INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Entwurf und Implementierung eines Leitungssimulators auf OSI-Schicht 2

Boris Lohner

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Markus Garschhammer

Abgabetermin: 27. Oktober 2003

INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Entwurf und Implementierung eines Leitungssimulators auf OSI-Schicht 2

Boris Lohner

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Markus Garschhammer

Abgabetermin: 27. Oktober 2003

Hiermit versichere ich, dass ich die vorliegende Ausarbeitung des Fortgeschrittenenpraktikums selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 27. Oktober 2003

. . . .
(Unterschrift des Kandidaten)

Zusammenfassung

Im Rahmen dieses Fortgeschrittenenpraktikums wurde ein Simulator für Übertragungswege auf OSI-Schicht 2 erstellt, der die dienstgüte-beeinträchtigenden Eigenschaften einer Netzwerkverbindung nachbildet: beschränkte Bandbreite, Verlust einzelner Frames, Duplizierung einzelner Frames und schwankende Übermittlungszeit einzelner Frames.

Der auf einem Linux-PC programmierte Simulator agiert auf Ethernet-Schicht (OSI-Layer 2) und kann somit beliebig in Netzwerke integriert werden, ohne dass Konfigurationsänderungen an den Versuchsrechnern notwendig werden, wie z.B. das Ändern von IP-Adressen oder Routing-Tabellen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

In zunehmendem Maße wird Software entwickelt und eingesetzt, die definierte Ansprüche an die Qualität der Datenübertragung (Quality of Service) über das Internet stellt. So soll z.B. bei einer Sprachübertragung über ein Rechnernetz gemäß den Forderungen der International Telecommunication Union (ITU) in ihrer Empfehlung G.114 die Verzögerung des Sprachsignals 150ms nicht überschreiten[ITU 96].

Die jeweiligen Anforderungen der Anwendung für ein zufriedenstellendes Funktionieren lassen sich durch theoretische Überlegungen oder praktische Messungen bestimmen. Bei Messungen in realen Netzen sind die Bedingungen aber nicht reproduzierbar, da die Dienstgüte des Übertragungsweges vom sich fortwährend ändernden Verhalten aller Netzteilnehmer beeinflusst wird.

Daher wurde im Rahmen dieses Fortgeschrittenenpraktikums ein Simulator für Übertragungswege in Rechnernetzen erstellt, der die qualitätseinschränkenden Eigenschaften eines Übertragungsweges (engl: link) reproduzierbar nachbildet:

- beschränkte Bandbreite
- Verlust einzelner Datenpakete
- Duplizierung einzelner Datenpakete
- schwankende Übermittlungszeit einzelner Pakete

Reproduzierbare Bedingungen im Netz sind aber nicht nur notwendig, um das Verhalten von Anwendungen zu testen, die hohe Ansprüche an die Dienstgüte stellen, sondern auch, um Messwerkzeuge zur Bestimmung der Dienstgüte zu entwickeln und zu kalibrieren.

So wurde der im Rahmen dieses Fortgeschrittenenpraktikums entwickelte Leitungssimulator auf dem Rechner *projekt7* von Cécile Neu im Rahmen ihrer Diplomarbeit “Design and implementation of a generic quality of service measurement and monitoring architecture”[Neu 02] wie in Abbildung 1.1 eingesetzt.

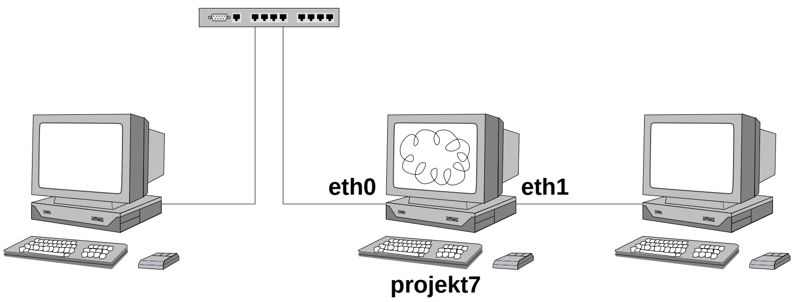

Abbildung 1.1: Einsatz des Simulators

Nach einer kurzen Einleitung beleuchten wir in Kapitel zwei die zu simulierenden Störeffekte und stellen ein paar Vorab-Überlegungen an.

Das dritte Kapitel wägt zunächst verschiedene Schnittstellen zum Empfangen und Senden von Daten auf dem Netzwerk ab, erklärt kurz die zum Verzögern verwendete Datenstruktur und erläutert abschließend die Funktionsweise des Simulators.

Im vierten Kapitel wird die konkrete Umsetzung dargestellt. Nach einer Beschreibung der Interaktion mit dem Netzwerk wird das präzise Erzeugen der Delays als Kernproblem beschrieben und - nach dem Verwerfen aller anderen Möglichkeiten - schließlich durch Busy-Waiting gelöst.

Zuletzt halten wir noch einen kurzen Ausblick auf mögliche Erweiterungen.

Kapitel 2

Anforderungsanalyse

2.1 Aufgabenstellung

In diesem Fortgeschrittenenpraktikum sollte auf Basis eines Linux-PCs ein Simulator entwickelt werden, der die in der Einleitung umrissenen Quality of Service (QoS)-Eigenschaften eines Local Area Network (LAN) oder Wide Area Network (WAN) auf IP-Ebene (OSI-Schicht 3) nachbildet.

Idealerweise soll der Simulator nicht nur künstlich schlechte Übertragungswege simulieren, sondern eine Nachbildung eines realen Leitungsverhaltens ermöglichen und dabei möglichst nahe das anderweitig gewonnene QoS-Profil dieser realen Leitung nachbilden.

Wir betrachten zunächst, welche Effekte QoS-beeinträchtigend wirken können, untersuchen kurz die Erfüllbarkeit der Aufgabe und stellen abschließend einige technische Überlegungen an.

2.2 Zu simulierende Effekte

Bei der Datenübertragung über ein Rechnernetz können folgende Effekte auftreten, die wir in der Simulation nachbilden müssen:

Paketverlust ist das Verlorengehen eines Paketes auf dem Weg von Sender zum Empfänger, so dass es nicht am Ziel eintrifft. Mögliche Gründe für verworfene Pakete sind unter anderem überfüllte Warteschlangen an Zwischenstationen (Router und Switche) oder unerkannte Kollisionen mit anderen Sendern auf dem gleichen Medium.

Paketverdopplung bedeutet, dass der Empfänger zeitlich versetzt zwei identische Pakete empfängt, obwohl der Sender nur ein korrespondierendes Paket versandt hat. Das Verdoppeln von Datenpaketen ist wohl meist auf Konfigurationsfehler zurückzuführen und tritt somit im Gegensatz zum Paketverlust selten auf. Dennoch soll der Simulator auch die Möglichkeit zum gezielten Verdoppeln von Paketen bieten.

Paketverzögerung/Delay ist der Zeitunterschied zwischen dem Abschicken des Paketes und dem Empfang des Paketes. Die Paketverzögerung kann je nach Netzlast schwanken. Diese Schwankungen werden oft als Jitter bezeichnet.

Paketvertauschungen treten auf, wenn ein Paket ein anderes überholt und die Reihenfolge der Pakete dadurch durcheinander gerät. Die Paketvertauschung ist kein separat zu simulierender Effekt, sondern ergibt sich aus dem Zusammenspiel eines entsprechenden Verzögerungsprofils zusammen mit einem Datenstrom: Wenn z.B., wie in Abbildung 2.1 dargestellt, ein Paket B zum Zeitpunkt t=1ms in den Simulator eintreten wird und um 7ms zusätzlich verzögert wird, nach einer weiteren ms aber bereits das nächste Paket C zum Zeitpunkt t=2ms eintrifft und nur um 3ms verzögert wird, so verlässt Paket C den Simulator zum Zeitpunkt t=5ms, und hat somit Paket B überholt, da dieses erst zu t=8ms weitergesendet wird.

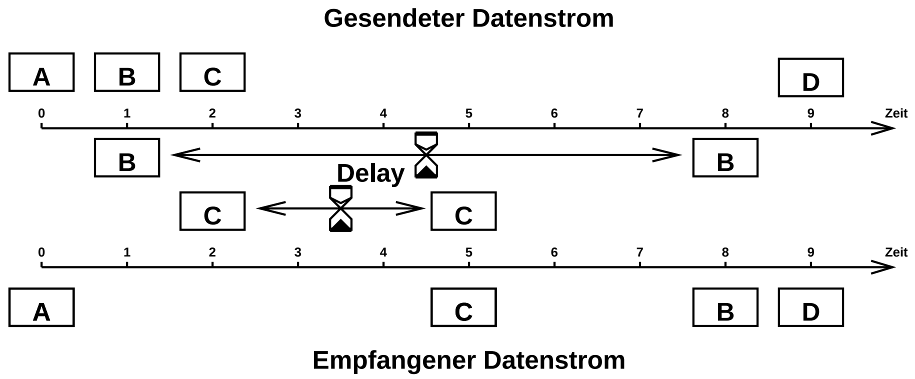

Abbildung 2.1: Paketvertauschung als Konsequenz schwankender Paketverzögerung

In der Realität wird ein hoher Delay allerdings meist durch "Stau" an einem überlasteten Router verursacht, so dass es keine Möglichkeit für später eintreffende Pakete gibt, die bereits wartenden Pakete zu überholen.

Daher sollte der Simulator in der Lage sein, eine strenge First In First Out (FIFO) Reihenfolge einzuhalten, indem später eintreffende Pakete entsprechend länger verzögert, auch wenn dadurch das gewünschte Verzögerungsprofil möglicherweise leidet.

Werden mehrere unterschiedliche Übertragungswege parallel genutzt (z.B. Kanalbündelung bei ISDN oder Etherchannelling von zwei oder mehr Ports zur Steigerung der Bandbreite zwischen zwei Switchen) so kann es dennoch vorkommen, dass Pakete einander überholen.

Das Erzwingen einer strengen FIFO Reihenfolge sollte also abschaltbar sein.

Beschränkte Bandbreite ist eine weitere QoS-beeinträchtigende Eigenschaft einer Leitung, aufgrund der in einem vorgegebenen Zeitraum nur endliche Datenmengen übertragen werden können. Die beschränkte Bandbreite ist kein Effekt, sondern die eigentliche Ursache von Paketverzögerung und infolge überlaufender Warteschlangen auch von Paketverlust.

2.3 Zwei mögliche Betriebsarten zur Vermeidung unvereinbarer Anforderungen

Die Beschränkung der Bandbreite zusätzlich zur Angabe eines Leitungsprofils mit Verzögerungen und Verlustwahrscheinlichkeiten kann zu unlösbaren Anforderungen führen, wenn zu viele Daten übertragen werden sollen.

Die Simulation einer Beschränkung der Bandbreite zusammen mit einem vorgegebenen Leitungsprofil aus Verzögerungswerten und Verlustereignissen kann unter Umständen nicht exakt erfüllbar sein: Soll beispielsweise eine verlustfreie Leitung mit 64kBit/s und konstantem Delay von 30ms simuliert werden, so führt die Einspeisung eines mehrere MByte langen Datenstroms mit 128kBit/s Übertragungsrate am Simulatoreingang zwangsläufig zur Verletzung mindestens einer der drei Vorgaben:

- Überschreitung der vorgegebenen Bandbreitenbeschränkung unter Vermeidung zusätzlicher Paketverluste oder zusätzlicher Verzögerung (Im Beispiel würde der Datenstrom wird mit seiner Eingangsrate von 128kBit/s um 30ms verzögert durchgereicht und die Übertragungsrate durch den Simulator hindurch würde 128kBit/s statt der vorgegebenen 64kBit/s betragen)

- Zusätzliche Paketverluste bei Einhaltung der Bandbreite und des Verzögerungsprofils (Im Beispiel könnte der Simulator jedes zweite Paket verwerfen und die verbleibende Hälfte der Pakte um 30ms verzögert weiterleiten, womit die vom Simulator geforderte Bandbreite von 64kBit/s eingehalten würde, aber die geforderte Verlustfreiheit verletzt wäre)
- Zusätzliche Paketverzögerung bei Einhaltung der Bandbreite und Verlustraten (Im Beispiel könnte der Simulator den Datenstrom puffern und mit 64kBit/s weitersenden, was mit zunehmender Dauer zu steigenden Verzögerungen der Pakete führen würde. So würde die geforderte Verlustfreiheit gewahrt und die Bandbreitenbeschränkung eingehalten, aber dafür ein anderes Verzögerungsprofil erzeugt als verlangt wurde)

Wir entschieden uns für zusätzliche Verzögerungen unter Einhaltung der Bandbreite und der Verlustrate. Dieses Vorgehen ist allerdings unrealistisch, da die Pakete im Simulator aufgrund einer fehlenden Längenbeschränkung der Warteschlange quasi unbegrenzt aufgehoben werden, wohingegen in realen Netzen hohe Verzögerungen meist durch hohes Verkehrsaufkommen entstehen und somit die Warteschlangen deutlich schneller überlaufen, so dass die Pakete letztlich verloren gehen.

Um derartige Konflikte zu vermeiden, empfiehlt es sich, aus zwei verschiedenen Anwendungsfällen den geeigneten auszuwählen:

Reproduktion eines Verzögerungsprofils Hierzu ist es notwendig, dass auch der eingehende Datenstrom in seinem Zeitverhalten reproduziert wird. Meist wird man mit einem Testdatenstrom eine Leitung durchmessen, daraus ein Verzögerungsprofil erstellen und mit diesem und dem reproduzierten Datenstrom das gemessene Verhalten nachbilden. Eine Beschränkung der Bandbreite ist in diesem Szenario nicht notwendig und würde das gewünschte Verzögerungsprofil nur stören.

Simulation einer schmalbandigen Leitung Geht es dagegen nur um die Simulation einer Leitung mit geringer Bandbreite für unbekannte Datenströme, wird man die Bandbreite auf das gewünschte Maß beschränken. Ein Datenstrom der die vorgegebene Bandbreite nicht überschreitet wird quasi ohne Verzögerung den Simulator passieren (TODO Anhang Datenblatt Nullverzögerung). Wenn hierzu eine zusätzliche Verzögerung gewünscht wird, so gibt man nur einen festen Wert an, aber kein kompliziertes Profil aus etlichen hundert von Werten, da der eigentlich gewünschte Effekt durch die begrenzte Bandbreite auftreten wird, sobald der eingehende Datenstrom die vorgegebene Bandbreite überschreitet.

2.4 Richtung der Simulation

Es gibt zwei Möglichkeiten, wie der Simulator das Netzwerkverhalten beeinträchtigen kann: bidirektional und unidirektional.

Die Beeinträchtigung in beiden Richtungen erscheint zunächst als die realistischere, da in realen Netzen auch oft beide Richtungen ähnlich gestört sind. Aber das Verhalten des Simulators ist bei einem derartigen Ansatz höchst undurchsichtig, da der sichtbare Effekt auf die Verzögerung einer Antwort die Kombination zweier Effekte ist: Die Störung der Anfrage und die Störung der Reaktion.

Daher haben wir uns entschieden, den Simulator nur in einer Richtung wirken zu lassen. Dadurch erzielen wir ein besser durchschaubares Verhalten, das sich leichter reproduzieren lässt und somit den Anforderungen nach *reproduzierbarer* Nachbildung eines Netzwerks besser entspricht.

2.5 Abschließende technische Überlegungen

In Abschnitt 2.2 haben wir erkannt, dass der Simulator zur Nachbildung eines Übertragungsweges Pakete verschieden lange verzögern, duplizieren und verwerfen können soll.

Wir müssen also in der Lage sein

- Pakete vom Netz zu empfangen
- Pakete auf dem Netz zu versenden
- Pakete in einer geeigneten Datenstruktur für verschieden lange Zeiten zwischenzuspeichern

Als Plattform stand ein Linux PC zur Verfügung, der mit zwei zusätzlichen Netzwerkkarten ausgerüstet wurde. Ein Netzwerkinterface stand dem Simulator für seine eigene Netzanbindung zum Hausnetz des NM-Lehrstuhls zur Verfügung und nahm nicht an der Simulation teil, so dass ein ständiges Umkonfigurieren einer Netzwerkkarte vom Simulatorbetrieb auf Internetanbindung und zurück entfallen konnte (Abbildung 2.2).

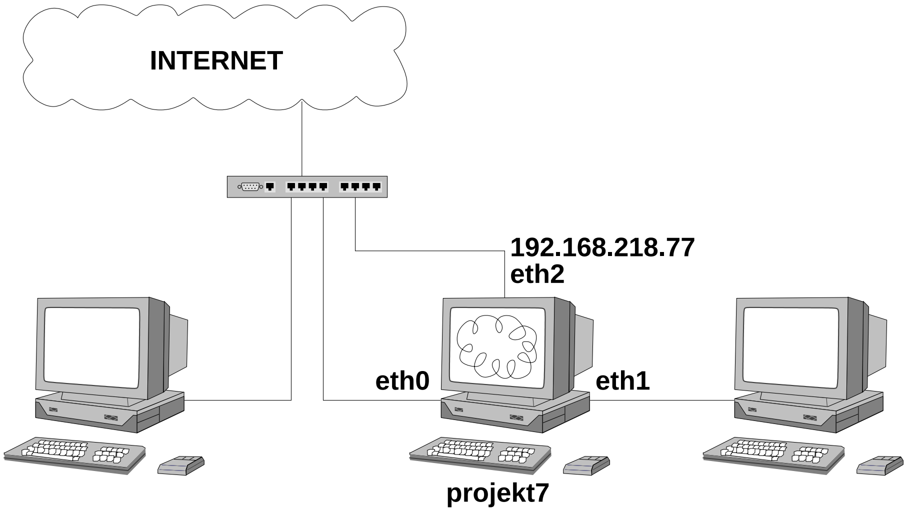

Abbildung 2.2: Von der Simulation unabhängige Internetanbindung des Simulators über separate dritte Netzwerkkarte

Auch konnte der Simulator so als Versuchsclient eingesetzt werden, ohne die Konfiguration der Simulationssoftware für diesen Spezialfall abzuändern (Abbildung 2.3).

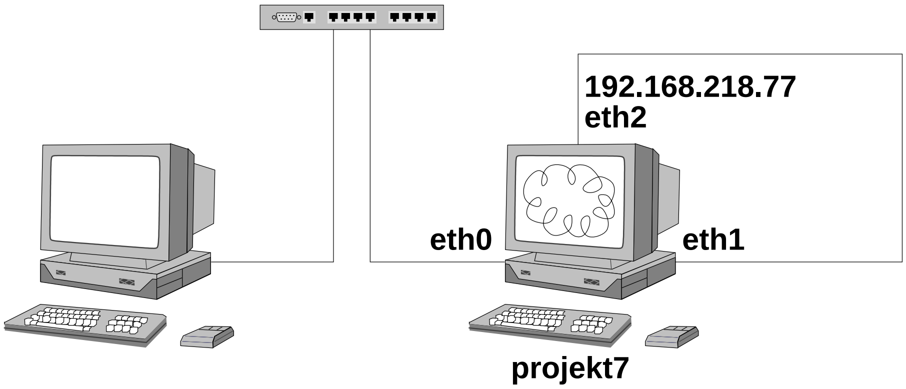

Abbildung 2.3: Einsatz von *projekt7* sowohl als Leitungssimulator als auch als Messstation

Kapitel 3

Design

Wie wir im vorigen Kapitel gesehen haben, benötigen wir Wege um Pakete vom Netz empfangen und erneut versenden zu können. Diese Nähe zum System legt uns C als Programmiersprache nahe. Außerdem benötigen wir eine geeignete Datenstruktur zur Zwischenspeicherung der künstlich zu verzögernden Frames.

Daher fiel die Wahl der Programmiersprache auf C++, da es die gesamte Funktionalität der Sprache C zur Verfügung stellt und somit ein direkter Zugriff auf das Socket-API des Linux Betriebssystems möglich ist und darüberhinaus durch die objektorientiert aufgebaute Standard Template Library (STL) über eine reichhaltige Klassenbibliothek verfügt, aus der wir die Container-Klassen zum Zwischenspeichern der Pakete verwenden werden.

Wir haben uns aus Gründen der Einfachheit dafür entschieden, den Simulator auf Ethernet-Schicht (OSI Layer 2) als Repeater agieren zu lassen. Daraus ergeben sich gegenüber der ursprünglichen Überlegung auf IP-Schicht (OSI-Layer 3) folgende Vorteile:

+ Keine Konfigurationsänderung der anderen Versuchsrechner
+ Abgesehen vom einzustellenden Simulationsverhalten keine Konfigurationsänderung auf der Simulatormaschine selbst
+ Keine prinzipielle Festlegung auf IPv4 als Schicht-3-Protokoll und dadurch größeres Einsatzfeld

Der Simulator kann somit einfach wie in Bild 1.1 dargestellt vor den/die Versuchsrechner ins Netz integriert werden und verschlechtert die Netzqualität auf reproduzierbare Weise.

3.1 Zugriff auf Ethernetframes

Es wurden verschiedene Wege betrachtet, die Frames mit dem Netz auszutauschen:

- Nutzung des Paketfilters `netfilter` (siehe http://www.netfilter.org) mit einer Regel, die ausgewählte Pakete über das `ulog`-Target an eine Anwendung im Userspace herausreicht (siehe Abbildung 3.1)
  Dieser Ansatz wurde verworfen, da das Reinjizieren der Pakete nicht geklärt werden konnte.
  Mit der Nutzung von `netfilter` hätte der Simulator auf IP-Schicht (OSI-Schicht 3) agiert. Die oben erwähnten Vorteile des Vorgehens auf Ethernet-Schicht (das alle weiteren Ansätze verfolgen) hätte man durch Mechanismen wie Proxy-ARP ebenso erzielen können. Der Simulator hätte in diesem Fall nicht als Bridge oder Repeater agiert, sondern als Pseudo-Bridge und man hätte seine Netzwerkkonfiguration bei jedem Einsatz anpassen müssen, um klar zu stellen, welcher IP-Bereich über

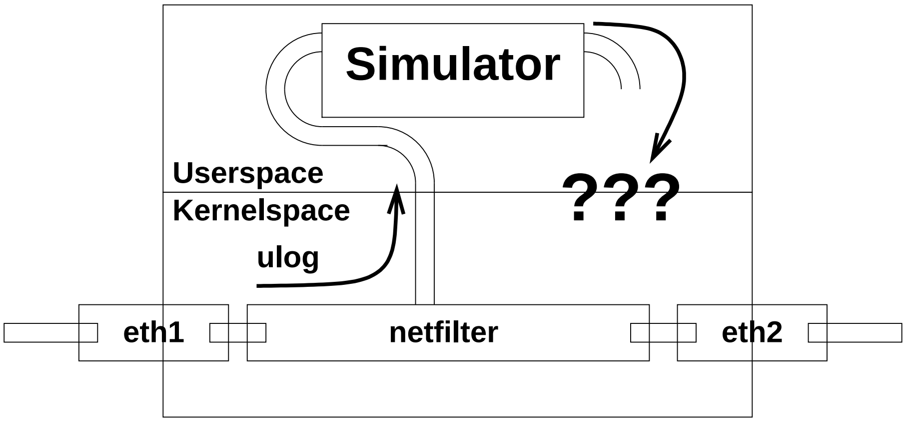

Abbildung 3.1: Unvollständiger Ansatz mit netfilter und ulog-Target

welches Netzwerkinterface erreichbar ist und für wen er daher ARP-Anfragen stellvertretend beantworten soll.

- Aufbau eines Userland-Tunnels mit zwei virtuellen Ethernet-Devices (Tap-Devices) die über den Bridge-Code im Kernel mit den beiden Simulator-Interfaces Frames austauschen, wie dies in Abbildung 3.2 schematisch dargestellt ist.
  Dieser Ansatz wurde für die Entwicklung eines ersten Prototypen verwendet und leidet etwas unter der komplizierten Konfiguration der Maschine:
  - Das Konfigurationswerkzeug zum Einrichten der Bridge `brctl` fehlt bei einigen Distributionen.
  - Die Bridge-Funktionalität ist nicht standardmäßig im Kernel aktiviert. Dies stört, wenn die Simulationssoftware auf einer anderen Maschine laufen soll.
  - An der Simulation sind zwei Ethernet-Devices, zwei Tap-Devices und zwei Bridge-Devices beteiligt, deren Konfiguration zwar über ein einfaches Skript gelöst wurde. Aber bei unvorhergesehenen Problemen stört das komplizierte Setup die Fehlersuche.

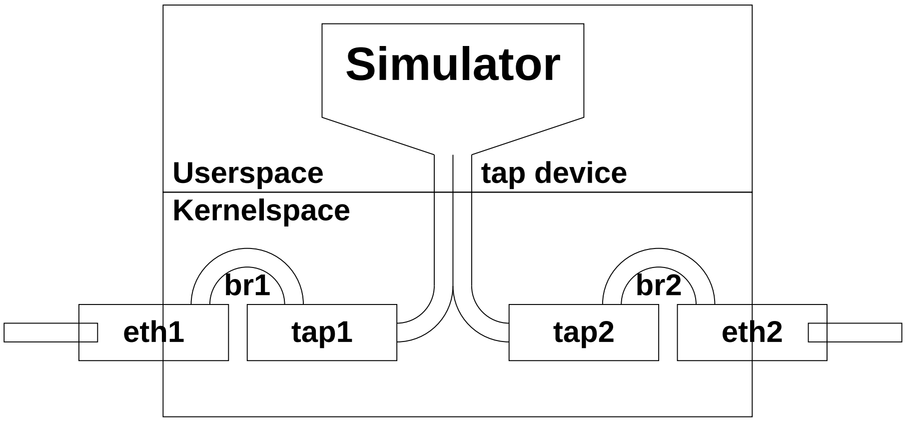

Abbildung 3.2: Prototypische Implementierung mittels Userland-Tunnel

- Modifizieren des Bridge-Codes im Kernel (siehe Abbildung 3.3)
  Dieser Ansatz wurde verworfen, da Entwicklung im Kernelspace deutlich aufwendiger ist als im Userspace: Man benötigt eine Entwicklermaschine, die stabil läuft, zur Übersetzung der neuen Kernel und eine Crash-n-Burn Maschine zum Testen der gebauten Kernel, die nach jedem Crash des mo-

difizierten Kernels mit dem neu verbesserten Kernel bestückt werden muss und neu gebootet werden muss. Software, die dies vereinfachen könnte (wie z.B. VMware) bewirkt nur eine Verschiebung der Kosten von Hardware zu Software. Da wir noch keinerlei Erfahrung bezüglich Kernelprogrammierung hatten und die ersten Versuche mit dem Prototypen im Userspace vielversprechend waren, gab es keinen Grund, diesen höheren Aufwand auf sich zu nehmen.

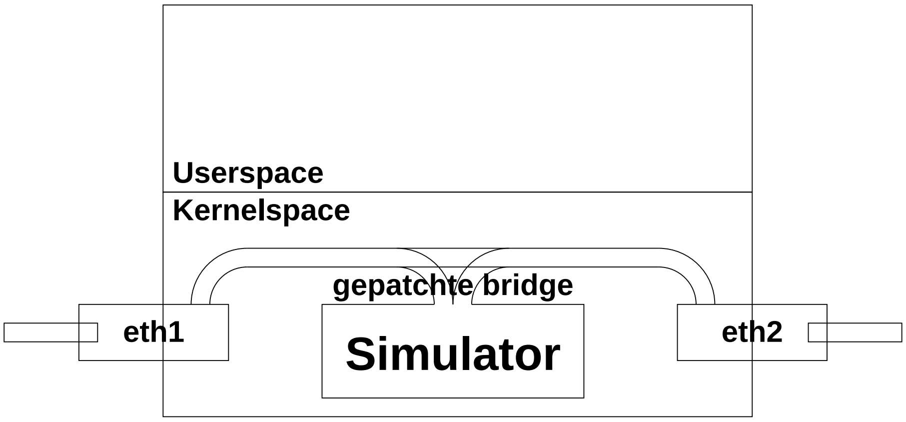

Abbildung 3.3: Der Simulator als Bridge im Kernel?

- Lesen und Schreiben der Frames über Packet-Sockets direkt von den Ethernetinterfaces, wie in Abbildung 3.4 dargestellt
  Dieser Ansatz verzichtet auf den nicht immer vorhandenen Bridge-Code im Kernel, sondern tauscht die Frames direkt mit den Ethernet-Karten aus. Im Vergleich mit Abbildung 3.2 erkennt man den Aufbau des Prototypen wieder, wobei die komplizierte Konfiguration des Kernels mit Bridges und virtuellen Ethernet Tap-Devices entfällt.

  Dieser Ansatz wurde für den Simulator gewählt und ist in Kapitel 4 detaillierter beschrieben.

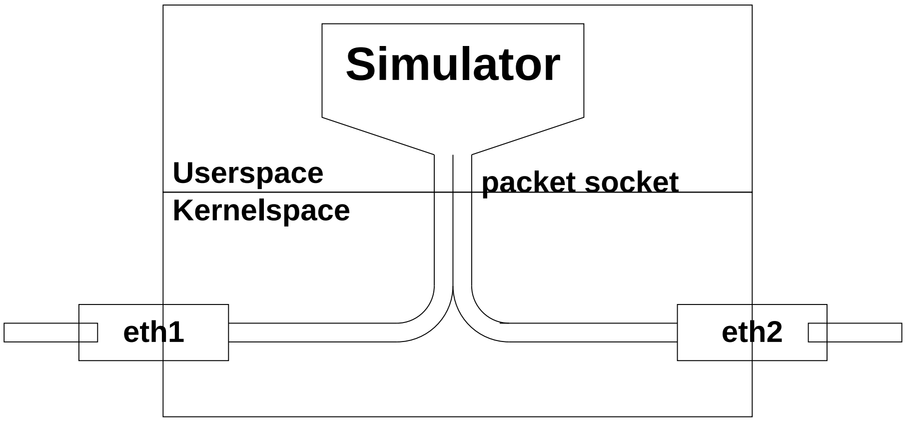

Abbildung 3.4: Der Simulator im Userspace mit dem Zugriff auf die Ethernetkarten

3.2 Realisierung des Frame-Puffers

Um die von einer Ethernetkarte empfangenen Frames eine gewisse Zeit zu verzögern, bevor diese auf der anderen Ethernetkarte weiterversendet werden, benötigen wir eine geeignete Containerstruktur. Diese

Containerstruktur soll eine Abbildung von Sendezeitpunkten auf die zu versendenden Frames herstellen, also Key/Value Paare verwalten, wobei die Schlüssel (Absendezeitpunkte) nicht eindeutig sein müssen, da zunächst nicht auszuschließen ist, dass zwei Frames zum gleichen Zeitpunkt versendet werden sollen, wenn die künstliche Verzögerung der beiden Frames gerade die unterschiedlichen Empfangszeiten ausgleicht. Technisch können wir die beiden Frames nur nacheinander versenden, aber bei der Verwendung einer Containerstruktur mit eindeutigem Schlüssel müssten wir derartige Schlüsselkollisionen separat abfangen und durch geringfügiges Ändern der Absendezeitpunkte lösen. Um uns diesen Aufwand zu ersparen benötigen wir eine Struktur, die mehrfache Schlüssel erlaubt.

Die Containerstruktur soll effizient das Einfügen beliebiger Schlüssel (gewünschter Absendezeitpunkt) und das Abrufen des/der kleinsten Schlüssel jeweils mit den assoziierten Daten (Ethernetframes) unterstützen. Die `multimap` aus der Standard Template Library (STL) erfüllt unsere Anforderungen, da sowohl das Einfügen als auch das Abfragen und Löschen des nächsten Elements in Abhängigkeit der gespeicherten Einträge n eine Komplexität von $\log n$ haben. Dies ist in ihrer Spezifikation zugesichert, wie sie beispielsweise bei SGI einsehbar ist [sgistlasscon].

3.3 Funktionsprinzip des Simulators

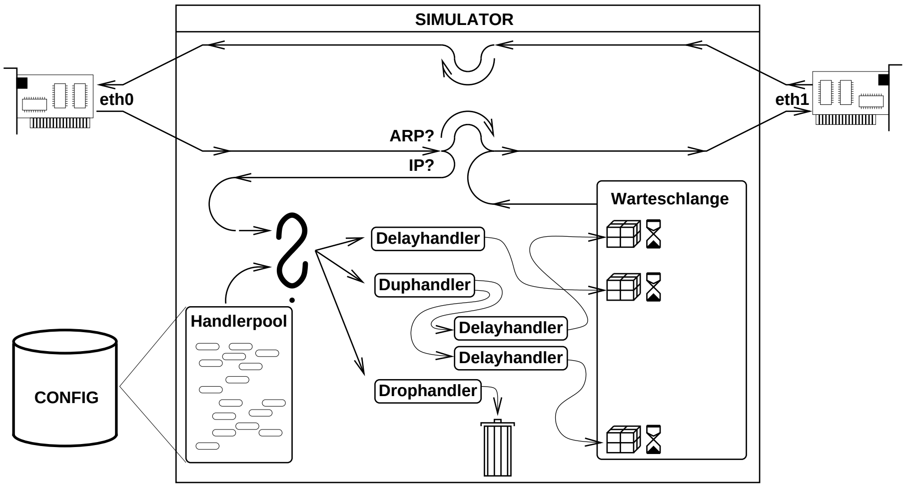

Abbildung 3.5: Funktionsprinzip des Simulators

Da die Simulation, wie in Abschnitt 2.4 auf Seite 5 erläutert wurde, nur in einer Richtung erfolgt, werden Ethernet-Frames von Interface 1 zu Interface 0 ohne zusätzliche Verzögerung weitergereicht.

In der anderen Richtung darf die Simulation aber nicht wahllos alle Frames erfassen, sondern nur die zum gewünschten Netzwerkprotokoll passenden (hier: IPv4). Zwar wären alle Protokolle gleichermaßen von einer schlechten Leitungsqualität betroffen, aber meist liegen die Ursachen für die eingeschränkte Leitungsqualität nicht im eigenen lokalen Netz sondern in der Fernanbindung. Die Aufgabenstellung in Abschnitt 2.1 erforderte die “Simulation von QoS-Eigenschaften … auf IP-Ebene“. Aus den am Beginn von Kapitel 3 genannten Gründen haben wir uns entschieden, den Simulator auf Ethernet-Schicht agieren zu lassen, aber dennoch sollen sich die Auswirkungen auf IPv4 beschränken.

Zwei Beispiele für andere Protokolle, die für den stabilen Betrieb des LAN notwendig sind und daher auch in Simulationsrichtung ohne Verzögerung weitergeleitet werden müssen sind:

ARP Das Address Resolution Protokoll ist notwendig für den Betrieb von IPv4 (und anderen Protokollen) in einem Ethernet, da es die Netzwerkadresse der höheren Schicht 3 auf die Hardware-Adresse (MAC-Adresse) des Zielsystems abbildet. Ohne diese Auflösung kann das Zielsystem nicht direkt angesprochen werden. Ein Verzögern oder gar Verwerfen von ARP-Paketen würde die Kommunikation zusätzlich stören, und somit das zu erzielende QoS-Profil negativ beeinträchtigen. Die simulierte Leitung wäre also schlechter als sie sein soll.

STP Das Spanning Tree Protokoll tritt in geswitchten Infrastrukturen auf und dient zum Eliminieren von Schleifen in der Netzwerktopologie. Eine zu starke Beeinflussung des STP-Verkehrs kann schlimmstenfalls zu einer Neuorganisation der Switche untereinander und damit einhergehend einem kurzzeitigen Ausfall einzelner Segmente führen.

Nachdem also der Protokolltyp des Frames bestimmt wurde und nur die gewünschten IPv4-Frames zur Simulation herangezogen werden, stellt sich die Frage nach deren weiterer Behandlung: Von den fünf in Abschnitt 2.2 gezeigten Effekten zeigen sich nur drei direkt als Phänomen beim Betrachten einzelner Pakete: Paketverlust, Paketverzögerung und Paketverdopplung. Die Bandbreitenbeschränkung wird wie in Abschnitt 2.3 angekündigt durch erhöhte Delays realisiert. Die Paketvertauschung zeigt sich erst beim Betrachten mehrerer Pakete durch das Zusammenspiel von verschiedenen Verzögerungen.

Bei objektorientierter Herangehensweise erstellen wir eine generische Handler-Klasse zur Behandlung der Frames und leiten davon drei Klassen für eben diese Effekte ab: Drophandler, Delayhandler und Duphandler. Das zu simulierende QoS-Profil wird aus einer Konfigurationsdatei bereits beim Programmstart eingelesen und für jeden Messwert des Profils wird eine entsprechende Handlerinstanz erzeugt und in einem Vorratspool abgelegt. Dabei besteht die in Abschnitt B.2 dokumentierte Konfigurationsdatei aus Verzögerungswerten zur Erzeugung von Delays und Störereignissen zum Verwerfen oder Verdoppeln von Paketen. Ein derartiges Profil lässt sich leicht aus einfachen Messungen gewinnen und zur Reproduktion des Netzverhaltens durch den Simulator verwenden.

Die aus der Konfigurationsdatei angelegten Instanzen der Handlerklassen im Handlerpool können nun zyklisch in der Reihenfolge des eingelesenen Profils durchlaufen werden, so dass ein periodisches und deterministisches Verhalten der simulierten Leitung erzielt wird, oder in zufälliger Reihenfolge zur Behandlung der Frames verwendet werden, wenn das deterministische periodische Verhalten unerwünscht ist. Die ausgewählte Handlerinstanz wird nun mit der Verarbeitung des Frames betraut.

Der für die Paketverdopplung zuständige Duphandler benötigt zwei weitere Handlerinstanzen, die die beiden Replikate des zu duplizierenden Frames weiterbehandeln. Zumeist werden dies zwei Delayhandler sein, die die beiden identischen Frames mit verschiedener Verzögerung weitersenden. Aber auch das Angeben weiterer Duphandler (mit zugehörigen Unterhandlern) ist denkbar, um eine Verdreifachung oder Ver-n-fachung zu erzielen.

Der Drophandler löscht einfach den Frame, ohne ihn vorher weiterzusenden.

Der Delayhandler bestimmt aus seinem bei der Erzeugung aus der Konfigurationsdatei festgelegten zusätzlichen Delay und dem Empfangszeitpunkt des Frames den Absendezeitpunkt des Frames zu dem er den Simulator idealerweise verlassen sollte. Falls keine Paketvertauschungen gestattet sind, ist sicherzustellen, dass der Frame keine anderen Frames überholt. Dazu wird sein Absendezeitpunkt gegebenenfalls soweit nach hinten verschoben, dass der Frame hinten in der Warteschlange eingereiht werden muss. Mit dem so bestimmten Absendezeitpunkt wird der Frame vom Delayhandler an die Warteschlange übergeben und dort einsortiert.

Wenn die Absendezeit für einen Frame in der Warteschlange gekommen ist, wird er aus der Warteschlange entfernt und auf der entsprechenden Ethernetkarte versendet. Dabei wird die vorgegebene beschränkte Bandbreite überwacht: Nach jedem versendeten Frame wird anhand der angegebenen Bandbreite und der Framegröße der frühestmögliche nächste Absendezeitpunkt bestimmt und erst dann kann der nächste Frame versendet werden. Diese zusätzliche Verzögerung ist wie in Abschnitt 2.3 dargelegt mit dem strengen Befolgen des gegebenen Verzögerungsprofils unvereinbar.

Schematisch ist der gesamte Vorgang der Frameverarbeitung in Abbildung 3.5 dargestellt.

Kapitel 4

Implementierung

Wir beschreiben nun die Details und Probleme, die sich beim Umsetzen des Designs aus dem vorigen Kapitel ergaben. Wir werden die Details beim Umgang mit dem Packet-Socket beleuchten und das Kernproblem des gesamten Projekts darstellen, die präzise Erzeugung von Delays. Aber zuerst fassen wir ein paar grundlegende Entscheidungen aus den vorigen Kapiteln nochmals zusammen:

Der Empfang und das Senden der Ethernet-Frames geschieht gemäß den Überlegungen aus Abschnitt 3.1 über Packet-Sockets [man7packet].

Da wir auf Ethernetschicht arbeiten (Layer 2) erhalten wir nicht nur IPv4 Pakete in den Ethernet-Frames. Andere Protokolle als IPv4 wollen wir aber in ihrem Fluss nicht beeinträchtigen, insbesondere hätte das Verzögern oder Verwerfen von Address Resolution Protocol (ARP) Paketen einen stark verfälschenden Einfluss auf die Simulation, wie wir in Abschnitt 3.3 erläuterten.

Der Delay wird aus den in Abschnitt 2.4 dargestellten Gründen der Übersichtlichkeit nur unidirektional realisiert.

4.1 Beschreibung der Frameverarbeitung

Entsprechend dem Entwurf aus dem vorigen Kapitel wird ein Frame von einer Netzwerkkarte empfangen und

- ohne Verzögerung weitergeleitet, falls er einen anderen Protokolltyp als IPv4 enthält oder in der “Rückrichtung” ankommt
- falls er verworfen werden soll, nicht weiter behandelt
- falls er (mit Verzögerung) weitergeleitet werden soll wird die aktuelle Zeit über den Systemaufruf `gettimeofday` bestimmt und der Absendezeitpunkt des Frames durch Addieren des anstehenden Delays auf die aktuelle Zeit berechnet. Falls keine Paketvertauschungen auftreten sollen, muss der Frame unter Umständen weiterverzögert werden, bis er der letzte Frame in der Warteschlange ist (die Konsequenzen dieses Vorgehens wurden in Abschnitt 2.3 dargelegt). Schließlich wird der Frame mit seinem Absendezeitpunkt als Schlüssel in die Warteschlange (die keine Schlange im FIFO-Sinne ist!) eingereiht und wartet darauf, weitergesendet zu werden.

Folgende Tücken fielen während der Implementierung besonders auf: Ein Packet-Socket reicht aus, um auf alle Interfaces zugreifen zu können. Das jeweilige Interface wird in der Link-Layer Adresse als Interface-Index angegeben. Auf dem Packet-Socket lesen wir alle Frames die der Maschine unterkommen. Nicht nur eingehende Frames sondern auch ausgehende, die von unserer Simulatormaschine selbst gesendet werden, inklusive der Frames, die das Simulationsprogramm gerade eben gesendet hat. Somit würde jeder Frame so lange kontinuierlich repliziert werden und das Netz fluten, bis der nächste Frame von außen empfangen wird [1]. Daher überprüfen wir, ob in der Link-Layer-Adresse der Pakettyp als PACKET_OUTGOING vermerkt ist und ignorieren in diesem Fall den Frame. Auch Frames die von anderen Interfaces stammen als den beiden an der Simulation beteiligten müssen ignoriert werden. Und schließlich hat ein Ethernet-Frame eine Payload von 1500 Byte, aber zusätzlich einen Header von 14 Byte, so dass wir 1514 Byte vom Packet-Socket lesen müssen.

Die Frames müssen die Warteschlange auch wieder verlassen und gesendet werden. Zum Absenden muss in der Link-Level Adresse, von der der Frame empfangen wurde, nur der Interface-Index abgeändert werden auf das jeweils andere Interface. Das pünktliche Versenden der Frames entpuppte sich allerdings als größere Herausforderung.

4.2 Probleme beim Erzeugen exakter Delays

Als zentrales Problem bei der Implementierung stellte sich das präzise Erzeugen der Delays heraus. Der Prototyp verwendete den `select`-Systemaufruf [man2select], um über eintreffende Frames informiert zu werden. Der `select`-Systemaufruf erhält als Parameter drei Mengen von Filedeskriptoren die auf “lesebereit”, “schreibbereit” und “außergewöhnliche Ereignisse” überprüft werden [man2select] und kehrt zurück, sobald einer (oder mehrere) Filedeskriptoren die geforderten Bedingungen erfüllen. Ein weiterer Parameter gibt eine maximale Wartezeit an (Timeout), nach der der Aufruf zurückkehrt, selbst wenn keine Zustandsänderung an den Filedeskriptoren auftrat.

In der Prototypischen Implementierung stellte sich heraus, dass die Delays nur mit einer Granularität von ca. 10ms (einem so genannten Jiffie) realisiert werden können. Als Timeout wurde jeweils die Differenz der Absendezeit des nächsten Frames in der Warteschlange zur aktuellen Zeit verwendet. Der `select`-Aufruf kehrte jedoch zu spät zurück.

Ein vergeblicher Lösungsversuch hätte darin bestanden, mehrere Threads zu verwenden, von denen einer Frames empfängt und einreiht und der andere die Frames versendet und dazwischen schläft, bis der nächste Frame an der Reihe ist.

Um das dafür notwendige hochpräzise Warten zu realisieren wurde ein Testprogramm `sleepprecision` geschrieben, das die Präzision verschiedener Timeout-Techniken untersucht, indem es verschiedene Warte-Techniken verwendet um für kurze Zeit zu pausieren und die tatsächlich vergangene Zeit berichtet:

Busy-Waiting Bei Busy-Waiting pausiert der Prozess nicht, sondern er überprüft in einer Schleife ob die Zeit zum Weiterlaufen gekommen ist. Die Präzision dieser Wartetechnik hängt ab von der Präzision, mit der die Uhrzeit bestimmt werden kann. Glücklicherweise hat der `gettimeofday` Systemaufruf auf unserer Plattform eine höhere Genauigkeit als Jiffies, im Gegensatz zu älteren Plattformen auf denen Busy-Waiting keinerlei Vorteile gegenüber den Jiffie-basierten Wartefunktionen gebracht hätte.

Select Diese oben beschriebene Funktion realisiert ihren Timeout auch nur mit Jiffie-Genauigkeit.

Nanosleep Dieser Systemaufruf könnte das Pausieren mit einer Genauigkeit von Nanosekunden ermöglichen. Die Dokumentation der Funktion erwähnt, dass die Genauigkeit leider nur im Jiffie-Bereich liegt. Allerdings werden Wartezeiten bis zu 2ms hochgenau erzeugt, indem die Funktion intern auf Busy-Waiting ausweicht. Dieses Verhalten wurde durch das Programm `sleepprecision` bestätigt, wie man in Abbildung 4.2 erkennen kann. Bizarrerweise ist `nanosleep` im Bereich über 2ms noch einen Jiffie schlechter als `select`, da die Dokumentation zusichert, dass auf keinen Fall weniger Zeit vergangen ist als angefordert. Die genaue Untersuchung der Wartetechniken mit dem Programm `sleepprecision` zeigte, dass `select` nach etwas weniger als einem Jiffie zurückkehrt. Da dieses Verhalten die Spezifikation von `nanosleep` verletzen würde, erhöht `nanosleep` offenbar die Wartezeit generell um einen weiteren Jiffie.

[1] Dieses Flooding trat während der Entwicklung auf und führte zu Systemabstürzen auf einem PC im gleichen Netzsegment.

Die tatsächlich verstrichene Zeit ist gegen die geforderte Wartezeit in Abbildung 4.1 aufgetragen. In Abbildung 4.2 ist die Relative Abweichung des Ergebnisses von der Vorgabe dargestellt.

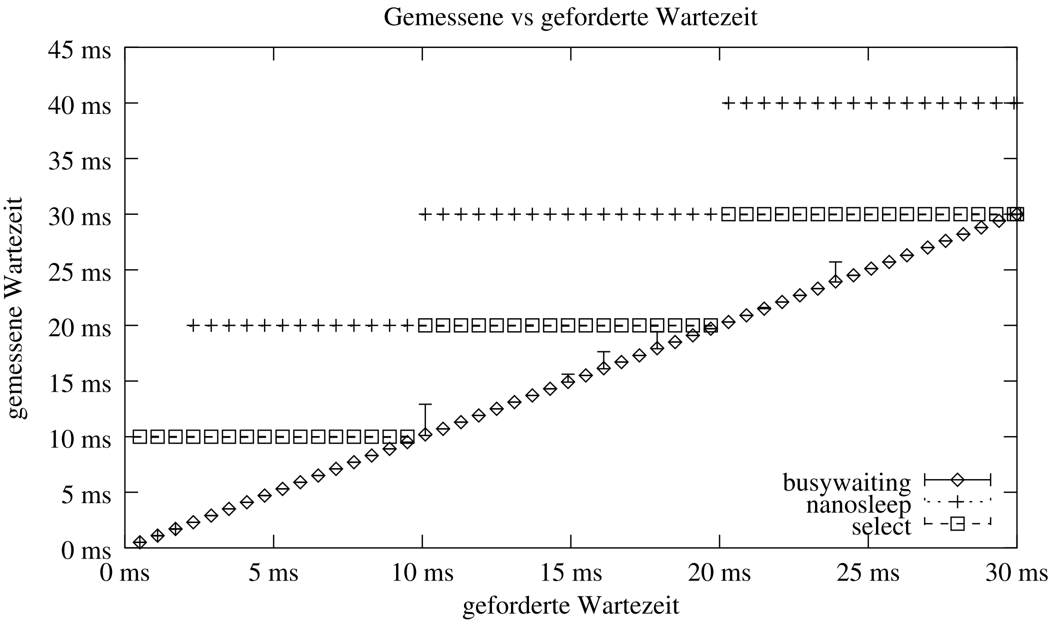

Abbildung 4.1: Tatsächliche vs. geforderte Wartezeit

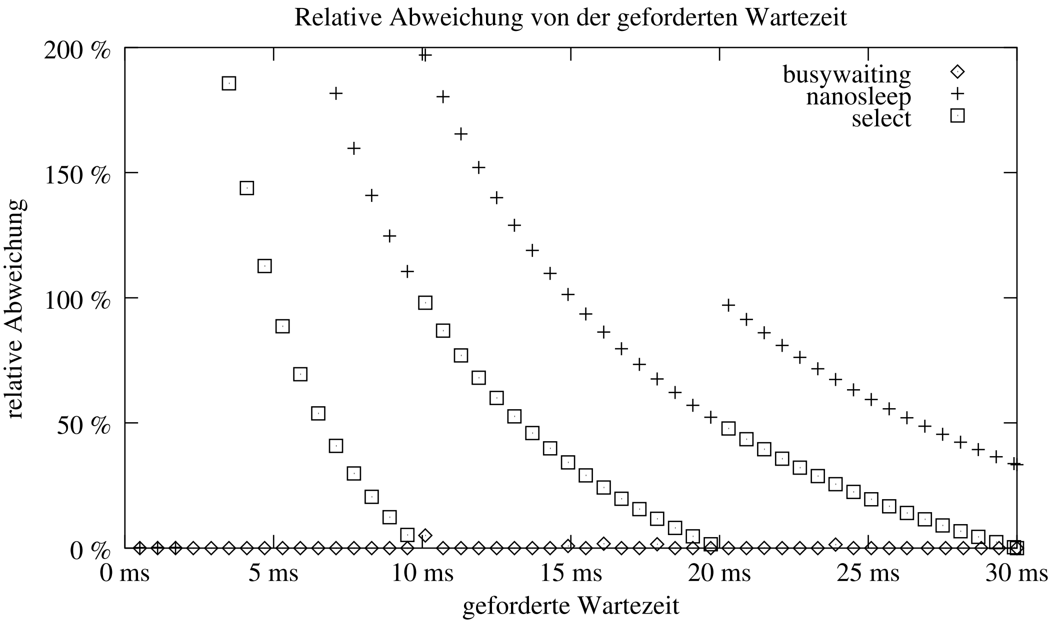

Abbildung 4.2: Relative Abweichung der tatsächlichen Wartezeit von der geforderten Wartezeit

Eine Analyse der Quelltexte des Linux-Kernels offenbarte, dass alle Funktionen, die in der Lage wären einen Prozess für eine bestimmte Zeit schlafenzulegen (z.B. `select`, `poll`, `sleep`, `nanosleep`), intern auf *Jiffies* basieren, eine Zeiteinheit auf Kernel-Ebene die auf der verwendeten i386-Plattform eine Granularität von 10ms aufweist [man2nanosleep]. Ein exemplarisches Beispiel aus dem Linux-Quelltext zeigen wir in Anhang A.1.

Gewünschter Zusatz-Delay	Erzielte Round-Trip Time Min	Avg	Max	MDev	Relative Abweichung
Bridge	0.851 ms	0.870 ms	1.048 ms	0.042 ms	N/A
0 ms	0.714 ms	0.734 ms	0.919 ms	0.028 ms	N/A
1 ms	1.830 ms	3.626 ms	114.940 ms	10.546 ms	188.3%
1 ms	1.829 ms	1.891 ms	3.120 ms	0.064 ms	14.8%
2 ms	2.841 ms	2.924 ms	18.450 ms	0.539 ms	9.0%
3 ms	3.852 ms	3.902 ms	4.334 ms	0.049 ms	5.3%
10 ms	10.852 ms	10.892 ms	11.326 ms	0.067 ms	1.5%

Tabelle 4.1: Simulierte Delays

Auf der Alpha-CPU haben Jiffies eine Granularität von 1/1024 Sekunden und die neuesten Kernelentwicklungen ab 2.5 [2] verwenden auch auf i386-Plattformen Jiffies von 1024. Damit wäre die Granularität der erzeugbaren Delays zwar eine Größenordnung besser, aber eine Alpha-Maschine stand nicht zur Verfügung, der neue 2.5er Kernel war noch nicht verfügbar und die Granularität wäre immer noch bei 1ms.

Das Problem besteht darin, einen genauen Zeitpunkt - den Absendezeitpunkt des nächsten Frames - nicht zu verpassen. *Busy waiting* stellt eine brauchbare Lösung dar, indem die zu überwachenden Filedeskriptoren auf nicht-blockierendes Lesen geschaltet werden (`O_NONBLOCK`) und in einer Schleife jeder Filedeskriptor auf anstehende Daten überprüft wird und die Uhrzeit mit dem Absendezeitpunkt des nächsten Frames aus der Warteschlange verglichen wird. Der Nachteil dieses Vorgehens liegt darin, dass die Simulatormaschine mit 100% CPU-Last arbeitet, obwohl sie fast gar nichts zu tun hat. Da die Simulatormaschine jedoch eine dedizierte Maschine für die Link-Simulation ist und keine weiteren Aufgaben zu erledigen hat, stellte dies kein echtes Hinderniss dar und ermöglichte die Simulation von Delays, die kleiner sind als 10ms.

In Tabelle 4.1 sind die erzielten Delays gegen die geforderten Delays aufgelistet. Die Messung erfolgte, indem durch den Befehl `ping ga.te.way.ip -i 0.001 -c 1000 -q` tausend Pakete im Abstand von einer Millisekunde durch den Simulator zum Gateway geschickt wurden. Als Vergleich zum Null-Delay wurde der Versuchsrechner für die allererste Messung alternativ als Bridge konfiguriert und die Messung wurde ohne die Simulator-Software durchgeführt.

Auffallend ist bei der ersten Messung der extreme Ausreißer. Derart große Ausreißer treten leider immer wieder auf. Davon abgesehen sind die Delays aber gut steuerbar.

Die relative Abweichung wurde berechnet als das Verhältnis des um den Null-Delay bei 0ms berichtigten Zusatz-Delays relativ zum geforderten Delay. Dabei sei $Messung(Wunsch)$ die durchschnittliche Round-Trip Time.

$$RelativeAbweichung = \frac{Messung(Wunsch) - Messung(0ms) - Wunsch}{Wunsch}$$

Man sieht, dass die Delays auch unterhalb eines Jiffies leidlich genau im Bereich kleiner 10% Abweichung liegen.

[2] Kernel 2.5 ist wie alle Kernel mit ungerader Minor Versionsnummer ein Entwicklerkernel, der nicht für den Produktiv-Betrieb vorgesehen ist. Erst 2.6 wird die Release sein und ist für Juni 2003 angekündigt.

Kapitel 5

Ausblick

In diesem Schlusskapitel werden einige denkbare Erweiterungen des Leitungssimulators umrissen.

Die Simulation könnte andere Protokolle als IPv4 beeinträchtigen, wie z.B. IPv6, IPX oder AppleTalk.

Durch die Verwendung von defragmentierenden Raw-Sockets anstelle von Packet-Sockets könnte der Simulator auf IP-Schicht komplette IP-Pakete betrachten. Im Gegensatz zur vorliegenden Implementierung würde dadurch bei zu simulierenden 25% Paketverlust ein Viertel aller volständigen IP-Pakete verworfen, und nicht ein Viertel aller Fragmente, was je nach Datenverkehr zu einem höheren Verlust an IP-Paketen führt, da beim Fehlen eines einzigen Fragmentes das gesamte IP-Paket am Ziel verworfen wird.

Für Anwendungen, die einen symmetrischen Delay erfordern, wie z.B. Uhrensynchronisationsverfahren, ist es unter Umständen wünschenswert, dass die Beeinträchtigung der Leitung in beiden Richtungen erfolgt.

Zum besseren Verständnis einiger Protokolle wäre es hilfreich, Datenpakete interaktiv zu beeinflussen und auf Tastendruck zu verwerfen, oder weiterzuleiten. Beispielsweise könnte man so detailliert verfolgen, wann TCP ein Paket erneut versendet und ob der `write`-Aufruf bei synchronem Schreiben bis zum Erhalt der Quittung (ACK) blockiert.

Anhang A

Auszüge aus dem Linux-Quelltext zur Delayproblematik

Die relevanten Code-Stellen wurden aus den Quellen des Linux Kernels in Version 2.4.9-13 entnommen.

Wir sehen in Zeile 61 von Listing A.1, wie `select` auf die Funktion `schedule_timeout` aufbaut. Alle anderen Warte-Funktionen verwenden den gleichen Jiffie-basierten Mechanismus, der in Listing A.2 abgedruckt ist.

Listing A.1: Kern der `select`-Funktion im Kernel (linux-2.4.9-13/fs/select.c, ab Zeile 165)

```
   int do_select(int n, fd_set_bits *fds, long *timeout)
 2 {
           poll_table table, *wait;
 4         int retval, i, off;
           long __timeout = *timeout;
 6
           read_lock(&current->files->file_lock);
 8         retval = max_select_fd(n, fds);
           read_unlock(&current->files->file_lock);
10
           if (retval < 0)
12                 return retval;
           n = retval;
14
           poll_initwait(&table);
16         wait = &table;
           if (!__timeout)
18                 wait = NULL;
           retval = 0;
20         for (;;) {
                   set_current_state(TASK_INTERRUPTIBLE);
22                 for (i = 0 ; i < n; i++) {
                           unsigned long bit = BIT(i);
24                         unsigned long mask;
                           struct file *file;
26
                           off = i / __NFDBITS;
28                         if (!(bit & BITS(fds, off)))
                                   continue;
30                         file = fget(i);
                           mask = POLLNVAL;
```

```
			if (file) {
				mask = DEFAULT_POLLMASK;
				if (file->f_op && file->f_op->poll)
					mask = file->f_op->poll(file, wait);
				fput(file);
			}
			if ((mask & POLLIN_SET) && ISSET(bit, __IN(fds,off))) {
				SET(bit, __RES_IN(fds,off));
				retval++;
				wait = NULL;
			}
			if ((mask & POLLOUT_SET) && ISSET(bit, __OUT(fds,off))) {
				SET(bit, __RES_OUT(fds,off));
				retval++;
				wait = NULL;
			}
			if ((mask & POLLEX_SET) && ISSET(bit, __EX(fds,off))) {
				SET(bit, __RES_EX(fds,off));
				retval++;
				wait = NULL;
			}
		}
		wait = NULL;
		if (retval || !__timeout || signal_pending(current))
			break;
		if(table.error) {
			retval = table.error;
			break;
		}
		__timeout = schedule_timeout(__timeout);
	}
	current->state = TASK_RUNNING;

	poll_freewait(&table);

	/*
	 * Up-to-date the caller timeout.
	 */
	*timeout = __timeout;
	return retval;
}
```

Listing A.2: schedule_timeout - die Jiffie-basierte Basis aller Wartefunktionen (linux-2.4.9-13/kernel/sched.c ab Zeile 368)

```
/**
 * schedule_timeout - sleep until timeout
 * @timeout: timeout value in jiffies
 *
 * Make the current task sleep until @timeout jiffies have
 * elapsed. The routine will return immediately unless
 * the current task state has been set (see set_current_state()).
 *
 * You can set the task state as follows -
 *
 * %TASK_UNINTERRUPTIBLE - at least @timeout jiffies are guaranteed to
 * pass before the routine returns. The routine will return 0
 *
```

```
 * %TASK_INTERRUPTIBLE - the routine may return early if a signal is
 * delivered to the current task. In this case the remaining time
 * in jiffies will be returned, or 0 if the timer expired in time
 *
 * The current task state is guaranteed to be TASK_RUNNING when this
 * routine returns.
 *
 * Specifying a @timeout value of %MAX_SCHEDULE_TIMEOUT will schedule
 * the CPU away without a bound on the timeout. In this case the return
 * value will be %MAX_SCHEDULE_TIMEOUT.
 *
 * In all cases the return value is guaranteed to be non-negative.
 */
signed long schedule_timeout(signed long timeout)
{
        struct timer_list timer;
        unsigned long expire;

        switch (timeout)
        {
        case MAX_SCHEDULE_TIMEOUT:
                /*
                 * These two special cases are useful to be comfortable
                 * in the caller. Nothing more. We could take
                 * MAX_SCHEDULE_TIMEOUT from one of the negative value
                 * but I’ d like to return a valid offset (>=0) to allow
                 * the caller to do everything it want with the retval.
                 */
                schedule();
                goto out;
        default:
                /*
                 * Another bit of PARANOID. Note that the retval will be
                 * 0 since no piece of kernel is supposed to do a check
                 * for a negative retval of schedule_timeout() (since it
                 * should never happens anyway). You just have the printk()
                 * that will tell you if something is gone wrong and where.
                 */
                if (timeout < 0)
                {
                        printk(KERN_ERR "schedule_timeout:␣wrong␣timeout␣"
                               "value␣%lx␣from␣%p\n", timeout,
                               __builtin_return_address(0));
                        current->state = TASK_RUNNING;
                        goto out;
                }
        }

        expire = timeout + jiffies;

        init_timer(&timer);
        timer.expires = expire;
        timer.data = (unsigned long) current;
        timer.function = process_timeout;

        add_timer(&timer);
        schedule();
        del_timer_sync(&timer);
```

```

        timeout = expire - jiffies;

 out:
        return timeout < 0 ? 0 : timeout;
}
```

Anhang B

Verwendung des Simulators

B.1 Aufruf des Simulators

Der Leitungssimulator wird auf der Kommandozeile mit vier Parametern aufgerufen:
simulator <nic1> <nic2> <config> <auswahlstrategie> <bandbreite> <vertauschung>
Die Parameter bedeuten:

nic1 Ethernetdevice am Ausgang mit Verzögerung

nic2 Ethernetdevice am Ausgang ohne Verzögerung

config Konfigurationsdatei mit dem zu simulierenden Leitungsprofil (siehe B.2)

auswahlstrategie Auswahlstrategie für die Handlerinstanzen. Mögliche Werte sind

- **`rr`** für zyklisches Abarbeiten des Handlerpools (Round Robin)
- **`rand`** für zufälliges Auswählen eines Handlers aus dem Pool (Random)

bandbreite Bandbreitenbeschränkung in Bits/s

vertauschung Paketvertauschungen verbieten oder zulassen. Mögliche Werte sind

- **`fifo`** wenn Vertauschungen verhindert werden sollen
- **`gigo`** wenn Vertauschungen passieren dürfen

B.2 Aufbau der Konfigurationsdatei

Die Konfigurationsdatei enthält das zu simulierende Leitungsprofil. Jede Zeile besteht aus genau einem der Befehle:

zahl eine ganze Zahl gibt den zusätzlichen Delay in Nanosekunden an und erzeugt einen Delayhandler

drop bewirkt das Verwerfen eines Frames durch den erzeugten Drophandler

dup bewirkt eine Paketverdopplung. Die beiden folgenden Zeilen werden als Unterhandler für die beiden resultierenden Frames verwendet. Es müssen aus den folgenden Zeilen zwei vollständige Handler hervorgehen!

B.3 Beispielkonfigurationsdateien

Listing B.1: 50 Prozent Paketverlust

```
0
drop
```

Listing B.2: Alle Frames zeitverzögert verdoppeln

```
dup
0
10000
```

Listing B.3: Paketverdreifachung

```
dup
dup
0
0
0
```

Listing B.4: Realistisches Beispiel

```
42700
40900
41900
40000
44300
41500
42500
41700
42300
41700
51300
42400
40600
42400
40000
drop
42100
42700
42000
41800
40500
40400
40000
39800
40400
40200
41900
42700
42100
42000
39900
40200
40100
39900
41900
```

Abkürzungsverzeichnis

ARP Address Resolution Protocol

FIFO First In First Out

ITU International Telecommunication Union

LAN Local Area Network

QoS Quality of Service

STL Standard Template Library

WAN Wide Area Network

Literaturverzeichnis

[ITU 96] ITU: *One way transmission time*, 1996.

[man2nanosleep] *Linux Programmer's Manual NANOSLEEP(2)*, April 1996.

[man2select] *Linux Programmer's Manual SELECT(2)*, Februar 2001.

[man7packet] KLEEN, ANDI: *Linux Programmer's Manual PACKET(7)*, April 1999.

[Neu 02] NEU, C.: *Design and implementation of a generic quality of service measurement and monitoring architecture*. Technischer Bericht, 2002.

[sgistlasscon] SILICON GRAPHICS, INC.: *http://www.sgi.com/tech/stl/AssociativeContainer.html*, Juni 2000, http://www.sgi.com/tech/stl/AssociativeContainer.html .

Index