Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Stephan Gorisch

Subsekunden-Netzwerkanalyse und Synthese mit Handgeräten

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Stephan Gorisch

Subsekunden-Netzwerkanalyse und Synthese mit Handgeräten

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Rainer Schoenen
Zweitgutachterin: Prof. Dr.-Ing. Aining Li

Abgegeben am 21. September 2016

Stephan Gorisch

Thema der Bachelorarbeit

Subsekunden-Netzwerkanalyse und Synthese mit Handgeräten

Stichworte

Verkehrsanalyse, Netzwerk, Sampleperiode, 10ms, Verkehrsgenerator, Einplatinencomputer, Raspberry Pi, Odroid, Zwischenankunftszeit, Paketlaufzeit, PCap

Kurzzusammenfassung

Entwickelt werden soll ein System zur Analyse von Netzwerkverkehr. Der Fokus liegt dabei auf einer möglichst kleinen Sampleperiode. Weiterhin wird untersucht, ob das System in Form eines Handgerätes eingesetzt werden kann.

Stephan Gorisch

Title of the paper

Sub-second network traffic analysis and synthesis with handheld devices

Keywords

Trafficanalyser, Network, Trafficgenerator, Sampleperiod, 10ms, Singleboard-Computer, Raspberry Pi, Odroid, Inter-Arrival-Time, Packet-Delay, PCap

Abstract

A system for analyzing network-traffic should be developed. The focus is on a smallest possible sample period. Furthermore the executability on a handheld device will be tested.

Inhaltsverzeichnis

1. Einleitung

1.1. Ziele der Arbeit

In diesem Projekt soll ein System zur Netzwerkverkehrsanalyse entwickelt werden, das den Nutzer in die Lage versetzt auch Ereignisse zu registrieren, die nur für eine sehr kurze Zeitspanne auftreten (Analyse im Subsekundenbereich). In bisher verfügbaren Lösungen wären derartige Ereignisse wegen der relativ großen Sampleperiode (teilweise im Minutenbereich) nicht sichtbar.

Um das Verhalten des Netzwerkes in bestimmten Situationen zu untersuchen, soll das System in der Lage sein, möglichst frei konfigurierbar, Netzwerkverkehr zu erzeugen.
Weiterhin sollen aus einem Paketfluss Verkehrsdaten erzeugt und geeignet ausgewertet werden. Analysiert werden soll sowohl Netzwerkverkehr, der aktiv erzeugt worden ist, als auch laufender Verkehr im Netzwerk, der dem Analysator bspw. per *Port-Mirroring* zugeleitet wird. Der Analysator soll auch in der Lage sein eine Folge von Paketen aufzuzeichnen und zu speichern, um eine nachträgliche Analyse (bspw. im Fehlerfall) zu ermöglichen.

Untersucht werden soll auch, ob das System auf einem *Einplatinen-Computer* lauffähig ist, also ob ein solcher ausreichend performant ist. Der Einplatinencomputer repräsentiert hier ein Handgerät, für das, vor allem wegen der Akkukapazität und der Gehäusegröße, eine vergleichbare Rechenleistung zu erwarten wäre.

1.2. Aufbau der Arbeit

Im direkten Anschluss an diese Einleitung folgt eine Marktübersicht für Handheld-Netzwerkanalysatoren. Anschließend beginnt der Grundlagen-Teil mit dem *OSI*-Schichtenmodell. Hier sollen besonders die Protokolle der Schichten 2 bis 4 beleuchtet werden, da diese für das Projekt bedeutend sind.
Sowohl bei der Erzeugung von Netzwerkverkehr, als auch bei der Analyse spielen Sockets

eine entscheidende Rolle, weswegen sie in einem eigenen Abschnitt beschrieben werden.
Wie in 1.1 erwähnt, soll die Analyse des Netzwerkverkehrs mit einer *Sampleperiode* von $T_S < 1s$ erfolgen, was Anpassungen am *Scheduler* erfordert. Daher wird das Scheduling-Verfahren des Linux-Kernel (seit Version 2.6.23) in Abschnitt 3.3 erläutert.
Den Abschluss des Grundlagenteils bilden Ausführungen zu den Themen Häufigkeitsverteilungen und Network Time Protocol (NTP), die für die Analyse der Paketlaufzeit und der Zwischenankunftszeit von Wichtigkeit sind.

Das 4. Kapitel beschäftigt sich mit dem Generieren von Netzwerkverkehr. Nach dem Formulieren der Anforderungen an den Generator wird das individuelle Erzeugen von einzelnen Paketen, sowie einer bestimmten Datenrate beschrieben. In 4.5 wird erläutert, welche Aufgabe der Paketgenerator bei der Analyse der Paketlaufzeit übernimmt.
Für den Paketgenerator wurde eine Schnittstelle zur Steuerung implementiert, die es unter anderem ermöglichen soll, die Anwendung aus einem externen Programm heraus zu steuern.
Wie in 1.1 beschrieben wird für die einzelnen Teile der Anwendung untersucht, in wie weit die Lauffähigkeit auf einem Einplatinen-Computer gegeben ist. So auch für den Paketgenerator in Abschnitt 4.7.

Nachdem in Abschnitt 3.3 die Arbeitsweise des Linux-Schedulers erläutert wurde, werden zum Einstieg in das 5. Kapitel die Optimierungen am *Scheduler* besprochen, die die Voraussetzungen für die Netzwerkanalyse mit möglichst kleiner *Sampleperiode* schaffen.
Im Weiteren werden zwei Möglichkeiten der Paketerfassung beschrieben. In 5.3 mit Hilfe der *Socket*-Schnittstelle, in 5.7 mit Hilfe der *PCap*-Programmierschnittstelle. Erläutert wird auch die Weiterverarbeitung der Ergebnisse (5.4), sowie die Erkennung verworfener Pakete (5.8).
Wie auch in Kapitel 4, wird die Steuerung beschrieben und eine Performance-Bewertung für den Einsatz der Anwendung auf einem *Einplatinen-Computer* vorgenommen.
Abschließend ($\rightarrow$ 5.12) wird anhand zweier Bespiele die Leistungsfähigkeit des Systems demonstriert.

Geschlossen wird die Arbeit mit einem Ausblick auf mögliche Erweiterungen bzw. Verbesserungen des Systems.

Hinweis zur Lesbarkeit: *Kursiv* geschriebene Begriffe sind im Glossar erläutert, in eckigen Klammern [ ] wird auf das Literaturverzeichnis verwiesen, ein Pfeil ($\rightarrow$ 5.8) verweist auf einen Abschnitt in dieser Arbeit (auch Tabellen, Listings, Abbildungen, usw.).

In einigen Abschnitten der Arbeit werden Vorgehensweisen anhand von Quellcode-Fragmenten erläutert. Die Code-Ausschnitte zeigen, der Übersicht halber, nur den für die Funktion maßgeblichen Code. Es wird hiermit ausdrücklich darauf hingewiesen, dass

dabei Teile des Source-Codes weggelassen werden, vor allem Code zur Fehlerbehandlung, Ausgaben und Log-Einträge. Zum Zwecke einer besseren Nachvollziehbarkeit, enthält jedes Listing in Zeile 1 eine Kommentarzeile, die angibt welcher Funktion der Code entstammt. Die kompletten Source-Codes finden sich auf der beigelegten DVD. Diese kann beim betreuenden Prüfer oder der Zweitgutachterin eingesehen werden.

1.3. Inhalt der beiligenden DVD

Auf der beigelegten DVD befinden sich folgende Inhalte:

- die vorliegende Arbeit im pdf-Format
- alle in der Arbeit enthaltenen Abbildungen in Originalgröße
- Source-Code-Dateien für Generator mit makefile
- Generator-Anwendung (kompiliert unter SuSE Linux 13.2 64 Bit)
- Source-Code-Dateien für Analysator mit readme und makefile
- Analysator-Anwendung (kompiliert unter SuSE Linux 13.2 64 Bit)
- Source-Code-Dateien für GNUPlot-Interface mit makefile
- Testskripte mit Beschreibung in readme

2. Marktübersicht Netzwerkanalysatoren Handgeräte

Im Folgenden sollen einige Handheld-Netzwerkanalysatoren anhand ihrer Funktionalität in aller Kürze vorgestellt werden.

NextGig TestPort

(Datenblatt → [2])

- Paketfilter nach *MAC*-Adresse, IP-Adresse, Port, Protokoll
- Paketerfassung im *PCap*-Format
- Paketgrößenstatistiken
- Paketzähler
- Remote Control
- kein Paketgenerator
- keine Live-Ratenauswertung

Fluke Linkrunner LRAT 1000

(Datenblatt → [1])

- *DHCP* und *DNS*-Tests
- *PoE*-Lasterzeugung
- Test der Athentifizierung nach 802.1X (siehe [23])
- Link-Status (Geschwindigkeit, Duplex)
- Switch und VLAN-Informationen
- Kabelprüfung und Tongenerator
- Durchsatztests

- kein Verkehrsgenerator
- keine Paketerfassung
- keine Messung der Paketlaufzeit

Fluke NTS2 Pro

- *PoE*-Tests und Lasterzeugung
- Erkennung von Stationen mit zugehörigen Adressen
- Erkennung von angebotenen Diensten
- Test der Authentifizierung nach 802.1X (siehe [23])
- Kabeltests
- Link-Status (Geschwindigkeit, Duplex)
- kein Verkehrsgenerator
- keine Paketerfassung

Exfo ETS1000

- Verkehrsgenerator bis *OSI*-Schicht 4
- Laufzeitmessung mit Loopback-Box
- Kabeltests
- keine Paketerfassung
- keine Ratenauswertung

IDEAL Networks SignalTEK/NaviTEK/LANExplorer

- Verkabelung- und Durchgangstests
- *PoE*-Messungen und Lasterzeugung
- *TDR*
- Verkehrsgenerator für bestimmte Anwendungen (bspw. VoIP, IPTV)
- Ping-Tests und Routenverfolgung
- Netscans (IP, *MAC*, offene Ports), Erkennung doppelter Adressen
- keine Paketerfassung

- keine Ratenauswertung

Von den genannten Herstellern sind weitere Geräte am Markt verfügbar, die sich im Funktionsumfang oft nur unwesentlich unterscheiden.

Allen Geräten fehlt ein konfigurierbarer Verkehrsgenerator (Einstellung der Datenrate, Paketgröße, Nutzdatenmuster, Traffic-Bursts, Kontrolle über die Header des Transportschicht- und Vermittlungsschichtprotokolls), sowie eine Live-Auswertung der Datenraten mit möglichst kleiner *Sampleperiode*.

3. Grundlagen

3.1. OSI-Schichtenmodell

3.1.1. Allgemeines

Mit Hilfe des *OSI*-Modells wird die Kommunikation über Netzwerkprotokolle in sieben Schichten abgebildet. Jede Schicht hat für die Kommunikation bestimmte Aufgaben (z. B. elektronische Signalübertragung). Diese Aufgaben werden von Netzwerkprotokollen erfüllt, die der jeweiligen Schicht zugehörig sind. Jede Schicht im OSI-Modell bietet der darüberliegenden einen Dienst an und greift auf einen Dienst der darunterliegenden Schicht zurück. Das jeweils verwendete Protokoll wird als Instanz einer Schicht bezeichnet [27].
Abbildung 3.1 zeigt den prinzipiellen Ablauf der Kommunikation. Sowohl sender- als auch empfängerseitig werden alle Schichten des *OSI*-Modells durchlaufen. Schicht 1-4 sind dabei transportorientiert, Schicht 5-7 anwendungsorientiert.

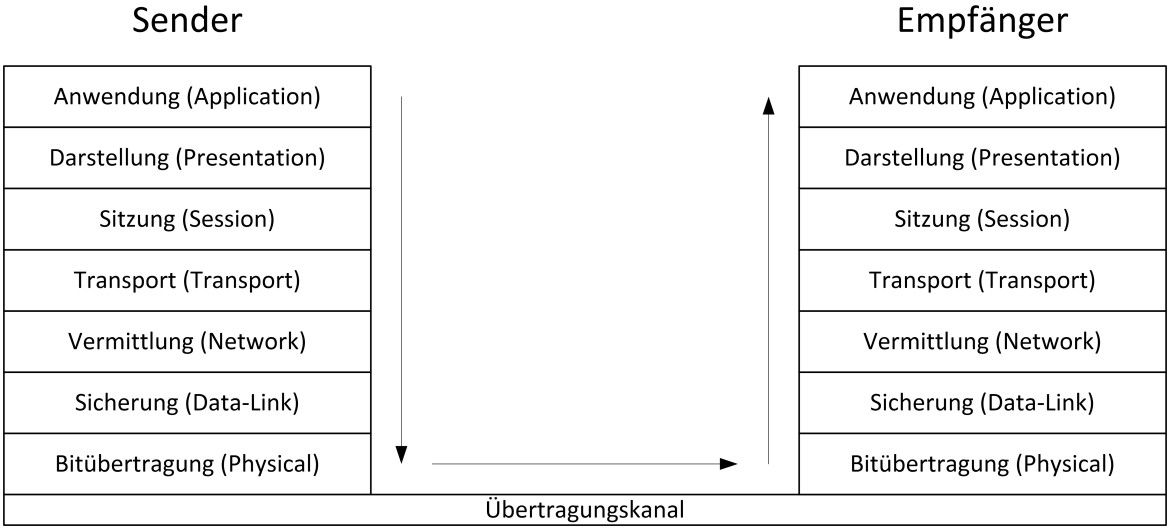

Abbildung 3.1.: Kommunikationsfluss im OSI-Schichtenmodell

Soll beispielsweise eine Information von einer Anwendung übertragen werden, würde von selbiger ein Datenpaket mit der Information generiert. Diesem Datenpaket würden alle Protokolle auf dem Weg von der Anwendungs- zur Bitübertragungsschicht Protokollinformationen des jeweiligen Protokolls hinzufügen. Es wird Schicht für Schicht in Protokollheader verpackt.
Aufgabe der Bitübertragungsschicht ist es, dieses Paket in übermittelbare Daten umzuwandeln und über das Übertragungsmedium (z. B. eine Funkstrecke) zu versenden. Auf der Empfängerseite würden die Protokollinformationen schichtweise wieder entfernt bis das Paket bei der richtigen Anwendung ankommt. Virtuell kommunizieren die Protokolle demnach auch mit der Partnerinstanz auf der Gegenseite [20].

Aufgrund der Relevanz für die vorliegende Arbeit wird im Folgenden auf die Aufgaben der Schichten 2-4 gesondert eingegangen und einschlägige Protokolle beschrieben.

3.1.2. Sicherungsschicht

Die Aufgabe eines Protokolls der Sicherungsschicht besteht darin, die von Schicht 3 kommenden Frames in einzelne Bit zu zerlegen, bzw. einzelne Bit zu einem Frame zusammenzusetzen und diesen an die Vermittlungsschicht weiterzureichen.

Abbildung 3.2 zeigt den Aufbau eines Frames im Falle von Ethernet nach IEEE 802.3 [24].

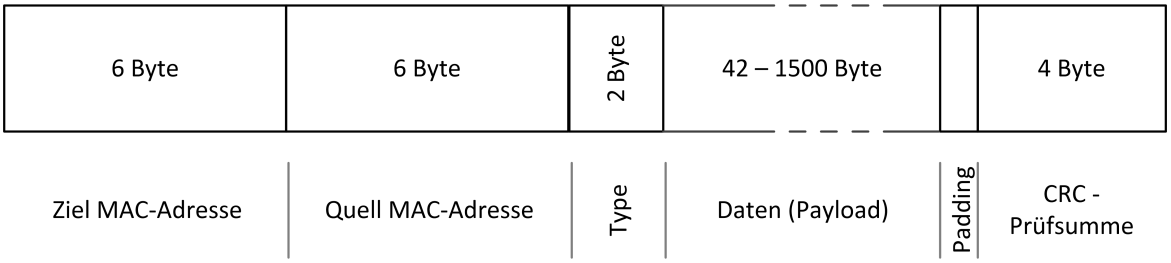

Abbildung 3.2.: Aufbau Ethernet-Frame

Ziel-/Quell *MAC*-Adresse
Die *MAC*-Adressen werden in der Regel byteweise hexadezimal dargestellt (bspw. E0:CB:4E:DD:A4:A0). Der ersten beiden Bit der *MAC*-Adresse haben eine besondere Bedeutung. Das niederwertigste Bit gibt an, ob es sich um eine Einzeladresse (Bit 0 nicht gesetzt) oder eine Multi-/Broadcast-Adresse handelt (Bit 0 gesetzt). Bit 1 gibt an, ob die Adresse nur lokal (Bit 1 gesetzt) oder global (Bit 1 nicht gesetzt) eindeutig ist.
Bei einer Einzeladresse folgt darauf die Kennung des *NIC*-Herstellers. Dafür stehen die Bits 2 bis 23 zur Verfügung. Daran schließt sich die individuelle Kennung der jeweiligen

Netzwerkschnittstelle an (Bit 24 bis 47).

Type
Für das Type-Feld stehen 2 Byte zur Verfügung. Angegeben wird, welches Protokoll innerhalb der Nutzdaten für die nächsthöhere Schicht verwendet wird. Für IPv4 lautet der Wert 0x0800, für IPv6 0x86DD.

Padding
Das Padding-Feld hat eine variable Größe. Es dient dazu eine Mindestpaketgröße von 64 Byte sicherzustellen. Da sich das Padding-Feld im Anschluss an die Nutzdaten befindet, muss das nächsthöhere Protokoll dafür sorgen, dass das Feld nicht als Nutzdaten interpretiert wird [22].

CRC-Prüfsumme
Die *CRC*-Prüfsumme wird vom Sender generiert und an den Frame angehängt. In die Berechnung einbezogen wird der gesamte Ethernet-Frame (außer dem *CRC*-Feld selbst). Der Empfänger des Pakets berechnet anhand der empfangenen Daten seinerseits die *CRC*-Prüfsumme und verwirft das Paket, falls sie nicht mit der aus dem *CRC*-Feld übereinstimmt.

3.1.3. Vermittlungsschicht

Die Aufgabe der Protokolle der Vermittlungsschicht ist das Routing der Datenpakete. Für die Wegfindung nötig ist eine Adressierung der Endgeräte. Wegen ihrer Relevanz werden die Schicht 3 - Protokolle *IPv4* und *IPv6* hier näher erläutert.

Mittelfristig wird das Internet Protocol Version 6 das der Version 4 ablösen. Der Hauptgrund liegt darin, dass *IPv4* nur eine Adresslänge von 32 Bit vorsieht, was für die schnell wachsende Anzahl an Teilnehmern schon derzeit zu wenige Adressen zulässt. Mit Verfahren wie *NAT* wird versucht diese Knappheit zu umgehen.

Abbildung 3.3 zeigt den Aufbau des IPv4-Headers.

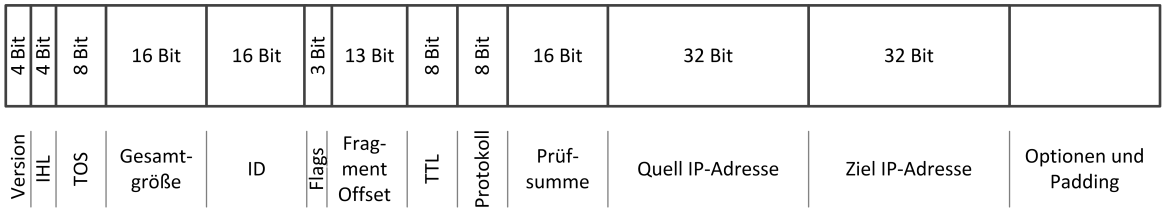

Abbildung 3.3.: Aufbau *IPv4*-Header

Version
Gibt die Version des Protokolls an (hier Version 4).

IHL
Gibt die Länge des Headers in Vielfachen von 4 Byte an.

ToS
Im *ToS*-Feld können Prioritätklassen und Dienstgüten festgelegt werden. Pakete könnten damit beim Routing unterschiedlich behandelt (z. B. kritische Dienste bevorzugt) werden. Das Feld wird von *IPv4* in der Regel nicht genutzt.

Gesamtgröße
Gibt die Gesamtlänge des *IPv4*-Pakets inklusive Header an.

ID - Identifikation
32-Bit Ganzzahl, die jedes Datenpaket eindeutig kennzeichnet. Wird das Paket fragmentiert erhalten alle Fragmente die ID des Ursprungspakets.

Flags
Im Flags-Feld werden Informationen zur Fragmentierung des Paketes übermittelt, bzw. eine Fragmentierung untersagt.

Fragmentation Offset
Gibt die Lage des aktuellen Fragments, relativ zum Anfangspaket, an. Damit kann die ursprüngliche Paketfolge wiederhergestellt werden.

TTL - Time To Live
Das *TTL*-Feld dient dazu dem Paket eine maximale Lebensdauer zu geben, um fehlgeleitete Pakete verwerfen zu können. Jedes mal, wenn das Paket einen Knoten passiert, wird der Wert um eins dekrementiert. Bei $TTL = 0$ wird das Paket verworfen.

Protokoll
Das Protokoll-Feld enthält die Protokollnummer des Protokolls der *OSI*-Schicht 4. Beispiele sind *TCP* (Protokollnummer 6) und *UDP* (Protokollnummer 17). Siehe dazu Abschnitt 3.1.4.

Prüfsumme
Es handlet sich um eine Prüfsumme, die nur aus den Headerdaten gebildet wird, also auch nur die Integrität des Headers sicherstellt. Da sich der *TTL*-Wert bei jedem Hop verändert, muss die Prüfsumme von jedem Knoten neu berechnet werden [19].

Quell- und Zieladressen
Wie bereits beschrieben sind für jede Adresse 32 Bit vorgesehen. Möglich sind demnach $2^{32} \approx 4,29 \cdot 10^9$ Adressen. Die Darstellung erfolgt in der Regel byteweise als Dezimalzahl und durch Punkte getrennt (z. B. 201.5.99.178). Welche Adressbereiche zum gleichen Netz gehören wird durch die Subnetzmaske festgelegt. Befinden sich zwei Adressen im gleichen Netz, ergibt die UND-Verknüpfung einer jeden Adresse mit der Subnetzmaske die gleiche Netzadresse. Dazu zwei Beispiele in der üblichen Notation und in Binärdarstellung.

Beispiel 1: IP-Adresse 172.16.12.10, Netzmaske 255.240.0.0

	172.16.12.10	$\rightarrow$	10101100 00010000 00001100 00001010
&	255.240.0.0	$\rightarrow$	11111111 11110000 00000000 00000000
	172.16.0.0	$\rightarrow$	10101100 00010000 00000000 00000000

Die IP-Adresse gehört zum Netz 172.16.0.0. Die gängige Schreibweise für dieses Netz lautet 172.16.0.0/12, da 12 Bit der Netzmaske gesetzt sind.

Beispiel 2: IP-Adresse 172.33.12.10, Netzmaske 255.240.0.0

	172.33.12.10	$\rightarrow$	10101100 00100001 00001100 00001010
&	255.240.0.0	$\rightarrow$	11111111 11110000 00000000 00000000
	172.32.0.0	$\rightarrow$	10101100 00100000 00000000 00000000

Die IP-Adresse gehört zum Netz 172.32.0.0. Die gängige Schreibweise für dieses Netz lautet 172.32.0.0/12.

Ohne Kenntnis über die Netzmaske könnte demnach nicht festgestellt werden, ob die Adressen zum gleichen Netz gehören.

Optionen
Im Optionsfeld können Routing-Kriterien, Sicherheitsinformationen oder auch Zeitstempel enthalten sein. Das Feld wird bis zum nächsten Vielfachen von 4 Byte mit Nullbits aufgefüllt (Padding), da die Headergröße im *IHL*-Feld in Vielfachen von 4 Byte angegeben ist.

Als zweites Protokoll der Vermittlungschicht soll *IPv6* erläutert werden. Abbildung 3.4 zeigt den Aufbau des *IPv6*-Headers.

Abbildung 3.4.: Aufbau *IPv6*-Header

Direkt ersichtlich ist, dass der Header im Vergleich zu *IPv4* wesentlich vereinfacht wurde. Die Headerlänge ist nicht mehr variabel sondern hat immer die feste Länge 40 Byte.
Im Folgenden die Kurzbeschreibungen zu den einzelnen Feldern des *IPv6*-Header.

Version
Gibt die Version des Protokolls an (hier Version 6).

Klasse
Das Feld ermöglicht eine Klassifizierung und damit eine Priorisierung des Datenverkehrs.

Flow Label
Mit dem Flow Label können Datenströme gezielt gekennzeichnet werden. Für diese können Ressourcen (bspw. Bandbreite) definiert werden. Anhand der Quell-Adresse und des Flow Labels wird der entsprechende Flow identifiziert und die zugehörigen Pakete passend behandelt.

Länge der Nutzdaten
Enthält die Länge der Nutzdaten (inkl. möglicher zusätzlicher Header) in Byte. Der Wert ist null, falls ein zusätzlicher Header den aktuellen Wert beinhaltet.

Nächster Header
Das Feld enthält entweder die Protokollnummer des in der Hierarchie des OSI-Modells nachfolgend höheren Protokolls oder verweist auf einen *IPv6*-Erweiterungsheader. Damit kann *IPv6* auch zukünftig angepasst werden, ohne das Grundgerüst verändern zu müssen.

Hop Limit
Das Feld ersetzt das *TTL*-Feld aus dem *IPv4*-Header und hat die gleiche Funktion.

Quell- und Ziel-Adresse
Die *IPv6*-Adressen sind viermal so lang wie die von *IPv4*. Das ermöglicht $2^{128} \approx 3,4 \cdot 10^{38}$ verschiedene Adressen. Die übliche Schreibweise ist nun hexadezimal und wortweise durch Doppelpunkte getrennt (z. B. 2003:57:ea18:a819:485c:79aa:3a2:281d). Verwendet wird außerdem eine Kurzschreibweise, die es erlaubt vorangestellte Nullen wegzulassen und eine

Folge von Worten, die ausschließlich aus Nullen besteht, mit zwei Doppelpunkten abzukürzen. Um Mehrdeutigkeiten auszuschließen ist das nur genau einmal erlaubt. Folgende Umwandlung in Kurzschreibweise wäre demnach möglich:

fe80:**0000:0000**:485c:**000**0:79aa:**0**3a2:281d $\rightarrow$ fe80::485c:0:79aa:3a2:281d

Anhand der Adresse kann ihr Gültigkeitsbereich abgelesen werden. Wichtige Bereiche sind Link-Local, für Adressen die in lokalen Netzsegmenten gültig sind und Global Unicast, für Adressen die Global gültig und eindeutig sind. Die ersten 2 Byte einer lokal gültigen Adresse lauten immer $fe80$, global gültige Adressen enthalten dort einen Wert im Bereich $2000..3fff$.

3.1.4. Transportschicht

Die Transportschicht hat 3 wesentliche Aufgaben [5]:

- Zuordnung der Datenströme zu den Anwendungen
- Bereitstellung eines Transportdienstes
- Stau- und Flusskontrolle

Anhand der Portnummer und der IP-Adresse wird die Anwendung und der jeweilige Host identifiziert, für die Datenpakete bestimmt sind. Ein Header eines Transportschichtprotokolls muss demnach zumindest den Ziel- und den Quellport enthalten, damit eine Kommunikation in beide Richtungen möglich ist.

Transportmechanismen können verbindungslos oder verbindungsorientiert realisiert sein.
Bei verbindungslosen Diensten gibt es weder eine Garantie für die richtige Reihenfolge der Pakete, noch dafür ob Pakete überhaupt den Empfänger erreichen. Demnach gibt es auch keine Übertragungswiederholungen.
Im Gegensatz dazu bieten verbindungsorientierte Dienste die Garantie der Einhaltung der Paketreihenfolge und wiederholen die Übertragung bei ausbleibenden Empfangsbestätigungen.

Die Flusskontrolle ermöglicht es, dass der Empfänger beim Auftreten einer Überlast die Übertragungsrate verringern kann. Mit der Staukontrolle wird auf Paketverluste im Netz reagiert und die Übertragungsrate durch den Sender gesenkt.

Im Anschluss werden die Transportschichtprotokolle *TCP* (verbindungsorientiert) und *UDP* (verbindungslos) erläutert.

Wie erwähnt handelt es sich bei *TCP* um ein verbindungsorientiertes Protokoll. Zum Aufbau der Verbindung wird der Three-Way-Handshake verwendet. Abbildung 3.5 zeigt den Ablauf des *TCP*-Verbindungsaufbaus.

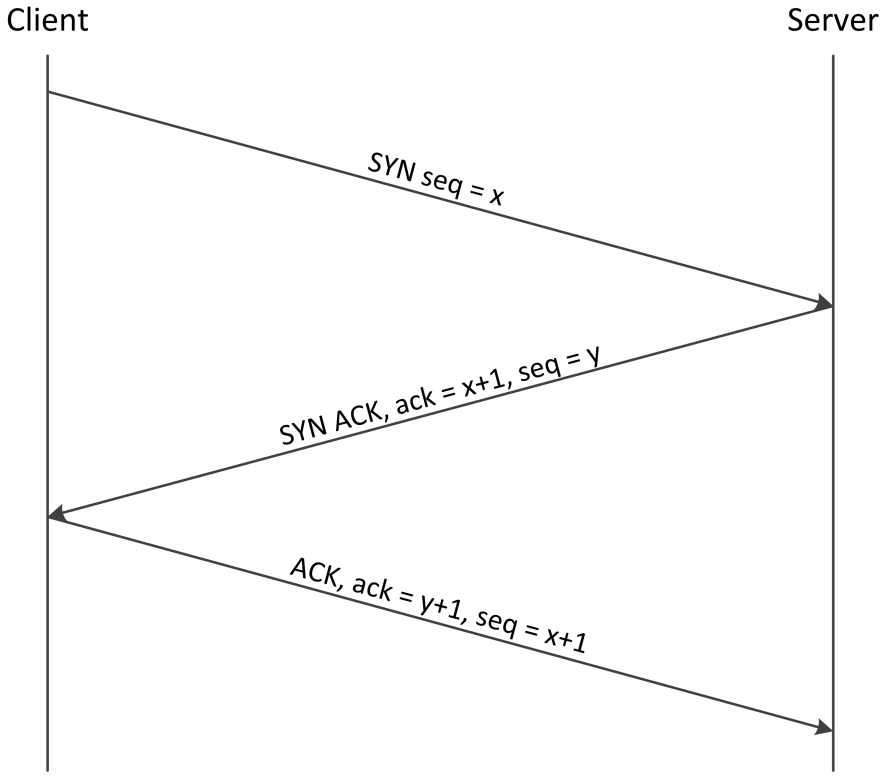

Abbildung 3.5.: Ablauf *TCP*-3-Way-Handshake

Der Verbindungsaufbau wird clientseitig durch das Versenden eines Paketes mit gesetztem SYN-Flag ($\rightarrow$ Abb. 3.6 und dazugehörige Erläuterungen) initiiert. Enthalten ist die initiale Sequenznummer (x) des Clients. Sollte dem entsprechenden Port serverseitig keine Anwendung zugeordnet sein, antwortet der Server mit einem Paket mit gesetztem RST-Flag (Reset), der Verbindungsaufbau wird abgebrochen. Im anderen Fall würde der Server im Antwortpaket ACK- (Acknowledge) und SYN-Flag setzen, um den Empfang des clientseitigen SYN-Pakets zu bestätigen und seinerseits die initiale Sequenznummer (y) zu übermitteln. Der Empfang dieses Pakets wird vom Client wiederum bestätigt.

Im Falle einer Bestätigung wird jeweils die nächste erwartete Sequenznummer als Acknowlegde-Nummer verwendet.

Die Sequenznummer wird für jedes verschickte Paket erhöht. Für Übertragungswiederholungen wird die ursprüngliche Sequenznummer verwendet, um der Gegenstelle die Zurodnung zu ermöglichen.

Abbildung 3.6 zeigt den Header eines *TCP*-Pakets.

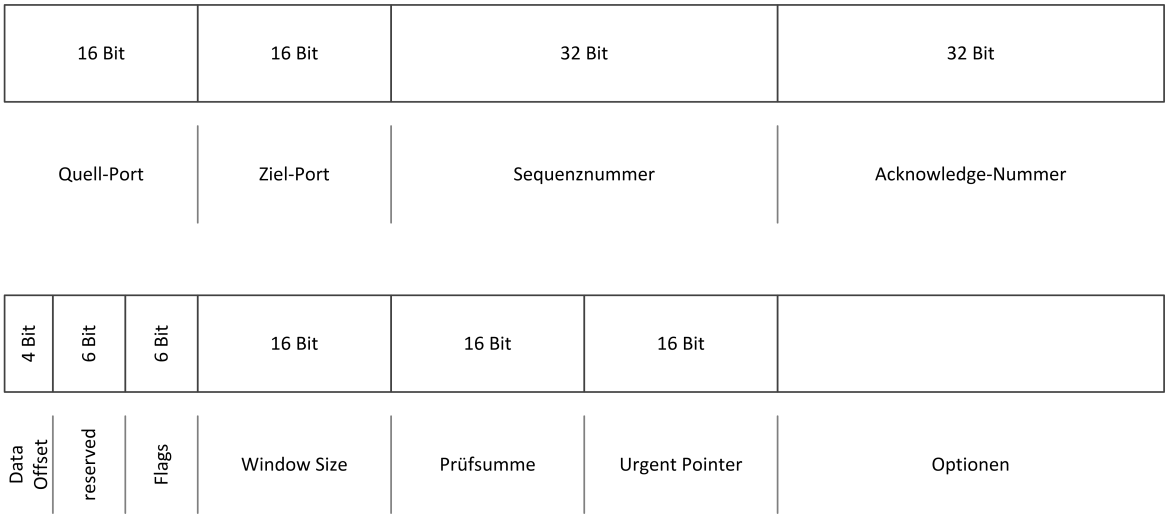

Abbildung 3.6.: Aufbau *TCP*-Header

Wie erwähnt sind Quell- und Zielport enthalten. Die möglichen Portnummern liegen im Bereich 0..65535, da je 16 Bit zur Verfügung stehen. Auch bereits angesprochen wurden Sequenz- und Acknowledge-Nummer, die in der weiteren Kommunikation wie beschrieben verwendet werden. Die übrigen Felder haben folgende Aufgaben:

Data Offset
Aufgrund der variablen Länge des *TCP*-Headers (Optionen) wird hier die Headerlänge in Vielfachen von 4 Byte angegeben.

reserved
Ungenutzt, in der Regel mit Nullen belegt.

Flags
Enthalten sind 6 Flags, die verschiedene Verwendung finden:

- URG (urgent): wenn gesetzt, werden die Daten (Headerende bis zu der im Feld Urgent Pointer angegebenen Stelle) sofort an die nächsthöhere Schicht weitergeleitet
- ACK (acknowledge): wenn gesetzt, wird der Empfang aller Pakete mit einer Sequenznummer kleiner der aktuellen Acknowledgenummer bestätigt
- PSH (push): wenn gesetzt, wird die Empfangsqueue umgangen, die Daten werden an die nächsthöhere Schicht weitergeleitet
- RST (reset): wenn gesetzt, wird die Verbindung ohne normalen Verbindungsabbau beendet

- SYN (synchronization): wird beim Verbindungsaufbau benutzt um die initiale Sequenznummer zu übermitteln
- FIN (finish): wird beim Verbindungsabbau genutzt, die letzte verwendete Sequenznummer wird mit übertragen

Window Size
Gibt an, welche Datenmenge übertragen werden darf, bevor eine Bestätigung durch den Empfänger erfolgen muss. Der Empfänger kann diesen Wert verringern, wodurch die Sendedatenrate sinkt und empfangsseitigen Überlastsituationen vorgebeugt wird.

Prüfsumme
Das Feld enthält eine Prüfsumme, die aus den Headerdaten und den Nutzdaten gebildet wird.

Urgent Pointer
Wie bereits beschrieben, gibt der Pointer die Stelle an, an der die Daten enden, die sofort an die nächsthöhere Schicht weitergereicht werden.

Als Beispiel für ein verbindungsloses Transportschicht-Protokoll wird im Folgenden *UDP* erläutert.

Versendet werden sogenannte Datagramme. Es gibt weder eine Fluss- noch eine Staukontrolle, verlorene Pakete werden nicht erneut übertragen. Die Datenübertragung ist vollständig ungesichert. Vorteile bringt ein verbindungsloses Protokoll, z. B. bei Anwendungen, die keinen Nutzen durch Übertragungswiederholungen hätten (bspw. *VoIP*) und bei denen die Datenübertragung nicht ins Stocken kommen darf, weil ein Paket neu angefordert werden muss.
Der *UDP*-Header (Abb. 3.7) fällt dementsprechend übersichtlich aus.

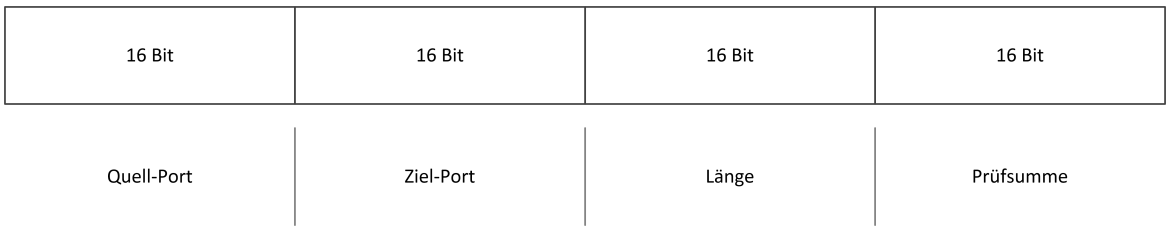

Abbildung 3.7.: Aufbau *UDP*-Header

Neben dem bereits von *TCP* bekannten Quell- und Zielport wird lediglich die Länge des Paketes (*UDP*-Header und Nutzdaten), sowie eine optionale Prüfsumme übertragen.

3.2. Sockets

Sockets sind Kommunikationsendpunkte, die für Anwendungen eine Schnittstelle zum Betriebssystem darstellen. In der Regel werden Stream- oder Datagramm-Sockets verwendet, die auf dem *TCP*- oder *UDP*-Protokoll aufsetzen.
Abbildung 3.8 zeigt eine Einordnung in das *OSI*-Schichtenmodell.

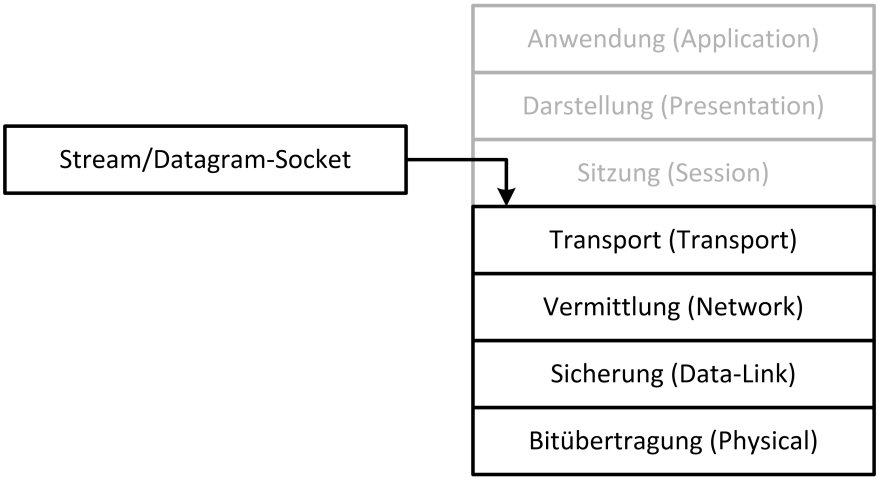

Abbildung 3.8.: Stream-/Datagram-Socket im *OSI*-Modell

Im Fall der beschriebenen Transportschicht-Protokolle ($\rightarrow$ 3.1.4) müssen lediglich Quell- und Zielport sowie Ziel-IP-Adresse durch die Anwendung bestimmt werden. Alles Weitere übernimmt die darunterliegende Netzwerk-Implementierung im Betriebssystem [30].

Eine Schicht darunter ist der RAW-Socket angesiedelt. Er setzt auf einem Vermittlungsschicht-Protokoll wie *IPv4* auf und ermöglicht die Implementierung eines eigenen Transportschicht-Protokolls (oder auch den Verzicht darauf). Natürlich kann manuell *TCP* oder *UDP* genutzt werden, der Paketheader muss dann aber vollständig selbst erzeugt werden, inklusive der Prüfsummenberechnung.

Abbildung 3.9 zeigt eine Einordnung des RAW-Sockets in das *OSI*-Schichtenmodell.

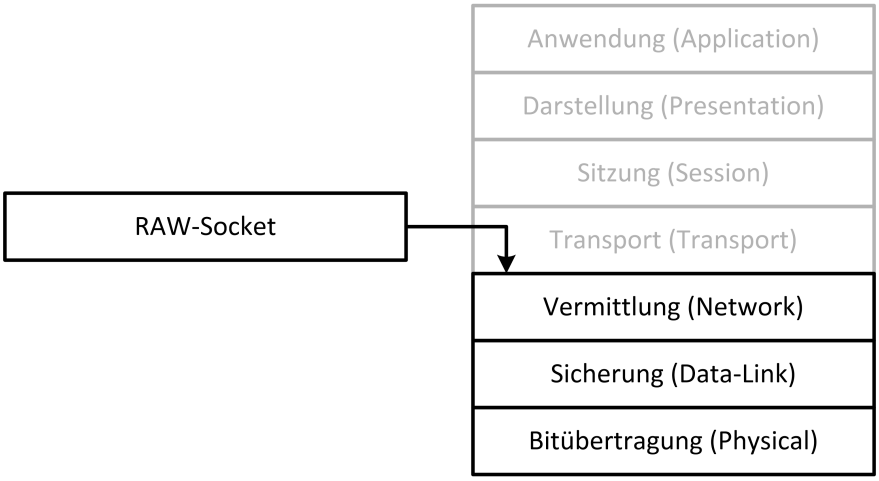

Abbildung 3.9.: RAW-Socket im *OSI*-Modell

Weiterhin kann auf den Header des Vermittlungschicht-Protokolls Einfluss genommen werden. Sämtliche Felder können manipuliert werden. Im Fall von *IPv4* kann die Prüfsummenberechnung und der Eintrag der Quell-IP-Adresse optional durch das Betriebssystem erfolgen.

Nochmals eine Schicht darunter ist der Packet-Socket anzusiedeln. Das Vermittlungschicht-Protokoll muss vollständig manuell implementiert werden (inklusive Prüfsummenberechnung).

Abbildung 3.10 zeigt eine Einordnung des Packet-Sockets in das *OSI*-Schichtenmodell.

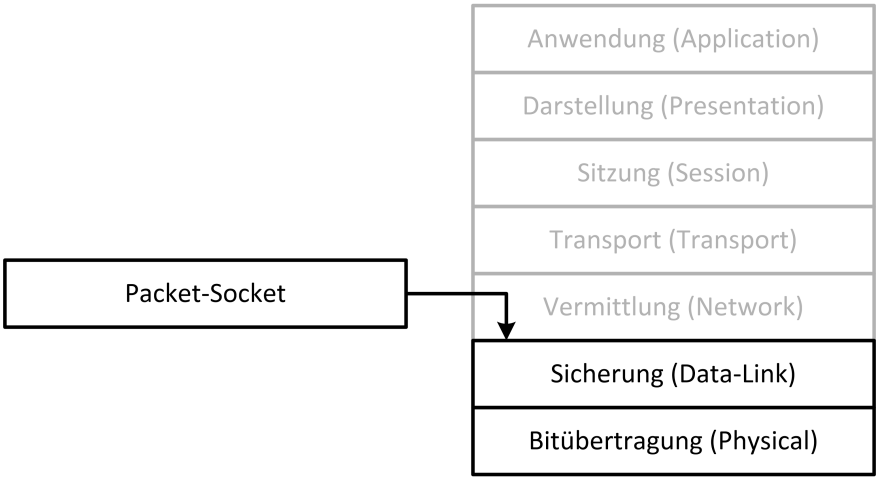

Abbildung 3.10.: Packet-Socket im *OSI*-Modell

Bei Verwendung von Ethernet als Protokoll der Sicherungsschicht kann die Quell- und Ziel-*MAC*-Adresse manipuliert werden. Die *CRC*-Prüfsumme kann manuell oder durch das Betriebssystem berechnet werden.

Mit Hilfe der verschiedenen Socket-Schnittstellen können individuell Pakete erzeugt, bzw. empfangene Pakete bis in die unteren Schichten ausgewertet werden. Für den Paketgenerator ($\rightarrow$ 4) wird ein RAW-Socket verwendet, für den Analysator ($\rightarrow$ 5) ein Packet-Socket, um eine Analyse bis zum *MAC*-Layer zu ermöglichen.

3.3. Scheduling

Die Aufgabe des Schedulers ist es, die zur Verfügung stehende Rechenzeit möglichst sinnvoll auf die lauffähigen *Tasks* aufzuteilen. Seit Linux Kernel 2.6.23 kommt der Completely Fair Scheduler (*CFS*) zum Einsatz. Ziel dieses Schedulers ist es, die CPU-Zeit möglichst fair zu verteilen. Bei *Tasks* mit identischer Gewichtung ist die Verteilung der Rechenzeit einfach nachzuvollziehen. Die zur Verfügung stehende CPU-Zeit (Scheduler-Periode) wird durch die Anzahl der lauffähigen *Tasks* geteilt. Das Ergebnis ist die Rechenzeit, die jeder *Task* in der aktuellen Scheduler-Periode erhält. Manipulierbar sind dabei die minimale CPU-Zeit t_{task_min}, die ein *Task* erhält, sowie die Standard-Scheduler-Periode $T_{Scheduler\_Default}$.
Abbildung 3.11 veranschaulicht die Aufteilung der Rechenzeit bei gleichgewichtigen *Tasks*.

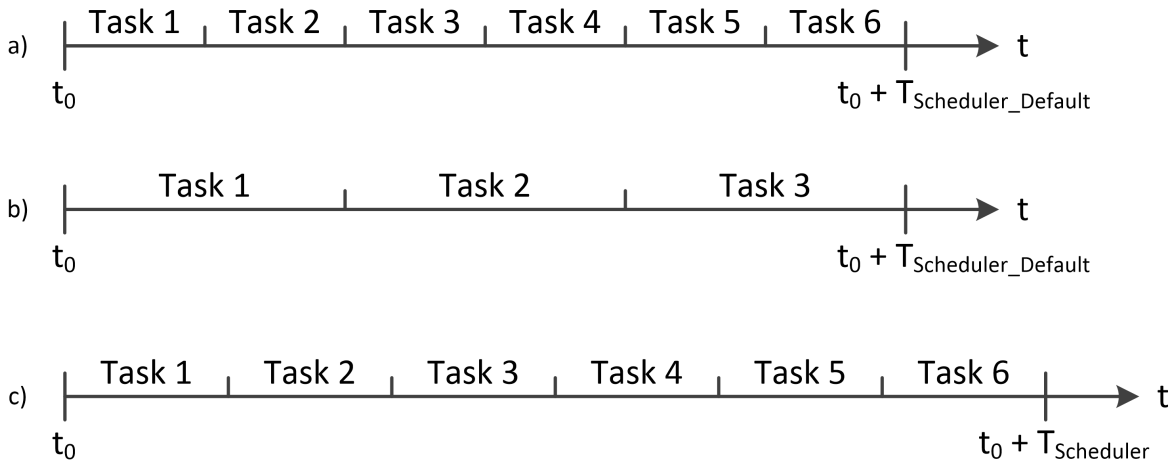

Abbildung 3.11.: *CFS* - Verteilung der Rechenzeit bei *Tasks* mit identischer Gewichtung

Dargestellt sind unterschiedliche Szenarien:

a) Die Scheduler-Periode ist gleich dem konfigurierten Standardwert, die Zeitscheiben der einzelnen *Tasks* sind größer als der eingestellte Minimalwert.

$\rightarrow T_{Scheduler} = T_{Scheduler\_Default}$
$\rightarrow t_{taskx} \geq t_{task\_min} = \frac{1}{6} \cdot T_{Scheduler\_Default}$

b) Wie a), aber mit nur 3 lauffähigen *Tasks*.
$\rightarrow T_{Scheduler} = T_{Scheduler\_Default}$
$\rightarrow t_{taskx} \geq t_{task\_min} = \frac{1}{3} \cdot T_{Scheduler\_Default}$

c) Wie a), die berechneten Zeitscheiben sind aber kleiner als der Minimalwert. Die Scheduler-Periode wird entsprechend angepasst.
$\rightarrow T_{Scheduler} > T_{Scheduler\_Default}$
$\rightarrow t_{taskx} = t_{task\_min} = \frac{1}{6} \cdot T_{Scheduler}$

Bisher waren alle *Tasks* gleich gewichtet. Die Gewichtung w eines Task wird nach folgender Formel berechnet [18]:

$$w(nice) = \frac{1024}{1,24^{nice}} \tag{3.1}$$

Demnach ergibt ein Nice-Wert von $nice = 0$ eine Gewichtung von $w = 1024$. Negative Nice-Werte erhöhen das Gewicht, der *Task* bekommt mehr Anteile an der zur Verfügung stehenden Rechenzeit. Positive Nice-Werte bewirken das Gegenteil. Der Nice-Wert eines *Task* kann beim Starten festgelegt oder zur Laufzeit verändert werden.
Die Rechenzeit eines *Task* in einer Scheduler-Periode berechnet sich nach:

$$t_{Task} = \frac{w_{Task}}{w_{gesamt}} \cdot T_{Scheduler} \tag{3.2}$$

Angenommen sei folgendes Beispiel (mit $w_{gesamt} = 7447$):

$$\begin{array}{llll}
nice_{Task1} = +5 & \rightarrow & w_{Task1} \approx 349 & \rightarrow \quad t_{Task1} \approx 0,047 \cdot T_{Scheduler} \\
nice_{Task2} = 0 & \rightarrow & w_{Task2} = 1024 & \rightarrow \quad t_{Task2} \approx 0,138 \cdot T_{Scheduler} \\
nice_{Task3} = 0 & \rightarrow & w_{Task3} = 1024 & \rightarrow \quad t_{Task3} \approx 0,138 \cdot T_{Scheduler} \\
nice_{Task4} = 0 & \rightarrow & w_{Task4} = 1024 & \rightarrow \quad t_{Task4} \approx 0,138 \cdot T_{Scheduler} \\
nice_{Task5} = -5 & \rightarrow & w_{Task5} \approx 3002 & \rightarrow \quad t_{Task5} \approx 0,403 \cdot T_{Scheduler} \\
nice_{Task6} = 0 & \rightarrow & w_{Task6} = 1024 & \rightarrow \quad t_{Task6} \approx 0,138 \cdot T_{Scheduler}
\end{array}$$

Abbildung 3.12 veranschaulicht die Aufteilung der Rechenzeit bei unterschiedlich gewichtigen *Tasks* für o. g. Beispiel.

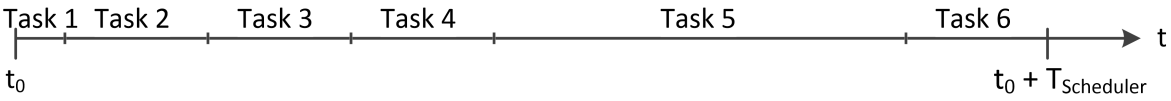

Abbildung 3.12.: *CFS* - Verteilung der Rechenzeit bei *Tasks* mit teilweise unterschiedlicher Gewichtung

Ersichtlich ist, dass jeder *Task* nur einmal pro Scheduler-Periode Rechenzeit bekommt. Zwischen seinen Zeitscheiben bekommt der *Task* n für die Dauer $t_n = T_{Scheduler} - t_{Taskn}$ keine CPU-Zeit.
Sollte von einem *Task* jedoch eine schnellere Reaktion (bspw. bei I/O-Events) erwartet werden, muss der *Task* in der Lage sein die Ausführung anderer *Tasks* zu unterbrechen.
Das kann im *CFS* realisiert werden, indem der Task einer anderen, höher priorisierten Scheduling-Klasse zugeordnet wird. In den bisher besprochenen Beispielen gehörten die *Tasks* der Klasse „fair_sched_class“ an, die nächsthöhere Klasse ist die sogenannte „rt_sched_class“ [18]. Ist ein *Task* dieser Klasse lauffähig, wird die Ausführung eines *Task* einer niederen Klassen angehalten und der *Task* der höheren Klassen ausgeführt, bis er keine Rechenzeit mehr benötigt oder von einem noch höher priorisierten *Task* unterbrochen wird.

Innerhalb der Klasse „rt_sched_class“ gibt es zwei verschiedene Scheduling-Strategien:

SCHED_FIFO
Der *Task* mit der höchsten Priorität innerhalb der Klasse wird ausgeführt bis er keine Rechenzeit mehr benötigt.

SCHED_RR
Die Rechenzeit wird periodisch, gewichtet anhand der Priorität, unter den *Tasks* der Klasse verteilt.

3.4. Network Time Protocol

Aufsetzend auf dem verbindungslosen Transportprotokoll *UDP* (→ 3.1.4) bietet das Network Time Protocol (*NTP*) eine Möglichkeit Systemuhren zu synchronisieren. Synchronisiert wird dabei gegen einen *NTP*-Server. Es gibt zwei Arten von *NTP*-Servern, jene mit und jene ohne direkten Anschluss an eine externe Uhr (z. B. per *GPS*-Empfänger oder *DCF77*-Empfänger). Das sogenannte Stratum gibt dabei an, wieviele Hops ein Server von einer externen Uhr entfernt ist. Ist die externe Uhr direkt an den Server angeschlossen handelt es sich um einen

Stratum 1-Server. Bezieht ein Server seine Zeit von einem Stratum 1-Server handelt es sich um einen Stratum 2-Server.
Abbildung 3.13 veranschaulicht eine mögliche Struktur von *NTP*-Servern und Clients.

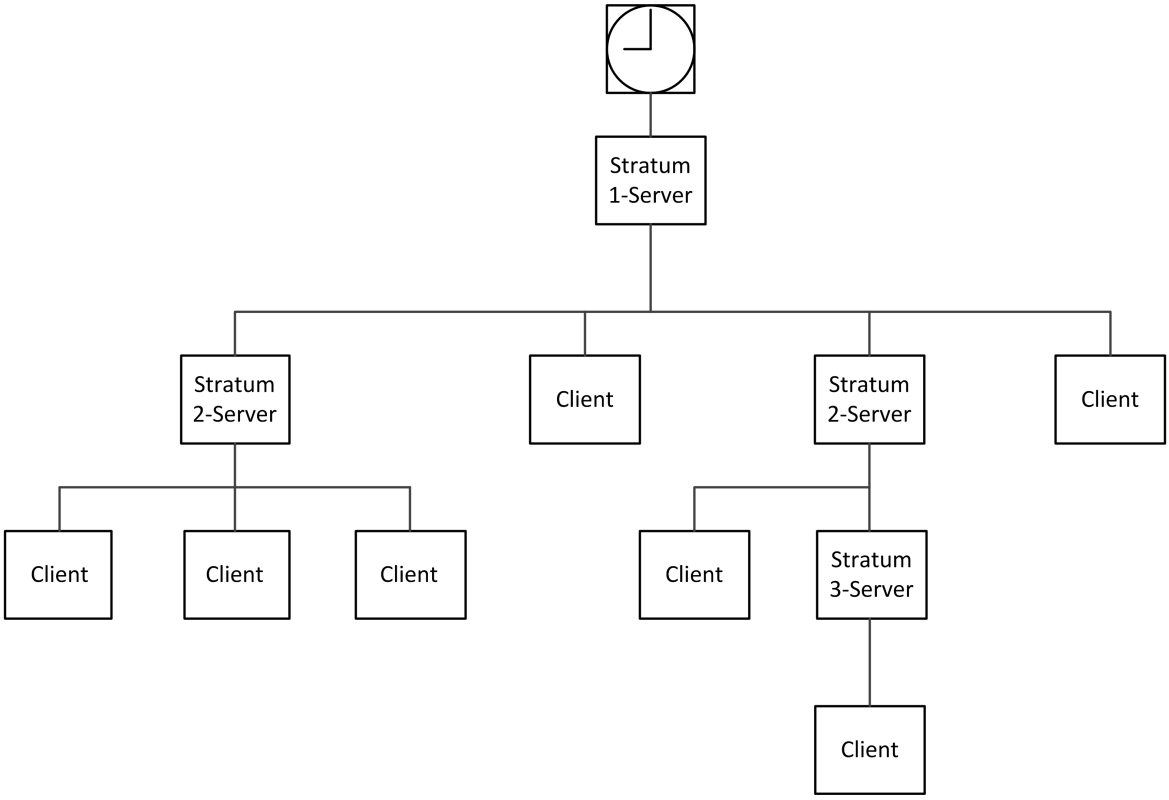

Abbildung 3.13.: *NTP* - Verschiedene Server-Hierarchien

Ein *NTP*-Server ist also immer auch Client, sofern er nicht eine eigene externe Uhr zur Verfügung hat.

Die Synchronisation erfolgt über das Versenden von Zeitstempeln. Problematisch dabei ist die Paketlaufzeit, der Zeitstempel ist also schon veraltet, wenn er beim Client ankommt. Angenommen wird nun, dass ein Paket für den Hin- und den Rückweg die gleiche Zeit benötigt. Der Ablauf ist folgender [10]:

- Client sendet ein Paket mit Zeitstempel t_{C0} an den Server
- Server stellt die Ankunftszeit des Paketes fest (t_{S0})
- Server erstellt ein Antwortpaket, das beide Zeitstempel sowie eine Fehlerangabe (t_{Err}) enthält

- Unmittelbar vor dem Versenden des Antwortpaketes fügt der Server seinen Ausgangszeitstempel t_{S1} hinzu, was den Fehler durch die benötigte Rechenzeit klein halten soll
- Client stellt den Zeitpunkt des Paketempfangs fest t_{C1}

Berechnen lässt sich daraus [10]:

- die Hälfte der Differenz der Paketlaufzeiten ohne Rechenzeit des Servers (t_{Offset}), also der Wert um den sich Client-Uhr und Server-Uhr unterscheiden

$$t_{Offset} = \frac{(t_{S0} - t_{C0}) - (t_{C1} - t_{S1})}{2} \tag{3.3}$$

- die Rundlaufzeit ohne Rechenzeit des Servers (t_{Delay})

$$t_{Delay} = (t_{C1} - t_{C0}) - (t_{S1} - t_{S0}) \tag{3.4}$$

- die Hälfte von t_{Delay}, also der größtmögliche Fehler von t_{Offset}

$$t_{Dispersion} = \frac{t_{Delay}}{2} + t_{Err} \tag{3.5}$$

Die Uhr des Clients muss demnach im Bereich $(t_{Offset} - t_{Dispersion})$ .. $(t_{Offset} + t_{Dispersion})$ angepasst werden. Davon ausgehend, dass der wahre Wert in der Mitte liegt, wird die Uhr angepasst [10]. Dabei gibt es keine Zeitsprünge, sondern die Client-Uhr wird verlangsamt bzw. beschleunigt, was auch der Grund dafür ist, dass der gesamte Vorgang, je nach erforderlicher Genauigkeit, sehr lange dauern kann.

Mit dem Kommando *ntpq -p* können (unter Linux mit installiertem *NTP*) die Statistiken der aktuellen *NTP*-Synchronisation abgerufen werden. Abbildung 3.14 zeigt die Ausgabe beispielhaft. Konfiguriert sind vier Zeitserver. Die wichtigsten Spalten sollen hier erläutert werden [7]:

- das Zeichen vor dem Servernamen gibt den Vertrauensstatus an
  * aktuelle Referenz
  + geht in die Zeitberechnung mit ein
  - liefert zu weit abweichende Werte
  # geeignet, aber in der Bewertungsreihenfolge weiter hinten
  x über längere Zeit keine verlässlichen Werte
- refid gibt die Zeitquelle des Servers an
- st gibt das Stratum des Servers an
- delay gibt die Rundlaufzeit in ms an
- offset gibt den aktuellen Offset der Systemzeit zur Serverzeit in ms an

```
linux-25xg:/home/stephan # ntpq -p
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
+ptbtime1.ptb.de .PTB.            1 u 1031 1024  377   20.724    5.360   0.579
*ptbtime2.ptb.de .PTB.            1 u  342 1024  377   21.022    5.582   0.463
+ntp0.rrze.uni-e .GPS.            1 u    3 1024  377   30.152    5.658   5.234
+ntp2.rrze.uni-e .GPS.            1 u  883 1024  377   29.944    6.372   0.951
```

Abbildung 3.14.: *NTP*-Statistiken

Die zugehörige *NTP*-Konfigurationsdatei befindet sich in Anhang D.

3.5. Häufigkeitsverteilungen

Die Abbildung einer Häufigkeitsverteilung stellt eine Methode zur Datenanalyse dar. Aus der Beobachtung der Ausprägung eines Merkmals (z. B. ein Messwert), kann eine Datenreihe (die sogenannte Urliste) gewonnen werden, die die aufgenommenen Werte enthält. Ausgewertet wird die Häufigkeit des Auftretens eines jeden Wertes. Die Häufigkeit h kann für das Merkmal a mit den Ausprägungen a_i absolut angegeben werden [15]:

$$h(a_i)\ mit\ i = 1, 2, .., k \tag{3.6}$$

Alternativ ist eine Angabe der Häufigkeit f relativ zur Gesamtzahl n der Werte (dem Probenumfang) möglich.

$$f(a_i) = \frac{h(a_i)}{n} \tag{3.7}$$

Bei stetigen Merkmalsausprägungen tritt jeder Wert nur einmal auf. Auch bei diskreten Merkmalen kann das der Fall sein, wenn die Anzahl der möglichen Werte größer ist als der Probenumfang. Der Gesamtbereich kann dann in Intervalle (Klassen) geteilt werden. In der Häufigkeitsverteilung wird in diesem Fall die Besetzung der Klasse angegeben.

Bei klassierten Daten bietet sich die Darstellung als Histogramm an. Abbildung 3.15 zeigt beipielhaft die Verteilung der Zwischenankunftszeit von Datenpaketen an einem Netzwerkinterface.

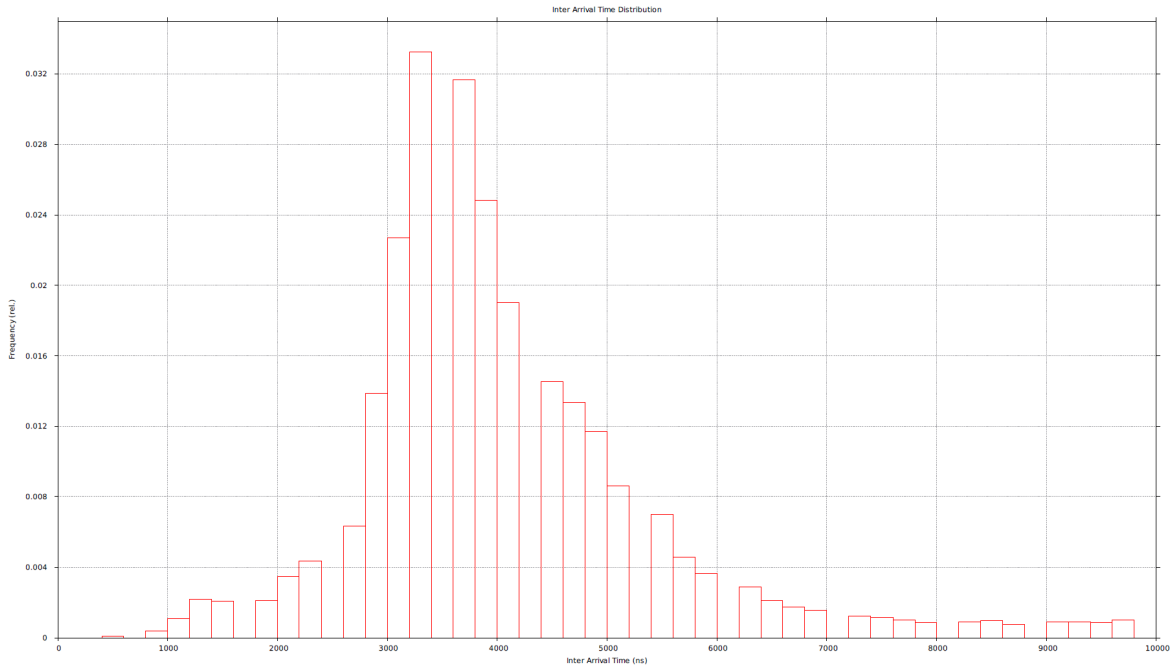

Abbildung 3.15.: Häufigkeitsverteilung Zwischenankunftszeit Datenpakete

Die Abbildung zeigt nur einen Ausschnitt des Messbereichs. Die Addition der dargestellten relativen Häufigkeiten ergibt daher einen Wert kleiner 1.
Anhand der Balkenbreite kann unmittelbar auf die Klassenbreite geschlossen werden.

4. Erzeugung von Netzwerkverkehr

4.1. Anforderungen

Einleitend wurde erwähnt, dass die Implementierung eines Netzwerkverkehrsgenerators erfolgen soll. Der Zweck ist die Verifizierung der Ergebnisse des Analysators, aber auch das gezielte Erzeugen von störender Netzwerklast, um mit dem Analysator die Auswirkungen zu untersuchen.

Anforderungen an den Generator sind folgende:

- einstellbare Datenrate
- *IPv4* als Vermittlungsschichtprotokoll mit vollständig konfigurierbarem Header
- optional *TCP* oder *UDP* als Transportschichtprotokoll mit jeweils vollständig konfigurierbarem Header
- verschiedene Nutzdatenmuster und einstellbare Nutzdatenlänge
- einstellbare Laufzeit
- steuerbar über Socket von extern

4.2. Aufbau

Abbildung 4.1 zeigt, stark vereinfacht, den Programmaufbau des Paketgenerators. Damit der Generator zu jedem Zeitpunkt steuerbar ist, wird in einem eigenen Thread auf Steuerbefehle gewartet. Das eigentliche Erzeugen des Netzwerkverkehrs findet im sogenannten Generator-Thread statt.

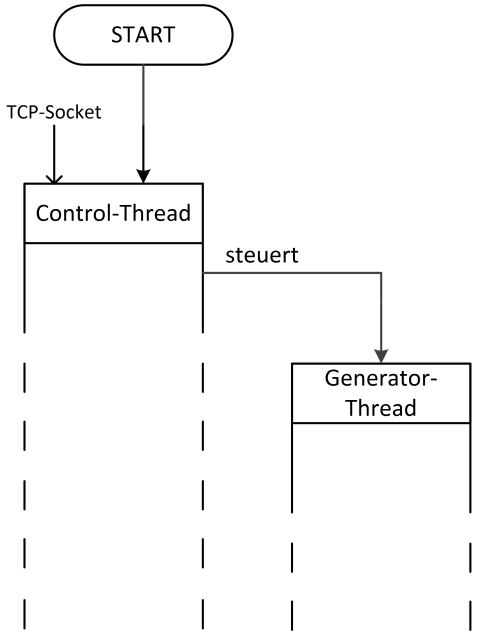

Abbildung 4.1.: Arbeitsweise Paketgenerator

Neben der Steuerbarkeit bietet dieses Design den weiteren Vorteil, dass jeder Thread für sich priorisiert werden kann, also die gewünschte Behandlung durch den Scheduler erfährt.

4.3. Paketerzeugung

Für die Paketerzeugung wird ein RAW-Socket ($\rightarrow$ 3.2) verwendet. Dieser bietet die Möglichkeit den Inhalt des zu versendenden Paketes zu manipulieren, bzw. erfordert, dass das Paket vom Vermittlungsschichtprotokoll aufwärts generiert wird. Der Ablauf der Paketgenerierung soll im Folgenden anhand von Code-Fragmenten erläutert werden.

Der erste Schritt ist das Erstellen des RAW-Sockets (→ Listing 4.1).

```
1 // from generator_thread.c
2 sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
```

Listing 4.1: Erstellen des RAW-Sockets

Die Funktion liefert einen Datei-Deskriptor für den Socket zurück. Die Argumente sorgen dafür, dass *IPv4* als Vermittlungsschicht-Protokoll verwendet wird und das ein RAW-Socket erstellt wird, also kein Transportprotokoll vorgesehen ist.

Für das Erstellen eines Paketes stehen in der Programmiersprache C verschiedene Strukturen zur Verfügung, die die Protokollheader abbilden (→ Anhang A, Listing A.1 bis A.5).

Bevor ein Paket versendet werden kann, wird es im Speicher zusammengestellt. Dazu wird ein Speicherbereich allokiert, den das Paket einnehmen kann (→ Listing 4.2).

```
1 // from generator_thread.c
2 char* packet = (char*) malloc(MAX_PACKET_SIZE);
```

Listing 4.2: Bereitstellung von Speicher für Paketgenerierung

Es wird Speicher für die maximal mögliche Paketgröße allokiert. Die echte Größe des Paketes wird beim Versenden angegeben.

In diesen Speicherbereich werden nun Zeiger auf die Protokollheaderstrukturen gesetzt, im Beispiel für ein *UDP*-Paket (→ Listing 4.3).

```
1 // from generator_thread.c
2 struct iphdr* ip = NULL; // pointer to IPv4-header
3 struct udphdr* udp = NULL; // pointer to UDP-header
4
5 ip = (struct iphdr*) packet; // set to its position
6 udp = (struct udphdr*)(packet + sizeof(struct iphdr)); // set to its position
```

Listing 4.3: Pointer auf Protokollheaderstrukuren im allokierten Speicherbereich

Anschließend wird ein Zeiger für den Beginn der Nutzdaten an das Ende des Headers des Transportschichtprotokolls gesetzt (→ Listing 4.4).

```
1 // from generator_thread.c
2 char* pl; // pointer to payload
3
4 pl = (char*)(packet + sizeof(struct iphdr) + sizeof(struct udphdr));
```

Listing 4.4: Pointer auf die Startadresse des Nutzdatenabschnitts

Im Ergebnis können die angesprochenen Bestandteile des Paketes manipuliert werden, indem die Werte den entsprechenden Strukturvariablen zugewiesen werden, beispielhaft in Listing 4.5 für das bereits besprochene *UDP*-Paket veranschaulicht.

```
1 // from generator_thread.c
2 ip->version = 4; // Protocol-Version
3 ip->ihl = 5; // header length
4 ip->id = 0; // ID, set by kernel
5 ip->ttl = ttl; // Time-to-live
6 ip->saddr = inet_addr(args->s_ip); // source-address, conversion from char*
7 ip->daddr = inet_addr(args->d_ip); // destination-address, conversion from char
    *
8 ip->check = 0; // Checksum, set by kernel
9 ip->tos = tos; // Type of Service
10
11 ip->tot_len = sizeof(struct iphdr) + sizeof(struct udphdr) + data_len; //
    length of Ipv4-Header and payload
12 udp->dport = htons(d_port); // Destination-Port
13 udp->sport = htons(s_port); // Source-Port
14 udp->len = htons(sizeof(struct udphdr) + data_len); // Length of L4-Header and
    payload
```

Listing 4.5: Füllen der Protokollheader

Wie in 3.2 erwähnt, werden Teile des *IPv4*-Headers durch die Verwendung des RAW-Sockets optional vom Betriebssystem generiert. Im Beispiel sind das die ID (Listing 4.5, Zeile 3) und die Prüfsumme (Listing 4.5, Zeile 7). Die Werte werden immer dann (unmittelbar vor dem Paketversand) vom Betriebssystem generiert, wenn sie den Wert 0 haben.

Im letzen Schritt werden die Nutzdaten in der vom User gesetzten Länge im entsprechenden Speicherbereich generiert (→ Listing 4.6). Im Beispiel werden Nullbits erzeugt.

```
1 // from generator_thread.c
2 memset(pl, 0, data_len);
```

Listing 4.6: Nutzdatengenerierung

Abbildung 4.2 zeigt den beschriebenen Ablauf der Paketerstellung.

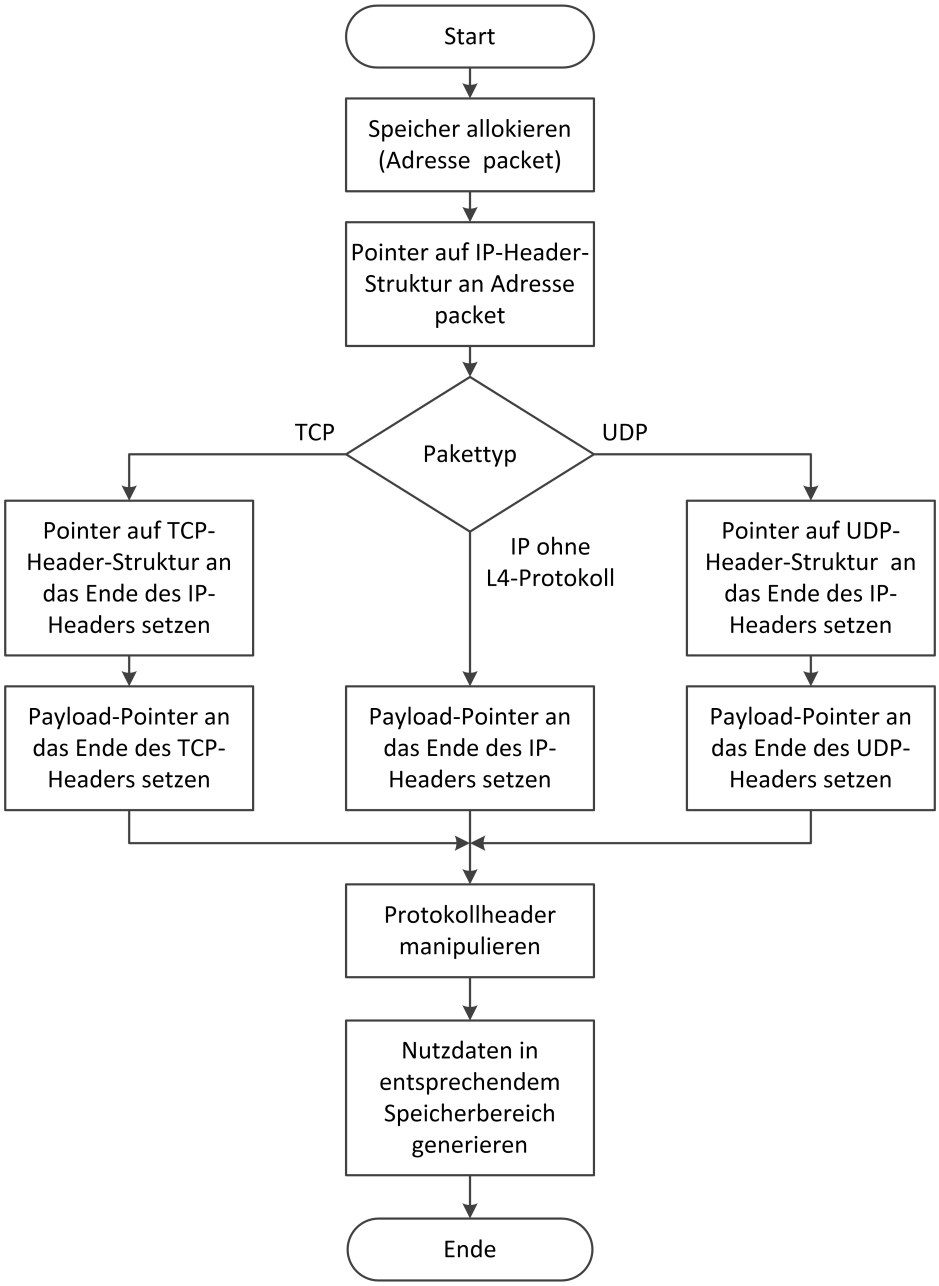

Abbildung 4.2.: Ablauf Paketerstellung

4.4. Paketversand

Im folgenden Abschnitt wird beschrieben, wie aus den Einzelpaketen eine Datenrate generiert wird.

Um die gewünschte Datenrate zu erzielen, muss eine passende Paketrate generiert werden. Der Zusammenhang ist folgender:

$$DR = PS \cdot PR \tag{4.1}$$

mit Paketrate PR in $\frac{1}{s}$, Datenrate DR in $\frac{Byte}{s}$ und Paketgröße PS in $Byte$.

Um den Fehler bei der Zeitmessung zwischen dem Versand zweier Pakete zu verringern, wird nach Ablauf einer festen Zeit t_{delta} die Anzahl Pakete versendet, die nötig ist, um bis zum aktuellen Zeitpunkt die gewünschte Datenrate zu generieren. Dazu wird die Startzeit t_{start} ermittelt und zu jedem Zeitpunkt $t_{start} + i \cdot t_{delta}$ die erforderliche Anzahl Pakete n_{burst} ermittelt, die im nächsten Burst versendet werden müssen. Für die Berechnung berücksichtigt wird die Zeit seit Start des i-ten Durchlaufs $i \cdot t_{delta}$, sowie die in den Bursts $1..i-1$ bereits versendeten Pakete $n_{packets}$:

$$n_{burst} = \frac{DR \cdot i \cdot t_{delta}}{PS} - n_{packets} \tag{4.2}$$

Abbildung 4.3 zeigt den zeitlichen Ablauf der Generierung einer vorgegebenen Datenrate.

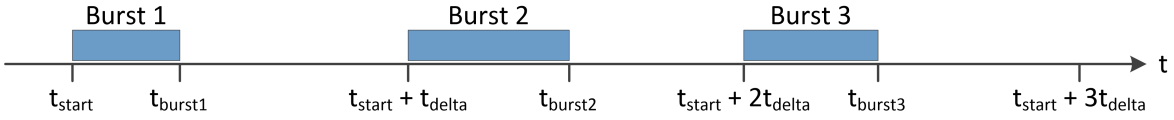

Abbildung 4.3.: Generierung der Datenrate

Wie Abbildung 4.3 entnommen werden kann, muss die Dauer des Paketbursts i ermittelt werden, um zum korrekten Zeitpunkt $t_{start} + (i+1) \cdot t_{delta}$ die erforderlich Paketanzahl für den Paketburst $i+1$ zu berechnen.

Daraus folgt der Programmablauf aus Abbildung 4.4.

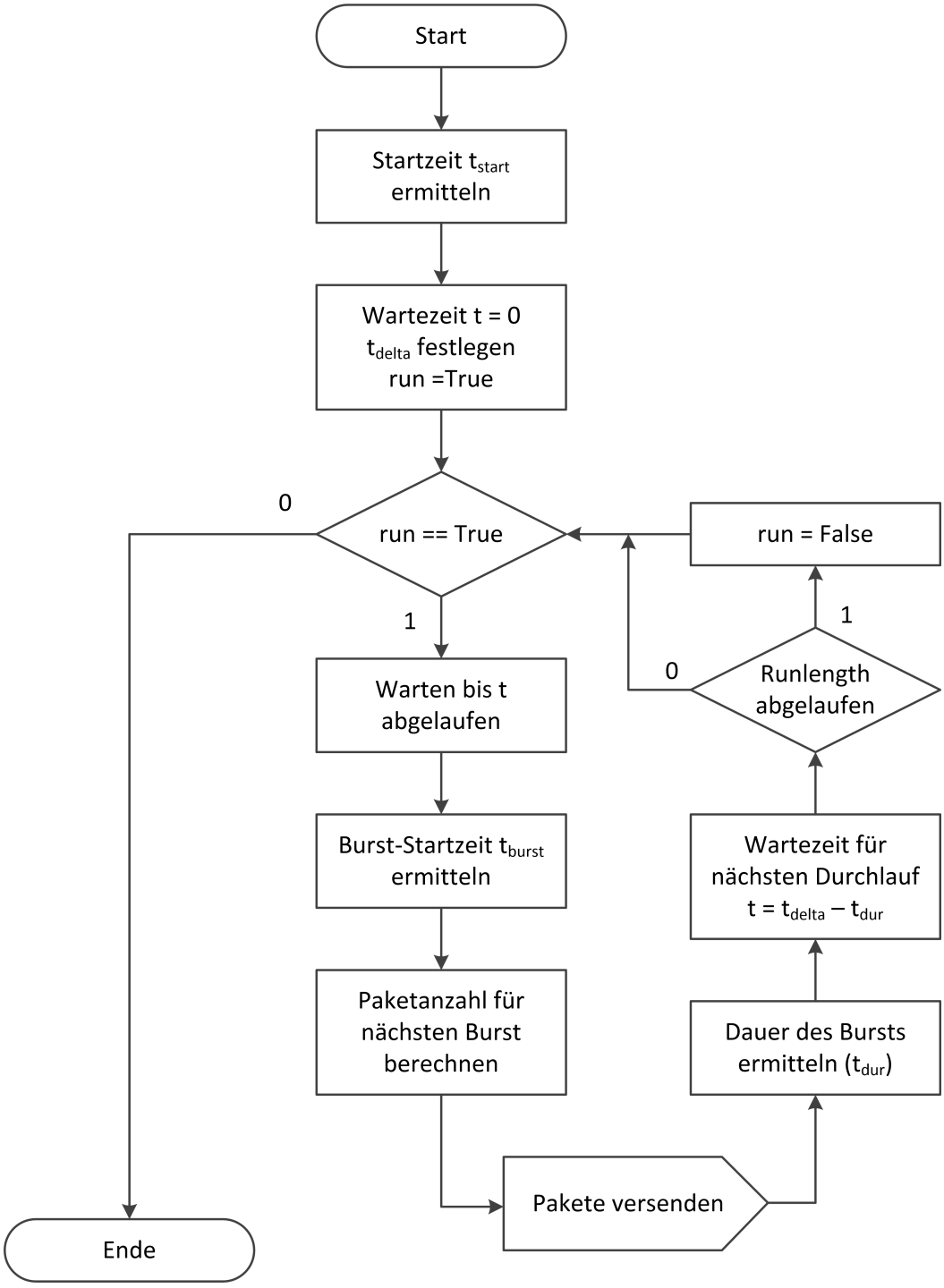

Abbildung 4.4.: Ablauf Generierung der Datenrate

Die Umsetzung dieses Ablaufs zeigt Listing 4.7.

```
1  // from generator_thread.c
2  while (run == true) {
3    select(0, NULL, NULL, NULL, &t); // wait for next burst
4    clock_gettime(CLOCK_REALTIME, &t_burst); // burst start time
5    subtract_tspec (&t_delta, &t_burst, &t_start); // time since generator-start
       in t_delta
6    burst = ((args->datarate * (t_delta.tv_sec + t_delta.tv_nsec * NANO)) /
       packetsize) - packets_sent;
7    // calculate number of packets for next burst
8    burst = min(MAX_BURST, burst); // burst <= MAX_BURST
9    for (i = 0; i < burst; i++) { // send packets
10     sendto(sock, packet, ip->tot_len, 0, (struct sockaddr*)&dest, sizeof(dest))
       ;
11   }
12   packets_sent += burst;
13   clock_gettime(CLOCK_REALTIME, &t_this); // current time
14   subtract_tspec(&t_duration, &t_this, &t_burst); // real burst duration in
       t_duration
15
16   subtract_tspec(&t_temp, &args->delta_t, &t_duration); // time to wait =
       delta_t - t_duration
17   if (t_temp.tv_sec < 0 || t_temp.tv_nsec < 0) {
18     t.tv_sec = 0;
19     t.tv_usec = 0;
20   }
21   else {
22     t.tv_sec = t_temp.tv_sec;
23     t.tv_usec = t_temp.tv_nsec/1000;
24   }
25   if (args->runlength != 0) { // end of run?
26     clock_gettime(CLOCK_REALTIME, &t_this); // current time
27     if ((t_this.tv_sec >= t_end.tv_sec) && (t_this.tv_nsec >= t_end.tv_nsec)) {
28       run = false;
29     }
30   }
31 }
```

Listing 4.7: Paketbursts zur Erzeugung einer vorgegebenen Datenrate

Für das Speichern der Zeiten werden jeweils Strukturen vom Typ *struct timespec* verwendet (→ Anhang A, Listing A.6). Zeile 5 zeigt die Berechnung von n_{burst}, in Zeile 9 wird diese Anzahl Pakete versendet. Wie in 4.3 erwähnt, erhält die Funktion *sendto()* die Paketgröße als Argument (ip->tot_len). In Zeile 15 wird, wie angesprochen, die Zeit berechnet, die bis zum nächsten Durchlauf noch gewartet werden muss.

Im Programmablauf (Abb. 4.4) nicht enthalten sind die Limitierung des Burst-Faktors auf den

Wert *MAX_BURST* (Zeile 7) und die Behandlung von negativen Zeiten (Zeile 16). Durch Ersteres wird sichergestellt, dass sich bei Überlastung des Generators (es können nicht so viele Pakete wie nötig generiert werden) kein Berg von Paketen anstaut und die Zeit *t_duration*, die ein Burst benötigt, beständig größer wird. Das hätte zur Folge, dass im nächsten Burst wiederum mehr Pakete gesendet werden müssten.

Zweiteres sorgt dafür, dass die Wartezeit (Zeile 2) in jedem Fall $t \geq 0$ ist.

Weiterhin können mehrere dieser Durchläufe nacheinander generiert werden. Der Anwender legt dabei die Dauer eines Durchlaufs t_{Run} sowie die Zeit zwischen den Durchläufen t_{Pause} fest (zugehörige Befehle siehe Anhang B). Abbildung 4.5 zeigt ein entsprechendes Beispiel.

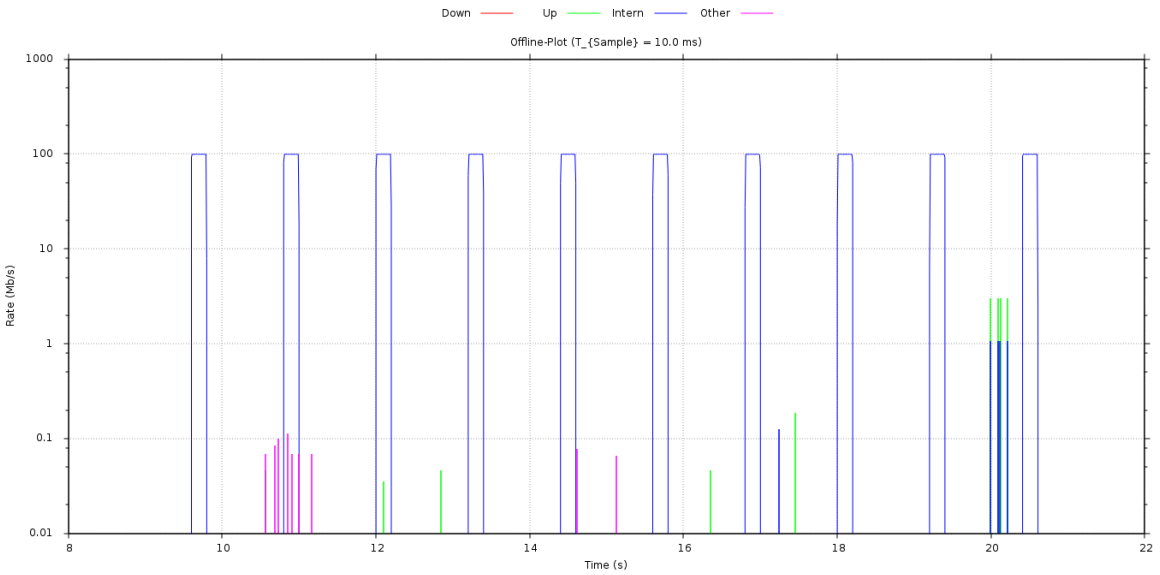

Abbildung 4.5.: Paketbursts (blau, mit Datenrate $DR = 100\ \frac{Mb}{s}$) mit $t_{Run} = 200ms$ und $t_{Pause} = 1s$

Um die Erkennung des generischen Netzwerkverkehrs zu ermöglichen, können die versendeten Pakete gesondert gekennzeichnet werden. Der Typ des Datenmusters kann auf den Wert *TAG* gesetzt werden. Zu Beginn der Nutzdaten wird dann eine bestimmte Zeichenkette gesetzt und die restliche Nutzdatenlänge mit Nullen aufgefüllt. Analysatorseitig werden diese Pakete erkannt und gesondert behandelt ($\rightarrow$ 5.3).

Ein weiteres Feature muss im derzeitigen Stadium als experimentell angesehen werden. Der Generator ist in der Lage Pakete mit exponentialverteilter Zwischenankunftszeit zu versenden. Zur Anwendung kommt dabei die Inversionsmethode, die es ermöglicht gleichverteilte Zufallszahlen in exponentialverteilte zu überführen.

Für die Exponentialverteilung gilt die Dichtefunktion [17]:

$$f(u) = \lambda \cdot e^{-\lambda \cdot u} \tag{4.3}$$

Daraus lässt sich die Verteilungsfunktion wie folgt berechnen:

$$F(x) = \int_{-\infty}^{x} f(u)\, du. \; = 1 - e^{-\lambda \cdot x} \tag{4.4}$$

Bildet man daraus mit $F(x) = y$ die Inverse ergibt sich:

$$x = -\frac{1}{\lambda} \cdot ln(1 - y) \tag{4.5}$$

Setzt man nun für y gleichverteilte Zufallszahlen im Intervall $[0, 1]$ ein, erhält man für x exponentialverteilte Werte. Listing 4.8 zeigt die Umsetzung im Code.

```
1  // from generator_thread.c
2  while (run == TRUE) {
3    pselect(0, NULL, NULL, NULL, &t_pselect, NULL); // timeout
4    equal = (double)(((double)(rand()))/((double)(RAND_MAX)));
5    // equal distributed random number [0,1]
6    exp = EXP_OFFSET + (EXP_MULT * ((-1/EXP_LAMBDA) * log(1-equal)));
7    // exp distributed number
8    t_pselect.tv_nsec = (long)(exp);
9
10   sendto(sock, packet, ip->tot_len, 0, (struct sockaddr*)&dest, sizeof(dest));
11   packets_sent++;
12 }
```

Listing 4.8: Paketversand mit exponentialverteilter Zwischenankunftszeit

Offset und Multiplikator sorgen dafür, dass die resultierenden Werte in einem gewünschten Bereich liegen und beeinflussen die sich ergebene Datenrate.

4.5. Laufzeitanalyse

Eine weitere Aufgabe des Paketgenerators ist das Erzeugen von Paketen, die eine Messung der Paketlaufzeit ermöglichen. Verwendet wird dafür ein Paket mit *IPv4* als Vermittlungsschichtprotokoll. Auf ein Transportschichtprotokoll wird verzichtet, da auf der Gegenseite (→ 5.5) ein Packet-Socket (→ 3.2) eingesetzt wird, dem sämtliche Pakete der Schnittstelle zugeleitet werden. Ein Anwendungsbezug ist somit nicht erforderlich.

Erzeugt werden Pakete, die einen Zeitstempel enthalten. Angegeben werden Millisekunden seit dem 1.1.1970, 0 Uhr *UTC* (sogenannte Unix-Epoche). Um die Pakete während einer laufenden Analyse einfach detektieren zu können, wird eine bestimmte Zeichenfolge („MAGIC PAYLOAD“) verwendet. Da der Gegenstelle das Format bekannt ist, kann der Zeitstempel einfach extrahiert werden. Listing 4.9 zeigt das Generieren des Pakets.

```
1  // from generator_thread.c
2  char* packet = (char*) malloc(MAX_PACKET_SIZE); // memory for packet to build
3  struct iphdr* ip = (struct iphdr*) packet; // ip header first
4  char* payload = (char*)(packet + sizeof(struct iphdr)); // pointer to payload
5
6  while (run == true) {
7    t.tv_sec = 0;
8    t.tv_usec = TIMESTAMP_PACKETS_PAUSE_US; // default 100ms
9    select(0, NULL, NULL, NULL, &t);
10   clock_gettime(CLOCK_REALTIME, &t_timestamp);
11   ms_since_epoch = (t_timestamp.tv_sec * 1000) + (u_int64_t)(t_timestamp.
       tv_nsec / 1000000); // convert to ms since epoch
12   sprintf(payload, "%s.%ld", MAGIC_PAYLOAD, ms_since_epoch);
13   ip->tot_len = sizeof(struct iphdr) + strlen(payload); // packet-length
14   sendto(sock, packet, ip->tot_len, 0, (struct sockaddr*)&dest, sizeof(dest));
15 }
```

Listing 4.9: Erzeugen und Versand eines Pakets mit Zeitstempel

Das Paket wird im Wesentlichen wie in 4.3 beschrieben generiert und die Headerfelder gefüllt. Die Nutzdaten müssen hier natürlich für jedes Paket einzeln erzeugt werden.

Um auch während der Laufzeitanalyse die normale Analyse des Netzwerkverkehrs parallel zu ermöglichen, werden die Zeitstempel-Pakete in relativ großen Abständen versendet (default $t_{delta} = 100ms$). Das vermeidet nicht nur eine verfälschende Belastung des Paketanalysators, sondern sorgt auch für einen möglichst kleinen Einfluss auf die Auslastung der übrigen Komponenten im Netzwerk.

4.6. Steuerung

Zur Steuerung des Generators ist ein *TCP*-Socket vorgesehen ($\rightarrow$ 4.1). Dieser nimmt während der Programmlaufzeit Befehle entgegen. Wie diese Befehle abgesetzt werden spielt damit keine Rolle. Von der Kommandozeile (bspw. per *netcat*) ist ebenso denkbar, wie von einer *GUI*-Anwendung. Je nach Netzwerkkonfiguration ist die Anwendung aus dem lokalen

Netz oder auch aus dem Internet erreichbar. Eine Fernsteuerung ist demnach problemlos möglich.

Um die Steuerung zu jedem Zeitpunkt zu ermöglichen, wird der Netzwerkverkehr in einem separaten Thread generiert. Im Hauptthread wird per *TCP*-Socket (Port 4000) weiter auf Befehle gewartet. Der Funktion, die in diesem eigenen Thread ausgeführt wird, kann aufgrund der Verwendung von *POSIX*-Threads nur ein einzelner Pointer als Argument übergeben werden [29]. Den Funktionsprototypen zeigt Listing 4.10.

```
1 // from generator2.h
2 void* generator_thread (void* arguments);
```

Listing 4.10: Funktionsprototyp für Generator-Thread

Als Argument wird ein Zeiger auf eine Struktur verwendet, die sämtliche Einstellungen für den Thread enthält. Anhang A (Listing A.7) zeigt die verwendete Struktur.

Eine Änderung der Einstellungen bei laufendem Generator-Thread würde sich direkt auswirken und hätte Probleme beim Erzeugen der Datenrate zur Folge. Die Einstellungen können daher nur bei gestopptem Generator-Thread geändert werden.

Sämtliche Steuerbefehle für den Paketgenerator sind in Anhang B, in Tabelle B.1 aufgeführt. Gesendet werden können einzelne Befehle oder eine Zeichenkette, die mehrere Befehle enthält. Als Trennzeichen sind Komma, Semikolon und Zeilenumbrüche zulässig (keine Leerzeichen). Die *TCP*-Steuerverbindung kann aufrecht erhalten oder nach jedem Befehl (bzw. einer Befehlssequenz) wieder abgebaut werden.

Ein Skript, das mehrere Befehle per *netcat* an einen lokal laufenden Generator sendet, ist beispielhaft in Listing 4.11 dargestellt. Hierbei werden einige Einstellungen gesetzt und der Generator gestartet.

```
1 #!/bin/bash
2 echo -n SETPTYPE IP,SETDATARATE 10240,SETDTYPE ONES,SETRUNLENGTH_S 30,
    SETDATALEN 256,START | netcat -w 1 localhost 3000
```

Listing 4.11: Beispielskript zur Generatorsteuerung

4.7. Performance-Bewertung Einplatinen-Computer

Wie in 4.4 beschrieben werden Pakete nicht einzeln verschickt, sondern in Form von Bursts. Der erwähnte Burst-Faktor kann nun genutzt werden, um die Performance eines Systems zu bewerten, das die Aufgabe des Paketgenerators übernehmen soll. Steigt der Burst-Faktor

kontinuierlich an, kann davon ausgegangen werden, dass das System nicht in der Lage ist die für die geforderte Datenrate benötigte Paketrate zu generieren. Um die maximal mögliche Paketrate zu ermitteln, wird eine möglichst kleine Paketgröße verwendet, da so schon eine relativ geringe Datenrate genügt.

Wie einleitend erwähnt, soll festgestellt werden, wie leistungsfähig ein *Einplatinen-Computer* im Vergleich zu einem vollwertigen Computer ist. Folgende Systeme stehen für den Vergleich zur Verfügung:

Tabelle 4.1.: Übersicht Systeme für Performance-Bewertung

System	CPU	RAM	NIC	OS
Raspberry Pi B	ARMv6 (1 Kern, 700 MHz)	512 MB	100 MBit/s	Debian Linux 8.0
Raspberry Pi 3B	ARMv8 (4 Kerne, 1.2 GHz)	1 GB	100 MBit/s	Debain Linux 8.0
Odroid C2	ARMv8 (4 Kerne, 2.0 GHz)	2 GB	1 GBit/s	Debian Linux 8.0
Lenovo X220	Intel i5 (2 Kerne, 2.5 GHz)	8 GB	1 GBit/s	SuSE Linux 13.2

Untersucht wird, wie sich Scheduler-Optimierungen ($\rightarrow$ 3.3) auswirken. Ermittelt werden sollen die maximale Paketrate und die zugehörige Datenrate für die vorgestellten Systeme mit folgenden Einstellungen:

- ohne Scheduler-Optimierung
- Generator-*Task* in der Scheduler-Klasse „rt_sched_class“ mit höchstmöglicher Priorität und Scheduling-Strategie SCHED_FIFO

Um die maximale Paketrate zu ermitteln, wird der Paketgenerator auf dem jeweiligen System mit folgenden Einstellungen gestartet:

- Pakettyp: IP
- Nutzdatenlänge: 32 Byte (daraus resultiert eine Paketgröße von 70 Byte)
- Datenrate 1000 MBit/s

Offensichtlich kann die Datenrate mit derart kleinen Paketen nicht erreicht werden, was dazu führt, dass der Generator die für das System maximale Paketrate produziert. Als Gegenstelle dient ein Desktop-System, dass die Pakete aufgrund einer falschen *IPv4*-Header-Prüfsumme direkt verwirft.

Die Ergebnisse zeigt Tabelle 4.2. Die maximalen Paketraten der *Einplatinen-Computer* unterscheiden sich recht deutlich. In jedem Fall kann die jeweilige Netzwerk-Schnittstelle bei entsprechender Paketgröße voll ausgelastet werden, dennoch ist die Performance der *Einplatinen-Computer* nicht mit der eines vollwertigen Systems vergleichbar.

Tabelle 4.2.: Ergebnis-Übersicht für Performance-Bewertung

System	Raspberry Pi B	Raspberry Pi 3B	Odroid C2	Lenovo X220
Max. Paketrate ohne Scheduler-Optimierung in $\frac{Pakete}{s}$	28000	42000	108000	492000
zugehörige Datenrate in $\frac{Mb}{s}$	14,95	22,43	57,68	262,76
Max. Paketrate mit Scheduler-Optimierung in $\frac{Pakete}{s}$	29000	47000	113000	500000
zugehörige Datenrate in $\frac{Mb}{s}$	15,49	25,10	60,35	267,03

5. Analyse des Netzwerkverkehrs

5.1. Anforderungen

Der Verkehrsanalysator ist die zentrale Komponente dieser Arbeit. Die, über allem stehende, Hauptanforderung leitet sich daher direkt aus dem Titel der Arbeit ab. Netzwerkverkehr soll mit einer *Sampleperiode* von $T_S < 1s$ analysiert werden. Dabei soll eine Klassifizierung des Verkehrs in die Klassen „Down“, „Up“, „Intern“ und „Other“ erfolgen. Die Analyse soll für alle Pakete, die *IPv4* oder *IPv6* als Vermittlungsschichtprotokoll nutzen, funktionieren. Das verwendete Netzwerkinterface muss dabei in den *Promiscuous Mode* gebracht werden, damit der Analysator auch an einem Mirror-Port arbeiten kann. Die erfassten Datenraten sollen darüber hinaus geeignet visualisiert werden und in einem passendem Format für eine spätere Auswertung zur Verfügung stehen.

Neben den Datenraten soll auch die Zwischenankunftszeit der Pakete erfasst werden. Die Darstellung soll in einem Histogramm erfolgen. Gleiches gilt für die Paketlaufzeit. Die vom Generator mit Zeitstempel versehenen Pakete müssen geeignet ausgewertet und visualisiert werden.

Für die Steuerung des Analysators gelten die gleichen Anforderungen wie für die Generatorsteuerung (siehe 4.1). Auch hier soll die Steuerung aus externen Programmen per Socketschnittstelle möglich sein.

5.2. Aufbau

Da auch bei aktuellen *Einplatinen-Computern* in der Regel mehrere Prozessorkerne zur Verfügung stehen, soll in jedem Fall zumindest die Paketerfassung in einem eigenen Thread stattfinden. Die aktuell gesetzten Einstellungen entscheiden darüber, ob es weitere nebenläufige Threads gibt, um z. B. Daten zu plotten, oder zum Zwecke einer späteren Auswertung in Dateien zu schreiben. Teilweise findet zwischen den Threads eine Kommunikation per *IPC* statt. Verwendet werden dafür *Named Pipes*.

Abbildung 5.1 zeigt vereinfacht den Programmaufbau.

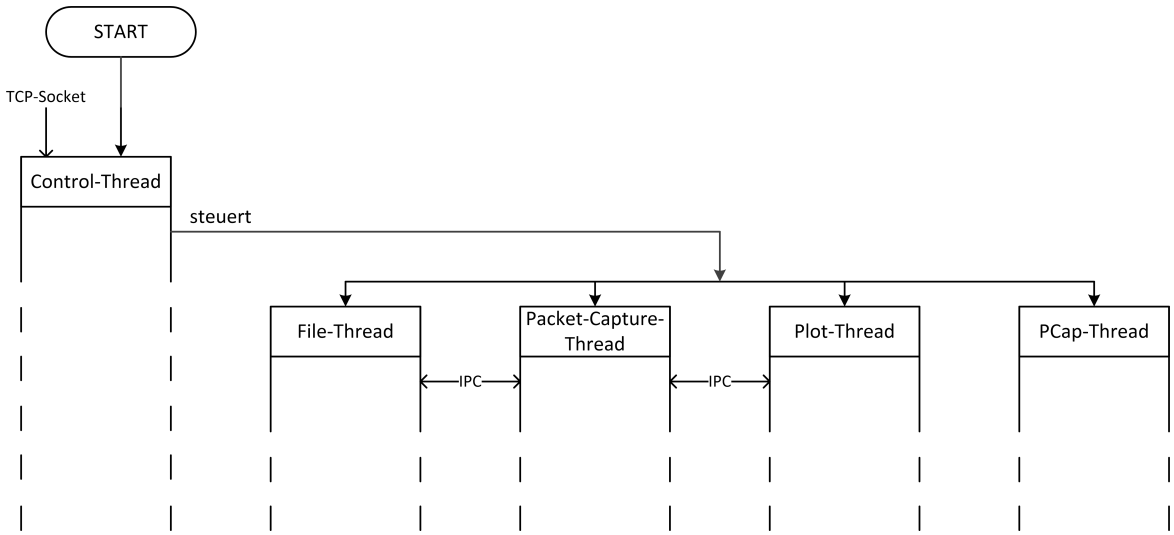

Abbildung 5.1.: Arbeitsweise Verkehrsanalysator

Das System kann somit an die Hardware angepasst werden, auf der es eingesetzt werden soll. Wie sich die Aktivierung bzw. Deaktivierung einzelner Features auf verschiedenen Systemen auswirkt, wird in Abschnitt 5.11 beschrieben.

5.3. Paketerfassung

Den Ablauf der Paketerfassung und Verarbeitung zeigt Abbildung 5.2. Grau dargestellte Bereiche sind, je nach Einstellung, optional. Die dargstellten Phasen werden anschließend beschrieben.

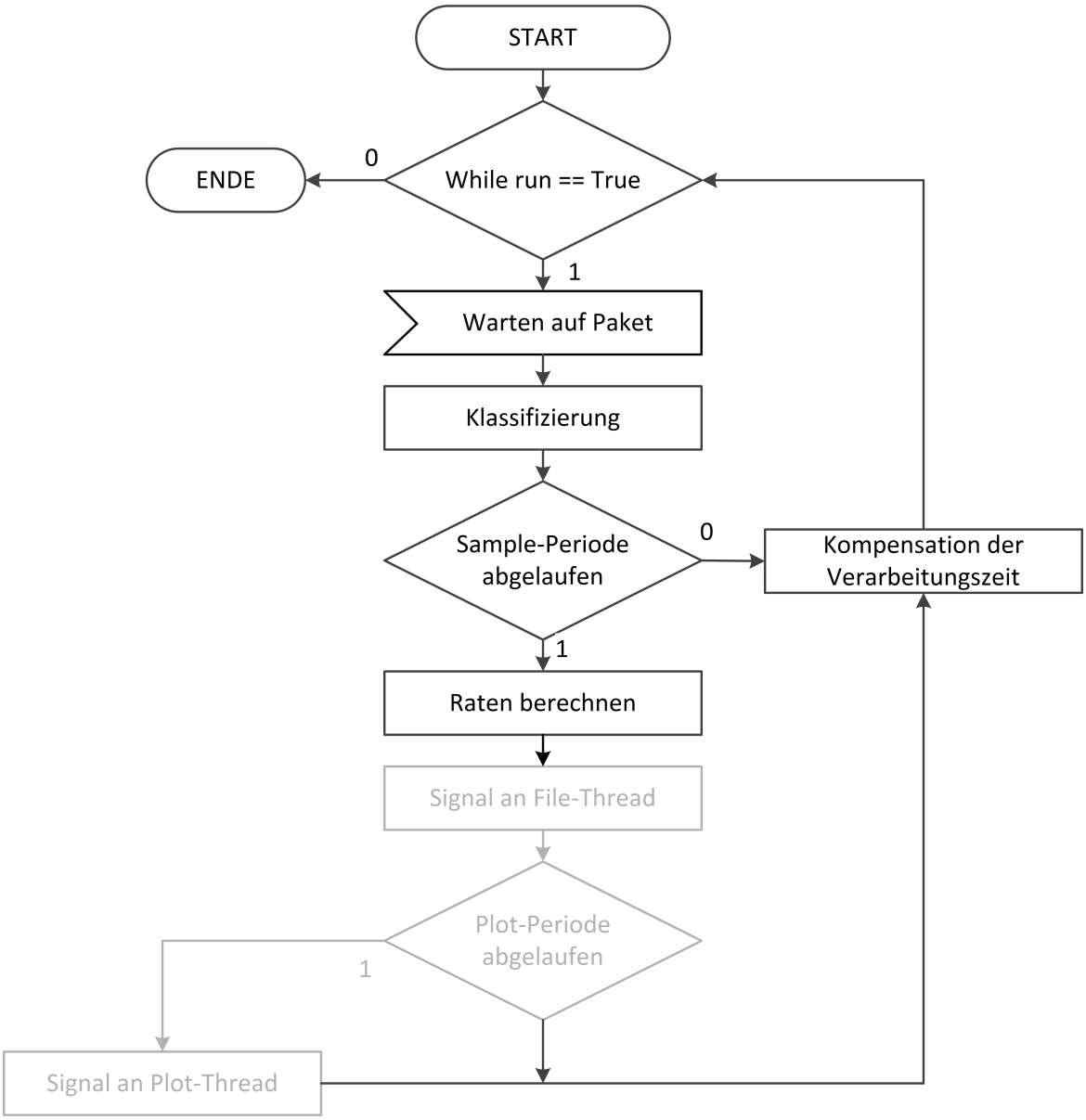

Abbildung 5.2.: Ablauf Paketerfassung

Zur Erfassung der Pakete kommt ein Packet-Socket zum Einsatz (→ 3.2). Wird dieser für ei-

ne Schnittstelle erstellt, werden ihm sämtliche an dieser Schnittstelle ankommenden Pakete zugeleitet, die das übergebene Protokoll benutzen.

Listing 5.1 erstellt einen Packet-Socket. Die übergebenen Argumente sorgen für das angesprochene Verhalten. Erfasst werden nur Pakete die Ethernet als Sicherungsschichtprotokoll nutzen.

```
1 // from rate_mode.c
2 sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

Listing 5.1: Erstellen eines Packet-Sockets

Der erzeugte Socket wird einem File-Deskriptor-Set hinzugefügt. Dieses Set kann mit der *select()*-Funktion auf Lesbarkeit überwacht werden. Die Funktion wird verlassen, wenn ein Paket empfangen wurde oder die Sample-Periode abgelaufen ist. Die in der *timespec*-Struktur (→ Listing A.6) gespeicherte Zeit wird durch die *select()*-Funktion verringert. Um auch die Verarbeitungszeit für den Ablauf der Sample-Periode zu berücksichtigen, wird selbige vom verbleibenden Teil der Sample-Periode abgezogen. Listing 5.2 zeigt das Schema. Die *timespec*-Struktur *t_select_diff* enthält die Verarbeitungszeit, die von *t_select* subtrahiert wird. Sollte daraus ein negativer Wert resultieren, würde die verbleibende Sample-Periode auf 0 gesetzt (nicht im Listing), so dass im nächsten Durchlauf am *select* nicht gewartet werden würde.

```
1  // from rate_mode.c
2  while (run == TRUE) {
3    select(fd_max, &read_fds, NULL, NULL, &t_select);
4    // wait for incoming packets or timeout
5    clock_gettime(CLOCK_REALTIME, &t_select_start);
6    /*
7     * Processing
8     */
9    clock_gettime(CLOCK_REALTIME, &t_select_end);
10   // end processing
11   subtract_tspec(&t_select_diff, &t_select_end, &t_select_start);
12   // calculate processing time (diff = end - start)
13   subtract_tspec(&t_select, &t_select, &t_select_diff);
14   // timeout left: t_select = t_selct - t_select_diff
15 }
```

Listing 5.2: Warten auf ankommende Pakete oder Ablauf der Sample-Periode

Wird ein Paket empfangen, wird es in einem Puffer gespeichert. In diesem Speicherbereich werden, ähnlich wie beim Vorgehen zur Paketerzeugung (→ 4.3), Zeiger auf die Paketheader gesetzt, um die einzelnen Header-Felder auslesen zu können. Aufgrund des

verwendeten Packet-Sockets muss hier beim Ethernet-Header begonnen werden. Listing 5.4 zeigt den Paketempfang und das Setzen des Pointers auf den Beginn des Ethernet-Headers (→ Listing A.1).

```
1 // from rate_mode.c
2 struct ethhdr *ethhdr;
3
4 bytes = recvfrom(sock, buffer, CAPTURE_BUF, MSG_TRUNC, NULL, NULL); // receive
    packet
5 ethhdr = (struct ethhdr *)(buffer); // ethernet-header
```

Listing 5.3: Paketempfang und Setzen des Pointers auf den Ethernet-Header

Nach der Erfassung eines Paketes wird es klassifiziert. Der Analyzer bietet dazu zwei Modi. Der User kann wählen, ob das Paket anhand des Sicherungs- oder des Vermittlungsschichtprotokolls analysiert werden soll.

Für beide Modi ist die Kenntnis der eigenen Adressen (*MAC*, *IPv4*, *IPv6*) erforderlich. Verwendet wird dafür die folgende Funktion:

```
1 int getifaddrs(struct ifaddrs *);
```

Listing 5.4: Ermitteln der Interface-Adressen

Die Funktion gibt einen Zeiger auf eine verkettete Liste mit Strukturen des Typs *struct ifaddrs* (→ Listing A.8) zurück. Jedes Element der Liste repräsentiert dabei eine Adresse. Ein Interface kann daher in mehreren Listenelementen vertreten sein.

Soll das Paket anhand des Sicherungsschichtprotokolls untersucht werden, wird unterschieden, ob der Analysator im *Promiscuous Mode* gestartet wurde oder nicht. Ist der *Promiscuous Mode* aktiv, kann anhand der *MAC*-Adressen nicht unterschieden werden, ob es sich um ein (aus Netzwerksicht) eingehendes, ausgehendes oder intern versendetes Paket handelt. Alle Pakete werden der Klasse „Other“ zugeordnet. Ist der *Promiscuous Mode* deaktiviert, können Quell- und Ziel-*MAC*-Adresse mit der eigenen *MAC*-Adresse verglichen werden. Damit können die Pakete in die Klassen „Upload“, „Download“ und „Other“ eingeteilt werden. Listing 5.5 zeigt die Zuordnung der empfangenen Bytes anhand der besprochenen Vorgehensweise.

```
1 // from rate_mode.c
2 if (promisc == ENABLED) { // promiscuous mode, no traffic-seperation (all in
    other)
3   bytes_other += bytes;
4 }
5 else { // not in promiscuous mode, traffic will be seperated in down/up/other
```

```
6   if (MAC_equals(ethhdr->h_dest, mac_addr) == TRUE) { // Down, dest-addr =
      own_Addr
7     bytes_down += bytes;
8   }
9   else if (MAC_equals(ethhdr->h_source, mac_addr) == TRUE) { // Up, src-addr =
      own_Addr
10    bytes_up += bytes;
11  }
12  else { // Other
13    bytes_other += bytes;
14  }
15 }
```

Listing 5.5: Klassifizierung anhand des Sicherungsschichtprotokolls

Soll die Paketanalyse anhand des Vermittlungsschichtprotokolls erfolgen gestaltet sich der Vorgang etwas umfangreicher. Das Protokoll-Feld des Ethernet-Headers gibt dabei Auskunft über das verwendete Vermittlungsschicht-Protokoll. Analysiert werden *IPv4* und *IPv6*. Alle Pakete mit anderen Vermittlungsschicht-Protokollen werden der Klasse „Other“ zugeordnet (→ Listing 5.6).

```
1 // from rate_mode.c
2 if (htons(ethhdr->h_proto) == 0x0800) {// IPv4
3 iphdr = (struct iphdr *)(buffer + sizeof(struct ethhdr));
4 /*
5  * Processing
6  */
7 }
8 else if (htons(ethhdr->h_proto) == 0x86dd) { // IPv6
9 ipv6hdr = (struct ipv6hdr *)(buffer + sizeof(struct ethhdr));
10 /*
11  * Processing
12  */
13 }
14 else { // OTHER (not IPv4 and not IPv6)
15   bytes_other += bytes;
16 }
```

Listing 5.6: Auswertung Protokoll-Felds im Ethernet-Header und Setzen der entsprechenden Pointer auf die Protokollheader

Im Falle von *IPv4* sieht die Verarbeitung folgendermaßen aus (→ 3.1.3):

- Klassifizierung „Other“, wenn Paket eine bestimmte Zeichenkette enthält, die es als generisches Paket kennzeichnet (→ 4.4)
- Klassifizierung „Intern“, wenn UND-Verknüpfung der Quell- und Ziel-IP-Adresse mit der Netzmaske des Analysators das gleiche Ergebnis haben
- Klassifizierung „Down“, wenn die UND-Verknüpfung der Ziel-IP-Adresse mit der Netzmaske des Analysators die Netzadresse des Analysators zum Ergebnis hat
- Klassifizierung „Up“, wenn die UND-Verknüpfung der Quell-IP-Adresse mit der Netzmaske des Analysators die Netzadresse des Analysators zum Ergebnis hat
- Sonst Klassifizierung „Other“

Listing 5.7 zeigt den entsprechenden Ausschnitt aus dem Quellcode.

```
1  // from rate_mode.c
2  if (strstr(payload, GEN_TAG)) {
3    // Packet from Generator when tagged
4    bytes_other += bytes;
5  }
6  else if ((iphdr->daddr & netmask_ipv4) == (iphdr->saddr & netmask_ipv4)) {
7    // INTERN -> same netaddress
8    bytes_intern += bytes;
9  }
10 // own netaddress is the same like DST-IPs netaddress
11 else if (netaddr_ipv4 == (iphdr->daddr & netmask_ipv4)) {
12   // DOWN
13   bytes_down += bytes;
14 }
15 // own netaddress is the same like SRC-IPs netaddress
16 else if (netaddr_ipv4 == (iphdr->saddr & netmask_ipv4)) {
17   // UP
18   bytes_up += bytes;
19 }
20 else { // OTHER
21   bytes_other += bytes;
22 }
```

Listing 5.7: Klassifizierung im Fall von *IPv4*

Die Klassifizierung im Fall von *IPv6* als Vermittlungsschichtprotokoll unterscheidet sich vom Vorgehen im Fall von *IPv4* recht stark. Nur wenn das Netzwerkinterface, auf dem die Pakete erfasst werden, über eine *IPv6*-Adresse aus dem Gültigkeitsbereich „global unique“ verfügt, kann der Netzwerkverkehr in die bekannten Klassen „Up, Down, Intern, Other“ eingeteilt werden. Sollte das Interface nicht über eine global gültige Adresse verfügen, werden für *IPv6*-

Pakete ersatzweise die Klassen „ipv6_gl“ für globalen und „ipv6_ll“ für lokalen Netzwerkverkehr verwendet. Grund ist, dass bei Nichtvorhandensein einer global gültigen *IPv6*-Adresse, nicht festgestellt werden kann, ob das jeweilige Paket das eigene Netzsegment verlässt oder es von außen erreicht hat.

Listing 5.8 zeigt die Umsetzung im Source-Code. Im ersten Schritt werden die ersten 2 Byte der *IPv6*-Quell- und Ziel-Adresse ausgewertet (Zeile 1-4 bzw. 15-18). Enthalten diese jeweils den Wert $0xfe80$ handelt es sich um ein lokales Paket, enthalten sie einen Wert im Bereich $0x2000..0x3fff$ handelt es sich um ein Paket, das das lokale Netzsegment verlässt, bzw. von außen erreicht hat. Ist eine global gültige *IPv6*-Adresse verfügbar, wird zur Unterscheidung zwischen Up- und Download in gleicher Weise verfahren, wie bei *IPv4*-Paketen. Im Unterschied dazu mussten aber, wegen der Komplexität der *IPv6*-Adressen, für die UND-Verknüpfung und die Prüfung auf Gleichheit zweier Adresse eigens Funktionen entwickelt werden (Listings 5.9 und 5.10).

```
1  // from rate_mode.c
2  if ((ipv6hdr->src.s6_addr[0] == 0xfe)
3      && (ipv6hdr->src.s6_addr[1] == 0x80)
4      && (ipv6hdr->dst.s6_addr[0] == 0xfe)
5      && (ipv6hdr->dst.s6_addr[1] == 0x80)) {
6    // Scope: link-local, both addresses start with fe80:...
7    // only if both partners are in link-local
8
9    if (ipv6_global == FALSE) { // use gl/ll
10     bytes_ipv6_ll += bytes;
11   }
12   else { // use down/up...
13     bytes_intern += bytes;
14   }
15 }
16 else if ((ipv6hdr->src.s6_addr[0] >= 0x20)
17     && (ipv6hdr->src.s6_addr[1] <= 0x3f)
18     && (ipv6hdr->dst.s6_addr[0] >= 0x20)
19     && (ipv6hdr->dst.s6_addr[1] <= 0x3f)) {
20   // Scope: global, addresses start with 2000:... - 3fff:...
21   if (ipv6_global == FALSE) { // use gl/ll
22     bytes_ipv6_gl += bytes;
23   }
24   else { // use down/up...
25     if (IPv6_equal(&global_netaddr_ipv6,
26       IPv6_and(&ipv6hdr->dst, &global_netmask_ipv6))) {
27       bytes_down += bytes;
28     }
29     else if (IPv6_equal(&global_netaddr_ipv6,
30       IPv6_and(&ipv6hdr->src, &global_netmask_ipv6))) {
31       bytes_up += bytes;
32     }
```

```
33      else {
34        // OTHER
35        bytes_other += bytes;
36      }
37    }
38  }
39  else {
40    // OTHER
41    bytes_other += bytes;
42  }
```

Listing 5.8: Klassifizierung im Fall von *IPv6*

```
1 // from analyzer.h
2 bool_t IPv6_equal(struct in6_addr *a, struct in6_addr *b);
```

Listing 5.9: Funktion zur Prüfung zweier *IPv6*-Adressen auf Gleichheit

```
1 // from analyzer.h
2 struct in6_addr *IPv6_and(struct in6_addr *ip1, struct in6_addr *ip2);
```

Listing 5.10: Funktion zur UND-Verknüpfung zweier *IPv6*-Adressen

Nach Ablauf einer Sample-Periode wird die aktuelle Zeit ermittelt und gespeichert, um jeweils die echte Dauer der Sample-Periode berechnen zu können. Aus dieser Differenzzeit und der erfassten Datenmenge je Klasse, kann die durchschnittliche Datenrate für die vergangene Sample-Periode berechnet werden. Diese Datenraten werden in Strukturen geschrieben, die für die weiterverarbeitenden Threads (in Datei schreiben, plotten) lesbar sind.

5.4. Verarbeitung der erfassten Daten

Wie in 5.2 beschrieben, findet die Weiterverarbeitung der gewonnenen Daten in eigenen Threads statt. Der Gedanke dabei ist, dass die Paketerfassung weiterlaufen kann während erfasste Daten noch verarbeitet werden. Damit soll erreicht werden, dass Paketverlust aufgrund eines vollen Empfangspuffers erst bei möglichst hohen Paketraten auftritt (→ 5.11), bei gleichzeitig kleiner *Sampleperiode*.

Die Weiterverarbeitung findet auf unterschiedliche Weise statt. Für die unmittelbare Visualisierung können die aktuellen Datenraten, sozusagen „Live“, geplottet werden. Verwendet wird dazu das Programm GNUPlot, für das mit gnuplot_i ein C-Interface existiert.

Wie beschrieben, werden am Ende jeder *Sampleperiode* die Datenraten in eine Struktur (→ Anhang A, Listing A.9) geschrieben, die für den Plot-Thread lesbar ist. Vorgehalten werden die letzten n Messwerte. Der Zeitraum, der geplottet werden kann, ergibt sich zu:

$$T_{Plot} = T_{Sample} \cdot n \tag{5.1}$$

Als Default-Einstellung wird $T_{Sample} = 10ms$ und $n = 3000$ verwendet. Damit zeigt der mitlaufende Plot jeweils die letzten $30s$.

Geplottet wird nicht zwangsläufig bei jeder Aktualisierung der Daten. Um den Rechenaufwand gering zu halten, kann die Aktualisierungsfrequenz des Plots im Bereich

$$\frac{1}{T_{Sample}} \leq f_{Plot} \leq 1\,\frac{1}{s} \tag{5.2}$$

festgelegt werden werden.

Listing 5.11 zeigt die wesentlichen Teile der Plot-Funktion. Die *read*-Funktion wird verlassen, sobald Daten in die *Named Pipe* geschrieben wurden. Damit wird vom Haupt-Thread signalisiert, dass der Plot aktualisiert werden soll. Da GNUPlot Daten aus Dateien plottet, werden die Daten passend formatiert in eine Datei geschrieben. Beim Plotten wird anschließend unterschieden, ob das verwendete Netzwerkinterface über eine global gültige *IPv6*-Adresse verfügt oder nicht. Danach wird entschieden, ob die Datenraten der Ersatzklassen (→ 5.3) geplottet werden oder nicht. Die Daten werden dann zeilenweise aus der Datei gelesen und im Plot dargestellt. Da die Datei nur die letzten n Datensätze enthält, ist sichergestellt, dass der Plot passend horizontal scrollt.

```
1  // from plot_thread.c
2  while(run == TRUE) {
3    read(rd_pipe, buf, bytes_read) < 0) { // blocking non-busy read() from Pipe
4
5    for (i = 0; i < n; i++) { // write to TMP-File
6          fprintf(plot_tmp_file, "%.1lf %.3lf %.3lf %.3lf %.3lf %.3lf %.3lf\n",
7          t[i], dr_down[i], dr_up[i], dr_intern[i], dr_ipv6_gl[i],
8          dr_ipv6_ll[i], dr_other[i]);
9    }
10   fclose(plot_tmp_file); // close file
11
12   if (g_args->ipv6 == FALSE) { // Host doesn't have valid gloabl IPv6-Address
13         gnuplot_cmd(g_args->gnuplot_handle,
14         " plot \"%s\" using 1:2 with lines title
15         \"IPv4 - Down\", \"%s\" using 1:3 with lines title
16         \"IPv4 - Up\", \"%s\" using 1:4 with lines title
17         \"IPv4 - Intern\", \"%s\" using 1:5 with lines title
18         \"IPv6 - Global\", \"%s\" using 1:6 with lines title
19         \"IPv6 - Local\", \"%s\" using 1:7 with lines title
```

```
20          \"Other\"",
21          g_args->tmp_filename, g_args->tmp_filename, g_args->tmp_filename,
22          g_args->tmp_filename, g_args->tmp_filename, g_args->tmp_filename);
23        }
24        else { // Host has valid gloabl IPv6-Address
25          gnuplot_cmd(g_args->gnuplot_handle,
26          " plot \"%s\" using 1:2 with lines title
27          \"Down\", \"%s\" using 1:3 with lines title
28          \"Up\", \"%s\" using 1:4 with lines title
29          \"Intern\", \"%s\" using 1:7 with lines title
30          \"Other\"", g_args->tmp_filename, g_args->tmp_filename,
31          g_args->tmp_filename, g_args->tmp_filename);
32        }
33  }
```

Listing 5.11: Plot-Funktion

Beispielhaft zeigen die Abbildungen 5.4 und 5.3 je einen Plot ohne bzw. mit verfügbarer global gültiger *IPv6*-Adresse und die damit verbundene unterschiedliche Klassifizierung des Netzwerkverkehrs.

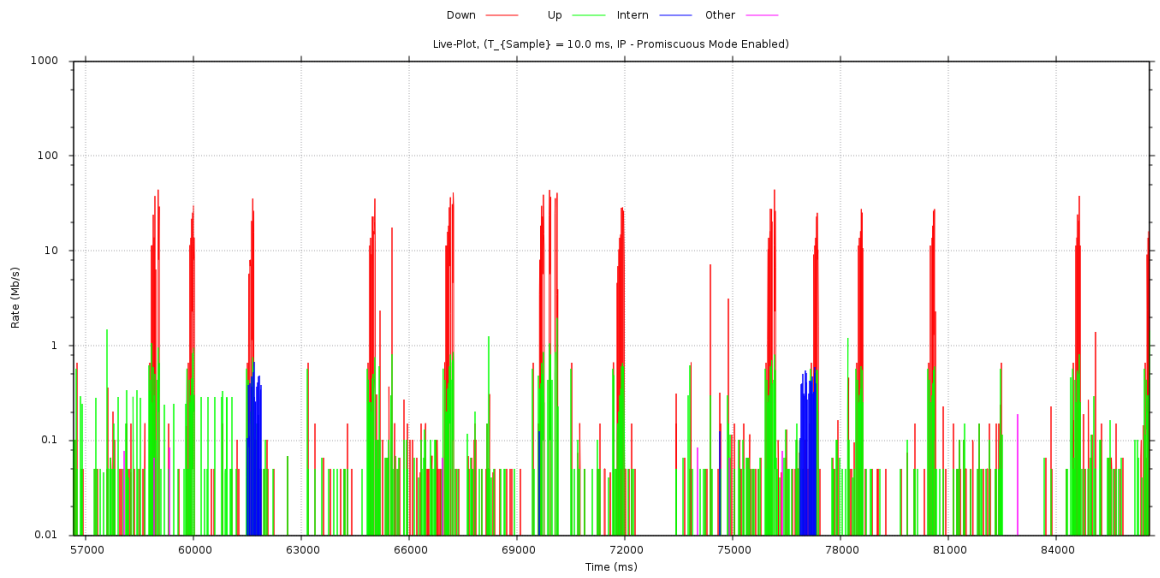

Abbildung 5.3.: Plot bei Vorhandensein einer global gültigen *IPv6*-Adresse (Downloaddatenrate in rot, Uploaddatenrate in grün, Datenrate Intern in blau)

Abbildung 5.4.: Plot bei Fehlen einer global gültigen *IPv6*-Adresse (Downloaddatenrate in rot, Uploaddatenrate in grün, Datenrate Intern in blau, Datenrate IPv6-Intern in türkis, Datenrate IPv6-global in pink)

Natürlich besteht die Möglichkeit die Plot-Funktion zu deaktivieren, was bspw. für leistungsschwache Systeme sinnvoll sein kann. Von der Plot-Funktion unabhängig können die anfallenden Daten binär in eine Datei geschrieben werden. Das Binärformat wurde gewählt, um die zu schreibende Datenmenge klein zu halten. Als Pendant zum Plot-Thread gibt es einen eigenen File-Thread, der im Wesentlichen gleichermaßen funktioniert. Per *Named Pipe* wird am Ende jeder *Sampleperiode* signalisiert, dass neue Daten in die Datei geschrieben werden sollen. Geschrieben wird hier, anders als bei der Plot-Funktion, natürlich nur der jeweils letzte Datensatz. Verwendet wird dazu eine Struktur, die einen Datensatz abbildet (siehe Anhang A, Listing A.11). Die Struktur hat eine Gesamtgröße von 72 Byte, gegenüber 360 Byte, wenn die Daten als Text in die Datei geschrieben würden (ausgehend von 5 Zeichen pro Wert). Bei der Standard-*Sampleperiode* von $T_S = 10ms$ liegt die erforderliche Schreibrate bei lediglich $7,2\frac{kB}{s}$.

Um bei langer Programmlaufzeit nicht unhandlich große Dateien zu produzieren, wird eine vom Anwender vorgegebene Anzahl Dateien vorgehalten, die ähnlich einem Ringspeicher in fester Reihenfolge nacheinander beschrieben werden. Die Datei wird jeweils bei Erreichen einer bestimmten Dateigröße gewechselt. Der Nutzer gibt vor, wie viel Speicherplatz insgesamt zur Verfügung stehen soll, wodurch sich die maximale Dateigröße ergibt. Außerdem kann daraus die Zeitspanne abgeleitet werden für die Daten verfügbar sind.

Aus den so gespeicherten Daten kann (sozusagen offline) ein Plot erzeugt werden. Start- und Endzeit können dabei durch den User vorgegeben werden. Per Default wird die gesam-

te zur Verfügung stehende Zeitspanne geplottet. Die Vorgabe der Zeitspanne kann ähnlich einer Zoom-Funktion genutzt werden. Die Plots aus den Abbildungen 5.5 bis 5.7 stammen aus der gleichen Verkehrsanalyse, zeigen aber zeitlich unterschiedlich große Ausschnitte.

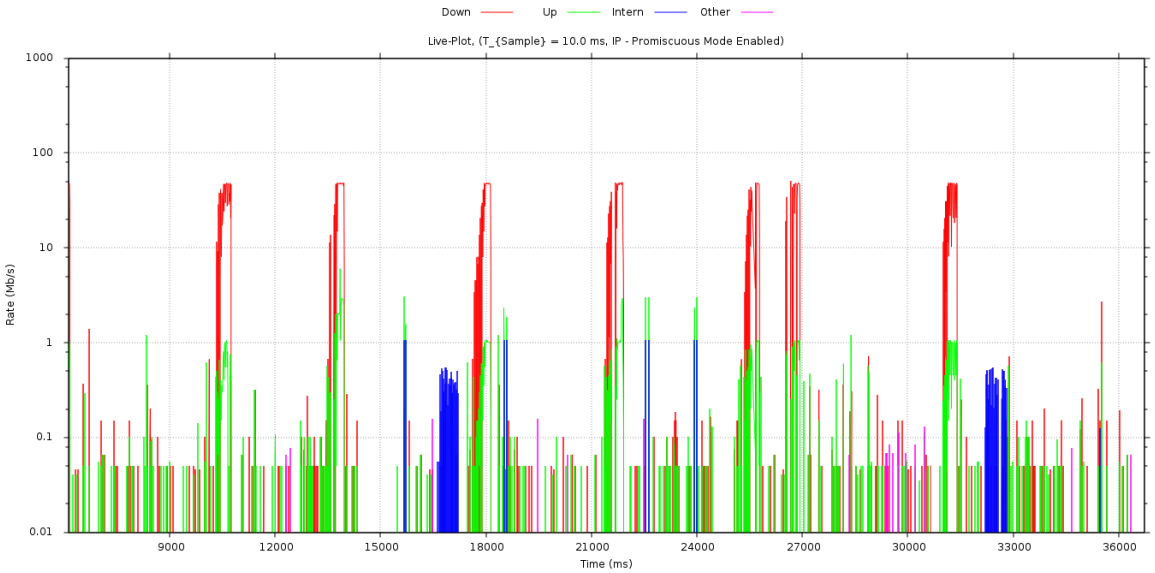

Abbildung 5.5.: Live Plot, während der laufenden Analyse (Downloaddatenrate in rot, Uploaddatenrate in grün, Datenrate Intern in blau)

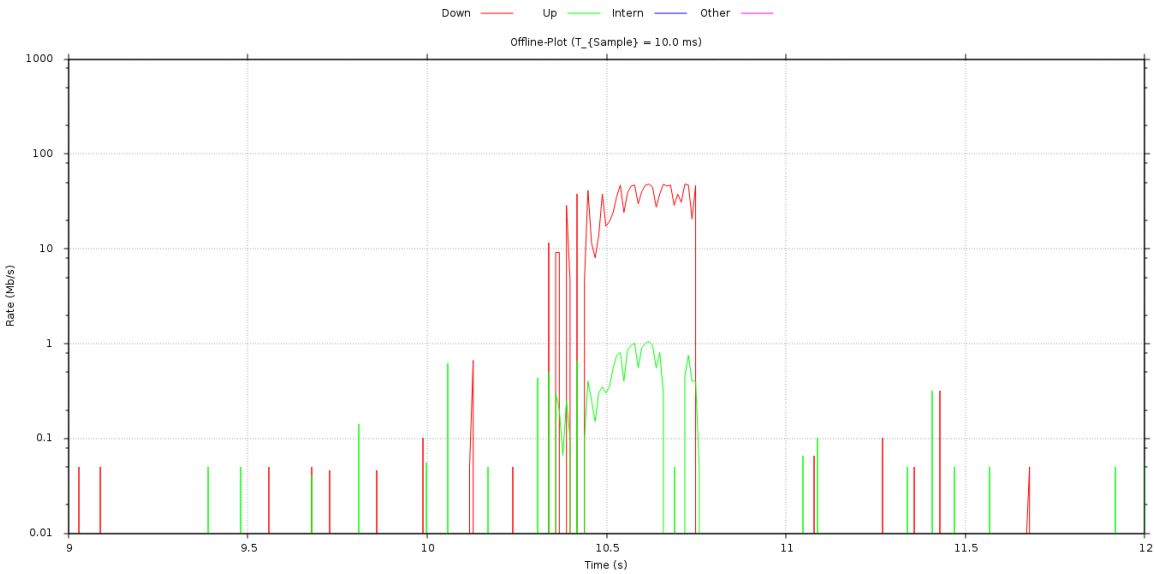

Abbildung 5.6.: Offlineplot, Ausschnitt Sekunde 9 bis 12 (Downloaddatenrate in rot, Uploaddatenrate in grün)

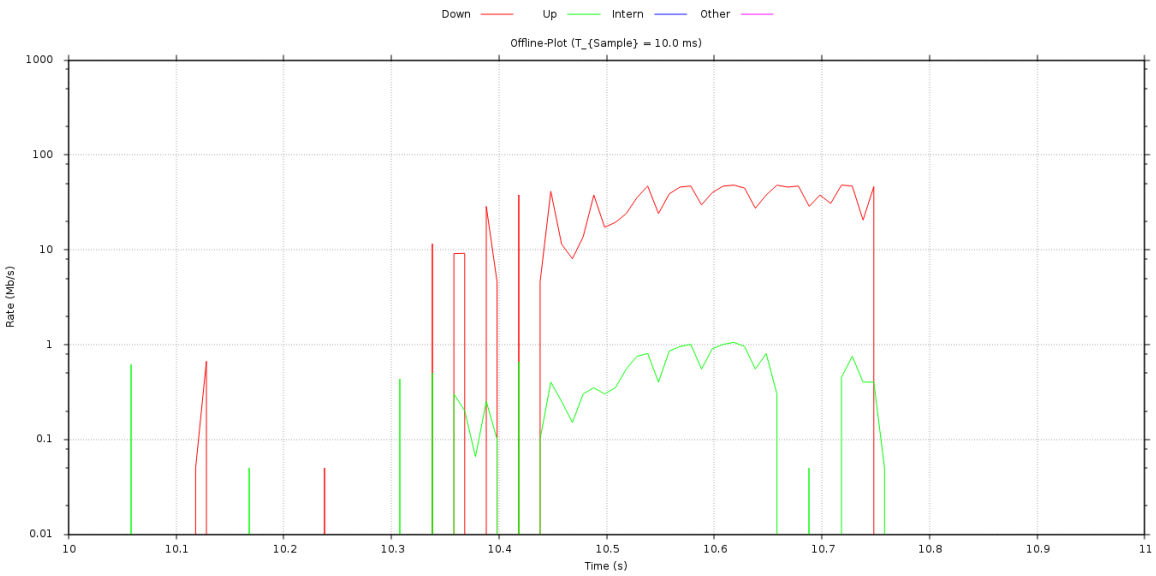

Abbildung 5.7.: Offlineplot, Ausschnitt Sekunde 10 bis 11 (Downloaddatenrate in rot, Uploaddatenrate in grün)

Schlussendlich können die gespeicherten Daten auch in ein CSV-Format konvertiert werden, bspw. für eine Auswertung mit Hilfe externer Tools. Listing 5.12 zeigt einen Ausschnitt aus dem CSV-Export zum Plot aus Abbildung 5.7. Die Datenfelder werden jeweils durch Leerzeichen getrennt (daher eigentlich richtigerweise *Space-separated*, statt *Comma-separated*).

```
#time datarate_down datarate_down datarate_intern datarate_other dropped_rel
10.32822 0.000 0.000 0.000 0.000 0.000000
10.33822 11.540 0.504 0.000 0.000 0.000000
10.34822 0.000 0.000 0.000 0.000 0.000000
10.35822 9.141 0.302 0.000 0.000 0.000000
10.36822 9.192 0.201 0.000 0.000 0.000000
10.37822 0.000 0.066 0.000 0.000 0.000000
10.38822 28.673 0.252 0.000 0.000 0.000000
10.39823 4.596 0.101 0.000 0.000 0.000000
10.40823 0.000 0.000 0.000 0.000 0.000000
10.41823 37.880 0.655 0.000 0.000 0.000000
10.42823 0.000 0.000 0.000 0.000 0.000000
10.43823 4.596 0.101 0.000 0.000 0.000000
10.44823 41.413 0.403 0.000 0.000 0.000000
10.45823 11.490 0.252 0.000 0.000 0.000000
10.46823 8.043 0.151 0.000 0.000 0.000000
10.47823 13.737 0.302 0.000 0.000 0.000000
10.48823 37.866 0.352 0.000 0.000 0.000000
10.49823 17.284 0.302 0.000 0.000 0.000000
10.50823 19.482 0.352 0.000 0.000 0.000000
10.51823 24.078 0.554 0.000 0.000 0.000000
10.52823 35.567 0.755 0.000 0.000 0.000000
10.53823 47.008 0.806 0.000 0.000 0.000000
10.54823 24.128 0.403 0.000 0.000 0.000000
10.55823 38.964 0.856 0.000 0.000 0.000000
10.56823 45.857 0.957 0.000 0.000 0.000000
10.57823 47.058 1.007 0.000 0.000 0.000000
10.58823 29.772 0.554 0.000 0.000 0.000000
10.59823 40.163 0.906 0.000 0.000 0.000000
10.60823 47.006 1.007 0.000 0.000 0.000000
10.61823 48.157 1.057 0.000 0.000 0.000000
10.62823 44.758 0.957 0.000 0.000 0.000000
10.63823 27.475 0.554 0.000 0.000 0.000000
10.64823 37.866 0.806 0.000 0.000 0.000000
10.65823 48.055 0.302 0.000 0.000 0.000000
10.66823 45.858 0.000 0.000 0.000 0.000000
10.67823 47.057 0.000 0.000 0.000 0.000000
10.68823 28.724 0.050 0.000 0.000 0.000000
10.69823 37.864 0.000 0.000 0.000 0.000000
10.70823 30.870 0.000 0.000 0.000 0.000000
10.71823 48.156 0.453 0.000 0.000 0.000000
10.72823 47.056 0.755 0.000 0.000 0.000000
10.73824 20.631 0.403 0.000 0.000 0.000000
10.74824 46.495 0.403 0.000 0.000 0.000000
10.75824 0.000 0.050 0.000 0.000 0.000000
```

Listing 5.12: CSV-Export

5.5. Analyse der Paketlaufzeit

Wie in 4.5 beschrieben, können durch den Paketgenerator *IPv4*-Pakete versendet werden, die einen Zeitstempel enthalten. Erreichen diese Pakete den Analysator, ist er in der Lage sie auszuwerten. Die Analyse der Paketlaufzeit kann parallel zur normalen Verkehrsanalyse stattfinden. Die entsprechende Option muss per Befehl (→ Anhang C) aktiviert werden. Ist die Laufzeitanalyse aktiviert, werden *IPv4*-Pakete deren Nutzdaten eine bestimme Zeichenfolge („MAGIC_PAYLOAD“) enthalten, gesondert ausgewertet und in der normalen Verkehrsanalyse nicht mehr berücksichtigt. Da für jedes Paket ohnehin die Ankunftszeit festgestellt wird, um die Verarbeitungszeit zu kompensieren (siehe Listing 5.2), muss lediglich der Zeitstempel aus den Nutzdaten extrahiert werden, um die Laufzeit berechnen zu können. Listing 5.13 zeigt den entsprechenden Teil des Codes.

```
1 // from rate_mode.c
2 if (runtime_analysis == TRUE && strstr(payload, MAGIC_PAYLOAD)) {
3   strtok(payload, ".");
4   ms_since_epoch_ts = strtok(NULL, ".");
5   ms_since_epoch_timestamp = strtol(ms_since_epoch_ts, NULL, 10);
6   runtime_ms = ms_since_epoch_arrival - ms_since_epoch_timestamp;
7   runtime_ms[(int)(runtime_ms/bin_width)]++;
8   n_packets++; // for rel. frequency
9 }
```

Listing 5.13: CSV-Export

Die Verarbeitung erfordert nur wenige Schritte. Sie hat ein Array zum Ergebnis, das die Klassen einer Häufigkeitsverteilung repräsentiert. Um diese Verteilung zu plotten steht eine eigene Plot-Funktion zur Verfügung, die parallel zur weiterlaufenden Verkehrs- und Paketlaufzeitanalyse ausgeführt werden kann. Abbildung 5.8 zeigt ein derart entstandenes Histogramm. Generator und Analysator sind dabei direkt (und als einzige Stationen) über einen Switch verbunden. Es wurde keine weitere Netzwerklast erzeugt.

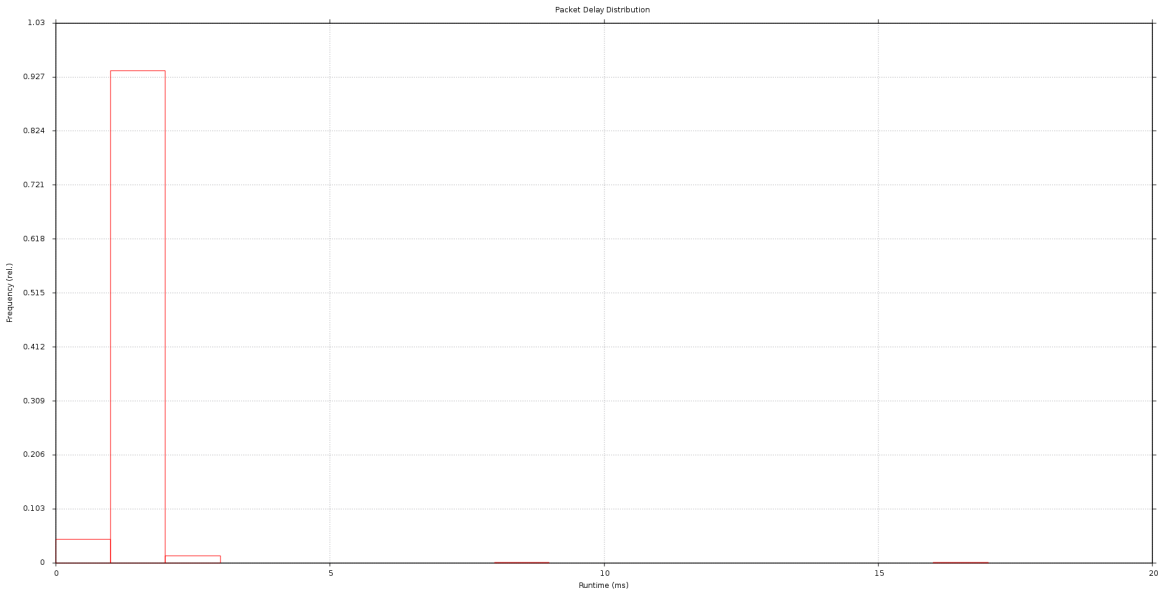

Abbildung 5.8.: Histogramm, Paketlaufzeit mit synchronisierten Systemuhren ($t_{offset} \approx 0,5ms$)

Abschnitt 3.4 beschreibt die Grundlagen der Zeitsynchronisation per *NTP*. Da die Synchronisation nicht binnen kurzer Zeit erfolgen kann und auch nicht beliebig genau ist, wird es dazu kommen, dass der Offset der Systemuhren von Generator und Analysator im Histogramm erkennbar ist. Ein Beispiel zeigt Abbildung 5.9. Die Systemuhr des Generators eilt hierbei derer des Analysators um ca. 210ms voraus.

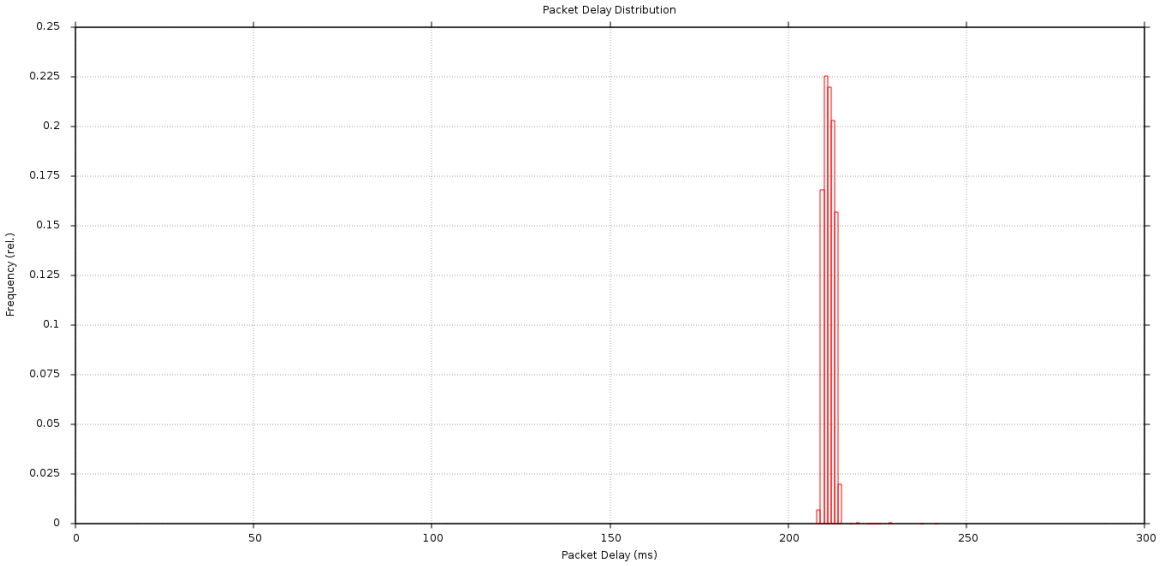

Abbildung 5.9.: Histogramm, Offsetbehaftete Paketlaufzeit mit $t_{offset} \approx 210ms$

Aber auch mit vorhandenem Offset kann die ermittelte Paketlaufzeit sinnvoll ausgewertet werden, da näherungsweise davon ausgegangen werden kann, dass der Offset über einen begrenzten Zeitraum konstant bleibt. Eine *NTP*-Synchronisation sollte in diesem Fall nicht aktiv sein, da diese dafür sorgen würde, dass der Offset gerade nicht konstant bleibt. Abbildung 5.10 zeigt dieses Verhalten. Es handelt sich um die gleiche Laufzeitanalyse wie in Abbildung 5.9, allerdings liegt ca. 1 Stunde *NTP*-Synchronisation zwischen den Plots. Die Paketlaufzeit bleibt dabei konstant, der Offset zwischen den Systemuhren wird aber geringer.

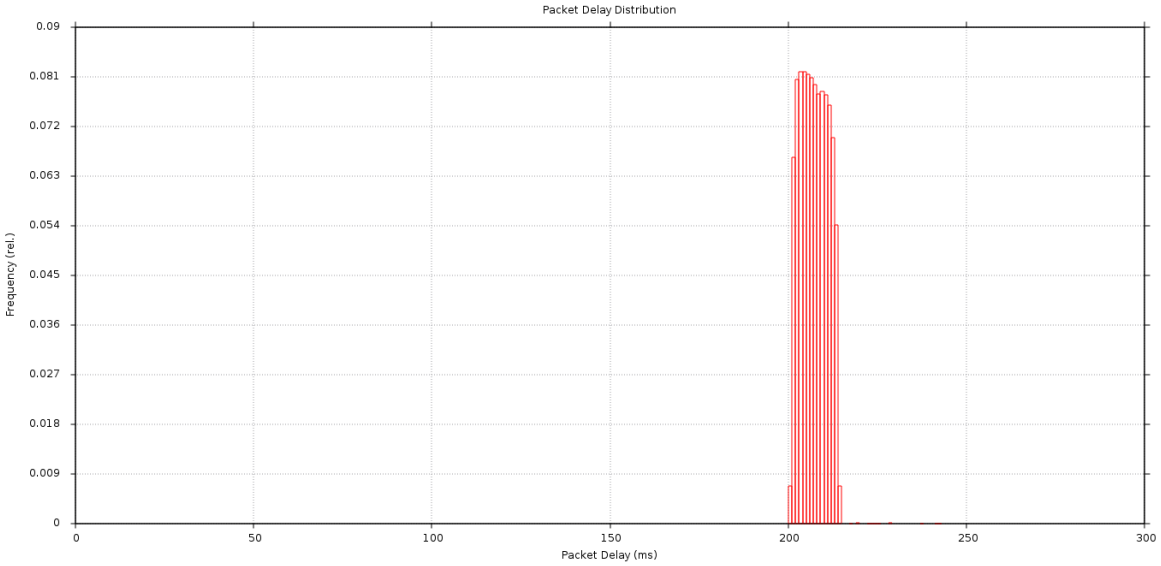

Abbildung 5.10.: Histogramm, Offsetbehaftete Paketlaufzeit mit laufender NTP-Synchronisation

5.6. Analyse der Zwischenankunftszeit

Um die Zwischenankunftszeit zu analysieren, ist es nötig, die Ankunftszeit eines jeden Pakets zu erfassen. Ist die Analyse der Zwischenankunftszeit aktiviert, wird die Ankunftszeit erfasst und die Ankunftszeit des vorherigen Pakets subtrahiert. Die Differenz wird mit Hilfe der Klassenbreite in einen Array-Index umgerechnet und der entsprechende Wert inkrementiert. Festgehalten wird auch die Gesamtzahl der erfassten Pakete, um eine relative Häufigkeit angeben zu können. Listing 5.14 zeigt die Umsetzung.

```
1  // from rate_mode.c
2  clock_gettime(CLOCK_REALTIME, &t_arrival); // packet arrival time
3  // arrival time in us since epoch
4  us_since_epoch_arrival = (t_arrival.tv_sec / MICRO) + (long)(t_arrival.tv_nsec
       / 1000);
5  // calculate iatime in us
6  iatime_us = us_since_epoch_arrival - us_since_epoch_last;
7  if ((iatime_us > 0) && (iatime_us < (n_bins * bin_width))) {
8    // increase counter of bin
9    iatime_us[(int)((us_since_epoch_arrival - us_since_epoch_last)/bin_width)]++;
10   n++; // total number of packets
11 }
12 // update arrival of last packet
```

```
13 us_since_epoch_last = us_since_epoch_arrival;
```

Listing 5.14: Erfassung der Zwischenankunftszeit

Die Häufigkeitsverteilung der Zwischenankunftszeit lässt sich als Histogramm plotten, im Folgenden durch 2 Beispiele veranschaulicht.

a) Paketgröße $PS = 1000\ Byte$
   Datenrate $DR_a = 10000\ \frac{kb}{s}$

b) Paketgröße $PS = 1000\ Byte$
   Datenrate $DR_b = 40000\ \frac{kb}{s}$

Um die gewünschte Datenrate mit der vorgegebenen Paketgröße zu erreichen, muss in bestimmten Abständen ein Paket versendet werden. In 4.4 ist beschrieben, wie die Datenrate erzeugt wird. Bei $t_{delta} = 10\mu s$ ist es demnach nicht nötig in jedem Burst ein Paket zu versenden, um die Datenrate zu erreichen. Die erwartete Zwischenankunftszeit berechnet sich wie folgt:

Für a)

$$IAT_a = \frac{PS}{DR_a} = \frac{1000B}{10000\frac{kb}{s}} = \frac{1000B}{1280000\frac{B}{s}} \approx 781\mu s \tag{5.3}$$

Für b)

$$IAT_b = \frac{PS}{DR_b} = \frac{1000B}{40000\frac{kb}{s}} = \frac{1000B}{5120000\frac{B}{s}} \approx 195\mu s \tag{5.4}$$

Die Abbildungen 5.11 bis 5.14 zeigen jeweils die Ratenverläufe und die Häufigkeitsverteilungen der Zwischenankunftszeiten zu a) und zu b). Anhand der Plots lässt sich die Berechnung verifizieren.

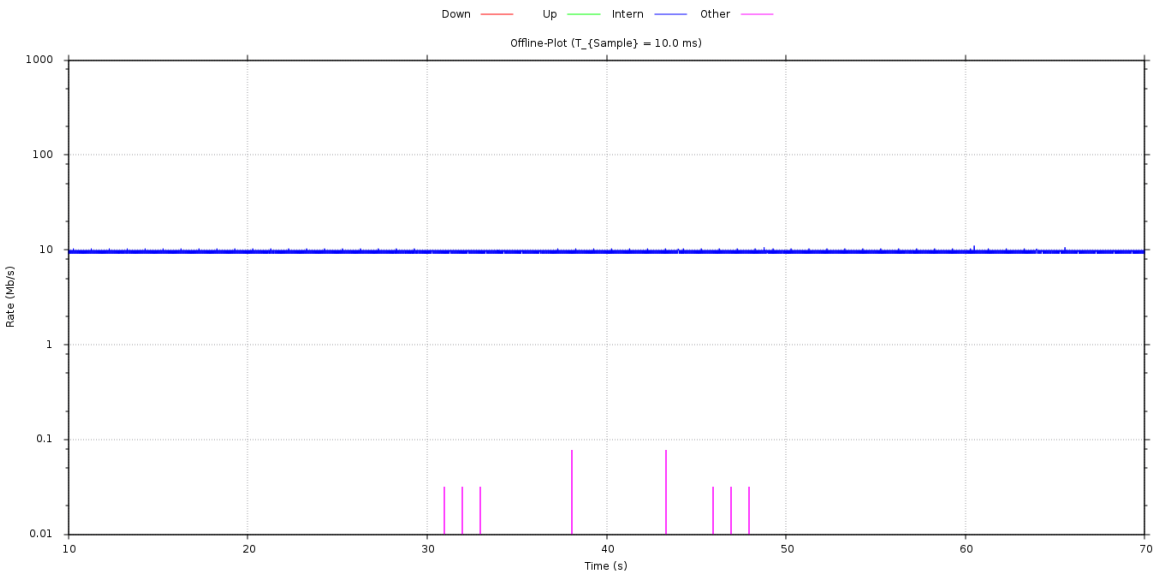

Abbildung 5.11.: Ratenverlauf zu a) (Generischer Verkehr in blau)

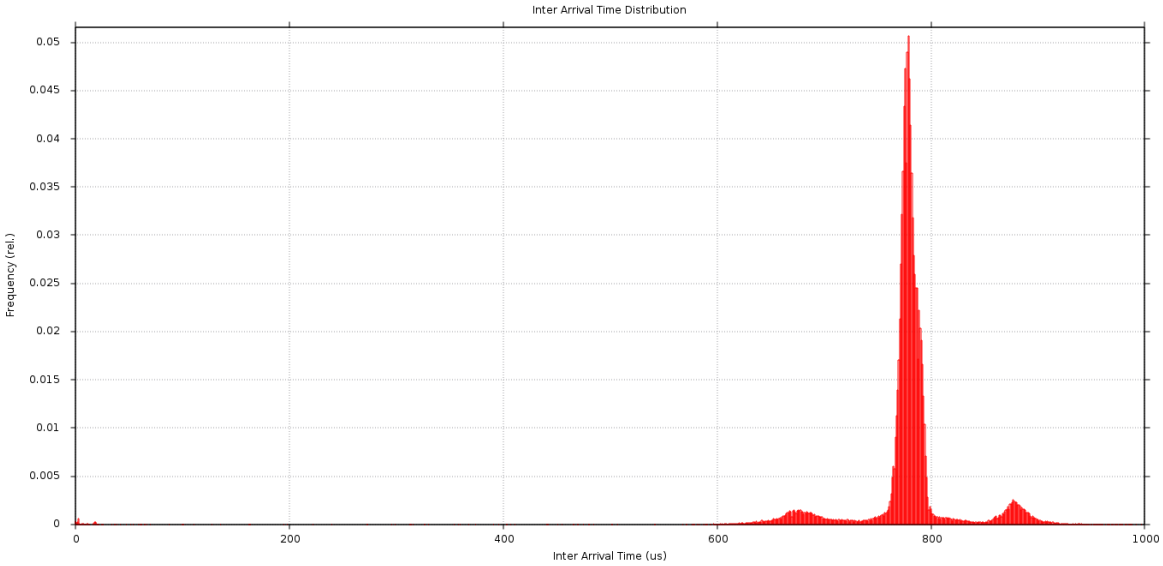

Abbildung 5.12.: Häufigkeitsverteilung Zwischenankunftszeit zu a)

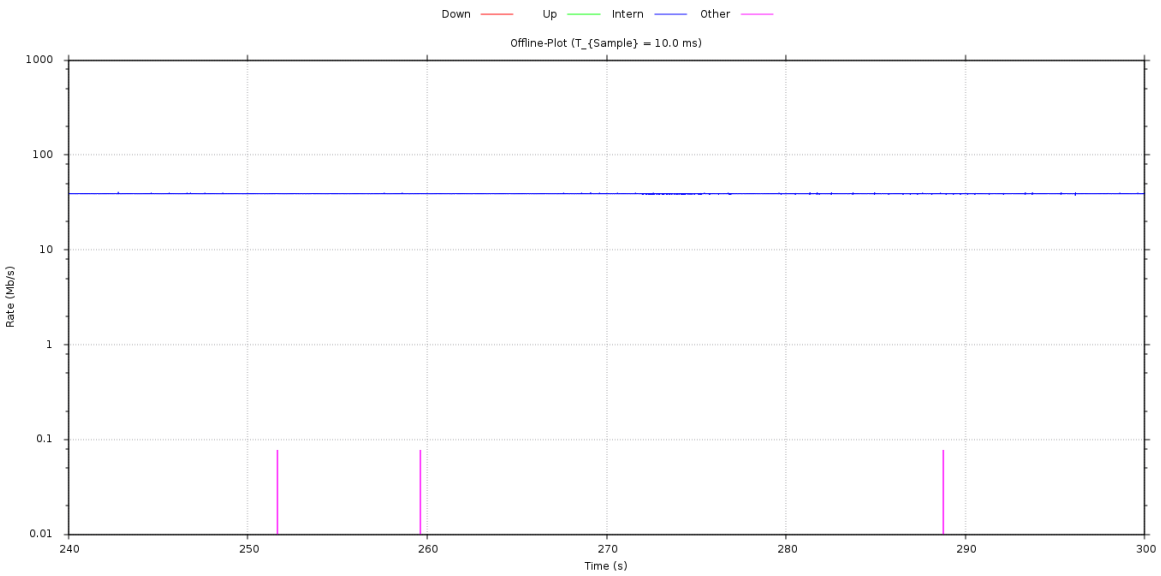

Abbildung 5.13.: Ratenverlauf zu b) (Generischer Verkehr in blau)

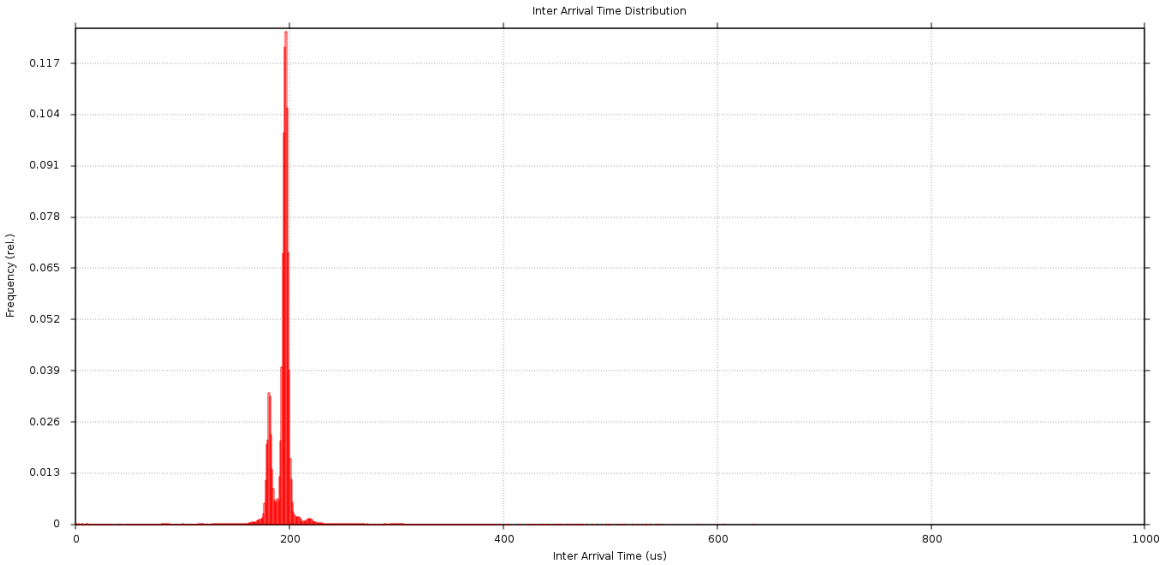

Abbildung 5.14.: Häufigkeitsverteilung Zwischenankunftszeit zu b)

In jeder Burst-Periode wird neu berechnet, wie viele Pakete versendet werden müssen, um die geforderte Datenrate zu erreichen. Für die Berechnung wird immer der gesamte Zeitraum seit Start des Generators berücksichtigt, so dass Schwankungen ausgeglichen werden können. Daher kommt es dazu, dass der zeitliche Abstand zwischen zwei Paketen größer oder auch kleiner als gewöhnlich ist.

Für die in 4.4 beschriebene Funktion des Paketversands mit exponentialverteilter Häufigkeit der Zwischenankunftszeit zeigt Abbildung 5.15 den entsprechenden Plot.

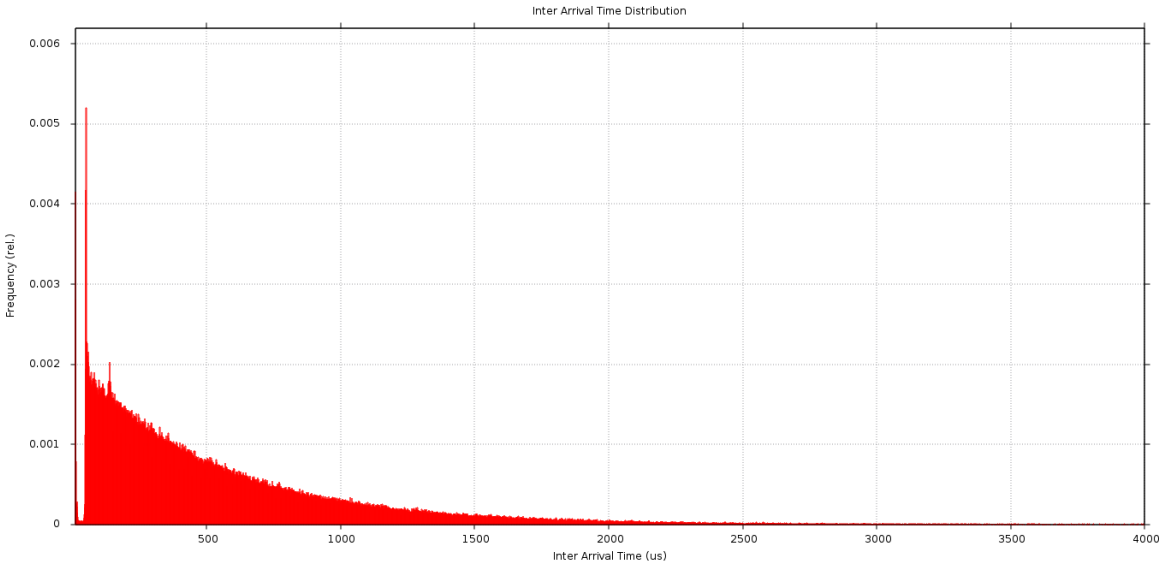

Abbildung 5.15.: Exponentialverteilung Zwischenankunftszeit

Häufigkeitsverteilungen für realen Netzwerkverkehr zeigen die Abbildungen 5.17 (Videostream) und 5.19 (Radiostream). Jeweils zugehörig sind die Plots der Datenraten in Abbildung 5.16 und 5.18. Während Abbildung 5.17 eine, möglicherweise für die Anwendung charakteristische, Verteilung zeigt, ist das für den Radio-Stream nicht der Fall (Abb. 5.19).

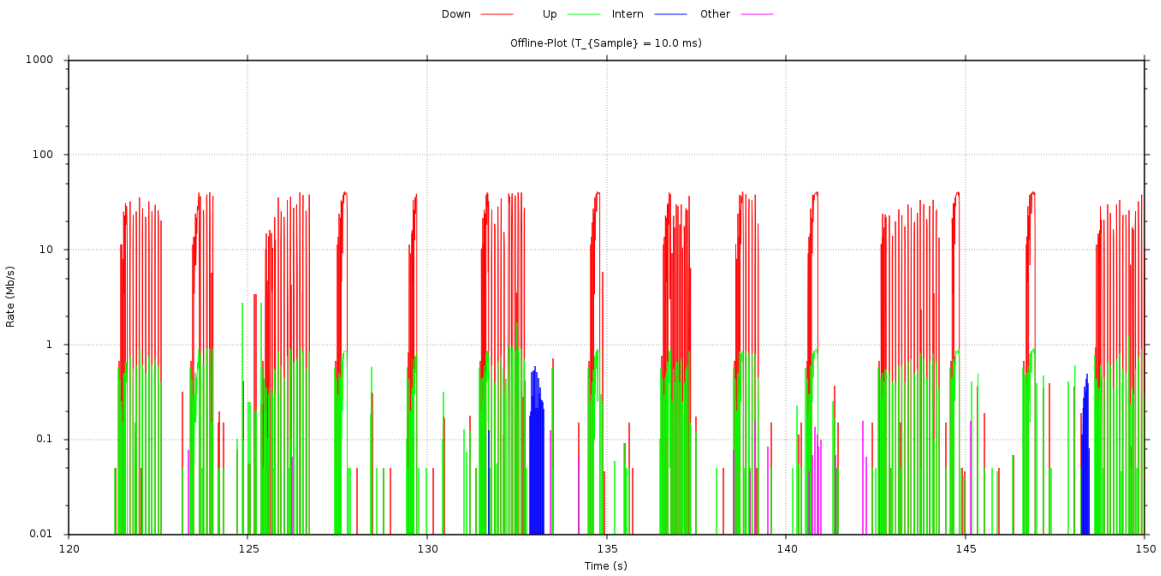

Abbildung 5.16.: Datenraten Videostream mit $DR_{Videostream} \approx 4\ \frac{Mb}{s}$ (Downloaddatenrate in rot, Uploaddatenrate in grün, Datenrate Intern in blau, Other in pink)

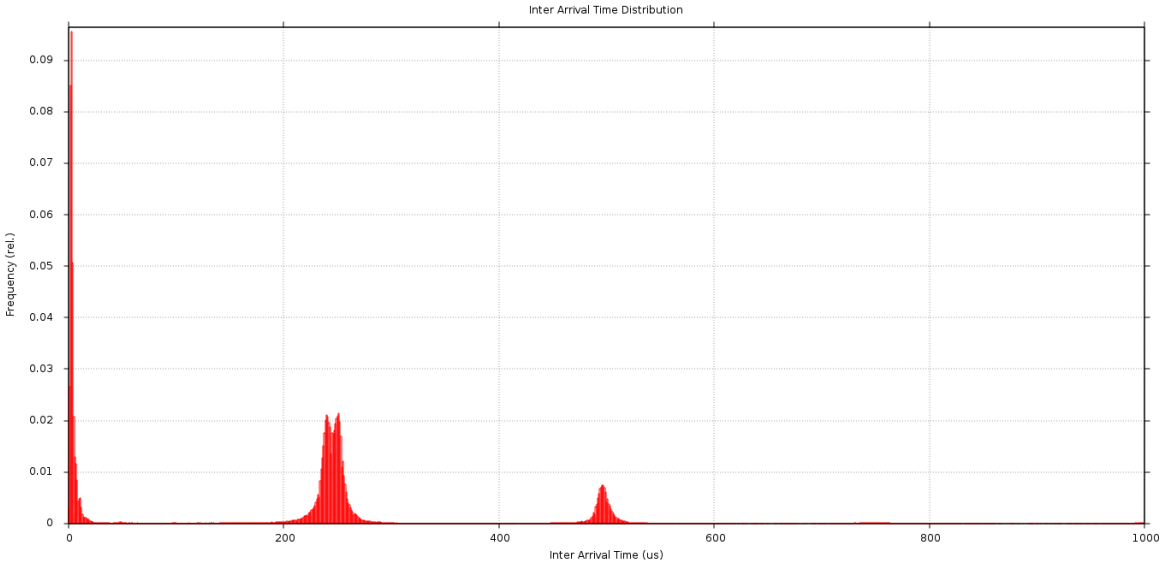

Abbildung 5.17.: Häufigkeitsverteilung Zwischenankunftszeit für Video-Stream mit (möglicherweise charakteristischen) Peaks bei $t_{IAT} \approx 250\mu s$ und $t_{IAT} \approx 500\mu s$

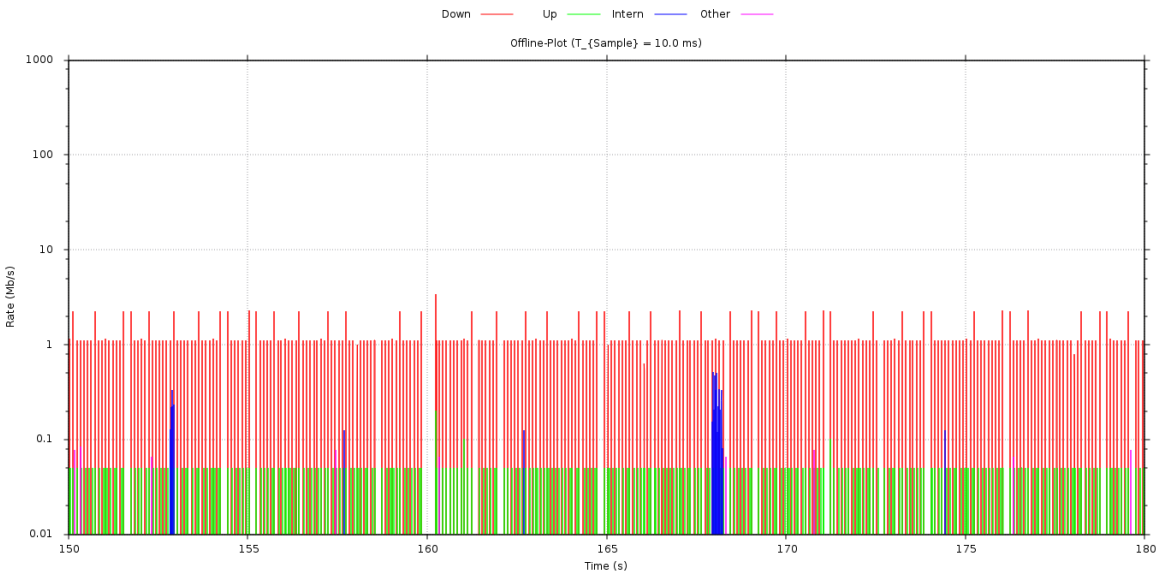

Abbildung 5.18.: Datenraten Radiostream mit $DR_{Radiostream} = 128\ \frac{kb}{s}$ (Downloaddatenrate in rot, Uploaddatenrate in grün, Datenrate Intern in blau, Other in pink)

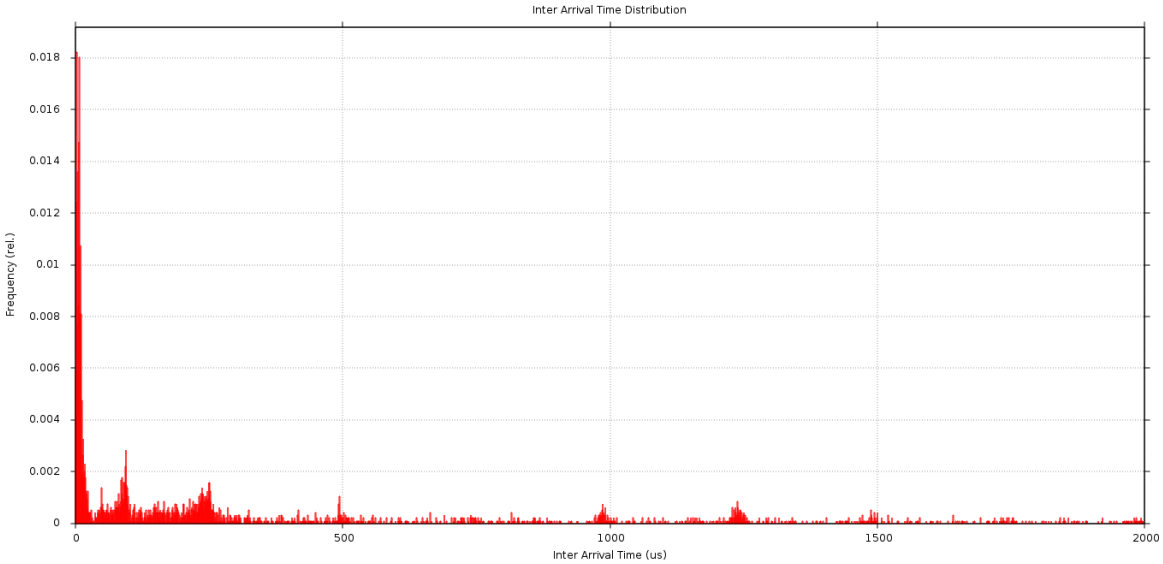

Abbildung 5.19.: Häufigkeitsverteilung Zwischenankunftszeit für Audio-Stream (kein besonderes Muster)

5.7. Paketerfassung und Speicherung mit Hilfe der PCap-Programmierschnittstelle

Parallel zur beschriebenen Paketanalyse können Paketsequenzen optional im *PCap*-Format mitgeschnitten und gespeichert werden. *PCap* bietet neben dem Dateiformat eine Programmierschnittstelle, die durch eine entsprechende Bibliothek nutzbar wird. Programme, wie bspw. Wireshark, sind in der Lage Paketsequenzen im *PCap*-Dateiformat zu verarbeiten. Damit wird eine weitere Form der nachträglichen Auswertung möglich.

Abbildung 5.20 zeigt den prinzipiellen Aufbau des Dateiformats. Im globalen Header werden Informationen wie Versionsnummer, Zeitzone oder auch Byteorder erfasst. Beim Paketheader handelt es sich um einen zusätzlichen Header, der jedem Paket vorangestellt wird. Dieser enthält einen Zeitstempel mit dem Zeitpunkt der Paketerfassung, sowie die gespeicherte und die originale Größe des Pakets. Erforderlich wird diese Angabe, da durch den User die Anzahl Byte vorgegeben werden kann, die von jedem Paket gespeichert werden sollen. Packet Data enthält das eigentliche Paket [21].

Abbildung 5.20.: Aufbau *PCap*-Datei-Format (Quelle: [21], GNU GPL)

Die Umsetzung im Quellcode zeigt Listing 5.15. Beim Erstellen der *PCap*-Session werden die wesentlichen Einstellungen übergeben [21]:

- *ifname* enthält den Namen des Netzwerkinterfaces
- *snaplen* enthält die Anzahl Byte, die von jedem Paket erfasst werden
- *promisc* gibt an, ob der *Promiscuous Mode* aktiv sein soll oder nicht
- *timeout_ms* gibt an, wie lange mittels *pcap_dispatch()* Pakete erfasst werden sollen
- *errbuf* ist ein Pointer auf allokierten Speicher, in dem ein Fehlerstring hinterlegt wird

```
1  // from pcap_mode.c
2  pcap_t *pcap_handle; // new handle
3  while (tun == TRUE) {
4    pcap_handle = pcap_open_live(ifname, snaplen, promisc, timeout_ms, errbuf);
5    // open PCap-session on interface iface
6    pcap_setdirection(t_args->pcap_handle, PCAP_D_IN);
7    // capture incoming packets
8    pcap_file = fopen(filename, "wb");
9    // file to store the records in
10   pcap_dump = pcap_dump_fopen(t_args->pcap_handle, t_args->pcap_file);
```

```
11   // opens file for writing
12   pcap_dispatch(t_args->pcap_handle, -1, &pcap_dump, (u_char *)(t_args->dump));
13   // caputre packets (duration = timeout_ms)
14   pcap_dump_flush(t_args->dump);
15   // write record to file
16 }
```

Listing 5.15: Paketerfassung mit der PCap-Schnittstelle

Auch hier soll die Dateigröße vom Anwender begrenzbar sein (→ 5.4). Der Nutzer kann dabei vorgeben bei welcher Dateigröße und nach welcher Zeit die Datei gewechselt wird. Je nach anfallender Paketrate und Einstellung kann zuerst die zeitliche oder zuerst die Größenbegrenzung greifen. Wiederum wird eine feste Anzahl Dateien, ähnlich einem Ringspeicher, rotiert.

5.8. Erkennung verworfener Pakete

Um mögliche Ungenauigkeiten der Verkehrsanalyse einschätzen zu können, ist es oft möglich die Anzahl verworfener Pakete zu erkennen. Verwendet wird dafür das Programm Ethtool, das über eine Gerätedatei ein Interface zur Verfügung stellt. Der Zugriff kann mit der Funktion ioctl() erfolgen. Listing 5.16 zeigt Teile der Umsetzung.

```
1 // from rx_overflow.c
2   long dropped;
3   struct ifreq ifr; // struct describing a network interface
4   strcpy(ifr.ifr_name, ifname); // set the current interface
5   struct ethtool_stats *estats = (struct ethtool_stats *)buf;
6   estats->cmd = ETHTOOL_GSTATS; // ethtool-command (GetStats)
7   ifr.ifr_data = (void *)estats; // point to the result-buffer
8   ioctl(sock_fd, SIOCETHTOOL, &ifr);
9   dropped =  estats->data[overrun_field];
```

Listing 5.16: Erkennung verworfener Pakete

Das entsprechende Interface wird durch die Struktur *struct ifreq* repräsentiert. Die Identifizierung geschieht mit Hilfe des Interface-Namens, den die entsprechende Strukturvariable als Wert bekommt. Die Struktur *struct ethtool_stats* wird an den Ethtool-Aufruf übergeben und zur Speicherung der Ergebnisse sowie zur Übergabe des Befehls (*ETHTOOL_GSTATS*) genutzt. Als Dateideskriptor dient ein geöffneter *Socket*. Für den Aufruf von *ioctl()* weiterhin nötig ist das Makro *SIOCETHTOOL*, das spezifiziert, dass es sich um einen Ethtool-Aufruf handelt [6].

In 4.6 wird beschrieben, dass die Verbindung zur Steuerung von Analysator und Generator optional aufrecht erhalten werden kann. Über diese Verbindung würde, wenn aufrecht erhalten, die steuernde Gegenstelle über Paketverlust informiert. Das entsprechende *TCP*-Paket enthält die Zeichenkette *x.xxxxxx (rel. Packet-Loss)*, wobei *x.xxxxxx* den relativen Paketverlust seit Analysator-Start beinhaltet. Weiterhin wird Paketverlust im Plot signalisiert (in Abb. 5.21 hervorgehoben).

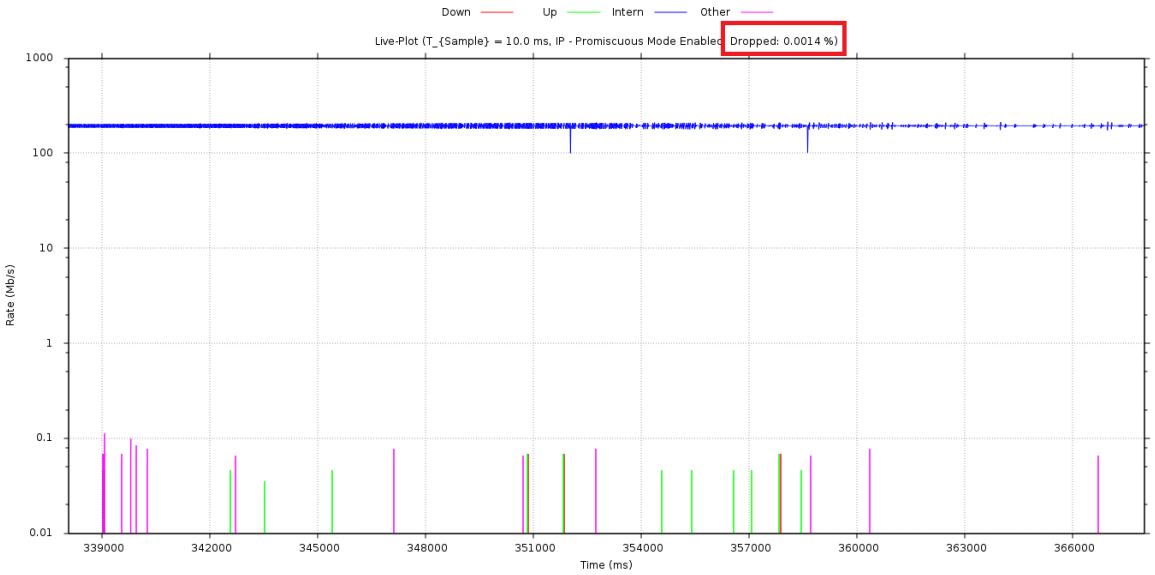

Abbildung 5.21.: Signalisierung von Paketverlust im Plot (rote Markierung)

Leider gibt es einige Einschränkungen bei der Nutzung dieser Methode. Die Statistik-Daten, die Ethtool zurückliefert, werden in einem Array gespeichert (*estats->data[]*). Bekannt sein muss nun, in welchem Array-Feld der gesuchte Wert gespeichert wird. Ein Aufruf von *ethtool -S interface_name* zeigt alle für das übergebene Interface verfügbaren Werte [4]. Welche Werte abrufbar und wie sie benannt sind, unterscheidet sich je nach Netzwerkinterface. Der Wert (z. B. *rx_no_buffer_count*) muss nun gesucht werden. Anschließend kann mit dem Befehl *SET_RX_OVERFLOW_FIELD x* das entsprechende Feld gesetzt werden. Ist das Feld nicht bekannt oder steht Ethtool nicht zur Verfügung, kann die Erkennung verworfener Pakete mittels *BUF_OVERFLOW_WARNING 0* deaktiviert werden.

5.9. Scheduler-Optimierung

In Abschnitt 3.3 wurden die Grundlagen zum *CFS* erläutert. Die Optimierungsmöglichkeiten, die dieser Scheduler bietet, werden genutzt um die Zeitgenauigkeit des Analysators zu ver-

bessern und eine möglichst kleine Sample-Periode zu ermöglichen. Die Anforderungen sind folgende:

- der Analysator-Thread soll möglichst sofort nachdem er Lauffähigkeit signalisiert hat, Rechenzeit erhalten, auch wenn dadurch andere *Tasks* unterbrochen werden müssen
- der Analysator soll nicht durch andere *Tasks* unterbrochen werden
- im Idealfall soll der Analysator-*Task* nicht mit anderen um CPU-Zeit konkurrieren müssen

Um diese Ziele zu erreichen, wird der *Task* nicht der Standard-Scheduling-Klasse „fair_sched_class“, sondern der Klasse „rt_sched_class“ zugeordnet. Innerhalb dieser Klasse wird er außerdem mit der höchstmöglichen Priorität versehen und die Scheduling-Strategie SCHED_FIFO angewendet. Damit ist gewährleistet, dass die Rechenzeit nicht periodisch neu verteilt wird (wie in der Scheduling-Strategie SCHED_RR), sondern der höchst priorisierte *Task* so lange die CPU beanspruchen kann, bis er keine Rechenzeit mehr benötigt. Die Umsetzung zeigt Listing 5.17. Der Funktion *pthread_setschedparam()* wird dabei durch die Funktion *pthread_self()* die Thread-ID des aufrufenden Threads übergeben.

```
1 // from rate_mode.c
2 // Scheduler Priority and Strategy for Rate-Thread
3 struct sched_param thread_param; // struct holds priority
4
5 thread_param.sched_priority = sched_get_priority_max(SCHED_FIFO);
6 pthread_setschedparam(pthread_self(), SCHED_FIFO, &thread_param);
```

Listing 5.17: Scheduling-Optimierung

Zur weiteren Optimierung und bei Vorhandensein mehrerer Prozessoren, kann ein Prozessor komplett vom Scheduling ausgenommen werden. Dazu kann eine Option des Linux Kernels genutzt werden (Listing 5.18):

```
1 isolcpus=0
```

Listing 5.18: Prozessoren vom Scheduling ausnehmen

Im Beispiel wird CPU0 vom Scheduling ausgenommen. Die Idee ist, dem Analysator-*Task* ausschließlich diese CPU zur Verfügung zu stellen. Theoretisch sollte der *Task* die CPU damit exklusiv nutzen können. Es gibt allerdings einige Betriebssystem-*Tasks*, die unabhängig vom Scheduler immer alle Prozessoren nutzen dürfen und darüber hinaus mit hoher Priorität in der Scheduling-Klasse „rt_sched_class“ ausgestattet sind. Nutzen diese die isolierte CPU, würde erneut eine Konkurrenzsituation entstehen. In Tests hat sich dieses Vorgehen nicht bewährt, da der Analysator-*Task* aufgrund seiner Bindung (Affinität) an die isolierte CPU keine Ausweichmöglichkeit hat, obwohl möglicherweise andere Prozessoren nicht ausgelastet sind.

Daher wird zwar eine CPU isoliert, dem Analysator-*Task* wird es aber ermöglicht alle Prozessoren (inklusive dem isolierten) zu nutzen. Listing 5.19 zeigt die Umsetzung. Die Variable *n_cpus* enthält hierbei die Anzahl der verfügbaren Prozessoren, die im Vorfeld ermittelt wurde. Die Funktion *CPU_SET()* fügt die CPU *i* einem Set von Prozessoren hinzu. Die Funktion *pthread_setaffinity_np()* bindet den aktuellen Thread an dieses Set von Prozessoren.

```
1 // from rate_mode.c
2 // CPU0 is isolated (kernel params)
3 cpu_set_t cpu_set;
4 CPU_ZERO(&cpu_set);
5 for (i = 0; i < n_cpus; i++) {
6   CPU_SET(i, &cpu_set);
7 }
8 pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpu_set);
```

Listing 5.19: Scheduling-Optimierung

5.10. Steuerung

In Abschnitt 4.6 wurde die Steuerung des Generators beschrieben. Die Steuerung des Analysators arbeitet auf die gleiche Weise. Verschieden ist lediglich der verwendete Port für den *TCP*-Socket (4000 statt 3000) sowie die zur Verfügung stehenden Befehle. Eine Übersicht der Analysator-Steuerbefehle mit Beschreibung befindet sich in Anhang C.

5.11. Performance-Bewertung Einplatinen-Computer

Auch für den Analysator soll eine Bewertung der Performance, der in Tabelle 4.1 aufgeführten Systeme, erfolgen. Für diesen Zweck steht eine Benchmark-Funktion zur Verfügung. Als Nutzdatentyp kann für den Generator der Wert *BENCH* gesetzt werden. Versendet werden dann *IPv4*-Pakete mit einer bestimmten Zeichenfolge in den Nutzdaten, die der Analysator im *Bench*-Modus gesondert behandelt. Ausgewertet werden die Anzahl der versendeten Pakete und die Anzahl der empfangenen Pakete. Wiederum wird eine (für das Analysator-System) offensichtlich nicht leistbare Paketrate genutzt. Anhand der erfassten Pakete, kann die maximale Paketrate des Analysator-Systems bestimmt werden. Als Toleranz werden maximal 0,1% Paketverlust zugelassen. Dazu folgendes fiktives Beispiel:

Paketrate Generator:

$$PR_{Gen} = 100000 \frac{Pakete}{s} \tag{5.5}$$

Laufzeit:

$$t_{bench} = 30s \tag{5.6}$$

Vom Analysator erfasste Pakete:

$$n_{packets} = 15000000\ Pakete \tag{5.7}$$

Maximale Analysator-Paketrate:

$$PR_{Analyzer,max} = \frac{n_{packets}}{t_{bench}} = 50000\ \frac{Pakete}{s} \tag{5.8}$$

Um die Leistungsfähigkeit der Systeme für einen bestimmten Anwendungsfall bewerten zu können, wurden die verschiedenen Funktionen des Analysators nacheinander einzeln aktiviert. Die Sampleperiode wurde nicht verändert und betrug $T_S = 10ms$.

Tabelle 5.1 zeigt die Ergebnisse als Übersicht.

Tabelle 5.1.: Maximale Paketrate ($\frac{1}{s}$) der getesteten Systeme

	Raspberry Pi B[a]	Raspberry Pi 3B	Odroid C2	Lenovo X220[b]
Nur Paketerfassung	-	18100	85000	228000
Paketerfassung und PCap-Record	-	9700	63000	228000
Paketerfassung und Ergebnisse in Binärdatei	-	14600	71000	228000
Paketerfassung und Live-Plot	-	15300	75000	228000
Paketerfassung und Analyse Paketlaufzeit	-	9800	70000	228000
Paketerfassung und Analyse Zwischenankunftszeit	-	12200	69000	228000

[a] System nicht ausreichend leistungsfähig
[b] Limitiert durch Paketgenerator

Der Raspberry Pi B war nicht leistungsfähig genug, um verwertbare Ergebnisse zu erzielen. Während des Benchmarkdurchlaufs war das System nicht ansprechbar, Paketraten von $PR_{Analyzer,max} < 500\ \frac{Pakete}{s}$ wurden ermittelt, sind aber aus den genannten Gründen nicht belastbar. Als Hauptursache kann die Verfügbarkeit von nur einem Prozessorkern angesehen werden.

Limitierend für den Raspberry Pi 3B wirkt sich sein 100Mb/s-Netzwerkinterface und der, im Vergleich mit dem Odroid C2, niedriger getaktete Prozessor aus. Die Größe des Arbeitsspeichers hat hier keinen Einfluss. In allen Durchläufen wurden vom Gesamtsystem nur wenige hundert Megabyte genutzt.

Für den Lenovo X220 konnte der Paketgenerator keine ausreichend hohe Datenrate liefern. Es wurden stets sämtliche Pakete erfasst, während der Generator bereits mit maximalem Burst-Faktor ($\rightarrow$ 4.4) arbeitet.

Die verfügbaren Funktionen wirken sich sehr unterschiedlich aus. Augenscheinlich wenig Einfluss hat die Live-Plot-Funktion, was daran liegt, dass sie in einem eigenen, niedrig priorisierten Thread arbeitet. Nichtsdestotrotz kann die Live-Plot-Funktion, aufgrund der vergleichsweise geringen Leistung der Prozessoren der *Einplatinen-Computer*, nur mit sehr niedriger Aktualisierungsfrequenz verwendet werden.

Verhältnismäßig stark limitierend wirken sich die Analyse der Paketlaufzeit und der Zwischenankunftszeit aus. Hier müssen die Nutzdaten jedes Pakets untersucht werden, was Verarbeitungszeit erhöht.

Das parallele Schreiben des Paketverlaufs in eine PCap-Datei hat den stärksten Einfluss. Hier wird jedes Paket kopiert und in eine Datei geschrieben. Der Thread arbeitet mit erhöhter Priorität (Scheduler-Klasse „rt_sched_class“). Das Schreiben der Datenraten in eine Binärdatei ermöglicht eine, im Vergleich, höhere Paketrate.

Tabelle 5.2 zeigt die sogenannte *Internet Mix* Paketverteilung.

Tabelle 5.2.: *Internet Mix* von Paketen (Quelle: [25])

Paketgröße	Anzahl	Anteil an der Gesamtanzahl	Bytes	Anteil an der Gesamtdatenmenge
40	7	58,33%	280	7%
576	4	33,33%	2304	56%
1500	1	8,33%	1500	37%

Legt man diese Verteilung zu Grunde, kann mit dem Odroid C2 bei einer angenommenen maximalen Paketrate von $PR_{Analyzer,max} = 60000\ \frac{Pakete}{s}$, eine Verbindung mit einer Datenrate von $DR \approx 20,42\ \frac{MB}{s}$ statistisch ohne Paketverlust analysiert werden. Andere Quellen

(bspw. [12]) gehen davon aus, dass die typische Internet-Paketverteilung 40% sehr kleine ($\approx 40\ Byte$) und 35% maximal große (Ethernet-*MTU*, $1500\ Byte$) enthält, was dem Analysator zugute käme.

5.12. Anwendung

Im Folgenden soll an zwei Bespielen gezeigt werden, warum eine möglichst kleine *Sampleperiode* wichtig ist und welche Störeinflüsse damit detektiert werden können.

Für das erste Beispiel erzeugt der Paketgenerator wiederholt für eine Dauer von $t_{Run} = 20ms$ eine Datenrate von $DR = 200\ \frac{Mb}{s}$. Zwischen jedem Durchlauf gibt es eine Pause von $t_{Pause} = 500ms$. Die Paketgröße beträgt 550 Byte, je Durchlauf werden damit ca. 950 Pakete versendet.

Da die *Sampleperiode* des Analysators variabel ist, kann nun untersucht werden, wie genau die Netzwerklast für verschiedene *Sampleperioden* abgebildet werden würde. Folgende *Sampleperioden* werden verwendet:

a) $T_{S,a} = 500\ ms$

b) $T_{S,b} = 100\ ms$

c) $T_{S,c} = 10\ ms$

d) $T_{S,d} = 1\ ms$

Für $T_{S,a} = 500\ ms$ wird eine Datenrate von $DR_a \approx 8\ \frac{Mb}{s}$ erwartet:

$$DR_a = \frac{t_{Run}}{T_{S,a}} \cdot DR = \frac{0,02s}{0,5s} \cdot 200\ \frac{Mb}{s} = 8\ \frac{Mb}{s} \tag{5.9}$$

Das Ergebnis ist im Plot (Abb. 5.22) dargestellt.

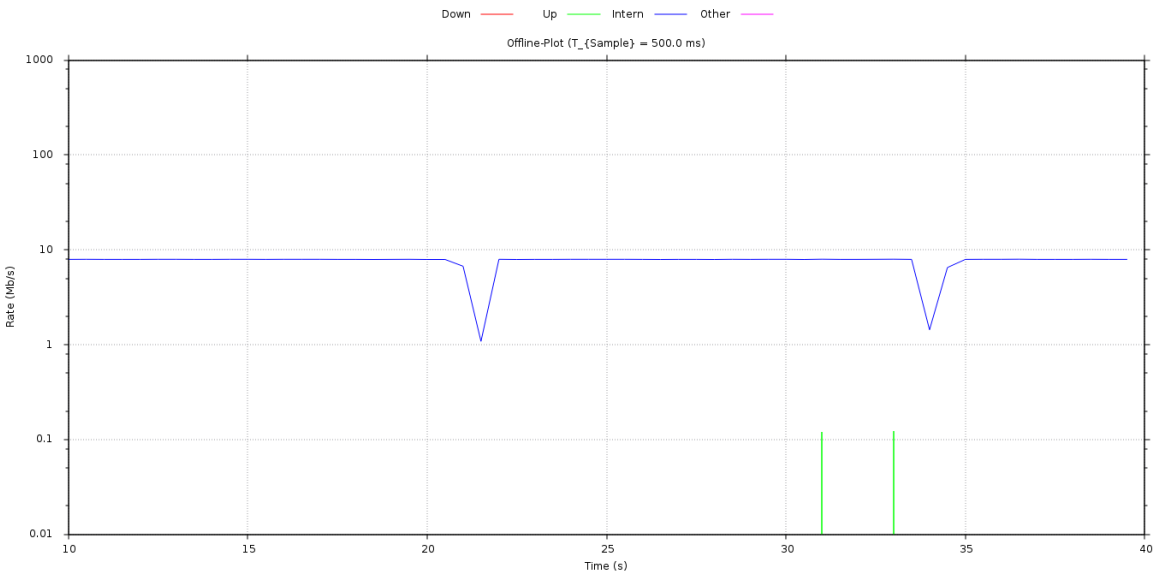

Abbildung 5.22.: Plot, Datenrate für Verkehrsanalyse mit $T_{S,a} = 500\ ms$

Für $T_{S,b} = 100\ ms$ wird eine Datenrate von $DR_b \approx 40\ \frac{Mb}{s}$ erwartet:

$$DR_a = \frac{t_{Run}}{T_{S,b}} \cdot DR = \frac{0,02s}{0,1s} \cdot 200\ \frac{Mb}{s} = 40\ \frac{Mb}{s} \tag{5.10}$$

Das Ergebnis ist im Plot (Abb. 5.23) dargestellt.

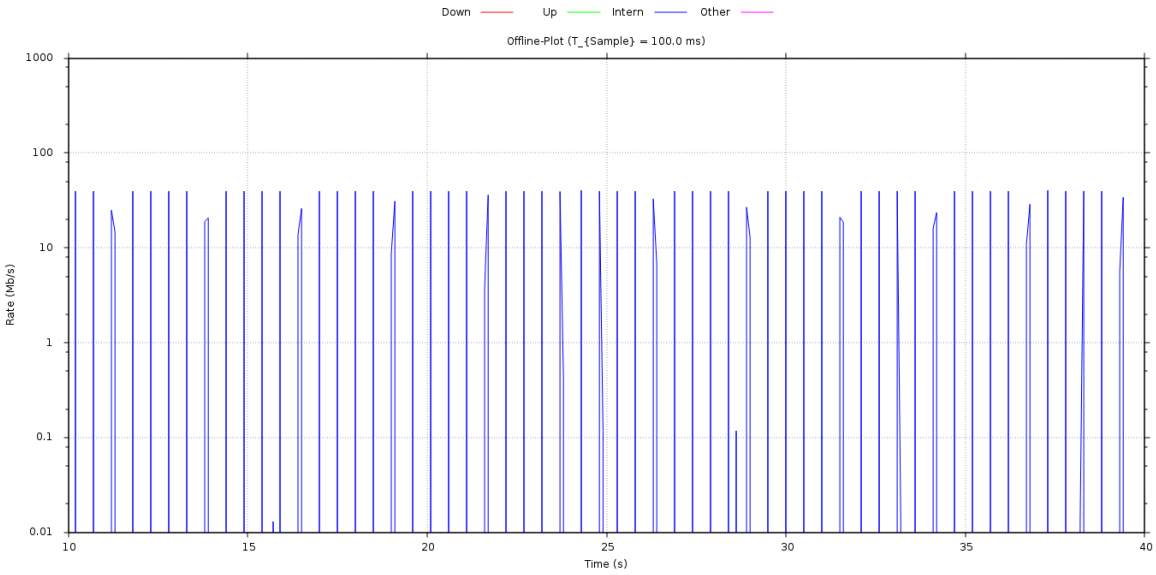

Abbildung 5.23.: Plot, Datenrate für Verkehrsanalyse mit $T_{S,b} = 100\ ms$

Für $T_{S,c} = 10\ ms$ wird eine Datenrate von $DR_c \approx 200\ \frac{Mb}{s}$ erwartet (mit $t_{Run} = T_{S,c}$, da $t_{Run} \leq T_S$ sein muss):

$$DR_a = \frac{t_{Run}}{T_{S,c}} \cdot DR = \frac{0,01s}{0,01s} \cdot 200\ \frac{Mb}{s} = 200\ \frac{Mb}{s} \tag{5.11}$$

Das Ergebnis ist im Plot (Abb. 5.24) dargestellt.

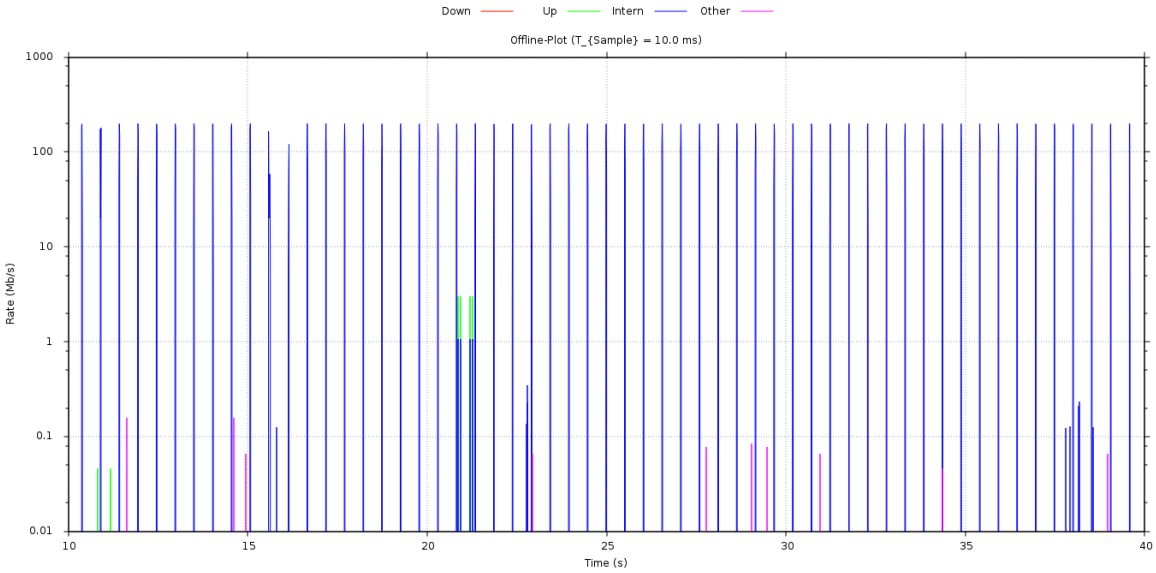

Abbildung 5.24.: Plot, Datenrate für Verkehrsanalyse mit $T_{S,c} = 10\ ms$

Für $T_{S,d} = 1\ ms$ wird eine Datenrate von $DR_d \approx 200\ \frac{Mb}{s}$ erwartet (mit $t_{Run} = T_{S,d}$, da $t_{Run} \leq T_S$ sein muss):

$$DR_a = \frac{t_{Run}}{T_{S,d}} \cdot DR = \frac{0,001s}{0,001s} \cdot 200\ \frac{Mb}{s} = 200\ \frac{Mb}{s} \tag{5.12}$$

Das Ergebnis ist im Plot (Abb. 5.25) dargestellt. Der Plot zeigt einen Zeitraum von nur $t_{Plot} = 1s$, um die einzelnen Durchläufe besser sichtbar zu machen.

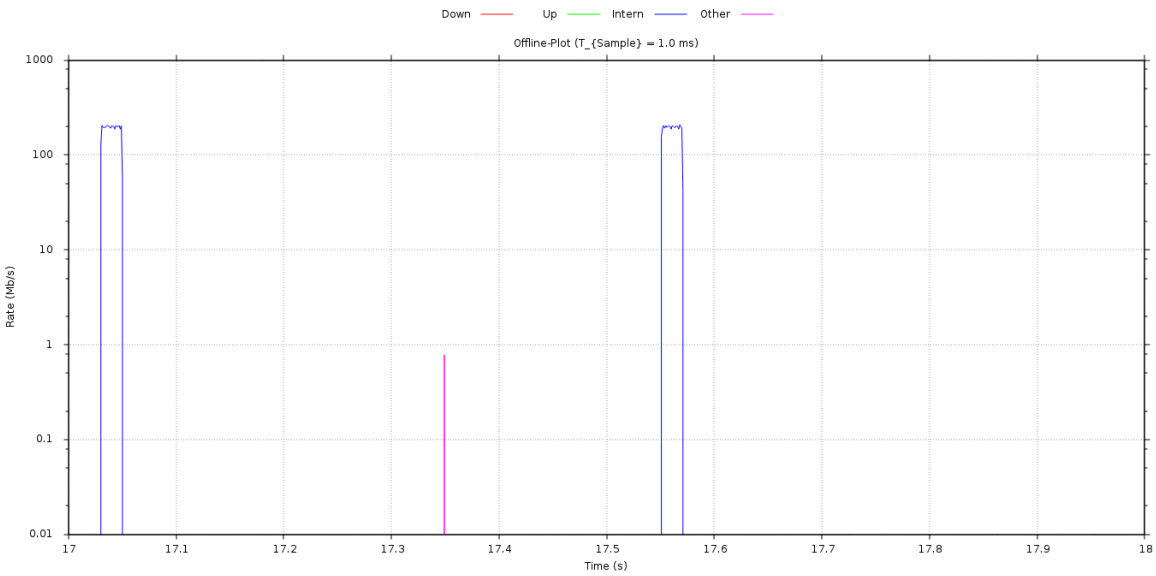

Abbildung 5.25.: Plot, Datenrate für Verkehrsanalyse mit $T_{S,d} = 1\ ms$ und $t_{Plot} = 1s$

Im Plot kann zusätzlich ein Durchlauf genauer betrachtet werden, um die Dauer t_{Run} zu verifizieren. Der Plot zeigt die generierte Netzwerklast (Abb. 5.26).

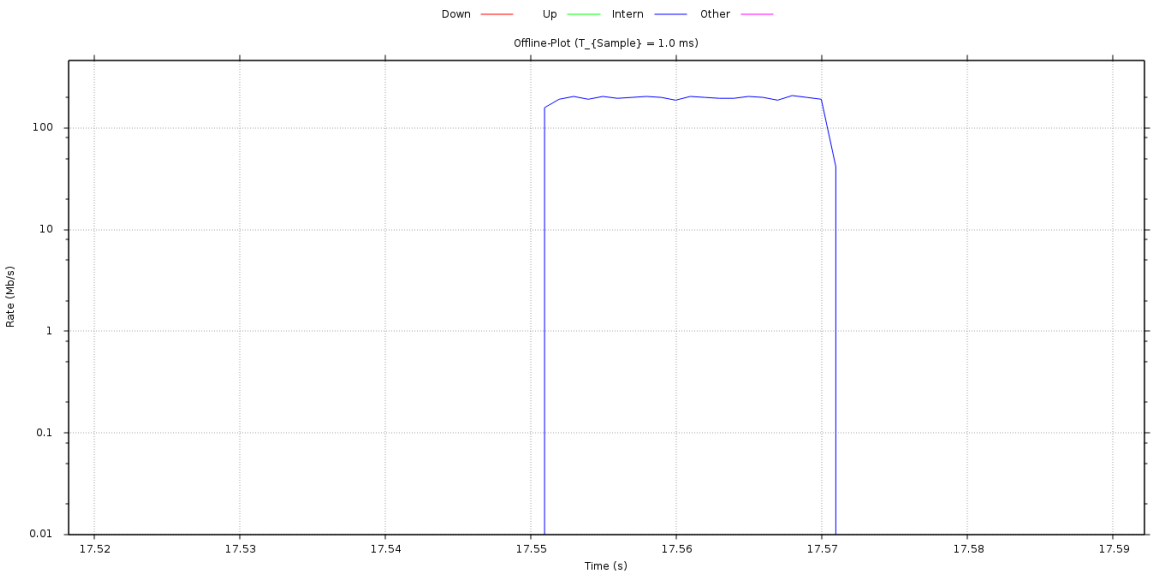

Abbildung 5.26.: Plot, Datenrate für Verkehrsanalyse mit $T_{S,d} = 1\ ms$, für einen Durchlauf vergrößert

An folgendem Testszenario (Übersicht Abb. 5.27) soll überprüft werden ab welcher Störungs-

dauer negative Auswirkungen zu beobachten sind. Störungen mit unterschiedlicher Länge werden vom Störgenerator erzeugt und an das Störziel gesendet. Um zu verhindern, dass die Pakete vom Router verworfen werden, wird eine portbasierte Weiterleitung zum Störziel eingerichtet. Für den Störverkehr werden daher *UDP*-Pakete verwendet. Die Pakete werden vom Störziel verworfen. Da sich Störziel und Download-Client im selben Netzsegment befinden, teilen sich Stör- und Nutzdaten mindestens die Verbindung zwischen Switch und Router (Uplink). Der Port des Switch, mit dem der Router verbunden ist, wird auf einen Mirror-Port gespiegelt, um den Netzwerkverkehr zu analysieren.

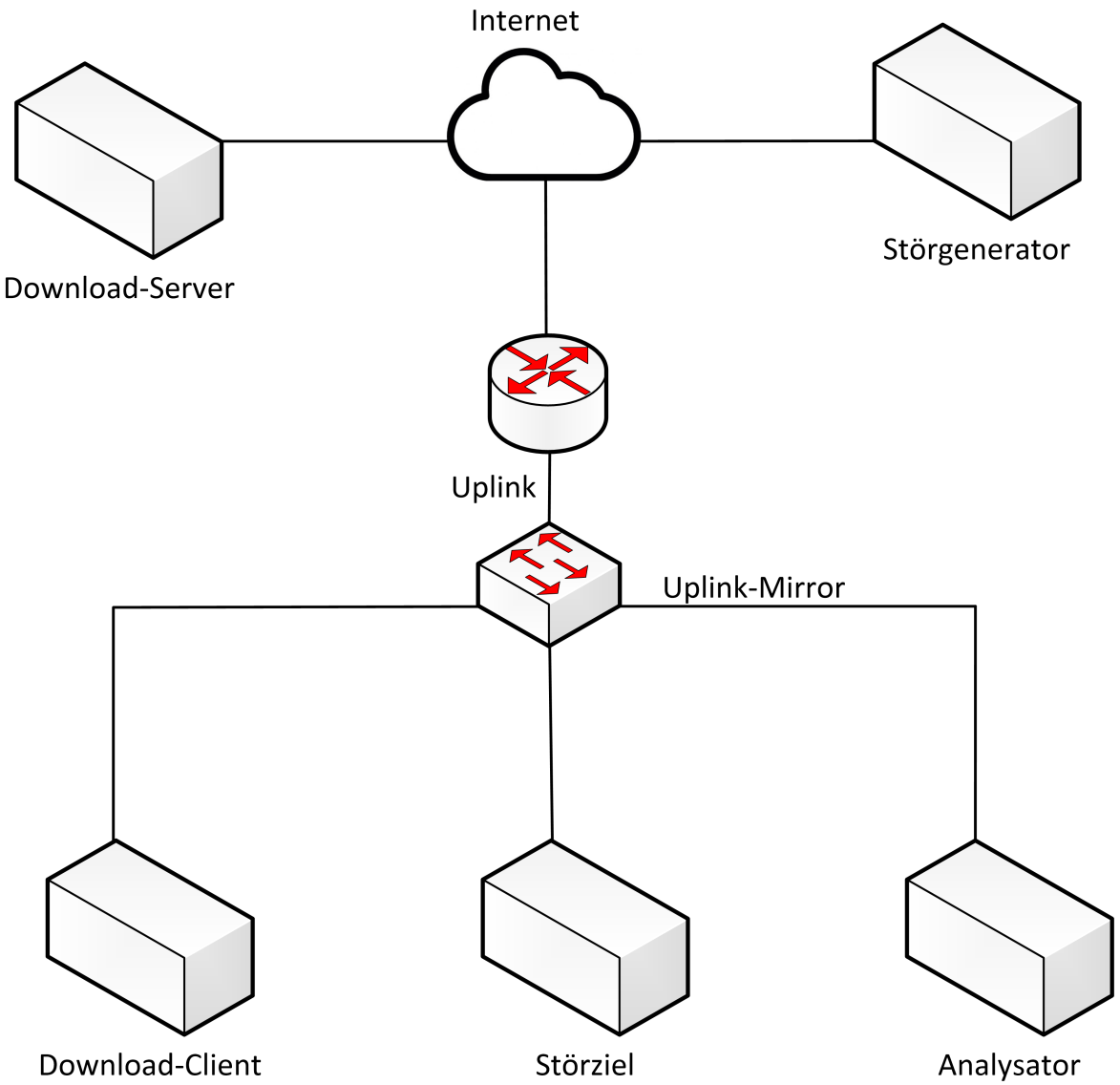

Abbildung 5.27.: Testszenario

Für verschiedene Störduauern t_{Run} soll untersucht werden, ob sie mit dem Analysator detektiert werden können. Die Störungen werden in der Verkehrsklasse „Other“ erfasst, da am Generator als Datentyp *TAG* gesetzt ist ($\rightarrow$ 4.4). Der Analysator kann daher den generischen Störverkehr vom Nutzverkehr unterscheiden. Damit wird ermöglicht, dass Störungen und Nutzverkehr in einem Plot gezeigt werden können. Als Nutzverkehr lädt der Download-Client vom Download-Server eine Datei. Für die Stördaten wird eine Paketgröße von $PS = 100\ Byte$ und einen Datenrate von $DR_{Stoer} = 20\ \frac{Mb}{s}$ verwendet. Zwischen den Störungen gibt es jeweils eine Pause von $t_{Pause} = 5s$.

Begonnen werden soll mit einer Störduauer von $t_{Run} = 100ms$. Um den Nutzen der kleinen *Sampleperiode* zu verdeutlichen, werden die Daten mit unterschiedlichen *Sampleperioden* erfasst. Die Abbildungen 5.28 bis 5.31 zeigen die Ergebnisse. Während bei $T_S = 500ms$ ein Einbruch der Downloadrate nahezu nicht erkennbar ist, werden bei $T_S = 5ms$ die Auswirkungen sehr deutlich. Zu Erkennen ist (Abb. 5.30 und 5.31), wie die Flusskontrolle arbeitet und das Receive-Window (siehe 3.1.4) bei Eintreten der Störung verkleinert, um es nach der Störung wieder zu vergrößern.

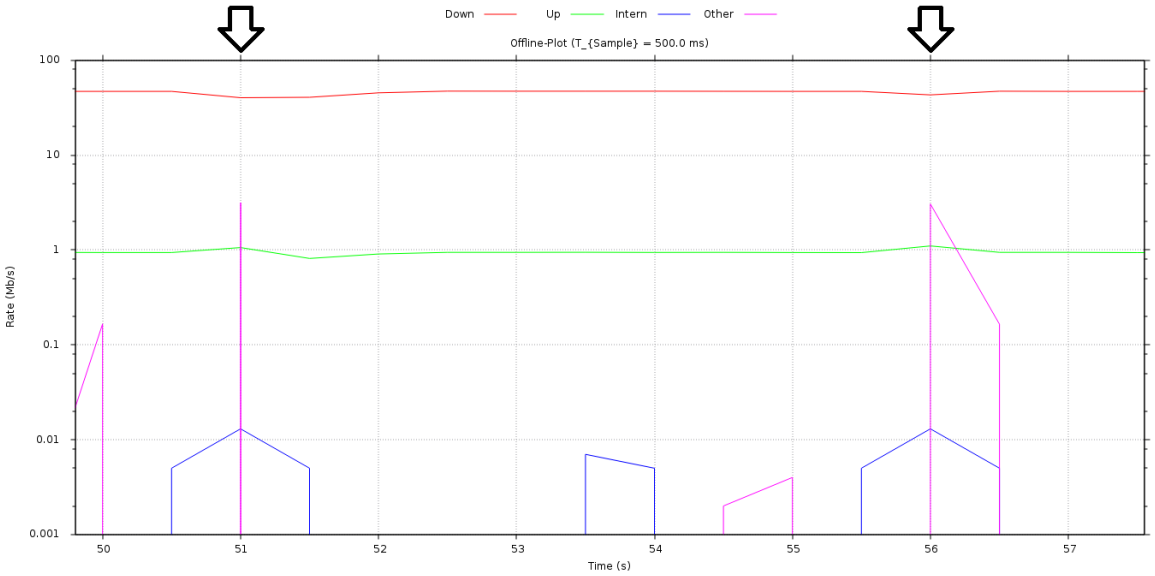

Abbildung 5.28.: Testszenario, Auswirkungen einer Störung mit Störduauer $t_{Run} = 100ms$, Erfassung der Datenraten mit $T_S = 500ms$ (Störungen in pink und mit Pfeilen markiert, Nutzdownloaddatenrate in rot, Nutzuploaddatenrate in grün)

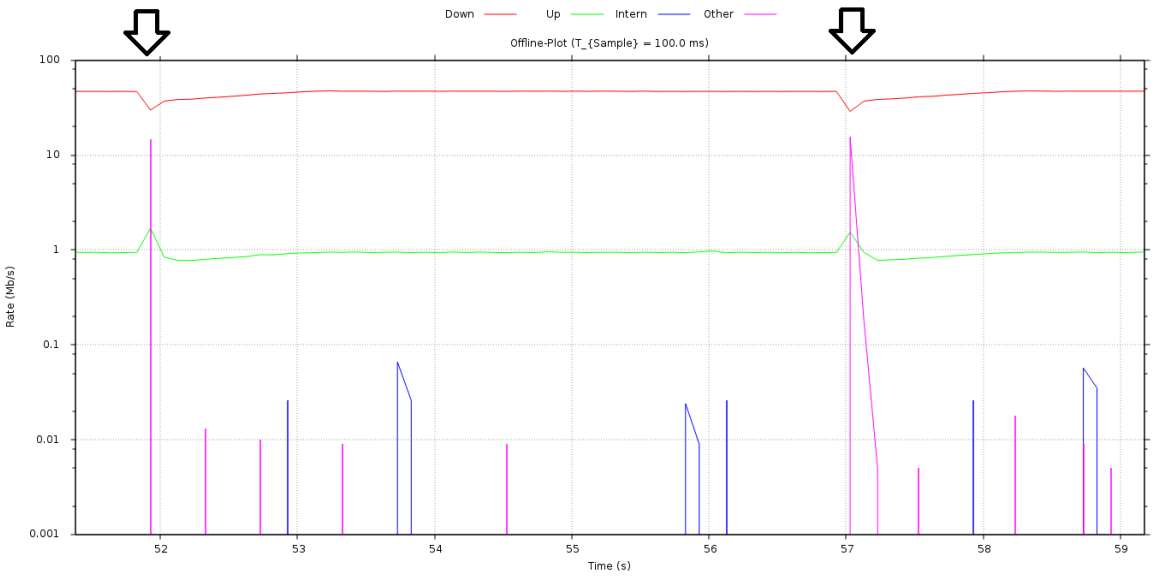

Abbildung 5.29.: Testszenario, Auswirkungen einer Störung mit Stördauer $t_{Run} = 100ms$, Erfassung der Datenraten mit $T_S = 100ms$ (Störungen in pink und mit Pfeilen markiert, Nutzdownloaddatenrate in rot, Nutzuploaddatenrate in grün)

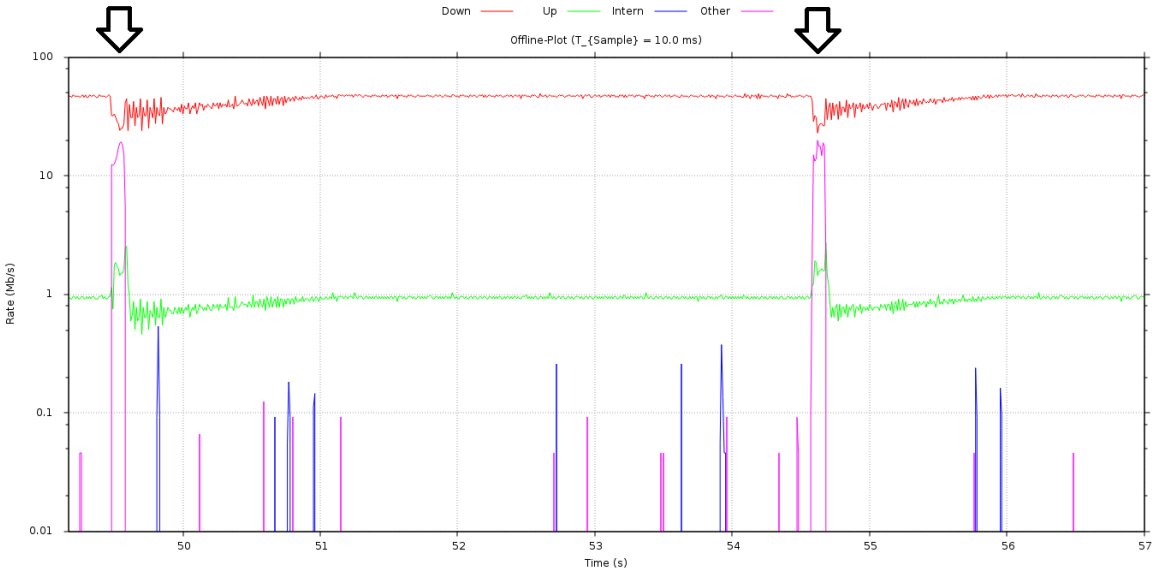

Abbildung 5.30.: Testszenario, Auswirkungen einer Störung mit Stördauer $t_{Run} = 100ms$, Erfassung der Datenraten mit $T_S = 10ms$ (Störungen in pink und mit Pfeilen markiert, Nutzdownloaddatenrate in rot, Nutzuploaddatenrate in grün)

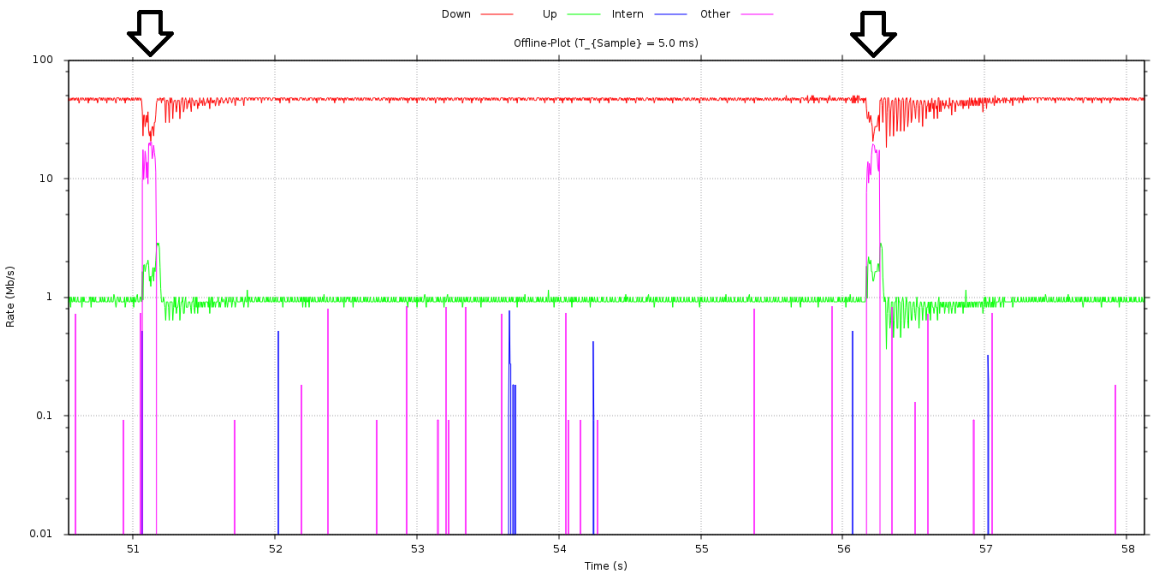

Abbildung 5.31.: Testszenario, Auswirkungen einer Störung mit Störduauer $t_{Run} = 100ms$, Erfassung der Datenraten mit $T_S = 5ms$ (Störungen in pink und mit Pfeilen markiert, Nutzdownloaddatenrate in rot, Nutzuploaddatenrate in grün)

Die Dauer der Störung wird nun verringert. Bei einer *Sampleperiode* von $T_S = 5ms$ soll untersucht werden, bis zu welcher Störduauer eine Auswirkung auf die Nutzdatenraten sichtbar bleibt. Die Abbildungen 5.32 bis 5.35 zeigen die Ergebnisse für Störduauern von $t_{Run} = 2ms..20ms$. Seriös erkennbar bleibt die Auswirkung der Störung für $t_{Run} = 5ms$. Für kleinere Störduauern würde die Störung nicht als solche erkannt, da im realen Umfeld der Störverkehr nicht hervorgehoben wäre.

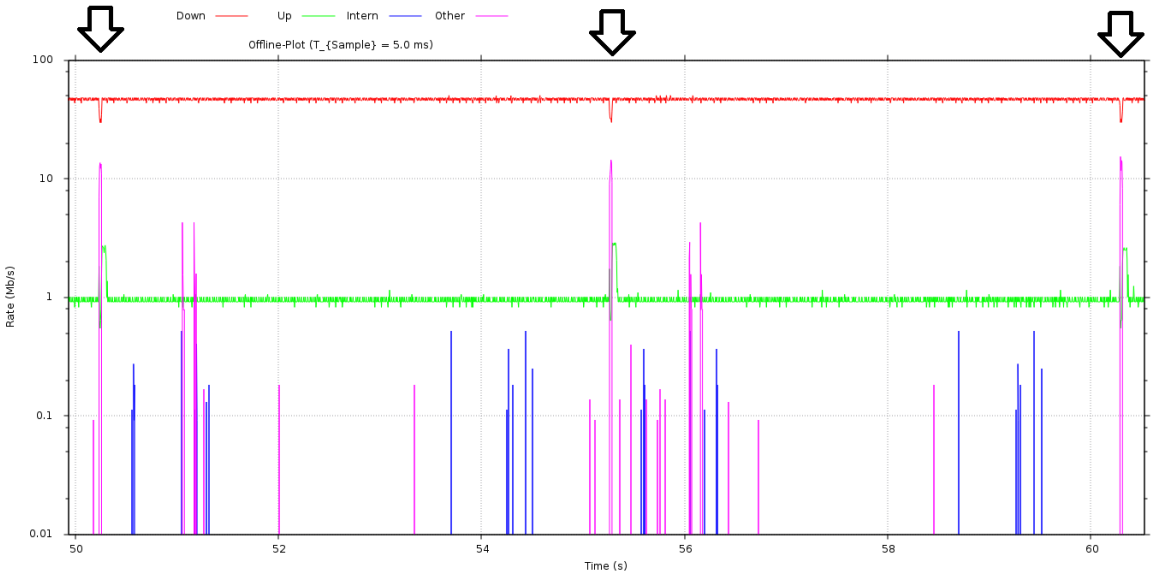

Abbildung 5.32.: Testszenario, Auswirkungen einer Störung mit Störduauer $t_{Run} = 20ms$, Erfassung der Datenraten mit $T_S = 5ms$ (Störungen in pink und mit Pfeilen markiert, Nutzdownloaddatenrate in rot, Nutzuploaddatenrate in grün)

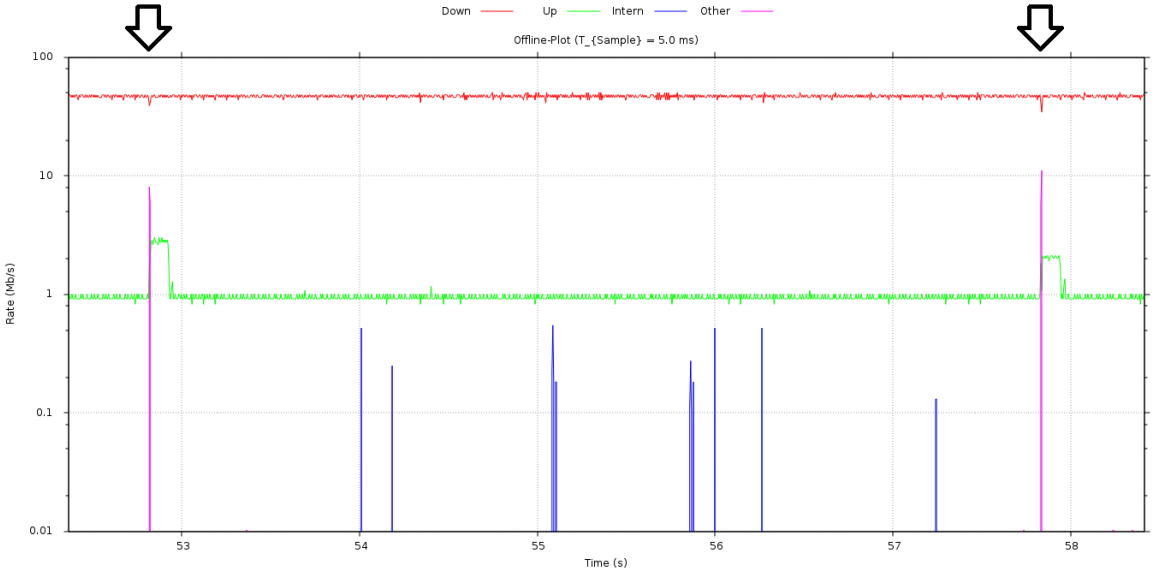

Abbildung 5.33.: Testszenario, Auswirkungen einer Störung mit Störduauer $t_{Run} = 10ms$, Erfassung der Datenraten mit $T_S = 5ms$ (Störungen in pink und mit Pfeilen markiert, Nutzdownloaddatenrate in rot, Nutzuploaddatenrate in grün)

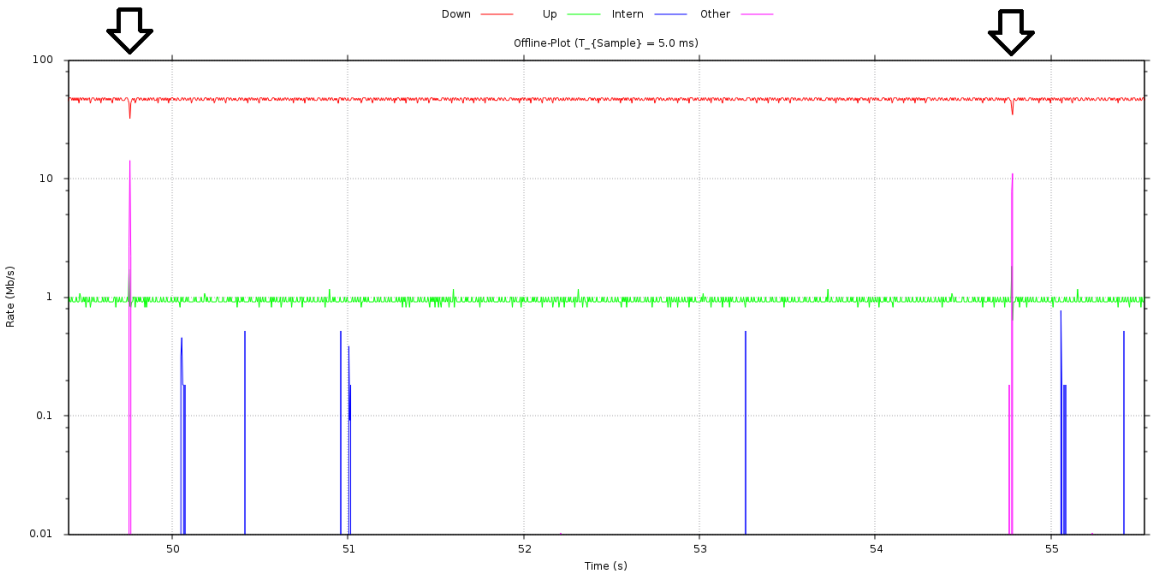

Abbildung 5.34.: Testszenario, Auswirkungen einer Störung mit Stördauer $t_{Run} = 5ms$, Erfassung der Datenraten mit $T_S = 5ms$ (Störungen in pink und mit Pfeilen markiert, Nutzdownloaddatenrate in rot, Nutzuploaddatenrate in grün)

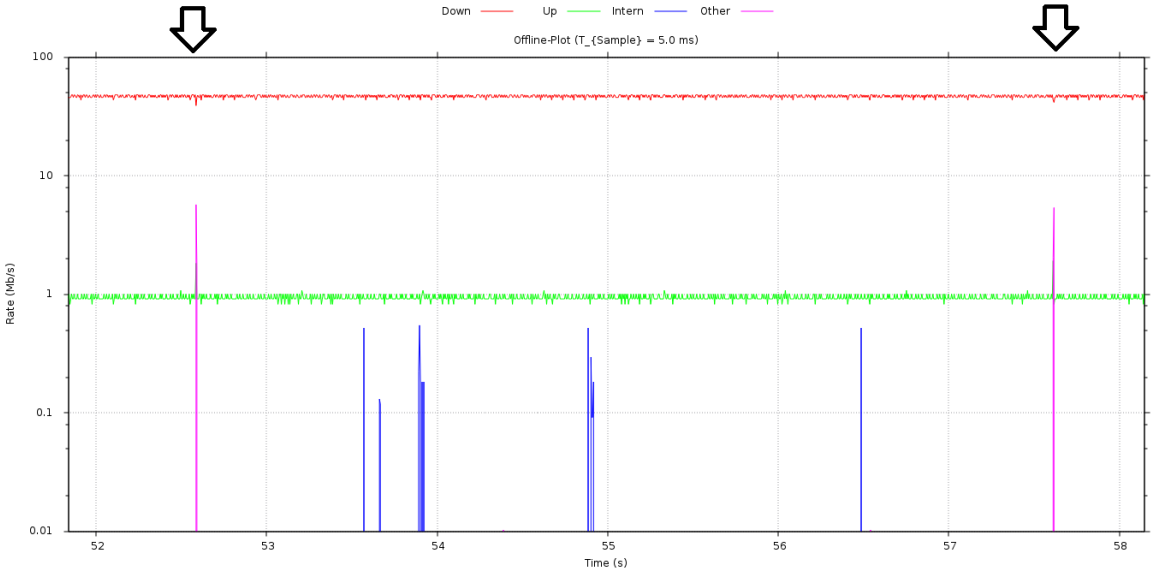

Abbildung 5.35.: Testszenario, Auswirkungen einer Störung mit Stördauer $t_{Run} = 2ms$, Erfassung der Datenraten mit $T_S = 5ms$ (Störungen in pink und mit Pfeilen markiert, Nutzdownloaddatenrate in rot, Nutzuploaddatenrate in grün)

6. Ausblick

Entwickelt wurde ein System, das den Anwender in die Lage versetzt, die Auswirkungen von störendem Netzwerkverkehr zu untersuchen. Ziel war es die aktuellen Datenraten mit einer möglichst kleinen *Sampleperiode* zu analysieren, die daraus entstehenden Vorteile wurden ausgiebig besprochen. Selbst auf einem *Einplatinen-Computer* kann der Analysator mit einer *Sampleperiode* von 10ms arbeiten, der Einsatz der Software auf einem Handgerät wäre demnach denkbar. Abhängig von der Leistungsfähigkeit der Hardware können Module der Anwendung deaktiviert werden, um Rechenleistung einzusparen. Für spätere Analysen, sozusagen offline, kann der Netzwerkverkehr in *PCap*-Records aufgezeichnet werden. Aufgrund der Verbreitung dieses Formates können die so gespeicherten Sequenzen mit Programmen wie bspw. Wireshark analysiert werden.

Generator und Analysator können über einen Streamsocket gesteuert werden, was die Kontrolle der Anwendungen aus anderen Programmen heraus ermöglicht. Zuerst genannt sei an dieser Stelle natürlich eine grafische Oberfläche. Als Erweiterung könnten Sockettyp und Port beim Start der Anwendung per Parameter übergeben werden. Weiterhin müsste für eine Absicherung gegen unberechtigte Zugriffe gesorgt werden.

Für Weiterentwicklungen des Generators wäre denkbar, dass die Einstellungen im laufenden Betrieb geändert werden können. Derzeit muss der Generator-Thread dazu gestoppt werden. Weiterhin nützlich wäre eine Paketerzeugung auf *MAC*-Layer und die Erzeugung von Paketen mit (in einem bestimmten Bereich) zufälliger Quell- und Ziel-Adresse. Außerdem könnten weitere Häufigkeitsverteilungen für die Zwischenankunftszeit implementiert werden. In Kombination mit Paketgrößen, die ebenfalls bestimmten Verteilungen gehorchen, könnte Netzwerkverkehr erzeugt werden, der einer realen Netzwerklast näher kommt. Die derzeitige Messung der Paketlaufzeit basiert auf einen Zeitstempel, der vom Generator verschickt und vom Analysator ausgewertet wird. Alternativ könnte ein Verfahren, ähnlich dem bei *NTP* verwendeten, getestet werden. Der Analysator würde dabei ein besonders gekennzeichnetes Paket an den Generator senden. Dieser würde es schlicht zurücksenden. Davon ausgehend, dass die Laufzeit je Richtung identisch ist, könnte die einfache Laufzeit näherungsweise ermittelt werden. Als Fehler bliebe die Verarbeitungszeit des Generators, eine Synchronisation der Systemuhren wäre nicht nötig.

Auch für den Analysator sind Erweiterungen denkbar. Aktuell wird für feststehende Events ein Eintrag in der Logdatei erzeugt. Hier könnten verschiedene Loglevel implementiert wer-

den, um bei problemlosem Betrieb für Übersicht zu sorgen bzw. die Fehlersuche zu vereinfachen. Abschnitt 5.6 beschreibt die Erfassung der Zwischenankunftszeiten. Einen entsprechenden Plot zeigt bspw. Abbildung 5.17. Um die Ursache für die Peaks bei bestimmten Zeiten zu ergründen, könnte eine Art Filter implementiert werden, das eine bestimmte Anzahl von Paketen vor und nach dem Auftreten einer bestimmten Zwischenankunftszeit (eines Bereichs) in ein *PCap*-Record speichert. Die Analyse könnte dann mit den bereits angesprochenen Programmen erfolgen. Darüber hinaus ist die Analyse weiterer Eigenschaften denkbar. Genannt seien hier die Paketgröße (Histogramm) und die Ankunftsreihenfolge, die anhand der Sequenznummern im *TCP*-Header ausgewertet werden könnte (wie oft muss neu sortiert werden). Anhand der Adressen könnte auch eine Sortierung der Stationen im Netzwerk nach verursachtem Netzwerkverkehr eingebaut werden, um die Top-Talker zu identifizieren und so möglicherweise auch ungewollte (schädliche) Netzwerklast zu finden.

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[1] *Datenblatt: LinkRunner AT*. – URL http://docs-europe.electrocomponents.com/webdocs/136f/0900766b8136fd0d.pdf. – Fluke Corporation; Online; Stand 13. September 2016

[2] *Datenblatt: TestPort Ethernet Protocol Analyzer*. – URL http://www.nextgigsystems.com/protocol_testing/handheld_traffic_generator_files/TestPort-pa-NextGig.pdf. – NextGig Systems; Online; Stand 13. September 2016

[3] *gnuplot_i reference Manual*. – URL http://ndevilla.free.fr/gnuplot/gnuplot_i/index.html. – Online; Stand 8. August 2016

[4] BROUGHTON, Jayson: *Fun with ethtool*. – URL http://www.linuxjournal.com/content/fun-ethtool. – Online; Stand 10. Juli 2016

[5] CARLE, Georg ; GÜNTHER, Stephan M. ; HEROLD, Nadine ; POSSELT, Stephan: *Grundlagen Rechnernetze und Verteilte Systeme*. – URL https://www.net.in.tum.de/fileadmin/TUM/teaching/grnvs/ss12/slides_chap4.pdf. – Vorlesungsskript TU München, SoSe 2012; Online; Stand 21. August 2016

[6] CORSETTI, Lisa: *Controlling Hardware with ioctls*. – URL http://www.linuxjournal.com/article/6908. – Online; Stand 10. Juli 2016

[7] ENDRES, Johannes: *Zeitdienst - NTP-Server und Client unter Linux einrichten*. – URL http://www.heise.de/netze/artikel/Zeitdienst-221732.html. – Online; Stand 12. September 2016

[8] FICHTNER, Klaus ; HEMMELING, Daniel ; KOHLMORGEN, Joachim ; LIESENFELD, Andre ; LUTZ, Heinz E. ; POHLMANN, Ralf ; SCHULZE, Mathias: *Netzwerke IPv6, RRZN Handbuch*. 1. Ausgabe. HERDT-Verlag für Bildungsmedien, 2014

[9] GRÄFE, Martin: *C und Linux*. 4. Auflage. Carl Hanser Verlag, 2010. – ISBN 978-3-446-42176-9

[10] HAGER, Orm: *Allgemeines zum Network Time Protocol*. – URL http://doc-tcpip.org/Ntp/basics.html. – Online; Stand 25. August 2016

[11] HÖNIG, Timo: *Der O(1)-Scheduler im Kernel 2.6*. 2004. – URL http://www.linux-magazin.de/Ausgaben/2004/02/Die-Reihenfolge-zaehlt. – Online; Stand 22. August 2016

[12] JOHN, Wolfgang ; TAFVELIN, Sven: *Analysis of Internet Backbone Traffic and Header Anomalies observed*. – URL http://conferences.sigcomm.org/imc/2007/papers/imc91.pdf. – Chalmers University of Technology; Online; Stand 12. September 2016

[13] KLEUKER, Stephan: *Grundkurs Software Engineering mit UML*. 3. Auflage. Springer Vieweg, 2013. – ISBN 978-3-658-00641-9

[14] KOBUS, Jacek ; SZKLARSKI, Rafał: *Completely Fair Scheduler and its tuning*. – URL https://www.fizyka.umk.pl/~jkob/prace-mag/cfs-tuning.pdf. – Online; Stand 20. Mai 2016

[15] MITTAG, Hans-Joachim: *Statistik - Eine interaktive Einführung*. Springer Verlag, 2011. – ISBN 978-3-642-17817-7

[16] PAPULA, Lothar: *Mathematische Formelsammlung*. 10. Auflage. Vieweg + Teubner, 2009. – ISBN 978-3-8348-0757-1

[17] PAPULA, Lothar: *Mathematik für Ingenieure und Naturwissenschaftler Band 3*. 6. Auflage. Vieweg + Teubner, 2011. – ISBN 978-3-8348-1227-8

[18] QUADE, Jürgen ; KUNST, Eva-Katharina: *Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 67*. 2013. – URL http://www.linux-magazin.de/Ausgaben/2013/04/Kern-Technik. – Online; Stand 12. August 2016

[19] RIGGERT, Wolfgang: *Rechnernetze*. 5. Auflage. Carl Hanser Verlag, 2014. – ISBN 978-3-446-44204-7

[20] SCHMIDT, Klaus ; DAUSCH, Martin: *Netzwerke Grundlagen, RRZN Handbuch*. 9. Ausgabe. HERDT-Verlag für Bildungsmedien, 2014

[21] WIKI, Wireshark: *Libpcap File Format*. – URL https://wiki.wireshark.org/Development/LibpcapFileFormat. – Online; Stand 1. September 2016

[22] WIKIPEDIA: *Ethernet*. – URL https://de.wikipedia.org/w/index.php?title=Ethernet&oldid=156747604. – Online; Stand 18. August 2016

[23] WIKIPEDIA: *IEEE 802.1X*. – URL https://de.wikipedia.org/w/index.php?title=IEEE_802.1X&oldid=152687785. – Online; Stand 13. September 2016

[24] WIKIPEDIA: *IEEE 802.3*. – URL https://en.wikipedia.org/w/index.php?title=IEEE_802.3&oldid=732218387. – Online; Stand 16. August 2016

[25] WIKIPEDIA: *Internet Mix*. – URL https://en.wikipedia.org/w/index.php?title=Internet_Mix&oldid=726870362. – Online; Stand 11. September 2016

[26] WIKIPEDIA: *MAC-Adresse*. – URL https://de.wikipedia.org/w/index.php?title=MAC-Adresse&oldid=155964461. – Online; Stand 18. August 2016

[27] WIKIPEDIA: *OSI-Modell*. – URL https://de.wikipedia.org/w/index.php?title=OSI-Modell&oldid=156289574. – Online; Stand 16. August 2016

[28] WIKIPEDIA: *Sockets (Software)*. – URL https://de.wikipedia.org/w/index.php?title=Socket_%28Software%29&oldid=155482291. – Online; Stand 10. August 2016

[29] WOLF, Jürgen: *C von A bis Z*. 3. Auflage. Galileo Press, 2009. – ISBN 978-3-8362-1411-7

[30] WOLF, Jürgen: *Linux-UNIX-Programmierung*. 3. Auflage. Galileo Press, 2009. – ISBN 978-3-8362-1366-0

[31] ZAHN, Markus: *Unix-Netzwerkprogrammierung*. Springer Verlag, 2006. – ISBN 3-540-00299-5

Glossar

CFS

Completely Fair Scheduler

CRC

Cyclic Redundancy Check, Verfahren um eine Prüfsumme zu generieren, die es zulässt fehlerhafte Daten zu erkennen

DCF77

Langwellensender, der Zeitinformationen aussendet (DCF - Rufzeichen Langwellensender, 77 - Zeit)

DHCP

Dynamic Host Control Protocol, Protokoll mit dem Adresskonfigurationsonformationen verteilt werden können

DNS

Domain Name System, Dienst für die Umsetzung von IP-Adressen in Domain-Namen

Einplatinen-Computer

Computersystem, bei dem alle Komponenten auf einer Platine verbaut sind

GPS

Global Positioning System

GUI

Graphical User Interface

IHL

IP-Header-Length, Länge des Headers eines IPv4-Pakets

Internet Mix

Angenommene Paketgrößenverteilung für Netzwerkverkehr ([25])

IPC

Inter Process Communication

IPv4

Internet Protocol Version 4, Protokoll der Vermittlungsschicht

IPv6

Internet Protocol Version 6, Protokoll der Vermittlungsschicht

MAC

Medium Access Control

MTU

Maximum Transfer Unit, maximale Paketgröße ohne Fragmentierung

Named Pipe

Datenverbindung zwischen Prozessen, arbeitet nach dem FIFO-Prinzip (First In First Out)

NAT

Network Address Translation, Verfahren um Rechnernetze zu verbinden

netcat

Kommandozeilentool, das Kommunikation über Netzwerkverbindungen ermöglicht

NIC

Network Interface Card

NTP

Network Time Protocol

OSI

Open Systems Interconnect

PCap

Packet-Capture

PoE

Power over Ethernet

Port-Mirroring

Funktion eines Netzelements um den Netzwerkverkehr eines Port an einen anderen zu spiegeln

POSIX

Portable Operating System Interface for UniX, Schnittstelle zwischen Anwendung und Betriebssystem

Promiscuous Mode

Empfangsmodus für Netzwerkinterfaces in dem auch fremdadressierte Pakete empfangen werden

Sampleperiode

Abtastperiode

Scheduler

Teil eines Betriebssystems, dass die Prozessorressourcen verwaltet

Socket

Kommunikationsendpunkt

Task

Prozess, Ausführung wird vom Scheduler verwaltet

TCP

Transmission Control Protocol, Protokoll der Transportschicht

TDR

Time Domain Reflectometry - Zeitbereichsreflektometrie: Analyse anhand von Impulsreflektionen

ToS

Type of Service

TTL

Time To Live

UDP

User Datagram Protocol, Protokoll der Transportschicht

UTC

Universal Time Coordinated, Koordinierte Weltzeit, Bezug zur Mitteleuropäischen Zeit: $MEZ = UTC + 1$

VoIP

Voice over Internet Protocol

A. Quellcodes

Wie einleitend erwähnt, befinden sich an dieser Stelle ausgewählte Quellcodes, auf die in der Arbeit verwiesen wird. Die vollständigen Source-Codes befinden sich auf der beiliegenden DVD.

Listing A.1: Ethernet-Header-Struktur aus netinet/ifether.h

```
1 #define ETH_ALEN  6   /* Octets in one ethernet addr   */
2
3 struct ethhdr {
4   unsigned char h_dest[ETH_ALEN]; /* destination eth addr */
5   unsigned char h_source[ETH_ALEN]; /* source ether addr  */
6   unsigned short  h_proto;        /* packet type ID field */
7 }
```

Listing A.2: *IPv4*-Header-Struktur aus linux/ip.h

```
1 /*
2  * IPv4 header.
3  */
4 struct iphdr {
5 #if __BYTE_ORDER == __LITTLE_ENDIAN
6     unsigned int ihl:4;
7     unsigned int version:4;
8 #elif __BYTE_ORDER == __BIG_ENDIAN
9     unsigned int version:4;
10    unsigned int ihl:4;
11 #else
12 # error    "Please fix <bits/endian.h>"
13 #endif
14    u_int8_t tos;
15    u_int16_t tot_len;
16    u_int16_t id;
17    u_int16_t frag_off;
18    u_int8_t ttl;
19    u_int8_t protocol;
20    u_int16_t check;
21    u_int32_t saddr;
22    u_int32_t daddr;
```

```
23     /*The options start here. */
24   };
```

Listing A.3: *TCP*-Header-Struktur aus netinet/tcp.h

```
1  /*
2   * TCP header.
3   */
4  struct tcphdr {
5      __extension__ union
6      {
7        struct
8        {
9      u_int16_t th_sport;         /* source port */
10     u_int16_t th_dport;         /* destination port */
11     tcp_seq th_seq;          /* sequence number */
12     tcp_seq th_ack;          /* acknowledgement number */
13 # if __BYTE_ORDER == __LITTLE_ENDIAN
14     u_int8_t th_x2:4;         /* (unused) */
15     u_int8_t th_off:4;         /* data offset */
16 # endif
17 # if __BYTE_ORDER == __BIG_ENDIAN
18     u_int8_t th_off:4;         /* data offset */
19     u_int8_t th_x2:4;         /* (unused) */
20 # endif
21     u_int8_t th_flags;
22 # define TH_FIN     0x01
23 # define TH_SYN     0x02
24 # define TH_RST     0x04
25 # define TH_PUSH    0x08
26 # define TH_ACK     0x10
27 # define TH_URG     0x20
28     u_int16_t th_win;         /* window */
29     u_int16_t th_sum;         /* checksum */
30     u_int16_t th_urp;         /* urgent pointer */
31       };
32       struct
33       {
34     u_int16_t source;
35     u_int16_t dest;
36     u_int32_t seq;
37     u_int32_t ack_seq;
38 # if __BYTE_ORDER == __LITTLE_ENDIAN
39     u_int16_t res1:4;
40     u_int16_t doff:4;
41     u_int16_t fin:1;
42     u_int16_t syn:1;
43     u_int16_t rst:1;
44     u_int16_t psh:1;
```

```
45      u_int16_t ack:1;
46      u_int16_t urg:1;
47      u_int16_t res2:2;
48  # elif __BYTE_ORDER == __BIG_ENDIAN
49      u_int16_t doff:4;
50      u_int16_t res1:4;
51      u_int16_t res2:2;
52      u_int16_t urg:1;
53      u_int16_t ack:1;
54      u_int16_t psh:1;
55      u_int16_t rst:1;
56      u_int16_t syn:1;
57      u_int16_t fin:1;
58  # else
59  # error "Adjust your <bits/endian.h> defines"
60  # endif
61      u_int16_t window;
62      u_int16_t check;
63      u_int16_t urg_ptr;
64        };
65    };
66  };
```

Listing A.4: *UDP*-Header-Struktur

```
1  /*
2   * UDP header.
3   */
4  struct udphdr {
5      u_int16_t sport;
6      u_int16_t dport;
7      u_int16_t len;
8      u_int16_t check;
9  };
```

Listing A.5: *IPv6*-Header-Struktur

```
1  /*
2   * IPv6 header.
3   */
4  struct ipv6hdr {
5      u_int32_t
6          version:4,
7          traffic_class:8,
8          flow_label:20;
9      u_int16_t length;
10     u_int8_t next_header;
11     u_int8_t hop_limit;
12     struct in6_addr src;
```

```
13     struct in6_addr dst;
14 };
```

Listing A.6: Timespec Struktur

```
1 /*
2  * Struct timespec
3  */
4 struct timespec {
5   time_t tv_sec; // seconds
6   long tv_nsec; // nanoseconds
7 }
```

Listing A.7: Zeiger auf Struktur als Argument für Paket-Generator-Thread

```
1 /* Arguments for Packet-Generator-Call (needs pointer as single argument) */
2 struct pg_args {
3   packet_type p_type;
4   data_type d_type;
5   u_int16_t data_len;
6   char s_ip[16];
7   char d_ip[16];
8   u_int8_t ttl;
9   u_int8_t tos;
10  u_int8_t syn;
11  u_int8_t rst;
12  u_int8_t urg;
13  u_int8_t psh;
14  u_int8_t ack;
15  u_int8_t fin;
16  u_int16_t s_port;
17  u_int16_t d_port;
18  status t_status; // thread status
19  u_int16_t runlength;
20  u_int32_t datarate;
21  tspec delta_t;
22  u_int16_t num_cpus;
23  bool_t dist_exp;
24  long packets_sent;
25  int burst;
26  int max_packet_rate;
27 };
```

Listing A.8: Struktur zur Beschreibung eines Netzwerk-Interfaces

```
1 struct ifaddrs {
2   struct ifaddrs  *ifa_next;    /* Next item in list */
3   char            *ifa_name;    /* Name of interface */
4   unsigned int     ifa_flags;   /* Flags from SIOCGIFFLAGS */
```

```
5   struct sockaddr *ifa_addr;    /* Address of interface */
6   struct sockaddr *ifa_netmask; /* Netmask of interface */
7   union {
8     struct sockaddr *ifu_broadaddr; /* Broadcast address of interface */
9     struct sockaddr *ifu_dstaddr; /* Point-to-point destination address */
10  } ifa_ifu;
11  #define         ifa_broadaddr ifa_ifu.ifu_broadaddr
12  #define         ifa_dstaddr   ifa_ifu.ifu_dstaddr
13  void            *ifa_data;    /* Address-specific data */
14 };
```

Listing A.9: Struktur mit Argumenten für Plot-Thread

```
1  struct plot_args {
2    double *dr_down;
3    double *dr_up;
4    double *dr_intern;
5    double *dr_other;
6    double *dr_ipv6_gl;
7    double *dr_ipv6_ll;
8    double *t;
9    int pos;
10   u_int16_t current_tmp_file;
11   bool_t first_plot_page;
12   u_int32_t num_rate_vals;
13   double t_last;
14   double t_this;
15   bool_t pipe_warning; // for plot-pipe
16   bool_t pipe_overrun; // for plot-pipe
17   u_int32_t rd_fifo;
18   FILE *plot_tmp_file;
19   char tmp_filename[MAX_FILENAME_LEN];
20   gnuplot_ctrl *gnuplot_handle;
21   u_int32_t t_sample_ms;
22   u_int32_t num_cpus;
23   //bool_t enable_buffer_overrun_warning;
24   long dropped;
25   double dropped_rel;
26   long n_packets;
27   bool_t ipv6;
28   promisc_t promisc;
29   layer_t analysis_layer;
30 };
```

Listing A.10: Struktur mit Argumenten für File-Thread

```
1  struct file_args {
2    int rd_fifo;
3    double bytes_down;
```

```
4   double bytes_up;
5   double bytes_intern;
6   double bytes_ipv6_gl;
7   double bytes_ipv6_ll;
8   double bytes_other;
9   double t_this;
10  FILE *file;
11  char filename[FILENAME_LEN];
12  u_int32_t current_file;
13  struct file_bin fb;
14  long bin_file_sz;
15  u_int32_t num_bin_files;
16  u_int32_t num_cpus;
17 };
```

Listing A.11: Struktur für Binärdatei

```
1  struct file_bin { // bin to file
2    u_int32_t t_sample_ms;
3    double t_this;
4    double dr_down;
5    double dr_up;
6    double dr_ipv6_gl;
7    double dr_ipv6_ll;
8    double dr_other;
9    double dr_intern;
10   double dropped_rel;
11 };
```

B. Steuerbefehle Generator

Tabelle B.1.: Steuerbefehle für den Paketgenerator

Befehl	Funktion	Bedingung	Argument(e)
START	Startet den Paketgenerator	Generator gestoppt	-
STOP	Stoppt den Paketgenerator	Generator gestartet	-
STATUS	Gibt Status des Generators zurück (RUNNING/STOPPED)	-	-
SETDATALEN x	Setzt Nutzdatengröße in Byte	Generator gestoppt	$0 \leq x \leq 1430$
SETSRCIP a.b.c.d SETDESTIP a.b.c.d	Setzt Quell-IP Setzt Ziel-IP	Generator gestoppt	$0 \leq a \leq 255$ $0 \leq b \leq 255$ $0 \leq c \leq 255$ $0 \leq d \leq 255$
SETTTL x	Setzt Time to live (L3-Header)	Generator gestoppt	$0 < x \leq 255$
SETSRCPORT x SETDESTPORT x	Setzt Quell-Port (L4-Header) Setzt Ziel-Port (L4-Header)	Generator gestoppt	$0 < x \leq 65535$ $0 < x \leq 65535$
SETPTYPE x	Setzt Pakettyp	Generator gestoppt	Werte für x: IP/TCP/UDP
SETDTYPE x	Setzt Nutzdatentyp	Generator gestoppt	Werte für x: ONES/ ZEROES/RANDOM/ALT TIMESTAMP/BENCH/TAG
SETTOS x	Setzt Type of Service (*IPv4*-Header)	Generator gestoppt	$0 < x \leq 255$
SETSYN x, SETACK x SETFIN x, SETURG x SETPSH x, SETRST x	Setzt Flag (*TCP*-Header)	Generator gestoppt	$x = 0/1$
SETRUNLENGTH_MS x	Setzt Generatorlaufzeit (ms pro Durchlauf)	Generator gestoppt	$x > 0$, inf. wenn $x = 0$
SETNRUNS x	Setzt Anzahl Durchläufe	Generator gestoppt	$0 < x \leq 10000$
SETTPAUSE_MS x	Setzt Pause zw. Durchläufen (ms)	Generator gestoppt	$0 < x \leq 100000$
SETDATARATE x	Setzt Datenrate (in kBit/s)	Generator gestoppt	$0 < x \leq 1024000$

C. Steuerbefehle Analysator

Tabelle C.1.: Steuerbefehle für den Analysator (1)

Befehl	Funktion	Bedingung	Argument(e)
START	Startet Analysator	Analysator gestoppt	
STOP	Stoppt Analysator	Analysator gestartet	
SETIFACE x	Setzt Netzwerkinterface	Analysator gestoppt	x $\rightarrow$ Interfacename Default: *eth0*
TOFILE x	(de)aktiviert Datenexport in Binärdatei (/BIN)	Analysator gestoppt	$x = 0/1$
SETNVALS x	Setzt Anzahl der Samples für Live-Plot	Analysator gestoppt	$0 < x \leq 10^6$
SETSAMPLEPERIOD_MS x	Setzt Sampleperiode in ms	Analysator gestoppt	$1 \leq x \leq 500$
SETNUMBINFILES x	Setzt Anzahl Binärdateien	Analysator gestoppt	$x > 0$
SETBINFILESZ_MB x	Setzt maximale Größe einer Binärdatei in MB	Analysator gestoppt	$x > 0$
PLOT x	(de)aktiviert Live-Plot	Analysator gestoppt	$x = 0/1$
PCAPREC x	(de)aktiviert Aufzeichnung im PCap-Format (/PCAP)	Analysator gestoppt	$x = 0/1$
SET_OP_TSTART_SEC x	Setzt Beginnzeit in s für Offlineplot	-	$x > 0$
SET_OP_TEND_SEC x	Setzt Endzeit in s für Offlineplot	-	$x > 0$
OFFLINEPLOT	Plottet Datenraten für $t_{start} \leq t \leq t_{end}$	TOFILE = 1	-
SET_ANALYSIS_LAYER x	Setzt Analyse-Schicht	Analysator gestoppt	$x = 2/3$
PROMISC x	(de)aktiviert Promiscuous Mode	Analysator gestoppt	$x = 0/1$

Tabelle C.2.: Steuerbefehle für den Analysator (2)

Befehl	Funktion	Bedingung	Argument(e)
SET_IAT_PLOT_TSTART_US x	Setzt Startzeit in μs für IAT-PLot	$x \geq 0$	-
SET_IAT_PLOT_TEND_US x	Setzt Endzeit in μs für IAT-PLot	$x > 0$	-
SET_IAT_PLOT_BIN_WIDTH_US x	Setzt Klassenbreite in μs für IAT-Analyse	$x \geq 1$	-
SET_IAT_PLOT_N_BINS_US x	Setzt Anzahl der Klassen für IAT-Analyse	$x \geq 1$	-
SET_IAT_PLOT_X_LOG x	(de)aktiviert logarithmische x-Achseneinteilung für IAT-Plot	-	$x = 0/1$
PLOT_IAT_DIST	Plottet Häufigkeitsverteilung für Zwischenankunftszeit	TOFILE = 1 IAT_ANALYSIS = 1	-
IAT_ANALYSIS x	aktiviert Analyse der Zwischenankunftszeit	Analysator gestoppt	$x = 0/1$
RESET_IAT_DIST	startet Messwertaufnahme für Häufigkeitsverteilung Zwischenankunftszeit neu	-	-
SET_RT_PLOT_TSTART_MS x	Setzt Startzeit ms für Plot Paketlaufzeit	-	$x \geq 0$
SET_RT_PLOT_TEND_MS x	Setzt Endzeit in ms für Plot Paketlaufzeit	-	$x > 0$
RT_ANALYSIS x	(de)aktiviert Analyse der Paketlaufzeit	Analysator gestoppt	$x = 0/1$
PLOT_RT_DIST	Plottet Häufigkeitsverteilung für Paketlaufzeit	TOFILE = 1 IAT_ANALYSIS = 1	-
SET_CSV_TSTART_SEC x	Setzt Startzeit für CSV-Export	-	$x \geq 0$
SET_CSV_TEND_SEC x	Setzt Endzeit für CSV-Export	-	$x > 0$
TO_CSV	Startet CSV-Export in Verzeichnis /CSV	TOFILE = 1	-

Tabelle C.3.: Steuerbefehle für den Analysator (3)

Befehl	Funktion	Bedingung	Argument(e)
SETPLOTREFRESHRATE_HZ x	Setzt die Aktualisierungsfrequenz für den Live-Plot (default: $f_{plot} = 10Hz$)	Analysator gestoppt	$0 < x \leq 100$
PCAP_SETNUMFILES x	Setzt Anzahl Dateien durch die rotiert wird	Analysator gestoppt	$0 < x \leq 1024$
PCAP_SETFILEROTATE_SEC x	Rotation zur nächsten PCap-Datei nach x Sekunden	Analysator gestoppt	$0 < x \leq 86400$
PCAPSETSNAPLEN_BYTE x	Anzahl Byte, die von jedem Paket gespeichert werden	Analysator gestoppt	$0 < x \leq 65535$
PCAPSETTIMEOUT_MS x	Setzt Timeout für PCap-Puffer (Zeit in der Pakete gepuffert werden, bevor sie in eine Datei geschrieben werden)	Analysator gestoppt	$10 < x \leq 6000$
PCAP_SETMAXSPACE_MB x	Setzt Maximalwert für den durch PCap-Dateien belegten Speicher	Analysator gestoppt	$0 < x \leq 8.388.608$ (8 TB)
BENCH x	(de)aktiviert Benchmark-Modus	Analysator gestoppt	x = 0/1
BUF_OVERFLOW_WARNING x	(de)aktiviert Warnung bei Paketverlust (derzeit Intel-NIC benötigt)	Analysator gestoppt	$x = 0/1$
SET_RX_OVERFLOW_FIELD x	Setzt Feld aus Ethtool-Struktur das verworfene Pakete enthält (ethtool -S *interface*)	Analysator $x \geq 0$ gestoppt	

D. NTP-Konfigurationsdatei

Listing D.1: NTP-Konfigurationsdatei /etc/ntp.conf

```
1  #///etc/ntp.conf
2
3  #// path for drift file
4  driftfile /var/lib/ntp/drift/ntp.drift
5
6  #// path for log file
7  logfile /var/log/ntp
8
9  #// path for keys file
10 keys /etc/ntp.keys
11
12 #// define trusted keys
13 trustedkey 1
14 requestkey 1
15 controlkey 1
16
17 #// servers
18 server ptbtime1.ptb.de
19 server ptbtime2.ptb.de
20 server ntp0.fau.de
21 server ntp2.fau.de
```

E. Hilfsmittel

- Erstellen eines Packet-Sockets und Versetzen eines Netzwerk-Interface in den Promiscuous Mode (im Quelltext kenntlich gemacht):
  http://stackoverflow.com/questions/114804/reading-from-a-promiscuous-network-device
- Umgang mit ioctl() und Ethtool, Beispielcode:
  http://stackoverflow.com/questions/16423249/ioctlsock-siocethtool-ifr-why-its-returning-1-all-the-time
- Studienarbeit RAW-Socket-Programmierung mit Beispielen:
  https://kola.opus.hbz-nrw.de/files/245/studienarbeit_oneside.pdf
- Netzwerkprogrammierung mit BSD-Sockets:
  www.zotteljedi.de/socket-buch/socket-buch.pdf
- Packet Sniffer Code in C using Linux Sockets (BSD) – Part 2:
  http://www.binarytides.com/packet-sniffer-code-in-c-using-linux-sockets-bsd-part-2/
- Send a raw Ethernet frame in Linux:
  https://gist.github.com/austinmarton/1922600
- Capturing Packet Data and Saving It to a File for Processing Later, IBM Knowledge Center:
  https://www.ibm.com/support/knowledgecenter/ssw_aix_61/com.ibm.aix.progcomc/libpcap_pcap2.htm
- pcap_dump, IBM Knowledge Center:
  https://www.ibm.com/support/knowledgecenter/ssw_aix_61/com.ibm.aix.basetrf1/pcap_dump.htm
- RAW ethernet programming:
  http://aschauf.landshut.org/fh/linux/udp_vs_raw/ch01s03.html
- Programming with pcap:
  http://www.tcpdump.org/pcap.html

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 21. September 2016	
Ort, Datum	Unterschrift