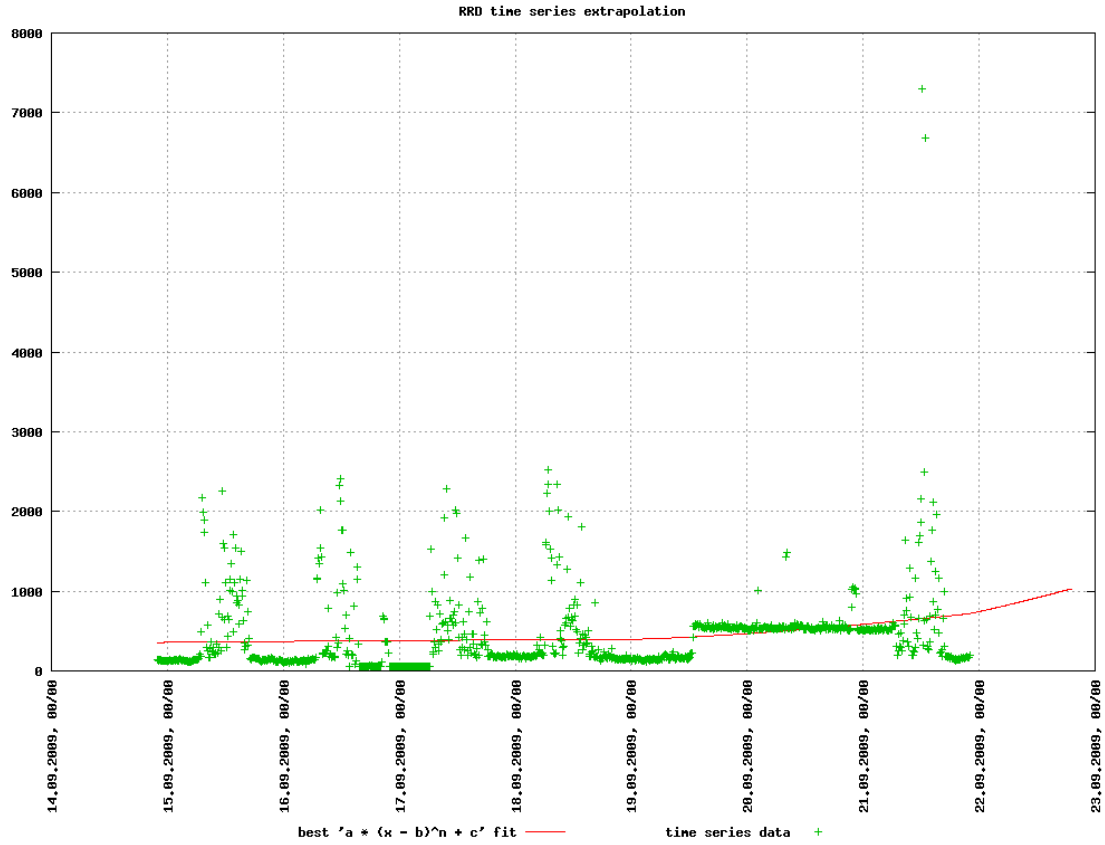


Abbildung 46: Resultate des Vorhersageframeworks, Monomiale Extrapolation: netsniff-ng, SSH-Verkehr in Pkts/min

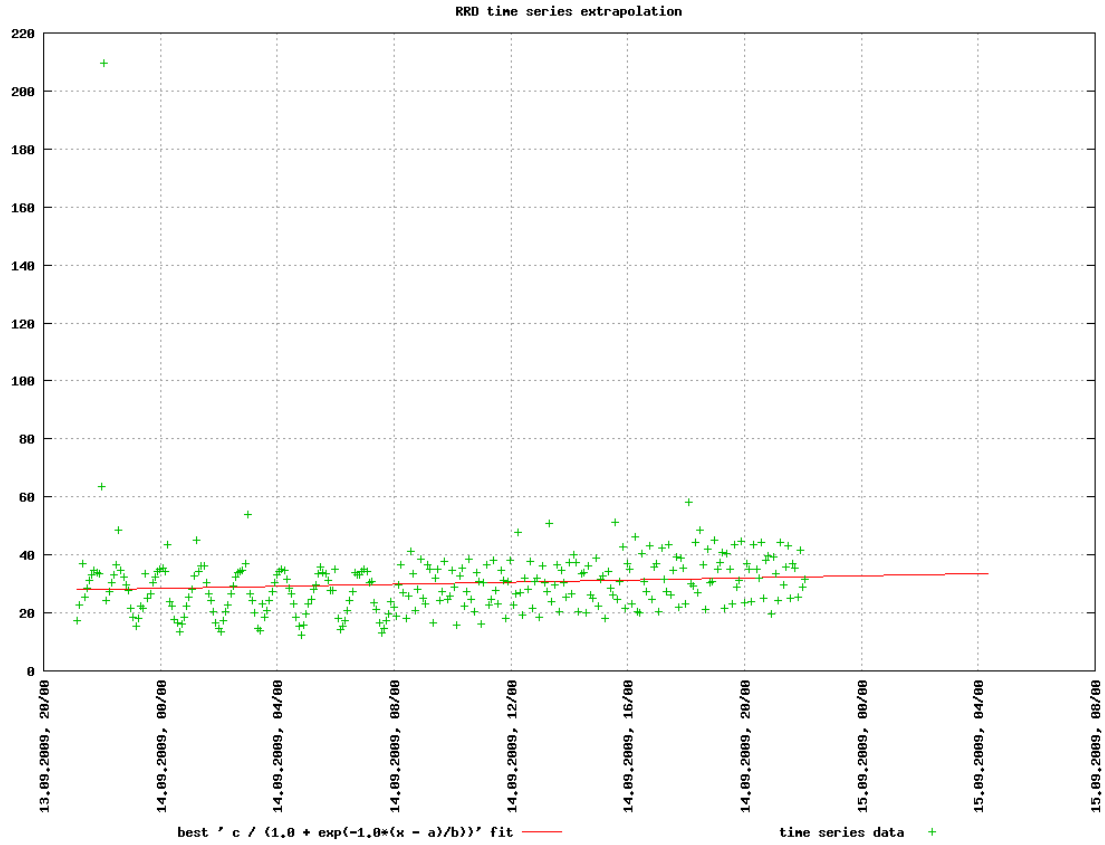


Abbildung 47: Resultate des Vorhersageframeworks, Exponentielle Extrapolation: netsniff-ng, Broadcast-Verkehr in Pkts/min

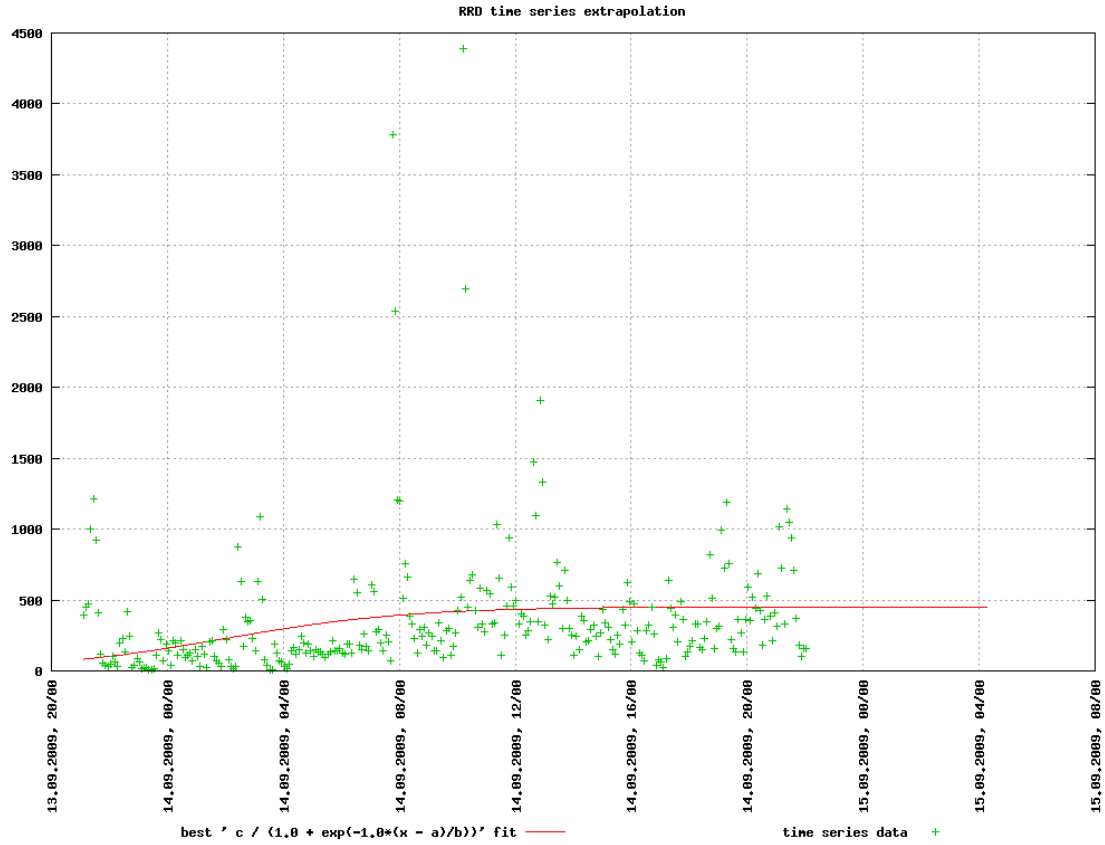


Abbildung 48: Resultate des Vorhersageframeworks, Exponentielle Extrapolation: netsniff-ng, Verkehr zum Webserver in KB/min

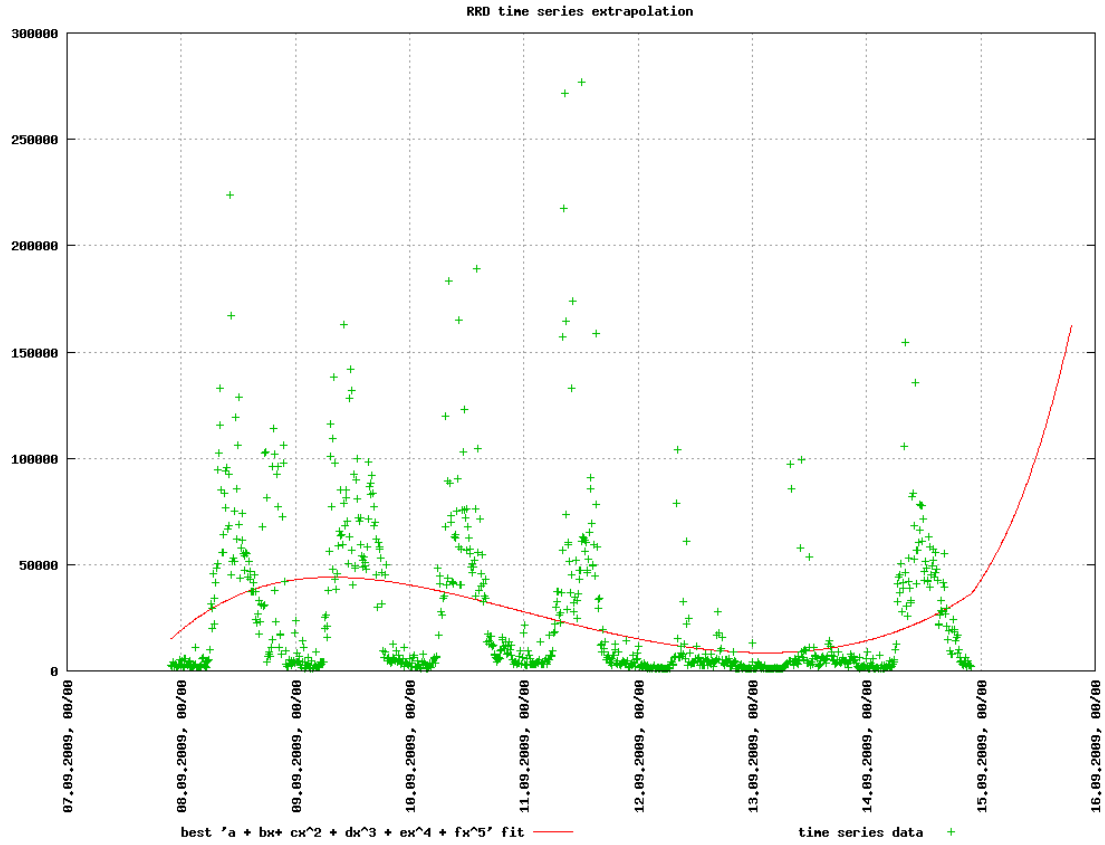


Abbildung 49: Resultate des Vorhersageframeworks, Polynomiale Extrapolation (Grad 5): netsniff-ng, Gesamter Verkehr in Pkts/min

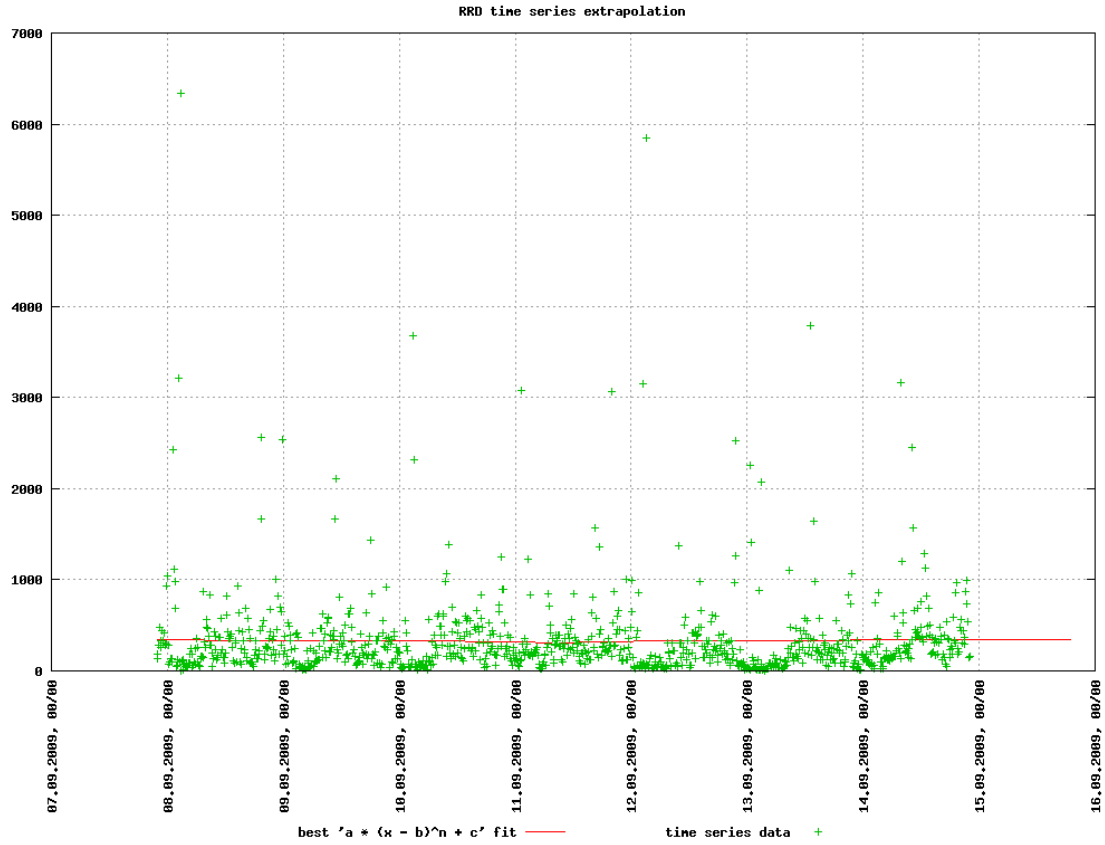


Abbildung 50: Resultate des Vorhersageframeworks, Monomiale Extrapolation: netsniff-ng, Verkehr zum Webserver in Pkts/min

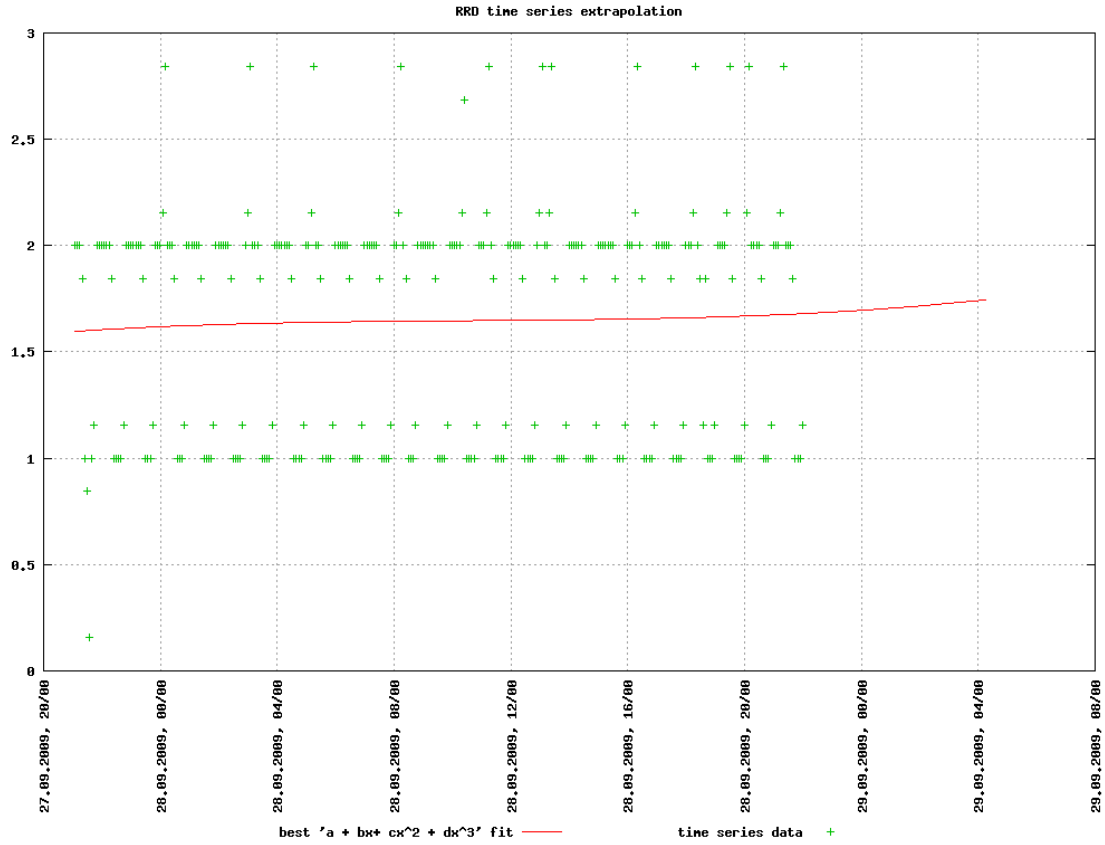


Abbildung 51: Resultate des Vorhersageframeworks, Polynomiale Extrapolation (Grad 3): netsniff-ng, Broadcast-Verkehr in KB/min

C Paketfilter für den Zimbra-Mailserver

Nachfolgend als Beispiel der im Rahmen dieser Arbeit entwickelte Filter für den Zimbra-Mailserver als BPF-Programmcode und nebenstehend die notwendige Transformation zum Opcode für *netsniff-ng*. Folgende Protokolle werden dabei beachtet und abgefangen:

HTTP (Port 80), IMAP over SSL (Port 993), SMTP over SSL (Port 465), POP3 (Port 110), SMTP (Port 25)

```
label   op      args            |->        op, jt, jf, k

        ldh     [12]                     { 0x28, 0, 0, 0x0000000c },
        jeq     #ETHERTYPE_IP, L1, L2    { 0x15, 0, 24, 0x00000800 },
 L1:    ld      [26]                     { 0x20, 0, 0, 0x0000001a },
        jeq     #0xc25fb7e7, L3, L4      { 0x15, 2, 0, 0xc25fb7e7 },
 L4:    ld      [30]                     { 0x20, 0, 0, 0x0000001e },
        jeq     #0xc25fb7e7, L3, L2      { 0x15, 0, 20, 0xc25fb7e7 },
 L3:    ldb     [23]                     { 0x30, 0, 0, 0x00000017 },
        jeq     #0x84, L5, L6            { 0x15, 2, 0, 0x00000084 },
 L6:    jeq     #0x6, L5, L7             { 0x15, 1, 0, 0x00000006 },
 L7:    jeq     #0x11, L5, L2            { 0x15, 0, 16, 0x00000011 },
 L5:    ldh     [20]                     { 0x28, 0, 0, 0x00000014 },
        jset    #0x1fff, L2, L8          { 0x45, 14, 0, 0x00001fff },
 L8:    ldxb    4*([14]&0xf)             { 0xb1, 0, 0, 0x0000000e },
        ldh     [x + 14]                 { 0x48, 0, 0, 0x0000000e },
        jeq     #0x50, L10, L11          { 0x15, 10, 0, 0x00000050 },
 L11:   jeq     #0x3E1, L10, L12         { 0x15, 9, 0, 0x000003e1 },
 L12:   jeq     #0x1D1, L10, L13         { 0x15, 8, 0, 0x000001d1 },
 L13:   jeq     #0x6E, L10, L14          { 0x15, 7, 0, 0x0000006e },
 L14:   jeq     #0x19, L10, L15          { 0x15, 6, 0, 0x00000019 },
 L15:   ldh     [x + 16]                 { 0x48, 0, 0, 0x00000010 },
        jeq     #0x50, L10, L17          { 0x15, 4, 0, 0x00000050 },
 L17:   jeq     #0x3E1, L10, L18         { 0x15, 3, 0, 0x000003e1 },
 L18:   jeq     #0x1D1, L10, L19         { 0x15, 2, 0, 0x000001d1 },
 L19:   jeq     #0x6E, L10, L20          { 0x15, 1, 0, 0x0000006e },
 L20:   jeq     #0x19, L10, L2           { 0x15, 0, 1, 0x00000019 },
 L10:   ret     #TRUE                    { 0x6, 0, 0, 0x00000060 },
 L2:    ret     #0                       { 0x6, 0, 0, 0x00000000 },
```

Abbildung 52: Filter für ankommende und abgehende SMTP-/IMAP-/HTTP- oder POP3-Pakete vom Internet, die mit dem Zimbra-Mail-Server *(194.95.183.231)* interagieren. *Auf der rechten Seite ist die für den Kernel lesbare Form des BPF-Codes.*

Literatur

[Alle 00] J. Allen. "Cricket Documentation". http://cricket.sourceforge.net/support/doc/, 2000.

[Anon 07] Anonymous. "Shared Memory between User Space and Kernel Space". http://kerneltrap.org/node/14326, 2007.

[Bart 08a] W. Barth, Ed. *Nagios-Anbindung von rrdtool basierten Datenbanken*, Netways Nagios Konferenz, 2008.

[Bart 08b] W. Barth. *Nagios: System and Network Monitoring, 2nd Edition*. No Starch Press, 2008.

[Bege 99] A. Begel, S. McCanne, and S. L. Graham. "BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture". Tech. Rep., University of California, Berkeley, 1999.

[Benv 05] C. Benvenuti. *Understanding Linux Network Internals*. O'Reilly, December 2005.

[Bosl 08] S. Boslaugh and P. A. Watters. *Statistics in a Nutshell*. O'Reilly, 2008.

[Bram 75] K. Brammer and G. Siffling. *Stochastische Grundlagen des Kalman-Bucy-Filters - Wahrscheinlichkeitsrechnung und Zufallsprozesse*. R. Oldernburg Verlag München Wien, 1975.

[Brut 00] J. D. Brutlag. "Aberrant Behavior Detection in Time Series for Network Monitoring". Tech. Rep., Proceedings of the Fourteenth Systems Administration Conference (LISA '00), USENIX, 2000.

[Burn 07] B. Burns, J. S. Granick, S. Manzuik, P. Guersch, D. Killion, N. Beauchesne, E. Moret, J. Sobrier, M. Lynn, E. Markham, C. Iezzoni, and P. Biondi. *Security Power Tools*. O'Reilly, 2007.

[Cama 09] U. A. Camaró. "The CONFIG_PACKET_MMAP option with the PACKET socket interface on 2.4 and 2.6 kernels". Linux Kernel Documentation, Documentation/networking/packet_mmap.txt, 2009.

[Chew 00] E. Chew and P. Greenspun. "Is Suicide at MIT a Poisson Process?". Tech. Rep., Massachusetts Institute of Technology, 2000.

[Corb 04] J. Corbet. "Virtual Memory 1: The Problem". LWN, 2004.

[Cox 65] D. R. Cox and H. D. Miller. *The Theory of Stochastic Processes*. Chapman & Hall, 1965.

[Dahm 08] W. Dahmen and A. Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer, 2008.

[Degi 07] L. Degioanni, M. Baldi, F. Risso, and G. Varenni. "Profiling and Optimization of Software-Based Network-Analysis Applications". Tech. Rep., Dipartimento di Automatica e Informatica - Politecnico di Torino, 2007.

[Demb 08] A. Dembo. *Stochastic Processes (Lecture Notes)*. Department of Statistics, Stanford University, 2008.

[Deri 09a] L. Deri. "Improving Passive Packet Capture: Beyond Device Polling". Tech. Rep., NETikos S.p.A., 2009.

[Deri 09b] L. Deri, Ed. *Modern Packet Capture and Analysis: Multi-Core, Multi-Gigabit, and Beyond*, 06/2009.

[Efro 99] S. Efromovich. *Nonparametric Curve Estimation - Methods, Theory, and Applications*. Springer, 1999.

[Ehm 08] F. Ehm, Ed. *NagiosGrapher - Viewing the history of metrics*, CERN IT-GD-ITR, 2008.

[Eric 08] J. Erickson. *Hacking: The Art of Exploitation, 2nd Edition*. No Starch Press, 2008.

[Fahr 97] L. Fahrmeir, R. Künstler, I. Pigeot, and G. Tutz. *Statistik - Der Weg zur Datenanalyse*. Springer, 1997.

[Flac 07] M. Flacke. "Vergleich NRPE - check_by_ssh". http://www.nagios-wiki.de/nagios/howtos/nrpe_vs_ssh, 2007.

[Frey 06] B. Frey. *Statistics Hacks - Tips & Tools for Measuring the World and Beating the Odds*. O'Reilly, 2006.

[Gals 04] E. Galstad. "Nagios Documentation - Performance Data". http://nagios.sourceforge.net/docs/1_0/perfdata.html, 2004.

[Gals 09] E. Galstad and C. D. Team. "Nagios Core Version 3.x Documentation". http://nagios.sourceforge.net/docs/3_0/, 2009.

[Gelp 07] S. Gelper, R. Fried, and C. Croux. "Robust forecasting with exponential and Holt-Winters smoothing". Tech. Rep., Katholieke Universiteit Leuven, 2007.

[Gent 04] J. E. Gentle, W. Härdle, and Y. Mori, Eds. *Handbook of Computational Statistics: Concepts and Methods*. Springer, 2004.

[Ho 01] R. Ho, Ed. *gnuplot - or, How To Make Your Data Look Neat and Shiny*, Stanford University, 2001.

[Inso 01] G. Insolvibile. "The Linux Socket Filter: Sniffing Bytes over the Network". Linux Journal, 2001.

[Inso 02a] G. Insolvibile. "Inside the Linux Packet Filter". Linux Journal, 2002.

[Inso 02b] G. Insolvibile. "Inside the Linux Packet Filter, Part II". Linux Journal, 2002.

[Jr 06] E. S. G. Jr. "Exponential smoothing: The state of the art—Part II". Tech. Rep., Bauer College of Business and University of Houston, Texas, 2006.

[Kale 04] P. S. Kalekar. "Time series Forecasting using Holt-Winters Exponential Smoothing". Tech. Rep., Kanwal Rekhi School of Information Technology, 2004.

[Kao 98] E. P. C. Kao. *An Introduction to Stochastic Processes*. Duxbury Press / University of Houston, 1998.

[Kies 74] H. Kiesewetter and G. Maess. *Elementare Methoden der numerischen Mathematik*. Akademie-Verlag Berlin, 1974.

[Kocj 08] W. Kocjan. *Learning Nagios 3.0*. PACKT-Publishing, 2008.

[LaTo 08] K. LaTourette. "Notes on the Convergence of the Levenberg-Marquardt method". Tech. Rep., The University of Arizona, 2008.

[Lawt 98] R. Lawton. "How should additive Holt-Winters estimates be corrected?". Tech. Rep., International Journal of Forecasting, 1998.

[Loom 04a] D. G. Loomis. "Exponential Smoothing (Diebold, Chapter 4, Problem 4)". Illinois State University, 2004.

[Loom 04b] D. G. Loomis. "Overview of Quantitative Techniques (Moving Average, Trend Analysis, Exponential Smoothing, ARIMA models, Econometric models)". Illinois State University, 2004.

[Lour 09] M. Lourakis. “Levenberg-Marquardt nonlinear least squares algorithms in C/C++”. Institute of Computer Science, Foundation for Research and Technology - Hellas, Crete, 2009.

[Love 05] R. Love. *Linux-Kernel-Handbuch - Leitfaden zu Design und Implementierung von Kernel 2.6*. Addison-Wesely, 2005.

[Manw 03] S. Manwani. “ARP Cache Poisoning Detection and Prevention”. Tech. Rep., Department of Computer Science, San Jose State University, 2003.

[Maur 05] D. R. Mauro and K. J. Schmidt. *Essential SNMP: Help for System and Network Administrators*. O’Reilly, 2005.

[McCa 92] S. McCanne and V. Jacobson. “The BSD Packet Filter: A New Architekture for User-level Packet Capture”. Tech. Rep., Lawrence Berkeley Laboratory, 1992.

[Merr 07] E. Merritt. “Gnuplot Reference”. http://www.gnuplot.info/docs/node93.html, 2007.

[Mill 07] E. Miller. “Holt-Winters Forecasting Applied to Poisson Processes in Real-Time”. Tech. Rep., IMVU, Inc., 2007.

[NIST 06] U. C. D. T. A. NIST-Agency. “NIST/SEMATECH e-Handbook of Statistical Methods”. http://www.itl.nist.gov/div898/handbook/, 2006.

[Oeti 02] T. Oetiker, Ed. *RRDtool*, ETH Zürich, 2002.

[Oeti 98] T. Oetiker. “MRTG - The Multi Router Traffic Grapher”. Tech. Rep., Proceedings of the Twelfth Systems Administration Conference (LISA ’98), USENIX, 1998.

[Peik 04] C. Peikari and A. Chuvakin. *Security Warrior - Know Your Enemy*. O’Reilly, 2004.

[Pete 07] L. L. Peterson and B. S. Dave. *Computer Networks: A Systems Approach, Edition 4*. Morgan Kaufmann, 2007.

[Pres 07] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes - The Art of Scientific Computing, Third Edition*. Cambridge University Press, 2007.

[Reus 04] H. Reusser and R. Veselinović. *Promiscous Mode Detector*. Master’s thesis, Züricher Hochschule Winterthur, 2004.

[Riss 07] F. Risso and L. Degioanni. “An Architecture for High Performance Network Analysis”. Tech. Rep., Dipartimento di Automatica e Informatica – Politecnico di Torino, 2007.

[Rock 04] B. Rockwood. “Getting Started with RRDtool”. Tech. Rep., Cuddletech TekRef Series, 2004.

[Scha 04] K. Scharnbacher. *Statistik im Betrieb: Lehrbuch mit praktischen Beispielen*. Gabler, 2004.

[Schr 04] A. Schreiber, Ed. *Nagios - Wachsamer Schutzheiliger im Netz*, Landeskriminalamt Thüringen, Chemnitzer Linux-Tag, 2004.

[Schr 09] D. Schröder. *Elektrische Antriebe- und Regelung von Antriebssystemen*. Springer, 2009.

[Tayl 04] J. W. Taylor. “Volatility forecasting with smooth transition exponential smoothing”. Tech. Rep., Saïd Business School, University of Oxford, 2004.

[Tayl 06] J. W. Taylor, L. M. de Menezes, and P. E. McSharry. “A comparison of univariate methods for forecasting electricity demand up to a day ahead”. Tech. Rep., Saïd Business School, University of Oxford and Cass Business School, London and Dep. of Engineering, University of Oxford, 2006.

[Tayl 07] J. W. Taylor. “A Comparison of Univariate Time Series Methods for Forecasting Intraday Arrivals at a Call Center”. Tech. Rep., Saïd Business School, University of Oxford, 2007.

[Team 09] N. P. D. Team. “Nagios plug-in development guidelines”. http://nagiosplug.sourceforge.net/developer-guidelines.html, 2009.

[Venk 08] S. Venkateswaran. *Essential Linux Device Drivers*. Prentice Hall, 2008.

[Wats 07] R. N. M. Watson and C. S. J. Peron, Eds. *Zero-Copy BPF Buffers*, Computer Laboratory, University of Cambridge and Seccuris Inc., 2007.

[Zucc] W. Zucchini and O. Nenadić. “Time Series Analysis with R - Part I”.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Leipzig, den 14. Oktober 2009

_____

14. Oktober 2009, Build 6306