ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Trace Control for Netem

Porting the Trace Based Network Emulator RplTrc to the Linux Kernel 2.6 Network Emulator Netem

Ariane Keller

Semester Thesis SA-2006-15
April 2006 to July 2006

Advisor: Rainer Baumann, Dr. Ulrich Fiedler
Supervisor: Prof. B. Plattner

Zusammenfassung

Das Testen neuer Netzerkprotokolle wird mit dem Wachstum und der Vielfältigkeit von Netzwerken stetig komplexer. Mit Hilfe von Netzwerkemulatoren ist es möglich, in einem einfachen Versuchsaufbau die Protokolle reproduzierbar zu beurteilen.

Die kostengünstigsten Emulatoren sind Open-Source-Emulatoren, welche auf einem Linux-Betriebssystem implementiert sind. Sie sind in der Lage, Paketverzögerungen, -verluste und -duplikationen zu emulieren, jedoch werden statistische Eigenschaften wie long-range dependency und self-similarity of cross-traffic nicht berücksichtigt. Um auch diese Eigenschaften zu emulieren, wurde am Institut für Technische Informatik der ETH Zürich der Netzwerkemulator RplTrc entwickelt. Bei RplTrc werden vor der Emulation Messungen an einem Netzwerk durchgeführt und alle Werte für Paketverzögerungen, -verluste und -duplikationen in einem Tracefile registriert. Bei der Emulation werden die Pakete gemäss den Anweisungen eines solchen Tracefiles bearbeitet.

RplTrc ist für die alte Linux-Kernel-Version 2.4 implementiert. In der Kernel-Version 2.6 ist standardmässig ein Netzwerkemulator (Netem) vorhanden. In dieser Semesterarbeit wurde der Netzwerkemulator Netem um die Eigenschaften von RplTrc erweitert. Unser Ziel war die Aufnahme in den Kernel-Source-Code. Damit haben wir einen weit verbreiteten Netzwerkemulator entwickelt, der stets von den neusten Erweiterungen vom Linux-Kernel profitiert und welcher Netzwerkeigenschaften wie long-range dependency und self-similarity of cross-traffic korrekt emuliert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Netzwerke, wie zum Beispiel das Internet, haben verschiedene Charakteristiken. Um neue Netzwerkprotokolle, wie zum Beispiel für Voice-Over-IP, zu testen, sind vor allem die Paketverzögerung zwischen Ursprungs- und Zielort, der Verlust und das Duplizieren von Paketen von Interesse. All diese Effekte variieren stark mit der Auslastung des Internets: bei einer starken Netzwerkauslastung sind die Router nicht mehr fähig, alle Pakete korrekt zu behandeln. Diese Dynamik sowie die Grösse des Internets erschweren es Netzwerkprotokolle zu testen. Zudem ist es kaum möglich, reproduzierbare Ergebnisse zu erhalten.

Netzwerkemulation, das heisst die Reproduktion der Netzwerkdynamik in einer "Box" wie in Abbildung 1.1 dargestellt, ermöglicht reproduzierbare Studien mit einem einfachen Versuchsaufbau. Häufig reicht die Leistungsfähigkeit käuflicher PCs aus, um eine Netzwerkemulation ohne Spezialhardware zu implementieren.

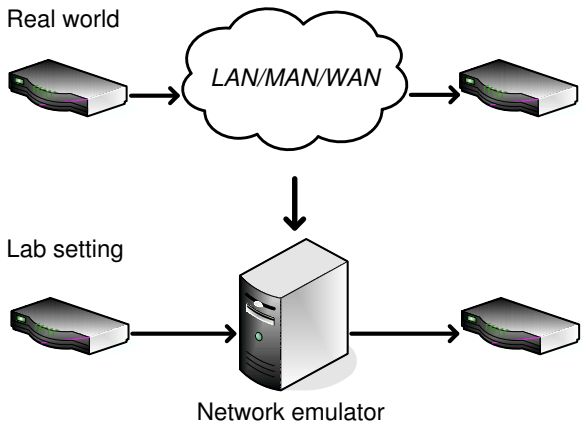

Abbildung 1.1: Network Environment

Dabei muss allerdings folgendes beachtet werden:

- Die Paketverarbeitung sollte in Echtzeit geschehen und
- Eigenschaften der Netzwerkdynamik wie long-range dependency [1] müssen korrekt modelliert werden.

Paketverarbeitung in Echtzeit ist nur mit einem Realtime-Betriebssystem möglich. Die Implementation eines Netzwerkemulators für ein Realtime-Betriebs-System ist ziemlich aufwändig. Zusätzlich ist die Installation und Bedienung eines Realtime-Betriebssystems für "Laien" schwierig. Deshalb entscheiden sich viele Programmierer ihren Netzwerkemulator im Linux Kernelspace zu implementieren.

Schwierig zu modellieren ist vor allem die long-range dependency der Verzögerungen der einzelnen Pakete. In allen bekannten Emulatoren wie NIST Net [2] und Netem [3] ist diese Eigenschaft nicht modelliert. Diese Emulatoren lesen die Werte für Paketverzögerung, -duplikation und -verlust aus relativ kleinen statistischen Tabellen. Bei diesem Ansatz werden long-range dependencies nicht berücksichtigt [1].

Um auch long-range dependencies zu modellieren wird folgender Ansatz verwendet: Das Internet wird einige Zeit überwacht, dabei werden alle Werte für Verzögerungen etc. in einer Datei registriert. Neben echten Messungen können auch Werte aus Netzwerksimulationen oder Netzwerkberechnungen verwendet werden. Bei der Netzwerkemulation wird dieses "Tracefile" genommen und jedes Paket gemäss einem der registrierten Werte verzögert. Die so erhaltene Emulation weist die korrekten Merkmale der Internetdynamik zu einem bestimmten Zeitpunkt auf.

In der Masterarbeit von Thomas Hug [4] wurde dieser Ansatz für den Netzwerkemulator NIST Net programmiert. NIST Net ist ein Netzwerkemulator auf der Basis des Linux-Kernel 2.4. Die Trace-Erweiterung heisst RplTrc.

Im Linux-Kernel 2.6 ist der Netzwerkemulator Netem im Standard-Source-Code enthalten. Dies ist interessant, da der Linux-Kernel weit verbreitet ist und die weitere Entwicklung vom Linux-Kernel auf dieser Version vorangehen wird. Zudem ist es einfacher den Kernel für die Netzwerkemulation zu konfigurieren, als den NIST Net Emulator zu installieren. Darum wurde in dieser Semesterarbeit die NIST Net Trace-Erweiterung von Thomas Hug auf den Netzwerkemulator des Linux-Kernel 2.6 portiert. Dabei wurde auf ein sauberes Design Wert gelegt, welches möglichst wenig Kommunikation zwischen den einzelnen Komponenten benötigt. Es wurde auch darauf geachtet, dass mit fehlerhaft eingegebenen Kommandos korrekt umgegangen wird. Im Vergleich zur NIST Net Trace-Erweiterung besteht für den Benutzer eine grössere Freiheit bei der Konfiguration.

Um künftigen Netem-Benutzern die Netzwerkemulation zu erleichtern wurde ein Manual geschrieben. Dieses umfasst sowohl Netem mit seiner Trace-Erweiterung als auch das Linux traffic control tool tc, welches für die Konfiguration von Netem benötigt wird. Bis anhin gab es dafür nur eine sehr umfangreiche, aber dennoch unvollständige Dokumentation.

Diese Arbeit ist folgendermassen aufgebaut: Kapitel 2 gibt einen Überblick über das Linux Paket handling und über die Netzwerkemulatoren NIST Net, RplTrc und Netem. Die Funktionsweise von Netem ist in Kapitel 3 genauer beschrieben. In Kapitel 4 wird die in dieser Arbeit entwickelte Netem Trace-Erweiterung TCN behandelt. Die Evaluation dieser Trace-Erweiterung ist in Kapitel 5 zusammengefasst. Das Manual zu den Themen "tc packet filtering" und "netem configuration" ist im Anhang zu finden.

Kapitel 2

Ausgangspunkt: Linux, NIST Net und Netem

Dieses Kapitel beschreibt für diese Arbeit wichtige, bereits vorhandene Software. Im ersten Teil wird kurz auf das Linux Paket handling eingegangen. Dabei wird das Konzept der Qdiscs (queueing disciplines) eingeführt, welches vom Netzwerkemulator Netem verwendet wird. Danach werden zwei Netzwerkemulatoren vorgestellt: Zuerst wird auf den Netzwerkemulator NIST Net mit seiner Trace-Erweiterung RplTrc eingegangen, danach wird die Netzwerkemulation vom Kernel 2.6 mit dem Linux iproute2 Paket und Netem behandelt.

2.1 Linux Paket handling

Das Betriebssystem Linux eignet sich sehr gut als Basis zur Netzwerkemulation, da

- es sich als Router konfigurieren lässt und
- sich Erweiterungen einfach einbinden lassen.

Ist ein Linuxrechner als Router konfiguriert, verarbeitet er Pakete nach Abbildung 2.1.

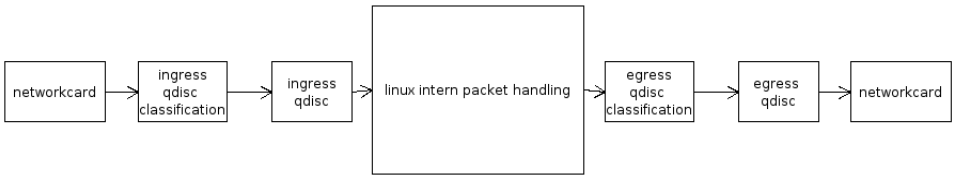

Abbildung 2.1: Linux-Kernel Packet Handling

Im ersten Schritt werden Pakete, die von anderen Computern gesendet werden, über eine Netzwerkkarte empfangen und auf Warteschlangen (englisch: queueing discipline, oder kurz Qdisc) aufgeteilt. Abbildung 2.2 enthält eine Definition vom Begriff Qdisc.

Im zweiten Schritt wird entschieden, ob dieser Computer das Ziel ist oder ob das Paket zu einem anderen Computer weitergeleitet werden muss. Im letzten Fall entscheidet der Linux-Kernel über das zu verwendende Netzwerkinterface und leitet das Paket an die entsprechende ausgehende Warteschlange (englisch: egress qdisc) des Netzwerkinterfaces weiter.

Im dritten und letzten Schritt, wird das Paket von der Warteschlange über das spezifizierte Netzwerkinterface weiter gesendet.

Normalerweise gibt es jeweils eine Warteschlange (Qdisc) pro Interface für einkommende (ingress) und eine für ausgehende (egress) Pakete. Diese Standard-Qdisc verarbeitet die Pakete nach dem FIFO (First In First Out) Prinzip. Es ist aber auch möglich, den Linux-Kernel so zu konfigurieren, dass er mehrere Qdiscs pro Interface erlaubt und dass diese die Pakete gemäss eigenen Policen abarbeiten. Aus Sicht des Kernels braucht es aber immer einen einzigen Ansprechpartner, die so genannte root Qdisc. Der Aufbau einer solchen Hierarchie ist in Abbildung 2.3 gezeigt. Pakete können mit Hilfe des Linux traffic control tool tc einer dieser Qdiscs zugewiesen werden. Das Filtern der Pakete zu der gewünschten Qdisc ist nach fast beliebigen Kriterien möglich. Im Anhang D sind einige Möglichkeiten zur tc Paketfilterung beschrieben. Der Funktionsweise von tc ist Abschnitt 3.1 gewidmet.

Definition Qdisc:
Eine Qdisc besteht aus zwei Teilen:

- einem Algorithmus der entscheidet wann welches Paket gesendet werden soll und
- einer Warteschlange, in welche die Pakete eingereiht werden, bis sie gesendet werden.

es gibt zwei Arten von Qdiscs:

- Qdiscs die Pakete an ein Netzwerkinterface senden und
- Qdiscs die Pakete in verschiedene Klassen aufteilen können. Diesen Klassen können dann wieder Qdiscs zugeordnet werden.

Abbildung 2.2: Definition Qdisc

2.2 Linux-Kernel 2.4

2.2.1 Netzwerkemulator NIST Net

NIST Net wurde entwickelt um eine kontrollierbare und reproduzierbare Umgebung zu schaffen, in welcher auf dem Internet Protokoll basierende Applikationen getestet werden können, ohne dafür ein komplexes Netzwerk zu benötigen. NIST Net ermöglicht das Nachbilden von Eigenschaften des Internets wie Paketverzögerung, Paketverlust und Paketduplikation. Die einzelnen Werte werden aus statistischen Tabellen gelesen.

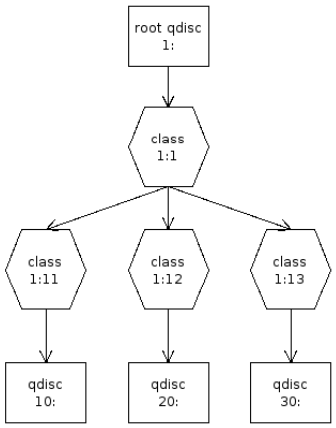

Abbildung 2.3: Qdisc Hierarchy

NIST Net besteht aus einem Userinterface und einem Linux Kernelmodul, das bei Bedarf in den Kernel geladen wird. Es ist jedoch nicht Bestandteil des offiziellen Kernel Source-Codes. NIST Net ermöglicht das Spezifizieren von einzelnen Flows [1]. Jeder Flow kann nach einer eigenen Verzögerungs-, Verlust- und Verdoppelungsstatistik bearbeitet werden. Um eine möglichst gute zeitliche Auflösung zu erhalten, wurde ein spezieller Timer mit einer Auflösung von 122μs verwendet.

2.2.2 NIST Net Trace-Erweiterung RplTrc

Um auch echt gemessene (oder mit einem Netzwerksimulator erzeugte) Paketverzögerungen, -verluste und -duplikationen an einen Flow anlegen zu können, wurde von Thomas Hug in seiner Masterarbeit [4] eine Erweiterung zu NIST Net erarbeitet. Vor der eigentlichen Netzwerkemulation werden die gemessenen Aktionswerte für Verzögerung, Verlust und Duplikation in so genannte Tracefiles gespeichert. Um auch long-range dependencies in Netzwerkdynamiken zu erfassen, was beim ursprünglichen NIST Net nicht möglich war, können diese Tracefiles beliebig lang sein.

Während der Netzwerkemulation werden die Tracefiles von RplTrc abgespielt. Dies ermöglicht eine Emulation, die alle Netzwerkdynamikeigenschaften, insbesondere auch long-range dependency korrekt widerspiegelt.

Um auch mit dem Linux-Kernel 2.6 long-range dependencies emulieren zu können, ist das Ziel dieser Arbeit eine Portierung der Trace-Erweiterung von Thomas Hug zu Netem, dem Linux-Kernel 2.6 Netzwerkemulator.

[1] Flow: Pakete die die selben Source und Destination IP Adressen mit den entsprechenden Ports besitzen.

2.3 Linux-Kernel 2.6

2.3.1 Das Linux iproute2 Paket

Das iproute2 Paket besteht aus einer Sammlung von Werkzeugen um die Netzwerkeigenschaften des Linux-Kernels zu konfigurieren und oder zu überwachen. Es ermöglicht in den Bereichen Routen, Filtern und Klassifizieren von Internet-Paketen sehr viel. Von speziellem Interesse für den Netzwerkemulator Netem ist das tool tc (kurz für traffic control). TC ermöglicht das Filtern von Paketen in so genannte Klassen. Als Filterkriterien kommen etwa IP Header Felder oder auch Datenfelder eines Paketes in Frage. Ebenfalls werden die queueing disciplines, die an diese Klassen angehängt werden, mit Hilfe von tc konfiguriert. Mit dem Mechanismus aus Filter, Klassen und Qdiscs lassen sich Internet-Pakete kategorisieren und je nach Kategorie verschieden behandeln. Es können beispielsweise alle Pakete mit IP-Source-Adresse A von einer anderen Instanz von Netem bearbeitet werden als solche mit IP-Source-Adresse B.

2.3.2 Netzwerkemulator Netem

Netem ist ein Linux-Kernelmodul, welches zu Netzwerkemulationen benutzt wird. Es ist in jedem Linux Betriebssystem ab der Kernelversion 2.6.7 standardmässig enthalten. Netem ist als Warteschlangendisziplin (Qdisc) implementiert, die von tc konfiguriert werden muss. Ähnlich zu NIST Net ermöglicht es das Verzögern, Verwerfen und Duplizieren von Internet-Paketen. Zusätzlich ist es möglich, gezielt Bitfehler in einzelne Pakete einzufügen. Das Modellieren von Bitfehlern ist vor allem für Tests von Wireless-Protokollen von Interesse, da dort die Bitfehlerrate relativ hoch ist. Da die Aktionswerte, wie bei NIST Net auch, aus statistischen Tabellen gelesen werden, ist die Modulation von long-range dependency nicht möglich. Deshalb wird in dieser Arbeit eine Erweiterung, ähnlich derer für NIST Net, implementiert, die es erlaubt, beliebig lange Tracefiles als Quelle für die Aktionswerte zu nehmen.

Kapitel 3

traffic control tool tc und Linux Netzwerkemulator Netem

In dieser Semesterarbeit wird die Netem Qdisc erweitert. Da Netem mit Hilfe des Linux traffic control tool tc konfiguriert wird, wird hier der Aufbau von tc kurz vorgestellt. Eine ausführliche Darstellung geht aber über den Rahmen dieser Arbeit hinaus. Im zweiten Teil dieses Kapitels wird der Aufbau von Netem beschrieben.

3.1 Traffic control tool tc

Obwohl es mit tc möglich ist, auf der einkommenden (ingress) Seite Pakete zu verarbeiten, wird normalerweise nur auf der ausgehenden (egress) Seite gearbeitet (siehe Abbildung 1.1). Wurde vom Linux-Kernel entschieden, über welches Netzwerkinterface, und somit über welche root Qdisc, ein Paket weitergeleitet werden soll, besteht die Möglichkeit, mit tc Filter diese Pakete nach fast beliebigen Kriterien in Klassen einzuordnen. Es können ganze Klassenbäume konstruiert werden. Um diese Pakete je nach Klasse anders zu behandeln, wird an den Blattklassen eine Qdisc angehängt. Die verschiedenen Qdiscs ermöglichen fast beliebiges Verändern des Paketstroms. Eine Anleitung zu tc mit Beispielen kann im Anhang D gefunden werden.

Da tc "nur" ein Konfigurationstool für das Paket handling im Kernel ist, besitzt es keinerlei Information über die aktuell vorhandenen Klassen, Qdiscs und Filter. Diese werden im Kernel registriert. Ob ein tc Kommando gültig ist, kann erst im Kernel entschieden werden, da dies von den bereits vorhandenen Klassen und Qdiscs abhängt.

Als Kommunikationsmittel zwischen Kernel und Userspace werden Netlinksockets verwendet. Diese Sockets sind in verschiedene Familien unterteilt. Tc verwendet die NETLINK_ROUTE Familie, die unter anderem zur Kontrolle der queueing disciplines und der Traffic-Klassen verwendet werden kann. Mehr Informationen über Netlinksockets können den Manpages netlink(3) und netlink(7) entnommen werden.

Der tc Source-Code ist so aufgebaut, dass beliebig viele neue Qdisc-, Filter- oder Klassentypen hinzugefügt werden können, ohne dass am eigentlichen tc Source-Code etwas geändert werden muss. Dies wird dadurch erreicht, dass alle Qdiscs als Kernelmodule implementiert sein müssen. Jedes Mal, wenn sie von tc geladen werden, registrieren sie sich in einer Liste. Danach kann durch Vergleichen der Commandline-Argumente mit dem Inhalt dieser Liste die Kontrolle an das entsprechende Modul übergeben werden.

3.1.1 Userinterface

Das Userinterface von tc wird zum Parsen der einzelnen Argumente verwendet. Es analysiert das eingegebene Kommando in vier Schritten (siehe Abbildung 3.1).

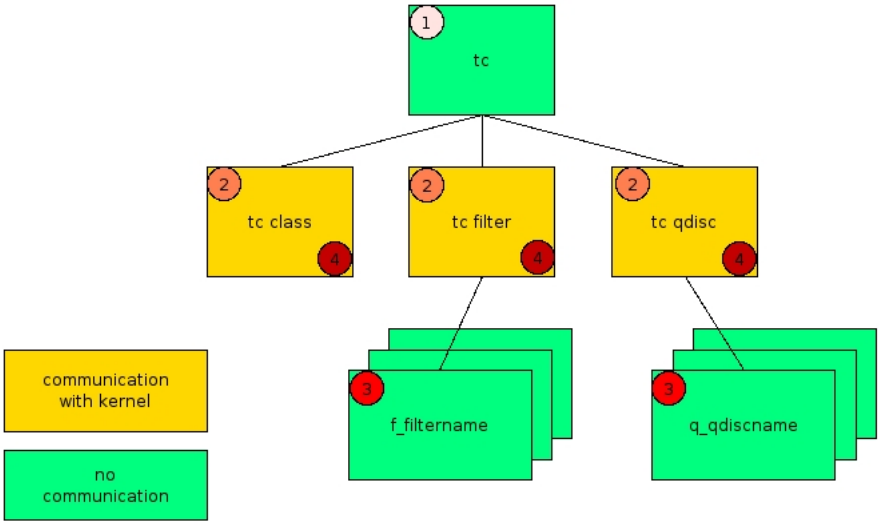

Abbildung 3.1: Structure of tc's Userinterface

1. Es wird festgestellt, ob eine Qdisc, ein Filter oder eine Klasse bearbeitet werden soll.
2. Dann werden die allgemeinen Parameter für dieses Gebiet überprüft.
3. Danach werden die spezifischen Parameter für die angegebene Qdisc oder den angegebenen Filter überprüft.
4. Schlussendlich werden alle Parameter dem Kernel übergeben, welcher überprüft, ob das Kommando ausgeführt werden kann.

Dies sei am Beispiel einer Erstellung einer Netem Qdisc näher erklärt. Das Anlegen einer Netem Qdisc am Ethernetinterface eth0, die jedes Paket um 100ms verzögert, erfolgt durch folgendes Kommando:
`tc qdisc add dev eth0 root netem delay 100ms`.

1. `tc qdisc`: Vom tc main Programm wird festgestellt, dass als zweites Argument "qdisc" angegeben wurde. Dies führt zum Aufruf des allgemeinen Parsingprogramms für Qdiscs.
2. `add dev eth0 root netem`: Das Qdisc-Parsingprogramm stellt fest, dass eine neue root Qdisc am Netzwerkinterface eth0 angelegt werden soll. Es stellt fest, dass eine Library namens q_netem.so existiert und linkt diese dynamisch dazu. Danach ruft es das jetzt bekannte Netem-Parsingprogramm auf.
3. `delay 100ms`: Das Netem-Parsingprogramm überprüft die verbleibenden Parameter auf ihre Gültigkeit. Danach beendet es sich, und die Kontrolle wird wieder an das allgemeine Qdisc-Parsingprogramm übergeben. Zu diesem Zeitpunkt ist noch nicht entschieden, ob das angegebene Kommando gültig ist oder nicht.
4. Das Qdisc-Parsingprogramm sendet alle geparsten Argumente über einen Netlinksocket an den Kernel. Dieser überprüft, ob das angegebene Interface und die angegebenen Klassen- oder Qdiscidentifier korrekt sind und gibt eine entsprechende Meldung zurück. Nun beendet sich das Qdisc-Parsingprogramm und das tc main Programm. Im Kernel wird das angegebene Kommando ausgeführt.

3.2 Netem

Netem besteht aus zwei Teilen:

- einem Kernelmodul welches eine Qdisc implementiert und
- einer Erweiterung zum tc-Userinterface, mit welcher das Netem Kernelmodul konfiguriert wird.

3.2.1 Userinterface

Das Netem-Userinterface parst die spezifischen Netem-Argumente. Es überprüft, ob alle angegebenen Argumente gültig sind, und setzt Defaultwerte für nicht angegebene Argumente. Da die Zeitmessung in Linux auf der Basis von Ticks, dem Intervall zwischen zwei Timerinterrupts, erfolgt, konvertiert es die angegebene Delayzeit von μs, ms oder s in Ticks.

3.2.2 Kernelmodul

Initialisierung

Wird das Netem-Modul eingefügt, registriert es sich in einer Liste, in welcher der Kernel alle geladenen Qdiscs verwaltet. Der Eintrag in diese Liste besteht aus einem struct, der Pointer auf alle Funktionen enthält, die Netem gegen aussen anbietet. So kann der Kernel auf diese Funktionen zugreifen.

Beim Instantiieren einer Netem-Qdisc wird die netem_init Funktion aufgerufen. In ihr wird zuerst ein eigener Timer und eine interne Warteschlange initialisiert. Danach wird die netem_change Funktion aufgerufen. In ihr werden die Kommandoargumente, die über einen Netlinksocket vom Userinterface an den Kernel gesandt wurden, analysiert und in einem struct vom Typ "netem_sched_data" gespeichert. Sofern die Daten gültig sind, arbeitet Netem mit diesen neuen Daten, andernfalls behält es die alten Werte bei.

Paketverarbeitung

Abbildung 3.2 zeigt die Abarbeitung eines Paketes durch das Netem Kernelmodul. Zu beachten ist, dass Netem auch eine root Qdisc sein kann, und dann direkt vom Kernel angesprochen wird. Hat der Linux-Kernel entschieden über welches Interface er ein Paket weiter senden will, ruft er die enqueue Funktion der root Qdisc auf. Diese behandelt das Paket nach ihren Kriterien und gibt es, durch Aufrufen der entsprechenden enqueue Funktion, an eine allfällig in der Qdisc-Hierarchie weiter unten liegende Qdisc weiter.

Wird die Netem enqueue Funktion aufgerufen, wird das Paket wie folgt verarbeitet:

- Ein Zufallszahlengenerator wird verwendet, um einen konkreten Aktionswert aus einer Verteilungstabelle zu erhalten.
- Duplizierte Pakete werden in die root Qdisc eingefügt, während das Original normal weiter bearbeitet wird.
- Pakete, die verworfen werden, werden nicht mehr weiterverarbeitet und ihr Speicher wird freigegeben.
- Für Pakete, denen ein Bitfehler eingefügt werden soll, wird zuerst die IP-Checksumme berechnet, danach an einer zufälligen Stelle ein Bit verändert.
- Bei Verzögerungen wird zuerst die aktuelle Zeit abgefragt und ihr dann der Delaywert hinzugefügt. Danach wird dieser "Timestamp" dem Paket angehängt und das Paket in eine interne Warteschlange eingefügt. Diese Warteschlange sortiert ihre Einträge gemäss den Timestamps der einzelnen Pakete.

Die Netem dequeue Funktion:
Soll von Netem ein Paket zurückgegeben werden, wird das vorderste aus seiner internen Warteschlange geholt und überprüft, ob sein Timestamp in der Vergangenheit liegt. Falls ja, wird es der aufrufenden Funktion zurückgegeben, falls nicht, wird es wieder in die interne Warteschlange eingefügt und deren Timer auf die entsprechende Sendezeit eingestellt. Der aufrufenden Funktion wird in diesem Fall NULL zurückgegeben. Läuft der Timer ab, wird am entsprechenden Interface ein Interrupt ausgelöst, der mitteilt, dass ein Paket zum Senden vorhanden ist.

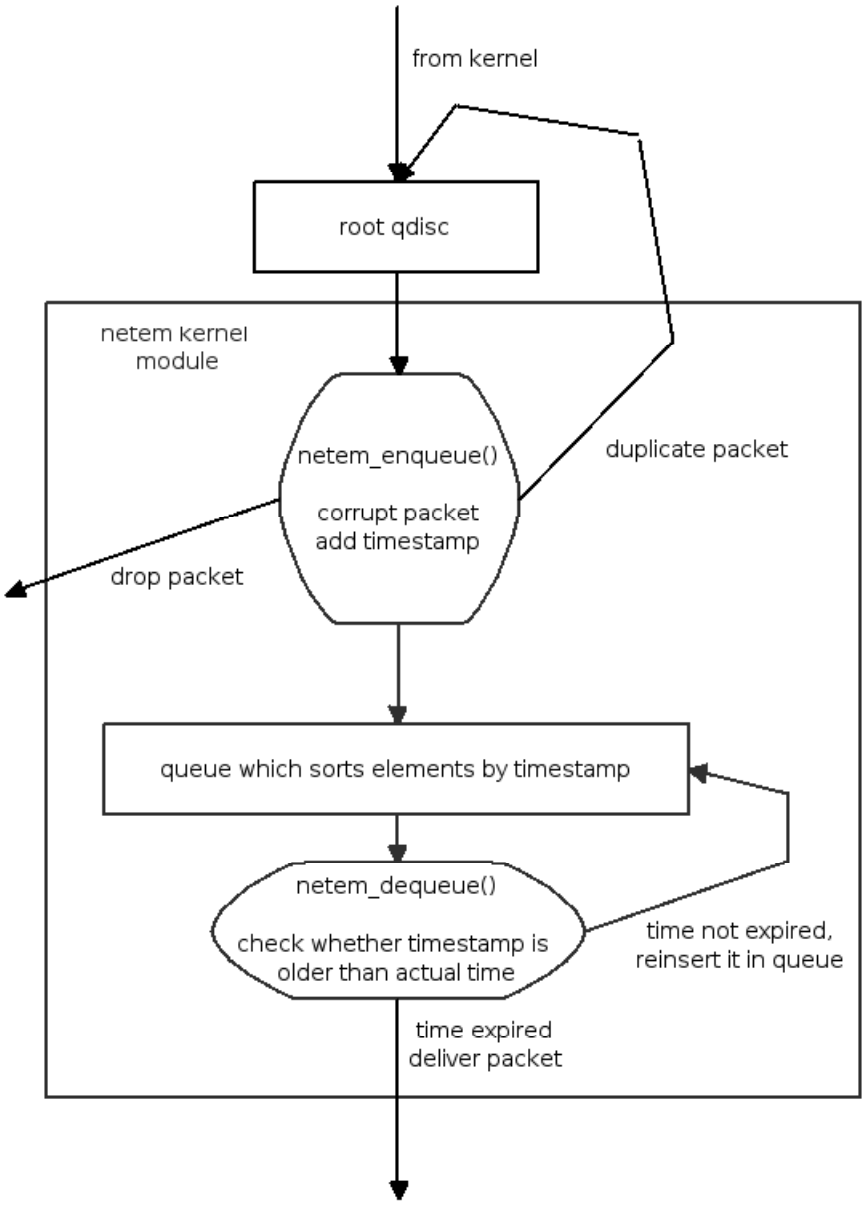

Abbildung 3.2: Netem Kernel Module

Kapitel 4

Die Netem Trace-Erweiterung

In dieser Semesterarbeit wurde das NIST Net Trace reader Modul RplTrc zu Netem portiert. Einige grundlegende Ideen und Konzepte für die Netem Trace-Erweiterung wurden von RplTrc übernommen.

Die Netem Trace-Erweiterung ist eine zusätzliche Ansteuerungsmöglichkeit für Netem. Es bietet die Möglichkeit für jedes Paket, das bearbeitet wird eine der Aktionen Verzögerung, Duplikation, Verwerfen oder Einfügen eines Bitfehlers einzeln zu spezifizieren. Diese Aktionen werden, in 32bit Werten kodiert, vorgängig in eine Datei geschrieben. Diese Datei nennt sich Tracefile. Die Generation von Tracefilen ist im Anhang D besprochen. Während der Netzwerkemulation werden nun diese Werte aus dem Tracefile gelesen, und die Pakete gemäss diesen Aktionen abgearbeitet.

Da im Kernel keine zeitaufwändigen Operationen ausgeführt werden dürfen (wie dies das Auslesen von Daten aus einem File ist), werden die Daten aus dem Tracefile im Userspace ausgelesen und danach in Portionen von 1'000 Werten dem Kernel übergeben. Dies wird vom Flowseedprozess gemacht.

Neben dem Flowseedprozess besteht die Netem Trace-Erweiterung aus einem erweiterten Userinterface, welches den Flowseedprozess startet, und einem erweiterten Kernelmodul, welcher die Aktionswerte des Flowseedprozesses entgegen nimmt und die einzelnen Aktionen den Paketen zuordnet. Abbildung 4.1 zeigt die grobe Struktur der Netem Trace-Erweiterung.

4.1 Erweiterungen im Netem Userinterface

Das Netem-Userinterface überprüft neu auch die Argumente der Trace-Erweiterung. Dies sind:

- Tracefilename
- Anzahl Wiederholungen des gesamten Tracefiles. Dies wird benötigt, um auch sehr lange Simulationen zu ermöglichen.
- Defaultwert, der angibt was mit einem Paket gemacht werden soll, falls kein Wert aus dem Tracefile vorhanden ist.

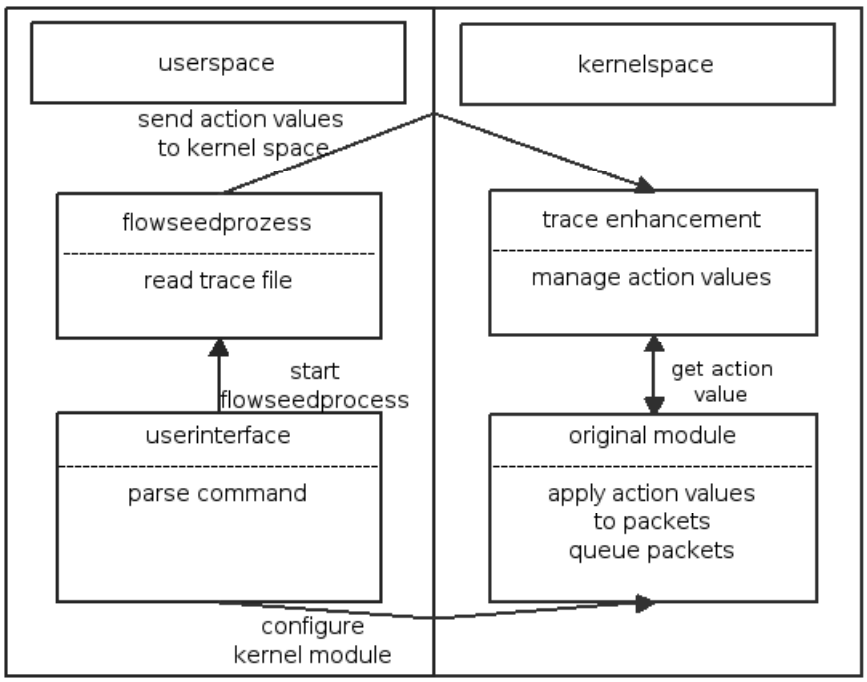

Abbildung 4.1: Enhanced Netem Overview

Wurde ein gültiger Tracefilename angegeben, startet das Netem-Userinterface den Flowseedprozess für das angegebene File. Für die spätere korrekte Zuordnung zwischen Qdisc und Flowseedprozess wird dem Linux-Kernel die Prozess ID des soeben generierten Flowseedprozesses mitgegeben.

Da die Delaywerte in den Tracefilen in μs angegeben sind, muss der korrekte Konvertierungsfaktor von μs nach Ticks ermittelt und dem Kernel übergeben werden. Dies wird mit Hilfe einer von Netem zur Verfügung gestellten Funktion gemacht.

Ebenfalls wird dem Kernel der Defaultwert, der angibt was mit einem Paket gemacht werden soll, falls kein Wert aus dem Tracefile vorhanden ist, mitgegeben.

4.2 Erweiterungen im Netem Kernelmodul

Das Kernelmodul besteht aus dem ursprünglichen Netem-Teil, welcher für das Verwerfen, Verzögern, Duplizieren und Einfügen von Bitfehlern zuständig ist, und der Trace-Erweiterung, die die Flowseedpakete empfängt und die entsprechenden Aktionswerte für das jeweilige Internet-Paket angibt.

4.2.1 Kommunikation zwischen Netem Kernelmodul und Flowseedprozess

In Abbildung 4.2 ist die Netem Trace-Erweiterung schematisch dargestellt.

Das Netem Kernelmodul benötigt, für jedes Paket das bearbeitet werden soll, einen Aktionswert, der bestimmt, wie mit dem Paket verfahren werden soll. Diese Werte werden vom Flowseedprozess im Userspace aus dem Tracefile ausgelesen.

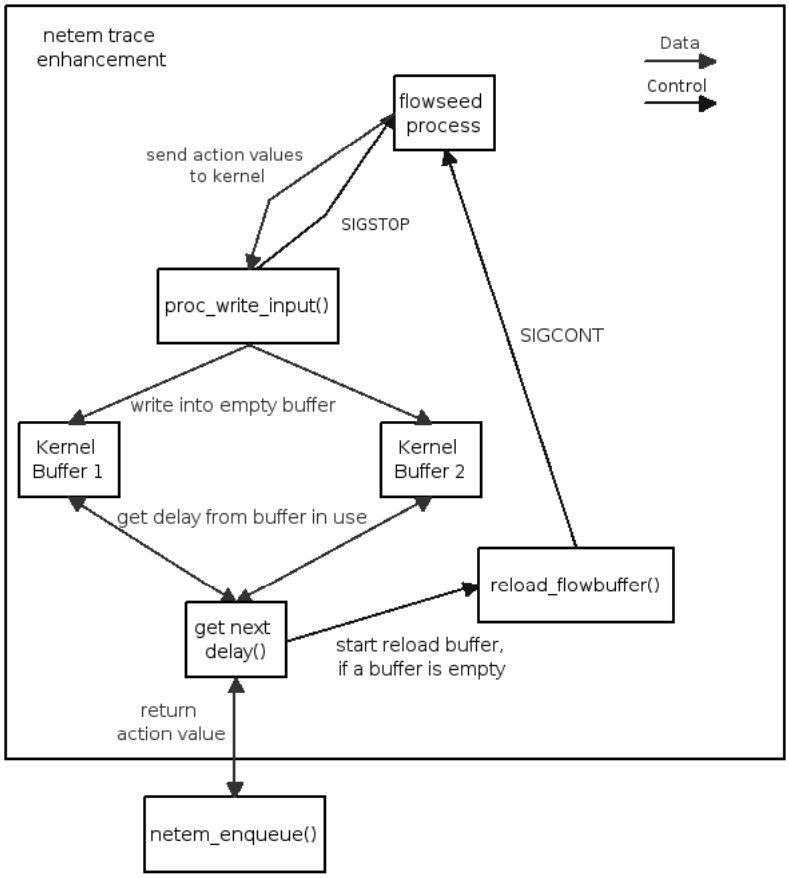

Abbildung 4.2: Netem Trace Enhancement

Um Daten vom Userspace zum Kernelspace zu transferieren, gibt es verschiedene Möglichkeiten. Wie Tests bei der Implementation von RplTrc ergaben, ist für diese Anwendung das Process File System (procfs) am geeignetsten [4, Seite 15]. Vom Standpunkt des Flowseedprozesses aus werden ununterbrochen Aktionswerte aus dem Tracefile gelesen und in Portionen zu 1'000 Werten dem Kernel übermittelt. Um nicht mit Daten überhäuft zu werden, sendet dieser dem Flowseedprozess ein SIGSTOP Signal, sobald er die Daten bekommen und in einen Buffer geschrieben hat. Da die procfs-write Operation des Flowseedprozesses "blocking" ist, und das SIGSTOP Signal vor dem return von write gesendet wird, legt sich der Flowseedprozess schlafen, bevor er weitere Daten aus dem Tracefile lesen kann.

Bei jedem Internet-Paket, das von der Netem Trace-Erweiterung bearbeitet wird, ruft die netem_enqueue Funktion die get_next_delay Funktion auf. Diese

liest einen Wert aus dem Buffer und gibt ihn an die enqueue Funktion zurück. Das Paket wird jetzt gemäss den Angaben dieses Wertes weiterverarbeitet.

Wird aus einem Kernelbuffer der letzte Aktionsswert gelesen, wird dem Flowseedprozess ein SIGCONT Signal gesendet, woraufhin dieser mit dem Auslesen der Daten aus dem Tracefile weiterfährt.

Damit die Netem Trace-Erweiterung ständig Pakete bearbeiten kann, müssen gleichzeitig Aktionswerte aus einem Buffer gelesen werden und neue Aktionswerte vom Tracefile in das Kernelmodul hinein gelesen werden. Dies wird mit zwei Buffern ermöglicht. Während aus dem einen Buffer Daten gelesen werden, werden in den anderen neue Daten geschrieben. Am Anfang werden beide gefüllt, und dann mit dem Lesen der Verzögerungswerte aus dem ersten Buffer begonnen. Sobald dieser leer ist, wird zum zweiten Buffer gewechselt und dort weiter gelesen. Gleichzeitig wird der erste Buffer wieder aufgefüllt. So kann beim Ende von Buffer zwei wieder beim Anfang von Buffer eins begonnen werden.

Im Netem Kernelmodul wird für jede angelegte Qdisc ein struct vom Typ "_flowbuffer" zur Verwaltung der beiden Buffer unterhalten. Er enthält unter anderem die Adressen der Buffer und die aktuelle Leseposition.

4.3 Herausforderungen

4.3.1 Keine Änderungen an tc

Wie bereits in Kapitel 3 erwähnt, wird das Netem-Userinterface über das tc Hauptprogramm und das allgemeine Qdisc-Parsingprogramm aufgerufen. Ein Ziel bei dieser Erweiterung von Netem ist, nur das eigentliche Userinterface von Netem zu verändern. Das wird aus Portabilitätsgründen gemacht und weil es sich um eine Erweiterung von Netem, und nicht tc im allgemeinen, handelt. Daraus ergeben sich einige Anforderungen an die Kommunikation der einzelnen Teile von Netem, insbesondere im Zusammenhang mit dem Flowseedprozess.

Da sich das Netem Userinterface beendet, bevor bekannt ist, ob das Kommando vom Kernel akzeptiert wird, ist das Userinterface ungeeignet um einen Prozess, der zur Paketverarbeitung benötigt wird, zu starten. Da aber am tc Source-Code nichts geändert werden soll, muss der Flowseedprozess dennoch aus dem Netem Userinterface gestartet werden. Schlägt das angegebene Kommando fehl, wurde der Flowseedprozess fälschlicherweise erzeugt und kann nicht mehr gestoppt werden.

Lösung

Das Netem Userinterface kennt nach dem Kreieren des Flowseedprozesses dessen Prozess ID (PID). Diese wird dem Kernel bei der Initialisierung einer Qdisc mitgeteilt. Bei fehlerhaften tc Kommandos kommt es zu keiner PID-Registration im Kernel, da die Initialisierung schon vorher abgebrochen wird. Der Flowseedprozess sendet dem Kernel neben den Aktionswerten auch seine PID. Werden Daten vom Kernel empfangen, wird überprüft ob ihm die angegebene PID bekannt ist, falls nicht, wird der dazugehörende Prozess mit einem SIGKILL Signal beendet, andernfalls werden die Daten akzeptiert.

Sendet jetzt ein Flowseedprozess, der zu einem fehlerhaften tc Kommando gehört, Daten, wird dieser durch ein SIGKILL Signal beendet, da seine PID nicht registriert wurde.

Dieses Verfahren bewirkt zusätzlich, dass niemand dem Netem Kernelmodul unaufgefordert Daten schicken kann, die zu einem Fehlverhalten führen könnten.

4.3.2 Koordination: Initialisierung Kernel, Start Flowseedprozess

Es muss sichergestellt werden, dass der Flowseedprozess erst Daten sendet, wenn seine PID im Netem Kernelmodul registriert ist. Andernfalls würde er vom Kernel durch ein SIGKILL Signal beendet. Dazu muss sich der Flowseedprozess mit der Registration seiner PID im Kernel synchronisieren.

Lösung

Sobald sich der Userinterface-Prozess beendet hat, sind entweder alle Parameter im Kernel registriert oder das Kommando ist fehlgeschlagen. In beiden Fällen kann nun der Flowseedprozess mit dem Senden seiner Daten beginnen. Wurde das Kommando erfolgreich registriert, ist seine Prozess ID im Netem Kernelmodul registriert und seine Daten werden akzeptiert. Ist das Kommando fehlgeschlagen, wird das Kernelmodul den Flowseedprozess mit einem SIGKILL Signal beenden.

Da der Flowseedprozess durch einen fork im Userinterfaceprozess entstanden ist, kann sich das Linux System mit parents, children und den jeweiligen Prozess IDs für die Synchronisation zwischen dem Beenden des Userinterfaces und dem Start des Sendens des Flowseedprozesses zunutze gemacht werden.

In Linux braucht jeder Prozess einen parent, dieser ist derjenige Prozess, von welchem der aktuelle Prozess gestartet wurde. Wird dieser parent beendet, wird als neuer parent der "grandparent", also der parent des ursprünglichen parent verwendet.

Für dieses Problem lässt sich das folgendermassen ausnutzen: das Userinterface speichert vor dem fork call seine Process ID in einer Variablen. Zwischen dem fork und dem execl call kennt das child noch sämtliche Variablen des parent. Unmittelbar nach dem fork call legt sich das child für eine sehr kurze Zeit schlafen, wenn es wieder aufwacht, vergleicht es seine parent Process ID mit derjenigen, die es gespeichert hat. Falls sie übereinstimmen, ist das Userinterface noch nicht beendet und somit die Registration im Kernel noch nicht abgeschlossen. Sind sie aber unterschiedlich, heisst das, dass sein ursprünglicher parent gestorben ist, und so die Registration abgeschlossen ist. Nun führt das child einen execl call aus und beginnt mit dem eigentlichen Flowseedprozess (siehe Abbildung 4.3).

4.3.3 Eindeutige Zuweisung Tracefile und Qdisc

Wenn mit Hilfe von tc mehrere Netem Qdiscs an ein Interface gelegt werden, muss sichergestellt sein, dass jeder Qdisc die Aktionswerte aus ihrem Tracefile zugeordnet werden, und nicht diejenigen eines Tracefiles einer anderen Qdisc.

Lösung

Im Kernelmodul gibt es ein Array mit einem Eintrag pro angelegter Qdisc. Dieser Eintrag besteht aus dem struct _flowbuffer, welcher zur Verwaltung der

```
int parentID, childID;
parentID=getpid();
switch(pid=fork()){
  case -1:{
              fprintf(stderr,"Cannot fork\n");
              return -1;
          }
  case 0: {                              //child
              //Warte bis der parent gestorben ist
              while(parentID==getppid()){
                 sleep(0);
              }
              //Flowseed Programm Aufruf
              execvl=execl(FLOWSEED,"flowseed", filename,*argv,0);
              if(execvl<0){
                  fprintf(stderr,"starting child failed\n");
                  return -1;
              }
          }
  default:{                              //parent
              fprintf(stderr, "I'm the parent\n");
          }
}
```

Abbildung 4.3: Synchronisation Parent Child

Buffer für die Aktionswerte benötigt wird. Bei der Initialisierung einer Qdisc wird überprüft welcher Eintrag im Array frei ist und dort der entsprechende _flowbuffer struct gespeichert. Netem stellt einen struct netem_sched_data zur Verfügung der jeder Qdisc eigen ist. In diesem struct wird eine Variable (q− >trace) gesetzt, die auf die entsprechende Position im Array der _flowbuffer structs verweist (siehe Abbildung 4.4). Mit Hilfe dieser Variablen kann auf den richtigen Buffer zugegriffen werden. Wird eine Qdisc gelöscht, wird der entsprechende Eintrag im Array freigegeben, und kann von einer neuen Qdisc wieder belegt werden. Alternativ ist denkbar, den _flowbuffer struct als weiteres Element direkt in den netem_sched_data struct zu speichern. Um das ursprüngliche Netem möglichst wenig zu beeinträchtigen, wurde die erste Variante gewählt.

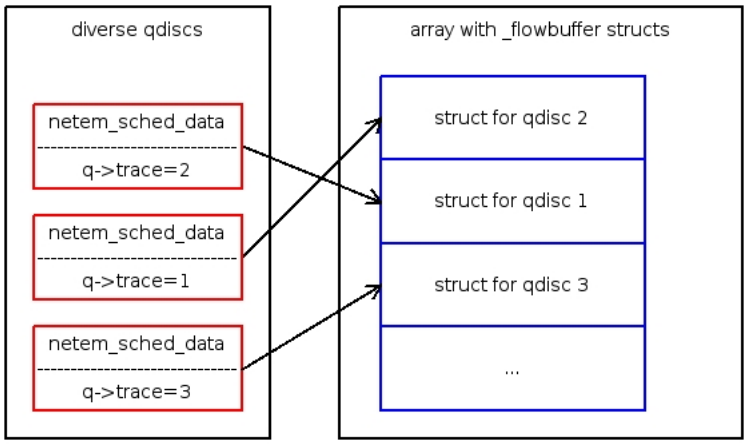

Abbildung 4.4: Mapping Between _flowbuffer Struct Array and Qdisc

4.3.4 Korrekte Statistik

Die Netem Trace-Erweiterung führt eine Statistik über alle durchgeführten Aktionen. Unter anderem wird gezählt, wie oft ein Aktionswert aus einem Buffer gelesen werden soll, dort aber kein Wert vorhanden ist. Es können zwei Fälle unterschieden werden:

1. Es sind tatsächlich keine Daten verfügbar, weil die Initialisierung noch nicht vollständig abgeschlossen ist und noch keine Daten vom Flowseed-prozess an den Kernel gesendet wurden oder weil das Tracefile zu Ende ist.

2. Das Tracefile enthält noch Daten, doch diese können aus irgendwelchen Gründen (zu hohe CPU Last zum Beispiel) nicht geladen werden. Dies wird in der Statistik als "Bufferunderrun" aufgeführt.

Eine korrekte Statistik muss zwischen diesen beiden Fällen unterscheiden können.

Lösung

Der Flowseedprozess hängt an jede Sendung von Aktionswerten an den Kernel ein Flag, ob die gesendeten Daten gültig sind. Stellt er fest, dass er keine Daten mehr zu senden hat, setzt er das Flag auf "false", sendet aber dennoch ein letztes Mal. Dieses Flag wird in den struct _flowbuffer geschrieben, welcher die Buffer verwaltet. Beim Laden des Moduls wird der Wert auf "keine gültigen Daten vorhanden" initialisiert.

Kann für ein Internet-Paket kein Wert aus dem Buffer gelesen werden, wird das entsprechende Flag zur Unterscheidung der beiden Fälle benutzt. Pakete, bei denen dieses Flag auf "false" gesetzt ist, werden in der Statistik als Pakete erfasst, für welche keine gültigen Daten vorhanden sind. Pakete, bei denen es auf "true" gesetzt ist, werden als Bufferunderruns gezählt.

Kapitel 5

Evaluation

Die Funktionsweise der Netem Trace-Erweiterung wurde wie folgt evaluiert:

1. Funktionale-Tests
2. Performance-Tests

Mit den funktionalen Tests wird gezeigt, dass die Netem Trace-Erweiterung das original Netem nicht beeinflusst und dass die erweiterten Funktionen korrekt implementiert sind. Die Performance-Tests zeigen, dass sich die Netem Trace-Erweiterung bei einer Last von bis zu 80'000 Paketen pro Sekunde in Bezug auf Paketdelayverteilung und Paketverlust im Wesentlichen gleich verhält wie das Standard-Netem.

5.1 Funktionale Verifikation

Es wurden die folgenden funktionalen Tests durchgeführt:

- Test, ob die Werte aus dem Tracefile korrekt gelesen und die Pakete nach diesen Werten behandelt werden.
- Anlegen und wieder Löschen einer Netem Trace-Qdisc.
- Anlegen einer Netem Trace-Qdisc, Wechseln zu einer Netem Delay-Qdisc, diese mehrmals Ändern und wieder Wechseln zu einer Trace-Qdisc.
- Anlegen einer Netem Trace-Qdisc und das dazugehörende Tracefile mehrmals Wechseln.
- Mehrere Delay- und Trace-Qdiscs gleichzeitig Anlegen und diese mehrmals Ändern.
- Eingeben von unkorrekten tc Kommandos.
- Test, ob die korrekten Defaultwerte zur Paketverarbeitung verwendet werden.
- Generieren eines zusätzlichen Flowseedprozesses, der dem Kernel Daten zu senden versucht.

Alle Tests wurden auch unter der Last von 80'000 zu verarbeitenden Paketen pro Sekunde bestanden.

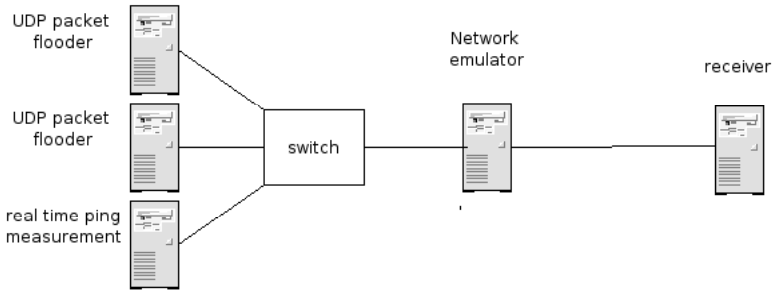

Abbildung 5.1: Evaluation Setup

5.2 Performance-Tests

Mit den Performance-Tests wird untersucht, inwiefern sich die Netem Trace-Erweiterung von der ursprünglichen Netem Version unter Last unterscheidet. Von besonderem Interesse ist der Paketverlust und die genaue Verzögerungsverteilung.

5.2.1 Versuchsaufbau

Der Versuchsaufbau der Performance-Tests ist aus Abbildung 5.1 ersichtlich. Um am Netzwerkemulator eine möglichst konstante Paketrate zu erzeugen, wurde die Paketgeneration auf zwei PCs aufgeteilt. Diese senden UDP-Pakete mit der dem Test entsprechenden Rate via Emulatorcomputer an den Empfängercomputer. Das Senden der UDP-Pakete wird von dem Programm pktgen, welches Bestandteil jedes Linux-Kernels ist, durchgeführt. Um die genauen Paketverzögerungszeiten zu messen, werden von einem Realtime-System (RTAI [5] mit RTnet [6]) Pingpakete an den Empfängercomputer gesendet. Dies ermöglicht eine zeitliche Auflösung von einer Mikrosekunde. Während einem Testdurchlauf werden neben den UDP-Paketen 100'000 Realtime-Pingpakete mit einer Rate von 100 Paketen pro Sekunde gesendet. Die Netem Trace-Erweiterung ist so konfiguriert, dass Pakete, für welche keine Aktionswerte vorhanden sind, verworfen werden.

Zur Messung der CPU-Auslastung wurde das Programm "top" verwendet. Der Emulatorcomputer hat folgende Kenndaten:

- CPU: AMD Athlon 2083MHz
- RAM: 1GB
- Netzwerkkarten: RealTek RTL8139D
- Betriebssystem: Debian sarge mit Kernel 2.6.16.19

5.2.2 Paketverlust

Die wichtigsten Ergebnisse sind in Tabelle 5.1 zusammengestellt. Bis zu einer Last von 80'000 Paketen pro Sekunde (P/s) tritt bei keinem Testszenario ein

signifikanter Paketverlust auf. Bei einer Paketverzögerung von 10ms ist der Paketverlust in der Ordnung von 10^{-5}. Dies führen wir auf die beschränkte Grösse der Netem-Warteschlange zurück. Bei den Tests war die Queuelänge auf den Standardwert von 1'000 Paketen begrenzt. Werden Pakete um 10ms verzögert, müssen unter idealen Bedingungen 800 Pakete in dieser Queue gespeichert werden. In der Realität erreicht diese Queuelänge aber in seltenen Fällen den Maximalwert. Dies kann auf folgende zwei Faktoren zurückgeführt werden:

- Pakete werden immer erst beim nächsten Timerinterrupt gesendet. Da die Timerauflösung 1ms beträgt, werden bis zum nächsten Interrupt zusätzlich 80 Pakete in die Warteschlange eingefügt.
- Es wird sowohl bei der Paketgeneration als auch beim Netzwerkemulator kein Realtime-Betriebssystem verwendet. Dies führt zu einer nicht ganz konstanten Paketrate, welche vom Netzwerkemulator verarbeitet werden muss. Zudem können im Emulator zusätzliche Verzögerungen entstehen, welche zu einem Anwachsen der Queue führen.

Pakete, die empfangen werden wenn die Netem-Warteschlange voll ist, werden verworfen.

Bei der CPU ist bei 80'000 zu verarbeitenden Paketen schon eine deutliche Auslastung aufgrund der zu behandelnden Interrupts feststellbar. Die höhere CPU-Auslastung von der Netem Trace-Erweiterung gegenüber dem Standard-Netem lässt sich durch den zusätzlichen Aufwand des Flowseedprozesses, der die Aktionswerte in den Kernel transferiert, erklären. Bei den 80'000 P/s werden 80 Mal in der Sekunde 1'000 Aktionswerten aus dem Tracefile gelesen und an den Kernel gesendet. Zusätzlich benötigt die Netem Trace-Erweiterung pro verarbeitetes Paket einen Funktionsaufruf mehr als das Standard-Netem. Erstaunlich ist, dass die CPU am stärksten belastet ist, wenn das Netem Modul gar nicht geladen ist. Dies führen wir darauf zurück, dass ohne Netzwerkemulator für jedes Paket, das gesendet wird, ein Interrupt ausgelöst werden muss. Der von Netem verwendete Linux Standardtimer hat eine zeitliche Auflösung von 1ms. Deshalb wird bei der Verwendung des Netzwerkemulators nur jede Millisekunde ein Interrupt ausgelöst. Dabei werden gleich alle zum Senden vorhandenen Pakete an den Treiber weitergeleitet. Dies ist bei der Paketrate von 80'000 Paketen pro Sekunde 80 Mal weniger als ohne Netzwerkemulator.

Bei einer Last von 90'000 P/s kann top die CPU-Auslastung teilweise nicht mehr korrekt anzeigen, weil die CPU bereits so stark ausgelastet ist, dass der top Prozess nicht mehr ausgeführt wird. Bei 90'000 P/s verliert die Netem Trace-Erweiterung deutlich mehr Pakete als das Standard-Netem. Dies ist darauf zurückzuführen dass, der Flowseedprozess wegen der hohen CPU-Auslastung nicht mehr ausgeführt wird und somit dem Kernel keine Aktionswerte mehr geliefert werden. Der Bereich von 90'000 P/s ist für die Netzwerkemulation mit dieser Hardware/Software-Konfiguration nicht mehr von Interesse, da die CPU auch ohne Emulator nicht mehr in der Lage ist, alle Pakete zu bearbeiten.

5.2.3 Paketverzögerungsverteilung

Abbildung 5.2 zeigt die Paketverzögerungsverteilung bei 80'000 P/s und einer angelegten Verzögerung von 10ms. Die Kurve der Netem Trace-Erweiterung ist derjenigen vom Standard-Netem sehr ähnlich. Bei beiden liegt der Mittelwert

Load (P/s)	Emulator	CPU Load	Packet loss
10'000	no emulator	0.3%	0%
	delay 1ms	0.3%	0%
	trace 1ms	0.3%	0%
	delay 10ms	0.3%	0%
	trace 10ms	0.3%	0%
80'000	no emulator	50%	0%
	delay 1ms	7%	0%
	trace 1ms	15%	0%
	delay 10ms	7%	$18*10^{-6}$
	trace 10ms	15%	$37*10^{-6}$
90'000	no emulator	100%	5%
	delay 1ms	100%	8%
	trace 1ms	100%	98%
	delay 10ms	100%	16%
	trace 10ms	100%	97 %

Tabelle 5.1: Packet Loss

ca. 1ms zu hoch. Dies kann durch das Zusammenspiel mehrerer Faktoren erklärt werden:

- Auch ohne Netem gibt es eine Verzögerung von 0.2ms. Diese kommt von der Verarbeitung der Pingpaket im Router und im Zielcomputer.
- Der Linux Timer hat eine Auflösung von 1 ms.
- Ein Paket wird von Netem immer erst dann geliefert, wenn sein Timestamp in der Vergangenheit liegt.

Tabelle 5.2 zeigt den Mittelwert und die Standardabweichung sowohl für die beiden Messungen mit Netem als auch für die Referenzmessung ohne Netem. Die höhere Standardabweichung von der Netem Trace-Erweiterung gegenüber dem Standard-Netem ist auf die erhöhte CPU Last zurückzuführen.

	Referenz	delay 10ms	trace 10ms
Mittelwert	200	10992.7	11005.9
Standardabweichung	25.7	61.9	163.7

Tabelle 5.2: Average and Standard Deviation in μs for 80'000 Packets/sec

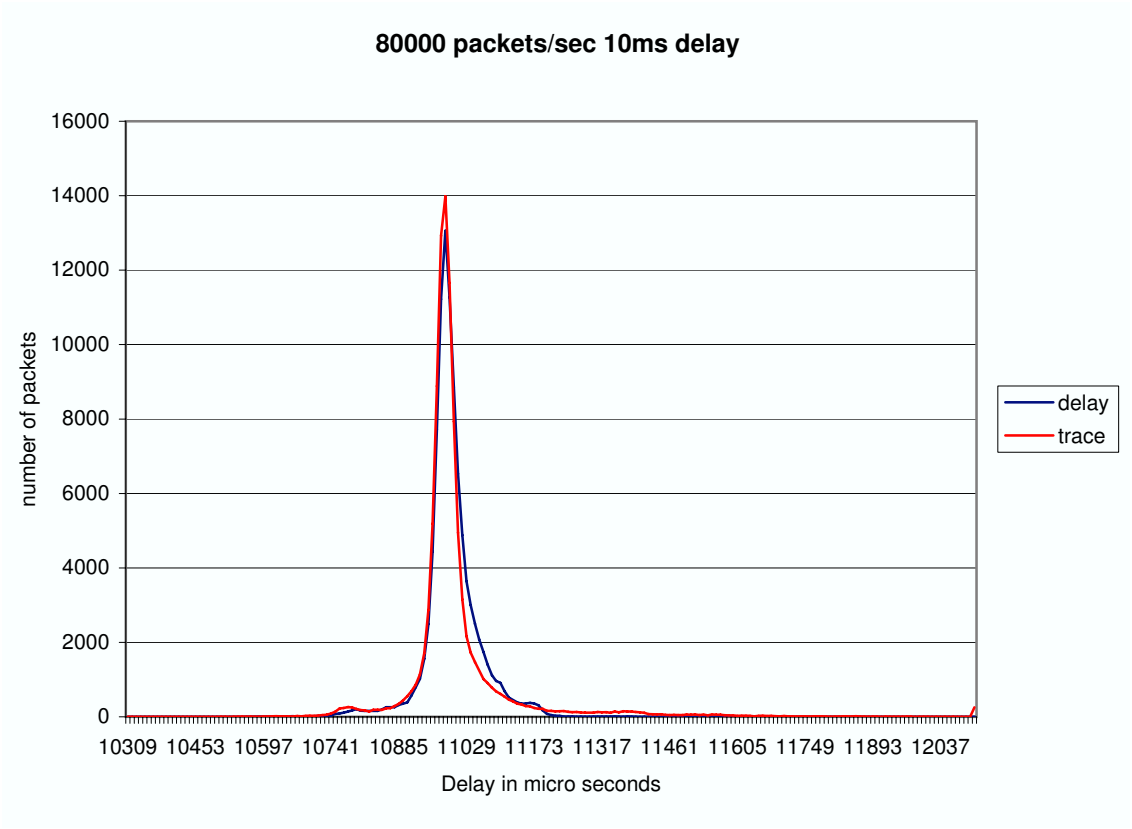

Abbildung 5.2: Packet Delay Distribution for 80'000 Packets/sec

Kapitel 6

Fazit

Das Ergebnis dieser Semesterarbeit ist ein Netzwerkemulator, welcher fähig ist, alle statistischen Eigenschaften des Internets, insbesondere die long-range dependency, korrekt zu modulieren. Dies wird dadurch erreicht, dass die Werte für Paketverzögerung, -verlust, -duplikation und -corruption (das Einfügen eines Bitfehlers) aus einem Tracefile gelesen werden. Dieses Tracefile wird im Voraus mit Hilfe von Messungen, Simulationen, oder Berechnungen erstellt.

Zu Beginn dieser Semesterarbeit wurde mit Hilfe des Source-Codes die Funktionsweise des Linux Netzwerkeumlationstool Netem und diejenige der NIST Net Trace-Erweiterung analysiert. In einem zweiten Schritt wurde ein Redesign von der NIST Net Trace Erweiterung gemacht, um dieses dann in Netem zu integrieren. Nach der Implementation wurden Tests durchgeführt, die das korrekte Funktionieren von der Netem Trace-Erweiterung auch bei bis zu 80'000 zu verarbeitenden Paketen pro Sekunde zeigen. Um den Gebrauch von Netem zu erleichtern wurde ein umfangreiches Manual geschrieben, in welchem die Themen "tc packet filtering" und "netem configuration" behandelt werden. Zusätzlich wurden zwei Patches hergestellt, die das einfache Upgraden auf die Netem Trace-Erweiterung ermöglichen. Der eine Patch enthält die Änderungen im Kernel Source-Code, der andere diejenigen am tc Source-Code.

Im Vergleich zwischen den beiden Trace-Erweiterungen, der einen für NIST Net und der in dieser Arbeit entwickelten für Netem, zeigen sich zahlreiche Unterschiede. Die Netem Trace-Erweiterung basiert auf dem Linux-Kernel 2.6. Dies ist für die Zukunft ein bedeutender Vorteil, da die weitere Entwicklung von Linux auf diesem Kernel vonstatten gehen wird, und nicht auf dem von NIST Net verwendeten Kernel 2.4. Mit der im Kernel 2.6 erhöhten Timerfrequenz von 1000Hz im Gegensatz zu den 100 Hz vom Kernel 2.4 wurde es für Netem möglich, den Standardtimer von Linux zu verwenden. Die zeitliche Auflösung ist zwar zur Zeit noch rund acht mal geringer als diejenige von NIST Net, welches die MC146818 real-time clock (RTC) als Timerinterruptquelle benutzt, doch ist sie für die meisten Anwendungen genügend. Jedoch wird die 2.6er Implementation von aktuellen Projekten profitieren können, welche beabsichtigen schnellere Timer in Linux zu verwenden [7]. Mit dem Einbinden des Netzwerkemulators in den offiziellen Kernel Source-Code wird eine weite Verbreitung erreicht.

Die in dieser Arbeit entwickelte Netem Trace-Erweiterung hat einige "Features" mehr als diejenige für NIST Net. Es kann beispielsweise angegeben werden wie oft ein Tracefile wiederholt wird, oder was der Defaultwert bei der

Paketverarbeitung ist.

Tabelle 6.1 fasst die wesentlichen Unterschiede zwischen der Netem Trace-Erweiterung und RplTrc zusammen

	Netem Trace Extension		RplTrc
+	Part of kernel 2.6	-	Addition to kernel 2.4
+	Packet filtering according arbitrarily fields	-	Packet filtering only per flow
+	Delay, duplication, loss, corruption	-	Delay, duplication, loss
-	Timer resolution 1ms	+	Timer resolution 122μs
+	Manual with examples	-	Only README file
+	Configurable options	-	No options

Tabelle 6.1: Comparison Between Netem Trace Extension and RplTrc

Bei der Evaluation der Netem Trace-Erweiterung hat sich ein stabiles und korrektes Verhalten gezeigt, bei dem keine wesentlichen Unterschiede zwischen der Netem Trace-Erweiterung und dem ursprünglichen Netem gefunden wurden.

In der folgenden Liste sind meine Beiträge zum Netzwerkemulator TCN zusammengefasst. Sie sind chronologisch geordnet und mit einigen Kommentaren versehen.

- **Redesign von RplTrc für die Integration in Netem.** Zuerst musste ich den Source-Code von RplTrc und denjenigen von Netem verstehen. Danach haben wir nach einem Design gesucht, welches sich möglichst gut in Netem integrieren lässt.
- **Erweiterung von RplTrc.** Neu kann ein Defaultwert für die Paketverarbeitung spezifiziert werden, und angegeben werden, wie oft das Tracefile wiederholt werden soll.
- **Implementation.** Alle Details müssen korrekt implementiert sein, sonst kann der Kernel hängen bleiben.
- **Performance Evaluation.** Die Installation, des für die Evaluation benötigten Realtime-Betriebssystems, war schwierig. Das Durchführen der einzelnen Tests und deren Auswertung war aufwändig.
- **Manual "tc packet filtering and netem".** Die Kommandos für komplexe Filterkriterien sind sehr kompliziert. Es gibt dazu keine Dokumentation.
- **Erstellen von Patches.** Es muss sichergestellt sein, dass die Patches korrekt sind. Um sie an kernel.org einzureichen, muss die korrekte Ansprechperson gefunden werden.
- **Erstellen einer Webpage für TCN.** Das Layout muss möglichst übersichtlich sein.
- **Erweiterung des Netem-GUI.** Das Netem-GUI [8] ist in PHP programmiert und hat mehr Zeilen Code als der ganze Netzwerkemulator. Die Erweiterung stellte sich als aufwändiger, als ursprünglich angenommen, heraus.

Kapitel 7

Further Work

Die zeitliche Auflösung von Netem ist durch den Linux Standardtimer gegeben. Aktuell liegt sie bei 1ms. Wegen dieser relativ groben zeitlichen Auflösung kann Netem nicht benutzt werden, um Netzwerke mit sehr kurzen Verzögerungen zu emulieren. Zur Zeit ist ein Projekt [7] im Gange, mit dem Ziel hochauflösende Timer für den Linux-Kernel zur Verfügung zu stellen. In Zukunft kann deshalb mit einer besseren zeitlichen Auflösung gerechnet werden.

Um die zeitliche Genauigkeit weiter zu verbessern, kann die Netem Trace-Erweiterung auf ein Realtime-Linux wie RTAI [5] portiert werden. Dies ist aber mit einem grossen Aufwand verbunden, da Realtime-Betriebssysteme anders aufgebaut sind und ein anderes API verwenden. Die Vorteile von einem Realtime-Linux liegen neben der besseren Genauigkeit auch darin, dass keine unvorhersehbaren Störungen wie Taskscheduling oder andere Interrupts auftreten können, welche die Pakete zusätzlich verzögern würden.

Netem ermöglicht das Einfügen eines einzelnen Bitfehlers an einer zufälligen Position im Paket. Um Wireless-Protokolle besser testen zu können, könnte man Netem so erweitern, dass man neben der Paketverzögerung auch einen Wert für die Anzahl und die genaue Position der Bitfehler aus dem Tracefile liest.

Literaturverzeichnis

[1] J. Beran, Statistics for Long-Memory Processes. Monographs on Statistics and Applied Probability, Chapman and Hall, 1994

[2] NIST Net, http://snad.ncsl.nist.gov/nistnet/

[3] Netem, http://linux-net.osdl.org/index.php/Netem

[4] Thomas Hug, Trace-Based Network Emulation, Master Thesis MA-2005-06 at ETH Zurich, 2005

[5] RTAI - the RealTime Application Interface for Linux from DIAPM, https://www.rtai.org/

[6] RTnet, Hard Real-Time Networking for Real-Time Linux, http://www.rts.uni-hannover.de/rtnet/

[7] High Res POSIX timers, sourceforge.net/projects/high-res-timers

[8] PHPnetemGUI, http://www.smyles.plus.com/phpnetemgui/

Anhang A

Ergebnisse der Performance-Evaluation

Dieser Appendix enthält alle Messungen für 10'000, 80'000 und 90'000 Pakete pro Sekunde. Die Messungen wurden sowohl für die Trace-Erweiterung als auch für das ursprüngliche Netem gemacht. Es wurden jeweils zwei Serien gemessen, die eine mit einer Verzögerung von 1ms und die andere mit einer Verzögerung von 10ms. Als Referenzmessung wurde eine Serie ohne Netzwerkemulator gemacht.

In einer Tabelle sind die durchschnittliche Verzögerung, die Standardabweichung der Verzögerung, der Minimal- und der Maximalwert der Verzögerung jeweils in Millisekunden angegeben. Die CPU-Auslastung des Emulator-PC und der Paketverlust kann ebenfalls aus der Tabelle gelesen werden.

Aus den Plots kann herausgelesen werden, wie oft ein Paket mit einer bestimmten Verzögerung empfangen wurde. Auf der x-Achse ist die Paketverzögerung in μs angegeben.

10'000 packets per second

	AVG	STD	MIN	MAX	CPU	Packet loss
ohne netem	0.155	0.014	0.13	0.499	0.3%	0
netem 1ms	1.701	0.303	1.149	34.259	0.3%	0
trace 1ms	1.701	0.3	1.158	30.679	0.3%	0
netem 10ms	10.96	0.178	10.152	39.954	0.3%	0
trace 10ms	10.939	0.183	9.869	41.932	0.3%	0

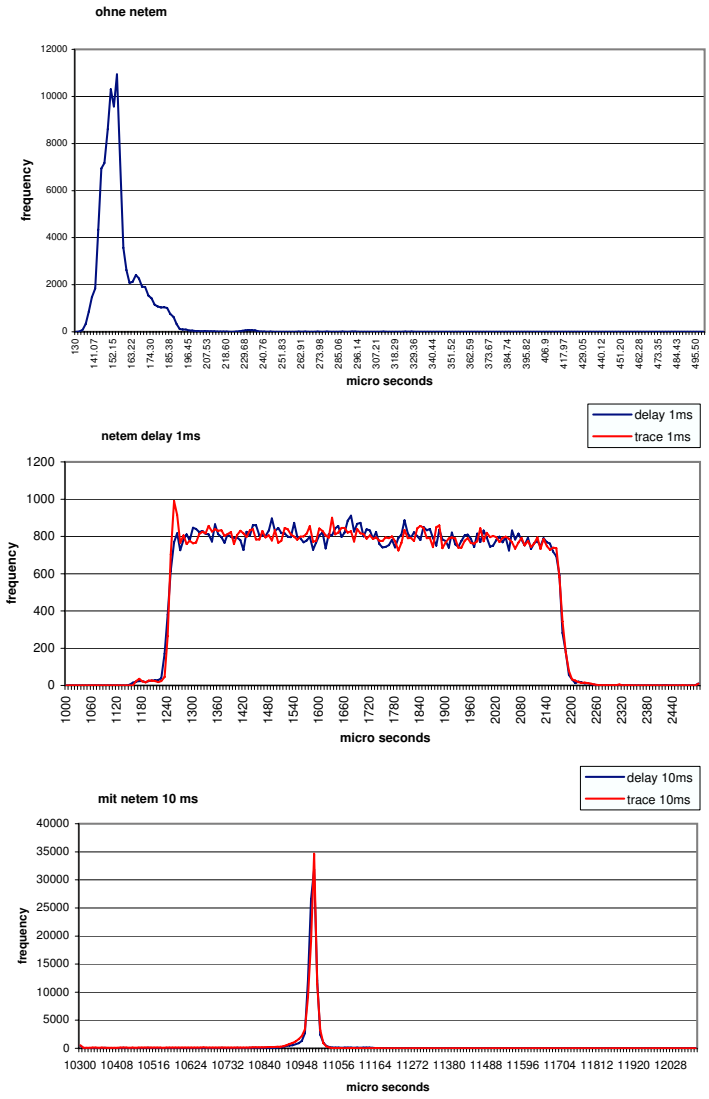

Abbildung A.1: Evaluation at 10'000 Packets/sec

80'000 Packets per second

	AVG	STD	MIN	MAX	CPU	Packet loss
ohne netem	0.2	0.026	0.134	2.019	50%	0
delay 1ms	2.022	0.103	1.191	4.329	7%	0
trace 1ms	2.024	0.111	1.167	8.018	15%	0
delay 10ms	10.993	0.063	10.252	15.234	7%	1.80E-05
trace 10ms	11.007	0.162	10.074	18.324	15%	3.70E-05

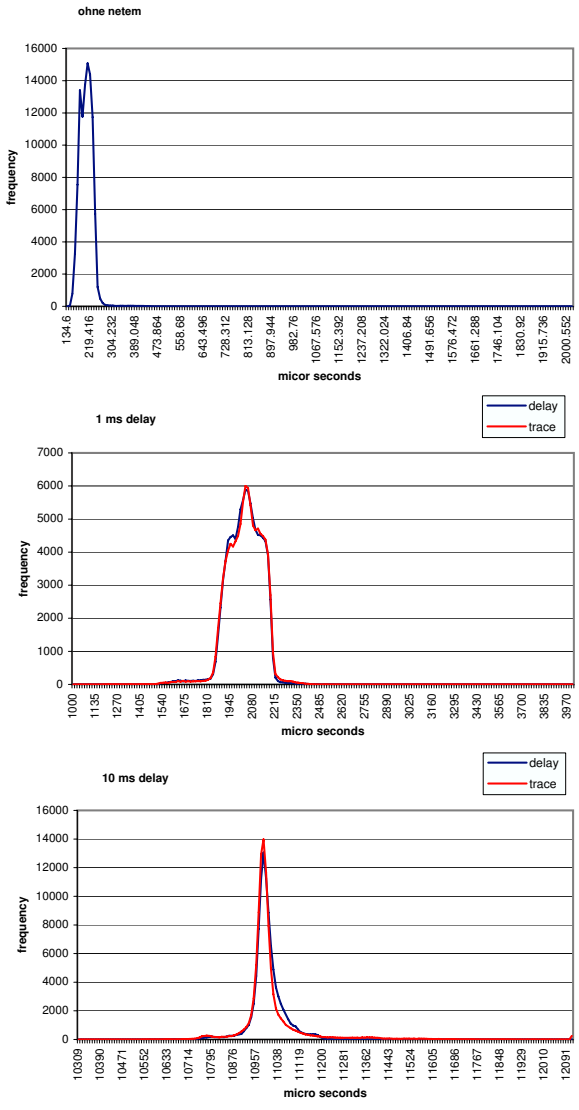

Abbildung A.2: Evaluation at 80'000 Packets/sec

90'000 packets pro second

	AVG	STD	CPU	Packet loss
ohne netem	8.9	3.9	100%	5%
netem 1ms	10.9	2.9	100%	8%
trace 1ms	9.8	3.5	100%	98%
netem 10ms	13.2	1	100%	16%
trace 10ms	13.1	0.9	100%	97%

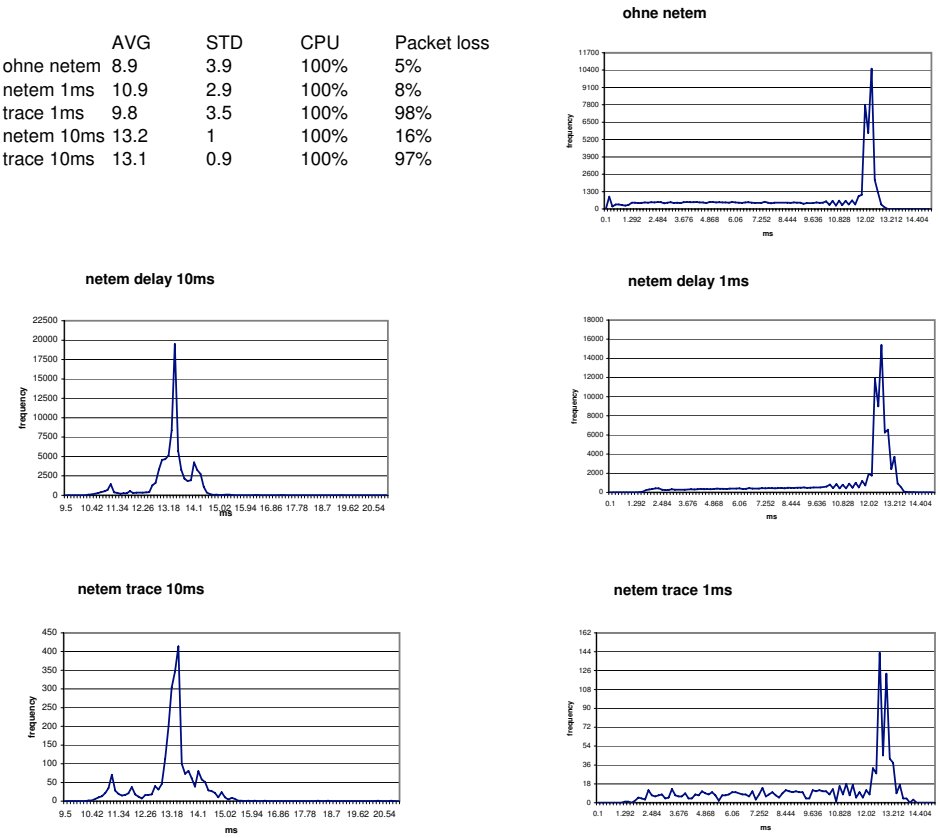

Abbildung A.3: Evaluation at 90'000 Packets/sec

Anhang B

Task Assignement

A Trace Reader Extension for Linux 2.6 Network Emulator Module (RplTrc for Netem)

Semester Thesis

Keller Ariane, arkeller@ee.ethz.ch

Advisor: Rainer Baumann, baumann@tik.ee.ethz.ch
Dr. Ulrich Fiedler, fiedler@tik.ee.ethz.ch
Simon Heimlicher, heimlicher@tik.ee.ethz.ch
Supervisor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

Issue Date: April 2006
Submission Date: October 2006

1. Introduction

Evaluating the performance of distributed applications, such as mobile telephone or video conferencing devices, has become increasingly complex as the diversity of scenarios and data rates in networks increase. In his Master Thesis, Thomas Hug implemented RplTrc, a network emulation tool, designed by Rainer Baumann and Ulrich Fielder. RplTrc uses traces to drive interceptions of the Linux protocol stack to delay, drop or duplicate packets. The length of traces is not limited. Thus, RplTrc is capable to account for important performance characteristics inherent to real networks such as long-range dependence and self-similarity of cross-traffic. Installed on a commodity PC, RplTrc can work at data rates of 100 MBit/s and more and has a precision in reproducing packet delays of the order of 100 μs. RplTrc thus enables extensive performance evaluation of distributed applications in lab environments. Given this context, this thesis leads towards porting the trace reader extension of RplTrc to the netem module of the Linux 2.6 kernel. If time allows, there are several options for improvements.

2. Assignment

2.1 Objectives

Porting and extending of a trace reader module developed for a network emulator that acts as a "network in a box" to perform controlled, reproducible experiments with network performance.

2.2 Tasks

- Get familiar with RplTrc's trace reader module.
- Get familiar with netem, the Linux 2.6 kernel network emulator module.

- Porte the trace reader module from RplTrc into the Linux 2.6 netem module. This especially includes the kernel module extension and the user interface as well as the user space process.
- Include bit errors into the trace reader module (optional). This allows for also emulating wireless networks. This could be done by extending a trace entry for 10 bit errors.
- Adapted the implementation to a real-time Linux (optional). An easy option would be to patch the kernel for real-time behaviors or to change the used timer. A more advanced option would be to port the netem with the trace reader module to a hard real-time Linux as RTAI.

3. Deliverables and Organization

- Whenever possible, student and advisor meet or telephone on a weekly basis to discuss the progress of work and next steps to undertake. The student should not hesitate to contact the advisor immediately, if problems/questions arise that can not be solved independently,
- At the end of the third week, a detailed time schedule of the semester thesis must be given and discussed with the advisor.
- In regular intervals (e.g. every two or three months) intermediate reports are due. These reports are linked to short presentations of 15 minutes to the professor and the advisor. In these presentations, the student has to discuss major aspects of the ongoing work including results, problems, and remaining work.
- At the end of this thesis, a presentation of 15 minutes must be given either in teleconference or in the communication systems group meeting. The presentation should carefully introduce settings and background of the work. Moreover, it should contain an overview of the major results and conclusions from the work.
- We encourage writing all reports in English. However, reports can also be written in German. The final report must contain a summary, the assignment and the time schedule. Its structure should include an introduction, a methods/design section, a results section and a conclusion section. Moreover, the final report should include a complete documentation of all produced software. Related work must be correctly referenced. See https://tikiwiki.ethz.ch/thesis or http://www.tik.ee.ethz.ch/~flury/tips.html for more tips on thesis writing. Three hard copies of the final report must be delivered to TIK.
- Any software which is produced in relation with this thesis needs to be delivered before ending the thesis. This includes all source code and documentation. The student agrees that the software may be published as open source. Moreover, the PDF and the source code employed to generate the final report also have to be delivered. This includes data to draw the figures preferred format for delivery is a CDROM.

4. References

A large collection of documents and code can be found under the following URL.
http://www.tik.ee.ethz.ch/~baumarai/shadow/fa=d233ppLDui74984651aqer

Anhang C

Timetable

Woche	Arbeit
1	Einarbeiten, Aufsetzen PC etc.
2	Verstehen von Netem und RplTrc
3	
4	Design und Implementation der Trace-Erweiterung
5	
6	
7	
8	Versehen der Filtermöglichkeiten von tc
9	Schreiben des Manuals
10	
11	Performance Tests
12	
13	Dokumentation, Bugfixes
14	
15	Zwischenpräsentation, Source-Code aufräumen
16	Endpräsentation, Patches erstellen

Anhang D

Manual Tc Packet Filtering and netem

Manual
tc Packet Filtering and netem

Ariane Keller
ETH Zurich

July 21, 2006

Contents

Chapter 1

Brief Introduction

This manual describes the usage of netem. Netem is a network emulator in the linux kernel 2.6.7 and higher that reproduces network dynamics by delaying, dropping, duplicating or corrupting packets. Netem is an extension of tc, the linux traffic control tool in the iproute2 package.

To understand this manual, which describes the configuration of the network emulator, we assume that you have some basic knowledge in IP networking and more specifically in IP packet handling in the Linux kernel. We recall that any linux machine running netem must be configured as a router (see section 4.1 for more details). The simplest network topology to use netem on a router is depicted in figure 1.1.

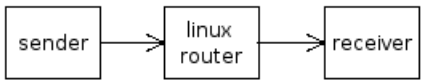

Figure 1.1: Network topology

Inside the router, IP packet handling is performed as follows. Packets enter a network interface card (NIC) and are classified and enqueued before entering linux internal packet handling. After packet handling, packets are classified and enqueued for transmission on the egress NIC.

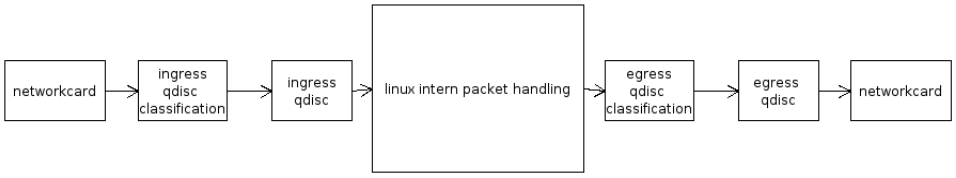

Figure 1.2: IP packet handling in the Linux kernel

Packet classification can be performed by analyzing packet header fields (source and destination IP addresses, port numbers, etc.) and payload. The

classification can be configured with the traffic control tool tc. Based on this classification, packets are enqueued in one of the ingress/egress queues. In the standard configuration there is solely one queue per interface and the packets are processed in a FIFO manner. The combination from queue and algorithm that decides when to send which packet is called qdisc (short for queueing discipline).

This manual starts with describing how one or multiple qdiscs can be configured on an egress interface and how to configure packet classification to identify specific flows before describing the configuration of the network emulation tool netem. This description includes the configuration of our enhancements for trace controled network emulation TCN. Moreover, we give information how to generate packet action traces that specify the amount of delay for each packet in the emulation as well as, which packets are to be dropped, duplicated, or corrupted.

Chapter 2

tc: Linux Advanced Routing and Traffic Control

Traffic control (tc) is part of the linux iproute2 package which allows the user to access networking features. The package itself has three main features: monitoring the system, traffic classification, and traffic manipulation. The tc part in the package can be used

- to configure qdiscs as well as
- to configure packet classification into qdiscs.

A general description of the iproute2 package can be found in an online manual at http://lartc.org/howto. To save time, this section repeats all relevant parts of creating qdiscs that can be found in the online manual before giving more details on packet classification. The first part of the section describes how different queueing disciplines (qdiscs) can be attached to one outgoing network interface. The second part explains how packets can be classified into the schedulers based on packet properties such as the source or destination ip address header field.

2.1 tc qdiscs and classes

2.1.1 Terminology

Queueing Discipline (qdisc)	packet queue with an algorithm that decides when to send which packet
Classless qdisc	qdisc with no configurable internal subdivision
Classful qdisc	qdisc that may contain classes classful qdiscs allow packet classification
Root qdisc	a root qdisc is attached to each network interface either classful or classless
egress qdisc	works on outgoing traffic only egress qdiscs are considered in this manual
ingress qdisc	works on incoming traffic see the lartc manual for more detail
Class	classes either contain other classes, or a qdisc is attached
Filter	classification can be performed using filters

2.1.2 General Commands

Generate a root qdisc:
`tc qdisc add dev DEV handle 1:  root QDISC [PARAMETER]`
Generate a non root qdisc:
`tc qdisc add dev DEV parent PARENTID handle HANDLEID QDISC [PARAMETER]`
Generate a class:
`tc class add dev DEV parent PARENTID classid CLASSID QDISC [PARAMETER]`

DEV:	interface at which packets leave, e.g. eth1
PARENTID:	id of the class to which the qdisc is attached e.g. X:Y
HANDLEID:	unique id, by which this qdisc is identified e.g. X:
CLASSID:	unique id, by which this class can be identified e.g. X:Y see section 2.1.3
QDISC:	type of the qdisc attached, see table 2.1
PARAMETER:	parameter specific to the qdisc attached

qdisc	description	type
pfifo_fast:	simple first in first out qdisc	classless
TBF:	Token Bucket Filter, limits the packet rate	classless
SFQ:	Stochastic Fairness Queueing, devides traffic into queues and sends packets in a round robin fashion	classless
PRIO:	allows packet prioritisation	classful
CBQ:	allows traffic shaping, very complex	classful
HTB:	derived from CBQ, but much easier to use	classful

Table 2.1: queueing disciplines (more details are found in the lartc manual)

2.1.3 Building a qdisc Tree

By default each interface has one egress (outgoing) FIFO qdisc (queueing discipline). To be able to treat some packets different than others, a hierarchy of qdiscs can be constructed. Furthermore different kinds of qdiscs exist, each with different properties and parameters that can be tuned. To build a tree, a classful root qdisc has to be chosen. In this example HTB (Hierarchical Token Bucket) is used, since the other qdiscs are either classless or prioritize some traffic (PRIO) or are to complicated (CBQ). For information about HTB see: http://luxik.cdi.cz/~devik/qos/htb/. At the leaves a classless qdisc can be attached. In this example netem is used, the network emulation qdisc, which is explained in detail in chapter 3.

A tree as shown in figure 2.1 with three leaf qdiscs and one root qdisc can be created as follows:
First the default root qdisc is replaced:
`tc qdisc add dev eth1 handle 1:  root htb`
then one root class and three children classes are created:

```
tc class add dev eth1 parent 1:  classid 1:1 htb rate 100Mbps
tc class add dev eth1 parent 1:1 classid 1:11 htb rate 100Mbps
tc class add dev eth1 parent 1:1 classid 1:12 htb rate 100Mbps
tc class add dev eth1 parent 1:1 classid 1:13 htb rate 100Mbps
```

The parentid is equal to the classid of the respective parent. The children's class ids have to have the same major number (number before the colon) as

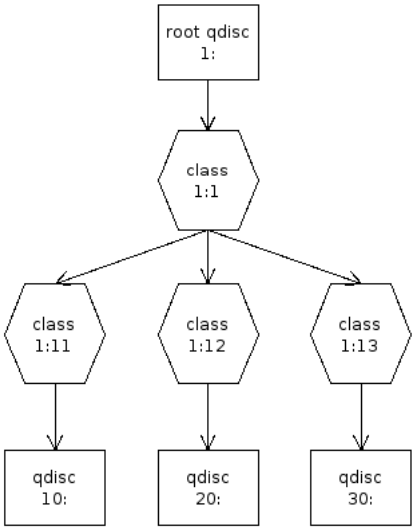

Figure 2.1: qdisc hierarchy.

their parent and a unique minor number (number after the colon). The qdisc is HTB with a maximal rate of 100 Megabits per second.
In the next step a qdisc is added to each class.

```
tc qdisc add dev eth1 parent 1:11 handle 10:  netem delay 100ms
tc qdisc add dev eth1 parent 1:12 handle 20:  netem
tc qdisc add dev eth1 parent 1:13 handle 30:  netem
```

The parent id is the id of the class to which the qdisc is attached. The handle is a unique identifier. Netem is chosen as a qdisc.
Unique numbers must just be unique within an interface.

2.1.4 Changing and Deleting qdiscs

The commands for changing and deleting qdiscs have the same structure as the add command. Qdisc parameters can be adapted using the change command. To change the 100ms delay from the qdisc with handle 10: (from the previous example) to 200ms the following command is used:

```
tc qdisc change dev eth1 parent 1:11 handle 10:  netem delay 200ms
```

To delete a complete qdisc tree only the root needs to be deleted:

```
tc qdisc del dev eth1 root
```

It is also possible to delete only a particular qdisc:

```
tc qdisc del dev eth1 parent 1:11 handle 10:
```

2.2 tc Filter Options

The ability of tc to filter packets is huge. Not only different filter types exist, but also the mechanism of referencing one particular filter is quite complex. The filter commands can be divided into two groups: simple filters and complex

filters. Simple filters are restricted in the way that they are referenced. Complex filters have identifier assigned and they have some more knowledge about the packets processed.

2.2.1 "Simple" Filter Commands

Simple filters allow the creation of filters that evaluate fields at a specified constant location. This implies that the IP header is assumed to be of constant size (20 bytes) and therefore must not include any options. Filter deletion can only be done for a complete priority band.

2.2.1.1 Command Structure

Add a filter:
`tc filter add dev DEV protocol PROTO parent ID prio PRIO FILTER match SELECTOR [FIELD] PATTERN MASK [at OFFSET] flowid FLOWID`
Delete filter:
`tc filter del DEV protocol PROTO parent ID prio PRIO`
Show filter:
`tc filter show dev DEV [protocol PROTO [parent ID [prio PRIO]]]`

DEV:	interface at which packets leave, e.g. eth1
PROTO:	protocol on which the filter must operate, e.g. ip, ipv6, arp, 802_3 see table 2.4 for all supported protocols
ID:	id of the class at which filtering should begin e.g. 1: to start at the root
PRIO:	determines the order in which filters are checked higher numbers → lower priority important: different protocols cannot have the same priority
FILTER:	specifies which filter is used, see section 2.2.1.2
SELECTOR:	depends on the filter, e.g. for u32: u32, u16, u8, ip, ip6
FIELD:	name of the field to be compared only for ip and ip6 selector
PATTERN:	value of the specified field (decimal or hexadecimal)
MASK:	indicates which bits are compared
OFFSET:	start to compare at the location specified by PROTO + OFFSET bytes, only for uX selectors
FLOWID:	references the class to which this filter is attached

2.2.1.2 Filter Overview

tc knows different filters for classifying packets, see table 2.2. The most interesting is the u32 filter which allows classification according to every value in a packet. This filter is discussed in more detail in the following sections.

filter	description
route:	bases the decision on which route the packet will be routed by
fw:	bases the decision on how the firewall has marked the packet
rsvp:	routes packets based on RSVP (ReSerVation Protocol, a reservation based extension to best effort service, not supported in the internet)
tcindex:	used in DSMARK qdisc (DSMARK is used for differentiated services, a priority based extension to the best effort service)
u32:	bases the decision on fields within the packet

Table 2.2: tc filter overview

2.2.1.3 u32 Filter

To simplify the filtering the u32 filter has different selectors:

- u32 filters according to arbitrary 32-bit fields in the packets
  the starting position is indicated with OFFSET (in bytes)
  OFFSET must be a multiple of 4. A mask of the same length
  is used to indicate the bits that should match.
  E.g. 0xFFFFFFFF: all 32 bits have to match
- u16 filters according to arbitrary 16-bit fields in the packets
  OFFSET has to be a multiple of 2.
  E.g. 0xFFFF: all 16 bits have to match, 0x0FF0: only bits 4 to 11 have to match
- u8 filters according to arbitrary 8-bit fields in the packets.
  E.g. 0xFF all 8 bits have to match, 0x0F only bits 4 to 8 have to match.
- ip bases decision on fields in the ipv4 and upper layer headers (for PROTO=ip)
  (see ipv4 traffic)
- ip6 bases decision on fields in the ipv6 and upper layer headers (for PROTO=ipv6)
  (see ipv6 traffic).

The desired values for OFFSET can be found by inspecting figure 2.2 and 2.3
for ipv4 and ipv6 packets respectively.

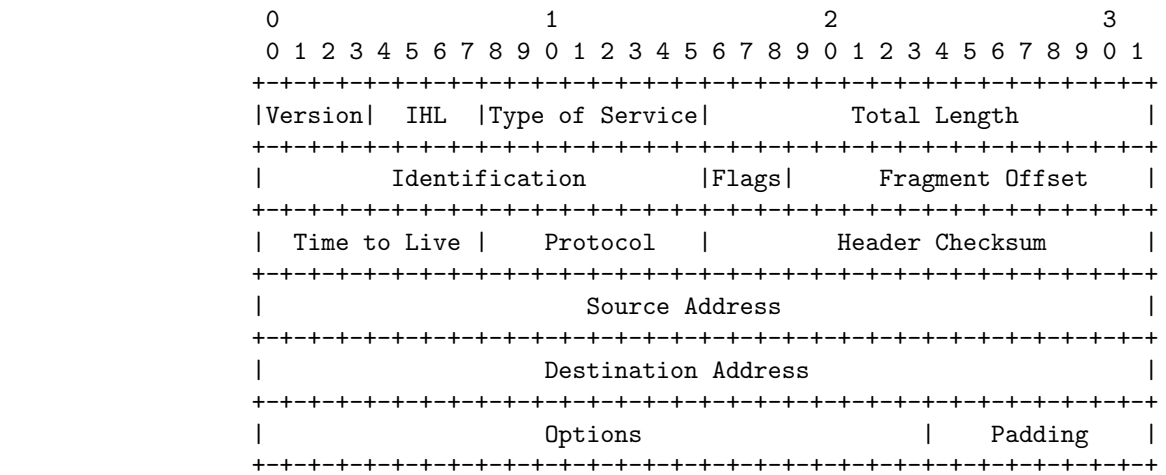

Figure 2.2: ipv4 header format

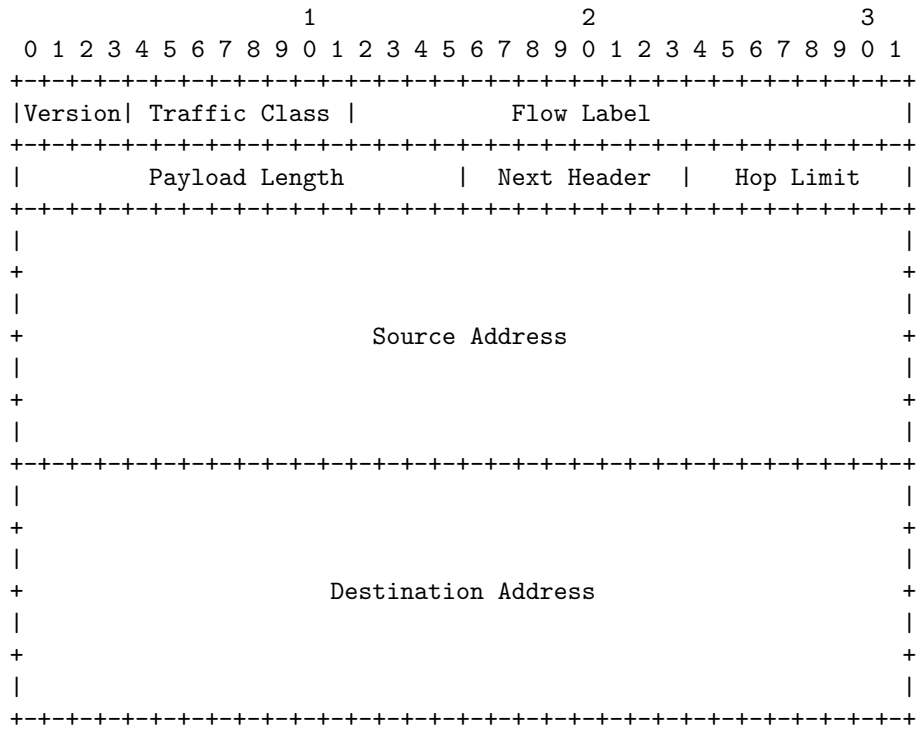

Figure 2.3: ipv6 header format

ipv4 Traffic
Simple filters assume the ipv4 header length to be constant (without options). Therefore incorrect results may be found if these filters are used for fields belonging to layer 4 (e.g. udp, tcp etc). The complex filters solve this problem by inspecting the ihl (internet header length) field (see section 2.2.3. For ipv4 traffic (PROTO=ip, SELECTOR=ip) the following fields are predefined:

field name	tc short	sample value	mask
IP source	src	10.0.0.2/24	na
IP destination	dst	10.0.1.2/24	na
IP header length	ihl	20	0xF
type of service	tos	10	0xFF
transport protocol (see table 2.3)	protocol	6	0xFF
packet is not fragmented	nofrag	na	na
packet is first (or only) fragment	firstfrag	na	na
don't fragment flag set	df	na	na
more fragments flag set	mf	na	na
udp/tcp source port	sport	5000	0xFFFF
udp/tcp dst port	dport	80	0xFFFF
icmp type	icmp_type	1	0xFF
icmp code	icmp_code	1	0xFF

Example for destination port filtering:
`tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 match ip dport`

protocol number	protocol
1	ICMP
2	IGMP
4	IP (encapsulation)
6	TCP
17	UDP
41	IPv6
58	IPv6-ICMP

Table 2.3: some protocol numbers as used in the ipv4 header. For a complete list see: www.iana.org/assignments/protocol-numbers

```
5000 0xffff flowid 1:11
```
Example for ip source address filtering:
```
tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 match ip src
10.0.0.2 flowid 1:11
```

ipv6 Traffic
The lartc manual says ipv6 filtering doesn't work, but I found some examples from people using it. I did not do any tests on my own.
For ipv6 traffic (PROTO=ipv6, SELECTOR=ip6) the following fields are predefined:

field name	tc short	possible values	mask
IP source	src	any ipv6 address	na
IP destination	dst	any ipv6 address	na
traffic class	priority	1	0xFF
next header	protocol	6	0xFF
flowlabel	flowlabel	1234	0xFFFFF
udp/tcp source port	sport	5000	0xFFFF
udp/tcp dst port	dport	80	0xFFFF
icmp type	icmp_type	1	0xFF
icmp code	icmp_code	1	0xFF

Example for destination port filtering:
```
tc filter add dev eth1 parent 1:  protocol ipv6 prio 2 u32 match ip6
dport 5000 0xffff flowid 1:11
```
Example for ip source address filtering:
```
tc filter add dev eth1 parent 1:  protocol ipv6 prio 2 u32 match ip6
src 2001:6a8:802:7::2 flowid 1:11
```

Other Traffic
There is a large list of accepted protocols (see table 2.4 for details). However, for all protocols other than ip and ipv6, no predefined fields are available. With a u32 selector arbitrary parts of a packet can be filtered. However, one has to know the structure of such a packet to place the "compare-pointer" to the correct location.

An Ethernet packet has the destination address in the first 6 bytes, followed by 6 bytes of source address and a 2 byte type field. Note that the source MAC address is already that of the router. To filter according to the MAC destination address one has to concatenate two u32 filters, since 6 bytes = 48 bits > 32 bits.

loop	pup	ip	irda	control
x25	arp	bpq	mobitex	tr_802_2
dec	dna_dl	dna_rc	ppptalk	localtalk
dna_rt	lat	cust	wan_ppp	ddcm
sca	rarp	atalk	snap	802_2
aarp	ipx	ipv6	all	ax25
802_3				

Table 2.4: Supported protocols for filtering

To match the Ethernet destination address 12:34:56:78:9A:BC two filters must be concatenated in the following manner:

```
tc filter add dev eth1 parent 1:  protocol 802_3 prio 3 u32 match u32
0x12345678 0xffffffff at 0 match u32 0x9abc0000 0xffff0000 at 4 flowid
1:11
```

2.2.2 Filtering Based on Multiple Criterions

Two possibilities exist to combine different filter rules. The logical AND restricts the packets on more than one field, the logical OR allows packets to have different values in one field.

Logical AND
Filtercriteria can be concatenated to allow a more specific filtering. To restrict packets on more than one field a filter with multiple match clauses can be used. To filter a packet according to its source ip address and its destination udp port the following statement can be used:

```
tc filter add dev eth1 protocol ip prio 1 u32 match ip protocol 17
0xff match ip dport 5000 0xffff match ip src 10.0.0.2 flowid 1:11
```

Logical OR
A class is allowed to have different filters. All packets that match one of these filters will be processed by the respective class. The filters are traversed according to their priorities and in the order they where created. All packets can be filtered based on destination ip address 10.0.0.2 and 10.0.1.2 to class 1:11 the following way:

```
tc filter add dev eth1 protocol ip prio 1 u32 match ip dst 10.0.0.2
flowid 1:11
tc filter add dev eth1 protocol ip prio 1 u32 match ip dst 10.0.1.2
flowid 1:11
```

Remember: different PRIO values must be used for different protocols!
To filter for an address range the common notation can be used:

```
tc filter add dev eth1 protocol ip prio 1 u32 match ip dst 10.0.1.0/24
```

which matches every packet going to an address in the range 10.0.1.0 to 10.0.1.255.
Filtering for a destination port range is done by adjusting the mask:

```
tc filter add dev eth1 protocol ip prio 1 u32 match ip dport 50000
0xff00
```

This matches every port between 49920 (0xC300) and 50175 (0xC3FF).

2.2.3 Complex Filter Commands

Complex filters make use of user defined handles and hash tables. A hashtable contains slots and the slots contain filter rules. This allows us to specify exactly one filter rule. The hash table memorises which protocol is used (e.g. ip). This allows the evaluation of the ihl (internet header length) field to get the correct start location of the upper layer protocol. In this section only ip traffic and u32 filtering is considered. Before adding a filter rule some prearrangements have to be taken.

2.2.3.1 Command Structure

First a classifier has to be created at the root. Each time a new PRIO value is introduced a new classifier id (CLASSIFIERID) is created. It starts with the value 800 and is incremented for each new PRIO.

```
tc filter add dev eth1 parent 1:0 prio PRIO protocol ip u32
```

A hash table has to be created, this allows the usage of the nexthdr+OFFSET option.

```
tc filter add dev eth1 parent 1:0 prio PRIO handle ID: u32 divisor
1
```

The hash table must be linked to the correct priority classifier:

```
tc filter add dev eth1 parent 1:  protocol ip prio PRIO u32 ht CLASSIFIERID::
match u8 0 0 offset at 0 mask 0x0f00 shift 6 link ID:
```

The actual filter is attached:

```
tc filter add dev eth1 protocol ip parent 1:0 prio PRIO handle 0xHANDLE
u32 ht ID: match SELECTOR [FIELD] PATTERN MASK [at OFFSET | nexthdr+OFFSET]
flowid FLOWID
```

Delete a specific filter:

```
tc filter del dev eth1 protocol ip parent 1:0 prio PRIO handle ID::HANDLE
u32
```

with:

PRIO:	Priority, determines the order in which filters are checked
ID:	Identifier of the hash table
HANDLE:	handle of a specific rule. If no handle is specified tc assigns one starting at 800
SELECTOR:	specifies the filter see section ipv4 and table table 2.5 note: udp and tcp are the same filter -> check the protocol separately
FIELD:	name of the header field to be compared
PATTERN:	value of the specified field
MASK:	indicates which bits are compared
OFFSET:	starts to compare at the ip header + OFFSET bytes
nexthdr+OFFSET:	starts to compare at the upper layer header + OFFSET bytes
FLOWID:	references the class to which this filter is attached

It is also possible to divide a hashtable into slots. This helps to find the correct filter rule fast. Some examples are found in the lartc manual. The syntax is as follows:

Create hash table:

```
tc filter add dev eth1 parent 1:0 prio PRIO protocol ip handle ID:
u32 divisor SLOTS
```

selector	field	description	filter
udp	src	udp source port	0xffff
	dst	udp destination port	
tcp	src	tcp source port	0xffff
	dst	tcp destination port	
icmp	type	icmp type field	0xff
	code	icmp code field	

Table 2.5: Filter selectors for complex filters only

Add a filter rule:
```
tc filter add dev eth1 protocol ip parent 1:0 prio PRIO handle 0xHANDLE
u32 ht ID:SLOTNR: match SELECTOR PATTERN MASK [at OFFSET | nexthdr+OFFSET]
flowid FLOWID
```
Delete a filter rule:
```
tc filter del dev eth1 protocol ip parent 1:0 prio PRIO handle ID:SLOTNR:HANDLE
u32
```
with:
SLOTNR: slot to which the rule is attached
SLOTS: number of slots in one hash table
I was not able to combine multiple slots with a working nexthdr function.

2.2.3.2 Examples

For all examples a tree as in figure 2.1 is assumed.
Setup the hash table for eth1, priority 1:
```
tc filter add dev eth1 parent 1:0 prio 1 protocol ip u32
tc filter add dev eth1 parent 1:0 prio 1 handle 1:  u32 divisor 1
tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 ht 800::  match
u8 0 0 offset at 0 mask 0x0f00 shift 6 link 1:
```

Filter all traffic that leaves eth1 and has its udp source port equal to 50000 to class 1:11. NOTE: the udp src and tcp src filter are EXACTLY the same. To filter only UDP packets the transport protocol field in the ip header must be examined.
```
tc filter add dev eth1 parent 1:0 prio 1 u32 ht 1:  match udp src 50000
0xffff match ip protocol 17 0xff flowid 1:11
```

Select tcp/telnet traffic to 193.233.7.75 and direct it to class 1:11 (telnet uses port 23 = 0x17). The handle 123 is assigned.
```
tc filter add dev eth1 parent 1:0 prio 1 handle 0x123 u32 ht 1: match
ip dst 193.233.7.75  match tcp dst 0x17 0xffff  flowid 1:11
```

Delete the rule above:
```
tc filter del dev eth1 parent 1:0 prio 1 handle 1::123 u32
```

Chapter 3

netem

This chapter describes the usage of netem, the linux network emulator module. netem is part of each standard linux kernel 2.6.7 and later. However some features are only available in kernel version 2.6.16 and later. The standard part of netem allows packet handling according to statistical proberties. In addition a trace mode has been written though it is not part of the standard kernel. This trace mode allows the specification of an independent value for each packet to be processed. The first part of this chapter describes the general configuration of netem whereas the second part discusses the generation of tracefiles.

3.1 Basic Operation of netem

netem provides functionality for testing protocols by emulating network properties. netem can be configured to process all packets leaving a certain network interface.

Four basic operations are available:

delay	delays each packet
loss	drops some packets
duplication	duplicates some packets
corruption	introduces a single bit error at a random offset in a packet

3.1.1 Original netem

In standard mode the packet delay can be specified by a constant, a variation a correlation and a distribution table. Packet loss, duplication and corruption can be modelled using a percentage and a correlation. An online manual is available at http://linux-net.osdl.org/index.php/Netem.

3.1.2 Trace Control for Netem TCN

For each packet the operation can be specified completely independent from the other packets. By monitoring some internet traffic a tracefile can be produced. This tracefile contains the real characteristics of the internet traffic at the time of monitoring. The tracefile can be used as the source to modify the behavior

(delay, duplication, loss, corruption) of a packet. The generation of tracefiles is discussed in section 3.4.

3.2 Installation

As a first step the source code of the netem kernel module and the source code of tc has to be patched.
Switch to the source code directory and type:
`cat /path/to/patch/trace.patch | patch -p1`
This has to be done once for the kernel module and once for tc.

In a next step the linux kernel must be configured and built.
The linux kernel has an internal timer. The frequency of this timer defines the precision with which netem sends packets. Since linux kernel 2.6.13 the frequency of the timer can be set at compile time. The maximum value is 1000, this leads to a timerinterrupt every 1ms. The default value is 250 for kernel versions 2.6.13 and later. For the use of netem it is important to set the timer frequency to 1000 since otherwise the resolution of netem will be 4ms instead of the possible 1ms! The timer frequency can be set after executing make menuconfig .

```
Processor type and features --->
   Timer frequency (250HZ) --->
      ( ) 100 HZ
      ( ) 250 HZ
      (X) 1000 HZ
```

3.3 Using netem

Since netem is a qdisc as described in chapter2.1 it has to be configured with tc, the traffic control tool of linux. The simplest netem command adds a constant delay to every packet going out through a specific interface:
`tc qdisc add dev eth0 root netem delay 100ms`
The command to add a trace file involves a few parameters:
`tc qdisc add dev eth0 root netem trace FILE LOOP [DEFAULTACTION]`
with:

FILE:	the tracefile to be attached
LOOP:	how many times the trace file is traversed 0 means forever
DEFAULTACTION:	if no delay can be read from the tracefile the default value is taken 0: no delay, 1: drop the packet, default is 0

The following example adds the tracefile "testpattern.bin" to the root qdisc of eth1. The tracefile is repeated 100 times and then all packets are dropped.
`tc qdisc add dev eth1 root netem trace testpattern.bin 100 1`

3.4 Generation of Tracefiles

The trace file contains the packet actions to be performed. Some example tracefiles are available from http://tcn.hypert.net. There are .txt files that contain the original delay values as measured with network probing and .bin files that contain the netem compatible delay values.

```
To create your own trace file some tools are provided:
headgen
generates one packet action value for the trace file.
It takes the type and the delay as an argument.
e.g. headgen <head> <delay>
with <head> = 0 -> delay only
              1 -> drop packet
              2 -> duplicate packet
              3 -> corrupt packet
     <delay> = delay value in microseconds

txt2bin
converts the output form headgen to a netem readable form
e.g txt2bin <inputfile.txt> <outputfile.bin>
    with <inputfile.txt> = file with values as obtained by headgen, one per line
         <outputfile.bin> = file that must be given as argument to netem trace

bin2txt
takes a netem compatible file and converts it to the txt format.
The output is printed to the shell.
If you want to save the output in a file use a pipe.
e.g. bin2txt <inputfile.bin> [u]
     with <inputfile.bin> = Netem compatible file
          u = optional, output in understandable format e.g head and delay are
              reported separately
usage with a pipe:
     bin2txt <inputfile.bin> [u] | <outputfile.txt>

Example
A. The following values have been measured:
1. 1ms delay
2. 2ms delay
3. packet loss
4. 1ms delay
5. 1ms delay and duplication
6. 2ms delay
7. 4ms delay and corruption
8. 3ms delay

B. Obtain the corresponding values for the .txt file and write them in a file
   e. g. myvalues.txt
```

```
# headgen 0 1000   -> 1000
# headgen 0 2000   -> 2000
# headgen 1 0      -> 536870912
# headgen 0 1000   -> 1000
# headgen 2 1000   -> 1073742824
# headgen 0 2000   -> 2000
# headgen 3 4000   -> 1610616736
# headgen 0 2000   -> 2000

myvalues.txt:
1000
2000
536870912
1000
1073742824
2000
1610616736
2000

C. Generate the netem compatible file:
# txt2bin myvalues.txt myvalues.bin
The file myvalues.bin can be used as the trace argument for tc.

D. If you want to see what was originally in your file use bin2txt:
# bin2txt myvalues.bin
1000
2000
536870912
1000
1073742824
2000
1610616736
2000

# bin2txt myvalues.bin u
00000000000000000000001111101000   head:  0  delay:     1000 value:       1000
00000000000000000000011111010000   head:  0  delay:     2000 value:       2000
00100000000000000000000000000000   head:  1  delay:        0 value:  536870912
00000000000000000000001111101000   head:  0  delay:     1000 value:       1000
01000000000000000000001111101000   head:  2  delay:     1000 value: 1073742824
00000000000000000000011111010000   head:  0  delay:     2000 value:       2000
01100000000000000000111110100000   head:  3  delay:     4000 value: 1610616736
00000000000000000000011111010000   head:  0  delay:     2000 value:       2000
```

3.5 Statistics

netem trace mode collects statistics about different events. To dump this data use:

```
$ cat /proc/netem/stats
```

Note that only stats for flows with packet counter greater than zero are shown. The dump only works with a loaded netem kernel module and only gives an output if at least one flow has a packet counter greater than zero.
Example:

```
# cat /proc/netem/stats
 Statistics for Flow 0
 ----------------------
 Packet count:               158299
 Packets ok:                 158299
 Packets with normal Delay:  158299
 Duplicated Packets:         0
 Drops on Request:           0
 Corrupted Packets:          0
 No valid data available:    0
 Uninitialized access:       0
 Bufferunderruns:            0
 Use of empty Buffer:        0
 No empty Buffer:            0
 Read behind Buffer:         0
 Buffer1 reloads:            8
 Buffer2 reloads:            8
 Switches to Buffer1:        8
 Switches to Buffer2:        8
 Switches to empty Buffer1:  0
 Switches to empty Buffer2:  0
```

The statistic for one flow is reset every time a tracefile is changed for that flow. The complete statistic can be reset manually by typing:

```
$ echo > /proc/netem/stats
```

Use

```
$ watch -n1 cat /proc/netem/stats
```

for continuous statistics. End with CTRL-C.

Chapter 4

Examples

The first part of this chapter describes, how computers must be configured to allow a network emulation. The second part gives some ready to use examples, that show all relevant features of packet filtering.

4.1 Hardware Configuration

To be able to test two network devices a PC with netem and with at least two network cards is required (see figure 4.1).

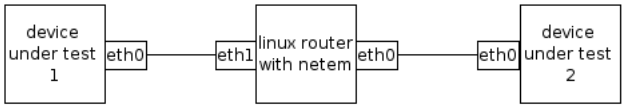

Figure 4.1: Network configuration

Example IP configuration:

- linux router
  - IP address eth1: 10.0.1.1
  - IP address eth2: 10.0.0.1
  - netmask: 255.255.255.0 (for both devices)
- device under test 1
  - IP address eth0: 10.0.1.2
  - netmask: 255.255.255.0
  - gateway: 10.0.1.1
- device under test 2
  - IP address eth0: 10.0.0.2
  - netmask: 255.255.255.0
  - gateway: 10.0.0.1

Allow routing:

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

make it permanent by adding the line net/ipv4/ip_forward=1 to the file /etc/sysctl.conf
Set IP addresses:
`ifconfig eth0 10.0.0.2`
Set netmask:
`ifconfig eth0 netmask 255.255.255.0`
Set default gateway:
`route add default gw 10.0.0.1`
Set MAC address (normally not needed)
`ifconfig eth0 hw ether 00:01:6C:E9:38:59`
These changes can be saved in a shell script and executed after each reboot. In Debian distributions the configuration can also be written to the file /etc/network/interfaces. Suggested contents:
auto eth0
iface eth0 inet static
address 10.0.0.2
netmask 255.255.255.0
gateway 10.0.0.1
hwaddress ether 00:01:6C:E9:38:59
This script is automatically carried out upon reboot. The correct execution can be checked with `ifconfig eth0`. If the result isn't the expected one, one can type in the shell: `ifup eth0`.

4.2 Qdiscs, filter and netem

Example 1: Outgoing Interface
All packets leaving eth1 will be processed by netem. The tracefile "testpattern1.bin" is read once and afterwards all packets are dropped.

```
tc qdisc add dev eth1 root netem trace testpattern1.bin 1 1
```

Example 2: Tree
In this example all packets coming from the 10.0.1.0/24 network will be processed with the file "testpattern1.bin" and all packets coming form the 10.0.2.0/24 network will be processed with the file "testpattern2.bin". The configuration is shown in figure 4.2. The htb parameter r2q is set to 1700 to suppress some warnings. Ceil = 100Mbps assures that all classes get the bandwith available.

```
tc qdisc add dev eth1 handle 1:  root htb r2q 1700

tc class add dev eth1 parent 1:  classid 1:1 htb rate 100Mbps ceil
100Mbps

tc class add dev eth1 parent 1:1 classid 1:11 htb rate 100Mbps

tc class add dev eth1 parent 1:1 classid 1:12 htb rate 100Mbps

tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 match ip src
10.0.1.0/24 flowid 1:11

tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 match ip src
```

```
10.0.2.0/24 flowid 1:12

tc qdisc add dev eth1 parent 1:11 handle 10:  netem trace testpattern1.bin
0 1

tc qdisc add dev eth1 parent 1:12 handle 20:  netem trace testpattern2.bin
100 0
```

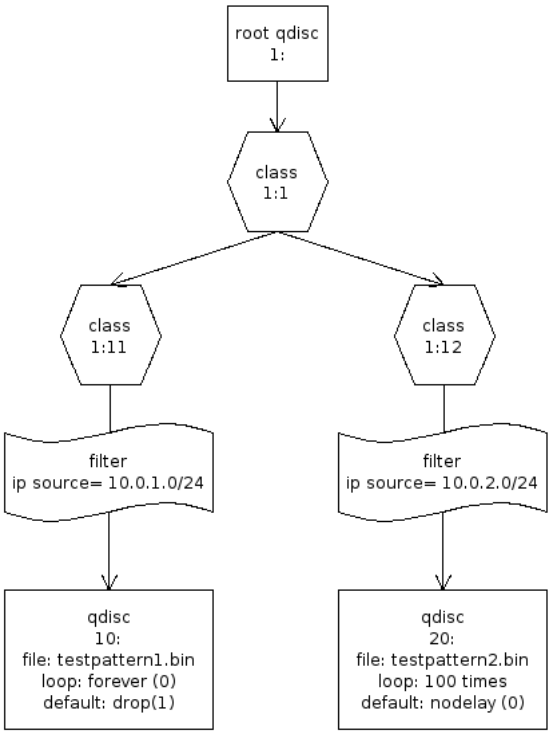

Figure 4.2: Example setup

Example 3: Complex Filter
Filter on eth1 all tcp traffic to 10.0.2.172 and port 80 or 20/21 (http and ftp traffic) that comes from subnet 10.0.1.0/24 to class 1:20. Treat packets as described in the file test.bin. This file is repeated for ever and the default action is dropping packets.
Creating qdiscs and classes:

```
tc qdisc add dev eth1 handle 1:  root htb r2q 1700

tc class add dev eth1 parent 1:  classid 1:1 htb rate 100Mbps ceil
100Mbps
```

```
tc class add dev eth1 parent 1:1 classid 1:20 htb rate 100Mbps

tc qdisc add dev eth1 parent 1:20 handle 12:  netem trace test.bin
0 1
```

Create filter:

```
tc filter add dev eth1 parent 1:0 prio 1 protocol ip u32

tc filter add dev eth1 parent 1:0 prio 1 handle 1:  u32 divisor 1

tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 ht 800::  match
u8 0 0 offset at 0 mask 0x0f00 shift 6 link 1:

tc filter add dev eth1 parent 1:0 prio 1 u32 ht 1:  match tcp dst 80
0xffff match ip protocol 6 0xff match ip src 10.0.1.0/24 match ip dst
10.0.2.172 flowid 1:20

tc filter add dev eth1 parent 1:0 prio 1 u32 ht 1:  match tcp dst 20
0xfffe match ip protocol 6 0xff match ip src 10.0.1.0/24 match ip dst
10.0.2.172 flowid 1:20
```

Chapter 5

Useful Links

- Linux advanced routing and traffic control howto: http://lartc.org/howto
- HTB manual: http://luxik.cdi.cz/ devik/qos/htb
- Netem: http://linux-net.osdl.org/index.php/Netem
- Trace control for netem: http://tcn.hypert.net