

Technische Universität München
Institut für Informatik · Lehrstuhl Prof. Feldmann



Analyse und Performancetests von Paket Replay Programmen

Sebastian Lange
(lange@in.tum.de)

15. November 2004

Ausarbeitung zum Systementwicklungsprojekt

Aufgabensteller: Prof. Anja Feldmann

Betreuer: Vlad Manilici, Holger Dreger

Inhaltsverzeichnis

1	Einleitung	3
2	Hintergrund	3
2.1	Weg eines Paketes	3
2.2	Aufbau eines Ethernetpaketes	6
3	Allgemeines zu den Programmen	8
3.1	Tcpreplay	9
3.2	NetVCR	10
3.2.1	Echtzeitanforderungen von NetVCR	11
3.2.2	DAG-Karten	12
3.3	Funktionenübersicht	12
4	Testmessungen	13
4.1	Hardware Umgebung	13
4.2	Software Umgebung	14
4.3	Tests mit maximaler Geschwindigkeit	15
4.4	Verlusttest	17
4.4.1	Vorgehensweise zur Bestimmung von Paketverlusten	18
4.4.2	Ergebnisse	19
4.5	Untersuchungen mit Inter-Timestamp Intervallen	20
4.5.1	Synthetisch generierte Traces	21
4.5.2	Realer Netzverkehr	23
4.6	Weitere Einflüsse	26
4.6.1	CPU Auslastung	26
4.6.2	Betriebssystem	27
5	Zusammenfassung	28
	Abbildungsverzeichnis	29
	Tabellenverzeichnis	30
	Literaturverzeichnis	31

1 Einleitung

Um Netzwerkgeräte, wie Router, Firewalls und Switches, oder auch ganze Netzwerkarchitekturen zu testen, benötigt man oft die künstliche Erzeugung von Netzverkehr. Dies kann man im Wesentlichen auf zwei Arten erreichen. Zum ersten können mit Hilfe von Paketgeneratoren einfach die gewünschten Pakete neu erstellt, und gegen das Zielgerät geschickt werden. Dagegen erzeugen andere Programme die Pakete, indem sie vorher abgespeicherten Netzverkehr verwenden, also die Quelldatei nur wieder einspielen. Deswegen werden sie hier im folgenden als Replayprogramme¹ bezeichnet. Zwei dieser Replayprogramme, *tcpreplay* und *NetVCR*, werden in dieser Arbeit näher betrachtet. Dabei geht es nicht nur um die grundlegenden Funktionen, sondern auch um die Qualität des Netzverkehrs, den sie erzeugen. Inwiefern kann man wirklich von einem Duplikat des Originals sprechen?

Das Ziel der Arbeit ist es, zu verstehen, wo die Fähigkeiten und Grenzen der Replayprogramme sind. Dabei geht es nicht nur um den maximalen Durchsatz, sondern auch um die Genauigkeit der Wiedergabe der Traces². Deswegen werden hier auch die Paketverluste untersucht, die sich vielleicht beim Wiedereinspielen einschleichen. Außerdem wird überprüft, ob und wie sich die Timestamp-Intervalle vom Original zum Replikat unterscheiden.

Mit den Testergebnissen können dann Ursachen von Veränderungen im erzeugten Netzverkehr erkannt, und Möglichkeiten bzw. Lösungen gefunden werden, die Qualität der Wiedergabertools noch zu verbessern.

2 Hintergrund

Bevor auf die einzelnen Programme detailliert eingegangen wird, werden vorher noch zwei grundlegende Themen erläutert, die im Zusammenhang des Paketeversickens wichtig sind. Zum einen ist dies der Weg eines Paketes vom Benutzer bis ins Netz und zum anderen der Aufbau eines Ethernetpaketes, bzw. seine minimale und maximale Größe.

2.1 Weg eines Paketes

Wenn ein Benutzer ein Replayprogramm aufruft, sieht er nur, dass die Pakete aus einer Datei eingelesen werden und irgendwann nach kurzer Zeit im Netz landen. Bzw. am Zielrechner wieder eingefangen (gesniff) werden können. Doch was passiert auf dem Weg des Paketes durch den Kernel? Folgende Abbildung 1 zeigt die verschiedenen Stationen, die das Paket zu durchlaufen hat.

¹replay: dt.: wiederholen

²trace: dt.: Spur; Bezeichnung für eine Datei mit aufgezeichnetem Netzverkehr

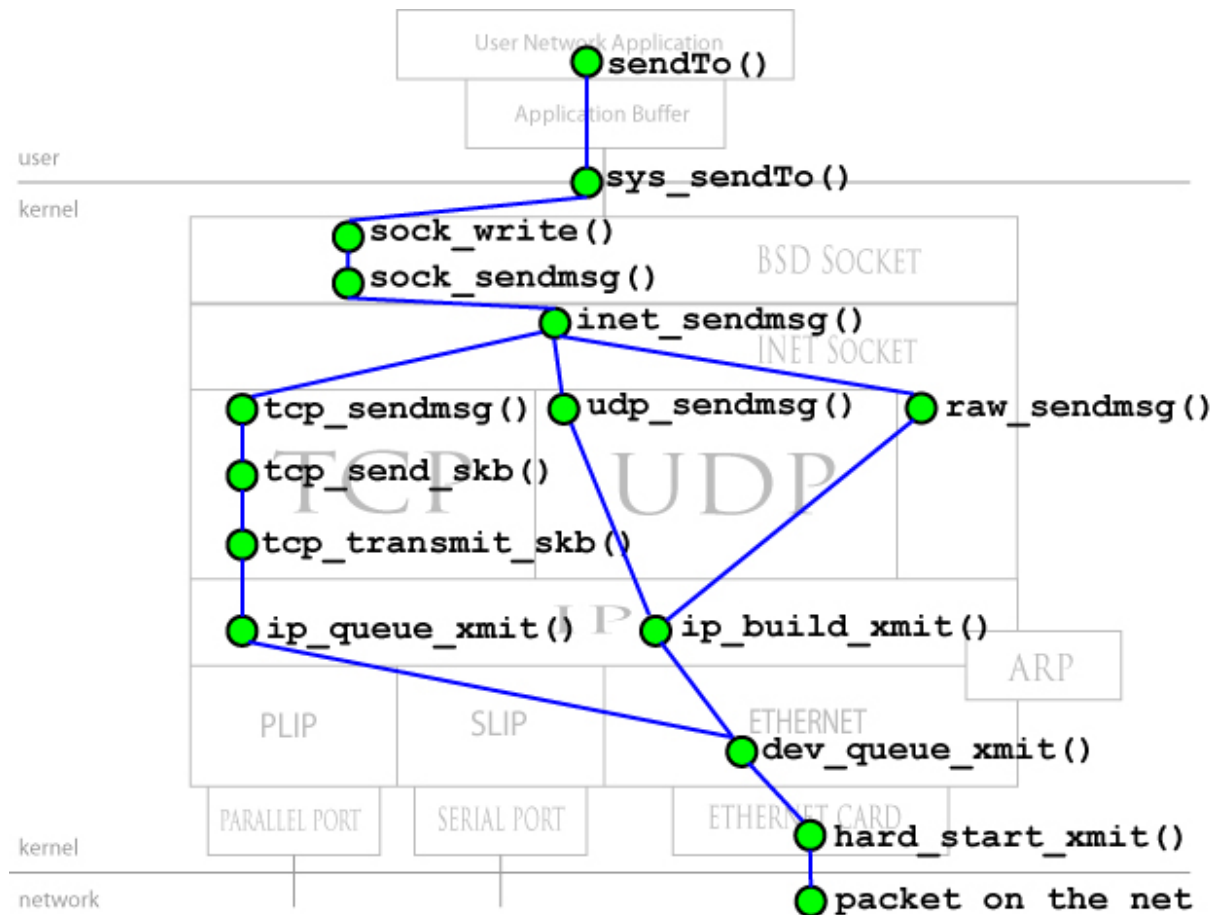


Abbildung 1: Weg eines Paketes durch den Kernel

Anfangen vom Userspace wird das Paket durch die Socket Struktur des Kerns geleitet. Danach „klettert“ es weiter nach unten durch die Transportschicht (transport layer; TCP, UDP), die Netzwerkschicht (network layer; IP) und schließlich die Verbindungsschicht (data link layer; ETHERNET), bis es tatsächlich durch das Kabel übertragen werden kann.

Der hier beschriebene Weg orientiert sich an der Reihenfolge der Funktionsaufrufe, die passieren, wenn man lokal erzeugten Netzverkehr schickt. Im Folgenden werden die einzelnen Funktionen kurz beschrieben und erläutert. Dazu wurden folgende Quellen herangezogen: [HH04], [RC01], [Sev03], [Sta04], [Wei00], [Aip01], [Win02]

- `sendTo()` [*userProgram.c*] Das ist der Aufruf des Benutzerprogramms. Hier wird der neu erzeugte Socket³, die Daten und die Länge des Puffers übergeben.
- `sys_sendTo()` [*net/socket.c*] Der Systemaufruf, äquivalent zum Aufruf des Benutzerprogramms.

³socket: eine bi-direktionale Softwareschnittstelle zur Interprozess- oder Netzwerkkommunikation.

- `sock_write()` [*net/socket.c*] Die Parameter des `sendTo()`-Aufrufs werden in eine Message-Struktur⁴ verpackt.
- `sock_sendmsg()` [*net/socket.c*] Der Pointer zu der Message-Struktur und zu einigen anderen Parametern wird der nächsten Funktion, eine Schicht tiefer, weitergereicht.
- `inet_sendmsg()` [*net/ipv4/at_inet.c*] Extrahiert den Socket und schickt das Paket an die richtige Funktion der Transportschicht. Also `tcp_` oder `udp_sendmsg`, wo die eigentliche Kommunikation beginnt. Denn bis jetzt wurden die Daten des Paketes nur durch verschiedene Abstraktionslevel gereicht.
- `tcp_sendmsg()` [*net/ipv4/tcp.c*] Schreibt die Daten aus der Message-Struktur in einen `sk_buff`⁵. Skbuff's (Socket Buffer) repräsentieren einzelne Datenpakete, die letztendlich verschickt werden. Außerdem werden hier noch einige Fehlerüberprüfungen gemacht und die Daten ins TCP-Paket kopiert.
- `tcp_send_skb()` [*net/ipv4/tcp_output.c*] Die TCP-Pakete werden in eine Warteschlange gesteckt und es wird entschieden, wann die Pakete zu schicken sind.
- `tcp_transmit_skb()` [*net/ipv4/tcp_output.c*] Diese Funktion schickt tatsächlich die TCP-Pakete ab. Hier werden die TCP-Header zusammgebaut und die Checksumme berechnet, damit die Pakete an die Vermittlungsschicht gesendet werden können.
- `ip_queue_xmit()` [*net/ipv4/ip_output.c*] Baut den IP-Header und berechnet die IP-Checksum. Falls das Paket zu groß ist, wird noch ein Aufruf zum Fragmentieren dazwischen geschoben (`ip_fragment()`). Sonst kommen die Pakete in eine weitere Warteschleife und werden danach eine Schicht tiefer gereicht (auf Ebene 2).
- `udp_sendmsg()` [*net/ipv4/udp.c*] Nach verschiedenen Fehlerüberprüfungen wird das UDP-Paket hier fertig erzeugt, und noch ohne `sk_buff` Struktur eine Ebene weiter tiefer, zu IP, gereicht.
- `raw_sendmsg()` [*net/ipv4/raw.c*] Nach verschiedenen Fehlerüberprüfungen wird das RAW-Paket hier erzeugt. Damit werden die Schichten TCP/UDP übergangen.
- `ip_build_xmit()` [*net/ipv4/ip_output.c*] Baut den IP-Header zusammen und berechnet die IP-Checksum. Falls das Paket fragmentiert werden muss, geschieht das hier mit der Funktion `ip_build_xmit_slow()`. Erst jetzt wird eine `sk_buff` Struktur erzeugt und mit den Daten des Paketes gefüllt, um gleich eine Schicht tiefer zu gelangen (auf Ebene 2).
- `dev_queue_xmit()` [*net/core/dev.c*] Hier kommt das Paket (in Form des `sk_buff`) in die letzte Warteschlange, bevor es zum Netzwerkadapter übertragen wird.

⁴Message-Struktur: siehe `usr/src/linux/include/linux/socket.h: struct msghdr{}`

⁵`sk_buff`: siehe `usr/src/linux/include/linux/skbuff.h`

- `hard_start_xmit()` [*drivers/net/deviceDriver.c*] Dieser letzte Funktionsaufruf gibt das Paket an den Netzwerkadapter weiter. Als nächstes sendet dieser das Paket ins Ethernet. Wenn das Paket komplett versendet wurde, wird ein Interrupt ausgelöst, um dies zu signalisieren.

Die unterschiedlichen Funktionen bei TCP und UDP sind wohl damit zu erklären, dass die Protokolle so unterschiedlich sind. TCP ist stateful und UDP nicht. Deswegen muss TCP auch in so vielen Warteschlangen zwischengepuffert werden. Die zwei Ebene-3-Funktionen (`ip_queue_xmit()`, `ip_build_xmit()`) machen im Prinzip beide das Gleiche. Wenn die Protokolle Ebene 2 erreichen, sind die Funktionen des Kernels wieder die selben.

Wie man sieht, passiert zwischen dem Aufruf des Benutzers und dem tatsächlichen Versenden des Paketes noch sehr viel im Kernel. Aus Performancegründen ist es daher nicht sinnvoll, die wiedereingespielten Pakete auch den gesamten Weg durchlaufen zu lassen. Außerdem würden da alle Header Informationen mit den hosteigenen Werten überschrieben werden. Deswegen werden *raw sockets* benutzt, um die Pakete so am TCP/IP Stack vorbei zu schleusen. Trotzdem kann es in diesen Funktionen bei schlechter Umsetzung zu erheblichen Verzögerungen in der Geschwindigkeit kommen und die zur Verfügung gestellte Bandbreite des Netzwerks kann nicht voll ausgenutzt werden.

2.2 Aufbau eines Ethernetpaketes

Um bestimmte Eigenschaften eines Netzwerkes, wie z.B. maximale Dichte des Verkehrs oder Inter-Packet-Gaps, zu messen, braucht man einheitliche Größen, mit denen man arbeiten kann. Bei Paketen wird deswegen häufig der MTU⁶ Wert betrachtet. Um zu wissen, wie groß ein Ethernetpaket maximal bzw. minimal sein darf, wird hier kurz die Zusammensetzung der Pakete erläutert. Denn oft finden sich im Internet verschiedene Angaben, die leicht zu Verwirrungen führen können.

Es wurde festgelegt, dass ein klassisches Ethernetpaket eine Größe von minimal 64 und maximal 1526 Byte haben darf [New04], [Ste94]. Falls ein Paket bei Ankunft auf der Netzwerkkarte kleiner ist, werden Null-Bytes eingefügt, um die minimale Paketlänge zu erreichen.

Folgende Abbildung² zeigt den Aufbau eines Ethernetpaketes. Wie man sehen kann, hat das Paket Felder, die eine bestimmte Größe haben müssen [Str03], [Ste94].

- *preamble*: 56-Bit Muster zur Synchronisation
- *start byte*: '10101011' Immer das selbe Byte, um den Start der eigentlichen Informationen zu kennzeichnen

⁶MTU: Maximum Transfer Unit

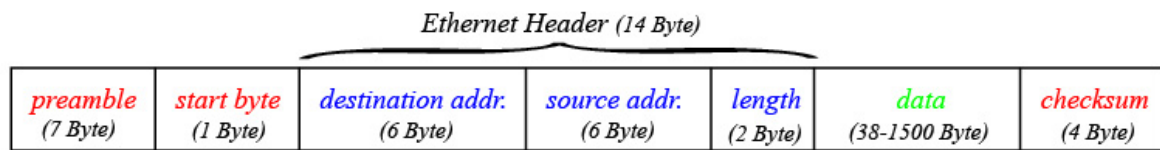


Abbildung 2: Aufbau eines Ethernetpaketes

- *source adress*: eindeutige physikalische 6-Byte MAC⁷ Adresse des sendenden Rechners
- *destination adress*: eindeutige physikalische 6-Byte MAC Adresse des Zielrechners
- *length*: Länge des Datenfeldes in Byte
- *checksum*: 32-Bit Prüfsumme

Zusammen sind das 26 Byte, was dem Datenfeld eine Größe von 38 - 1500 Byte übrig lässt. In mancher Literatur findet man deshalb den MTU Wert eines Ethernetpaketes mit 1500 Byte beschrieben, weil die 26 Byte immer fest vorhanden sind. Es kommt ebenfalls vor, dass 1514 Byte angegeben werden, was dann dem maximalen Datenfeld + dem 14 Byte Ethernetheader entspricht. Hier werden sozusagen nur die *preamble*, das Startbyte und die *checksum* nicht beachtet [Str03].

In das Datenfeld sind alle Informationen weiterer Protokolle aus höheren Schichten eingebettet, z.B. die des IP Protokolls. Die nachstehende Abbildung³ zeigt den Aufbau eines IP-Paketes, welches natürlich nur maximal 1500 Byte groß sein darf. Es muss ja in das Datenfeld des Ethernetpaketes passen. (Bei größeren Datenmengen wird das Paket fragmentiert. Mehr über Paketfragmentierung kann man z.B. unter <http://de.wikipedia.org/wiki/IP-Fragmentierung> nachlesen.) Die einzelnen Felder des IP Headers haben zusammen eine Größe von 20 Byte, werden hier aber nicht einzeln erläutert [Ste94]. Die genaue Beschreibung dazu kann man z.B. im RFC791 nachlesen.

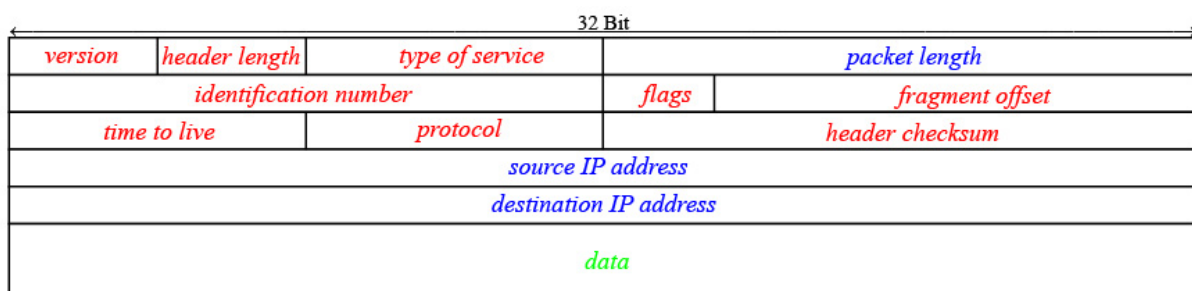


Abbildung 3: Aufbau eines IP-Paketes

⁷MAC-Adresse: Media Access Control; Hardware-Adresse eines jeden Netzwerkgerätes, die zur eindeutigen Identifikation des Geräts im Netzwerk dient.

Dem Datenfeld bleibt somit eine maximale Größe von 1480 Byte, in das Protokolle der vierten Schicht geschrieben werden, z.B. TCP und UDP. In folgenden beiden Abbildungen (4, 5) ist die Zusammensetzung von TCP und UDP Paketen gezeigt.

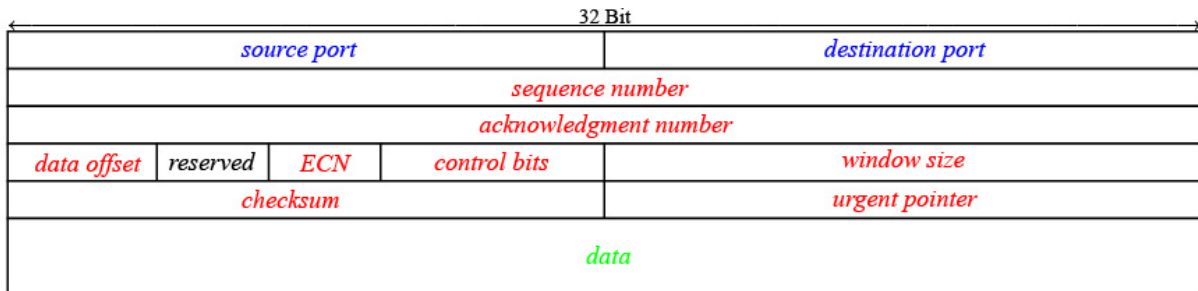


Abbildung 4: Aufbau eines TCP-Paketes

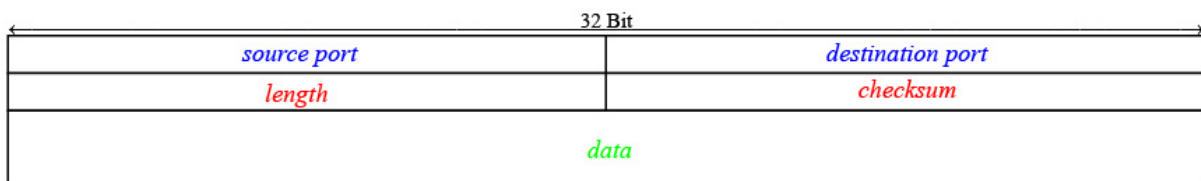


Abbildung 5: Aufbau eines UDP-Paketes

Ein TCP Header hat, genau wie IP, die Größe von 20 Byte, ein UDP Header nur 8 Byte. Damit bleiben für die Payload bei TCP 1460 Byte, bei UDP 1472 Byte übrig. Die genauen Beschreibungen der einzelnen Felder kann man unter [Ste94] oder den entsprechenden RFC's ([RFC793](#), [RFC768](#)) nachlesen.

Es gibt natürlich noch viele andere Protokolle, welche die maximale Nutzlast beschränken, wie z.B. das im IEEE 802.3 Standard definierte LLC⁸ Protokoll, welches das Datenfeld eines Ethernetpaketes noch einmal um 8 auf 1492 Byte reduziert. Allerdings führt die einzelne Betrachtung solcher Protokolle hier zu weit, und es wird auf passende Literatur verwiesen.

In meinen folgenden Tests habe ich mich an dem Ausgabeformat von *tcpdump* und *ethereal* orientiert, und bin von einem MTU Wert von 1514 Byte für ein Ethernetpaket ausgegangen. Alle folgenden Paketgrößenangaben sind tatsächlich auf dem Netz noch um 12 Byte größer (*preamble*, *checksum*; siehe oben).

3 Allgemeines zu den Programmen

Um Pakete ins Netz einzuspielen, benutzen die Replayprogramme folgenden Ansatz: zuerst muss eine Quelldatei erstellt werden, die dann wieder abgespielt werden kann. Entweder man speichert mit Programmen, z.B. *tcpdump*, realen Netzverkehr in einer Datei

⁸LLC: Logical Link Control; <http://www.org.abdn.ac.uk/users/gorry/course/lan-pages/llc.html>

ab, oder kann sie sich mit verschiedensten anderen Tools selbst zusammen setzen. Je nachdem, was für Netzverkehr gerade benötigt wird. Es haben sich im wesentlichen zwei Replayprogramme heraus kristallisiert, die jetzt beide kurz vorgestellt werden.

3.1 *Tcp*replay

Ursprünglich wurde *tcp*replay von Matt Undy von Anzen Computing geschrieben, und die 1.0 Version irgendwann 1999 veröffentlicht [Tur04]. Kurze Zeit danach wurde Anzen Computing von der Firma NFR aufgekauft, und die Weiterentwicklung von *tcp*replay gestoppt. Später, im Jahr 2001, begannen zur selben Zeit 2 Leute, an *tcp*replay weiter zu arbeiten. Matt Bing von NFR und Aaron Turner. Es wurden einige Patches dafür geschrieben, und die Beiden einigten sich darauf, bei der Entwicklung gemeinsam weiter zu arbeiten. Zum jetzigen Zeitpunkt entwickelt nur noch Aaron den Code aktiv weiter [Tur04].

*Tcp*replay ist wahrscheinlich das am weitesten verbreitete Programm, um Netzverkehr wieder ins Netz einzuspielen. Es verwendet Daten, die im *pcap*-Format in einer Datei abgespeichert ist. Libpcap⁹ ist eine plattformunabhängige Schnittstelle, mit der auf Benutzerebene Pakete bzw. Netzverkehr abgefangen werden kann. Genauso wie das weiter unten erwähnte *libnet* ist diese Bibliothek in C geschrieben.

Im Prinzip ist *tcp*replay das genaue Gegenstück zu *tcpdump*, mit dem der Verkehr abgespeichert („gedumpt“) wird. Bis jetzt unterstützt *tcp*replay nur Quelldateien im *pcap*-Format. Um auch andere Speicherformate benutzen zu können, gibt es Tools, die verschiedene Formate ineinander umwandeln können. [Tur04]

Über die Jahre, seit dem *tcp*replay das erste Mal veröffentlicht wurde, hat es einiges an Funktionalität dazu gewonnen. Deswegen wird auch eine frühe Version (*tcp*replay 1.0.1) und die momentan aktuellste Version (*tcp*replay 2.2.2) separat getestet. Hier werden nur die wichtigsten Eigenschaften des Programms erwähnt, alles weitere kann man in Tabelle 1 bzw. den Manual Pages nachlesen.

Angegeben werden muss eigentlich nur der Name der Quelldatei mit den *pcap*-Daten. Dann schickt *tcp*replay die Pakete auf dem Standard Interface raus [UBT03]. Allerdings kann man das Interface sowie die Geschwindigkeit genauer festlegen. So können die Pakete entweder um ein Vielfaches schneller verschickt werden, als sie aufgenommen wurden, oder in genau der selben Geschwindigkeit.

Mit der *exclude*-Funktion kann *tcp*replay 2.2.2 auch nur ausgewählte Pakete einer Quelldatei wieder einspielen, z.B. solche, die von einer bestimmten IP Adresse kommen [UBT03]. Außerdem ist eine *loop*-Funktion vorhanden, mit der die selbe Datei immer wieder von Vorne eingespielt werden kann. Wenn also ein kontinuierlicher Strom von Paketen benö-

⁹libpcap: siehe <http://sourceforge.net/projects/libpcap/>

tigt wird, muss man nicht eine einzige riesige Quelldatei auf dem Server liegen haben, sondern kann eine kleinere sich unendlich oft wiederholen lassen.

3.2 NetVCR

NetVCR bzw. *TCPivo* wurde auf der SIGCOMM¹⁰ 2003 vorgestellt. Zuerst hieß es *NetVCR*, und wurde auch unter diesem Namen programmiert. Doch ein anderes Produkt hatte bereits den Namen schützen lassen, deshalb musste es in *TCPivo* umbenannt werden [Fen]. Es wurde von Mitarbeitern des Systems Software Lab¹¹ der amerikanischen Universität Oregon Health & Science University programmiert, unter anderem von Wu-chang Feng, Wu-chi Feng und Jonathan Walpole.

NetVCR verwendet den selben Ansatz wie *tcpreplay*, nämlich Pakete aus einem abgespeicherten Tracefile wieder als Netzwerkverkehr einzuspielen. Allerdings benutzt es Dateien im ERF-Format¹². Um eine hohe Genauigkeit und Durchsatzrate zu erreichen, benutzt *NetVCR* neue Methoden des Tracefile-Managements und genaue Zeitmessungen, die effizient arbeiten und wenig Overhead produzieren [FGW03]. Wenn man außerdem einen extra gepatchten Linuxkernel benutzt, soll es sehr unabhängig von starker Hintergrundbelastung des Rechners sein [FGW03]. Also immer noch mit der selben Geschwindigkeit Pakete verschicken, wie wenn es alleine auf dem System läuft. Eine genauere Erklärung zu den Echtzeitanforderungen ist im Kapitel 3.2.1 gegeben.

Da die Entwickler ihr Augenmerk primär auf die Performanz ihres Programms gerichtet haben, ist *NetVCR* mit vergleichsweise wenigen, einstellbaren Optionen ausgestattet. Die Einspielgeschwindigkeit kann genauso wie bei *tcpreplay* entweder nach den gespeicherten Timestamps erfolgen, nach einer festgelegten Bitrate pro Sekunde oder mit Maximalgeschwindigkeit.

Außerdem kann man die Puffergröße verändern, die das Programm benutzt, damit es im späteren Verlauf in Echtzeit auf die Pakete zugreifen kann. Je größer man den Puffer macht, desto weniger Verzögerung hat man, weil *NetVCR* zu oft den Puffer neu laden/lesen muss [Net02].

Ein Nachteil von *NetVCR* ist, dass es keine *loop*-Funktion besitzt. Wenn also ein langer kontinuierlicher Strom von wieder eingespieltem Netzwerkverkehr benötigt wird, muss dieser vorher in einer entsprechend großen Datei abgespeichert worden sein. Auch kann man nach keinen bestimmten Paketen filtern, es wird der Verkehr nur so eingespielt, wie er im Tracefile geschrieben wurde. Möchte man dennoch etwas geändert haben, muss man mit anderen Tools das Tracefile vorher per Hand bearbeiten.

¹⁰SIGCOMM: siehe <http://www.informatik.uni-trier.de/ley/db/conf/sigcomm/>

¹¹systems software lab: siehe <http://www.cse.ogi.edu/sysl/>

¹²ERF-Format: extensible record format; siehe Kapitel 3.2.2

3.2.1 Echtzeitanforderungen von *NetVCR*

Es muss gesagt werden, dass *NetVCR* spezielle Realtime- bzw Schedulingtechniken benutzt. Um sie zu verstehen, wird erst grob das Prinzip des Linux Scheduling erklärt. In Linux wird für Benutzerprozesse ein Schedulingverfahren mit dynamischen Prioritäten verwendet. Jeder Prozess, der ausgeführt wird, hat eine bestimmte Priorität, die festlegt, wie wichtig er ist, und wie dringend er ausgeführt werden muss. Weil das Scheduling pre-emptive¹³ ist, kann der Kernel einem Prozess mit niedriger Priorität die CPU entziehen, und Einem mit höherer Priorität sofort Rechenzeit geben.

Weiterhin gibt es andere Schedulingverfahren für besondere Prozesse. Z.B. stellt die Echtzeiterweiterung zwei neue Strategien bereit, womit dann die Prozessbibliothek insg. drei Schedulingverfahren anbietet [Uni02].

- **SCHED_OTHER**: default Scheduling. Jeder Prozess hat eine Anfangspriorität, die vom Scheduler kontinuierlich verändert werden kann. Ohne Echtzeiterweiterung hätten alle Prozesse diese Schedulingstrategie.
- **SCHED_FIFO**: first-in first-out Scheduling. Bei mehreren Prozessen gleicher Priorität läuft derjenige, der zuerst angefangen hat, zuende, bevor der nächste Prozess an der Reihe ist.
- **SCHED_RR**: round-robin Scheduling. Bei mehreren Prozessen gleicher Priorität laufen die Prozesse abwechselnd immer für ein festgelegtes Zeitintervall.

Die Prioritätsklassen werden mit der Echtzeiterweiterung eingeführt und haben einen Integer Wert von 0 - 127 [Uwi99]. 0 ist die Niedrigste und reserviert für alle **SCHED_OTHER** Prozesse. Da **SCHED_OTHER** die Default-Schedulingstrategie für jeden Prozess ist, haben also normalerweise alle Prozesse die Prioritätsklasse 0. Dort konkurrieren sie dann mit ihrem Priority-Wert miteinander. Diesen Wert innerhalb der Prioritätsklasse 0 kann man z.B. auch mit dem `nice()`-Aufruf beeinflussen.

Zeitkritischen und Realtime Prozessen kann mit der Linuxfunktion `sched_setscheduler()` auch eine andere Prioritätsklasse zugewiesen werden. Allerdings werden hierfür Superuser (root)-Rechte benötigt. Diese Prioritätsklassen sind also die restlichen von 1 - 127, also auf jeden Fall wichtiger als **SCHED_OTHER** [Uwi99].

Wenn man sich den Quellcode von *NetVCR* anschaut, sieht man, dass genau das passiert. *NetVCR* weist sich selbst die Schedulingstrategie **SCHED_RR** mit der auf dem System höchsten Prioritätsklasse zu. Wenn es dann gestartet wird, werden also sofort alle Prozesse niedrigerer Priorität hinten angestellt, und warten, bis *NetVCR* fertig ist. Deswegen wird es auch trotz CPU Auslastung nicht langsamer, weil die auslastenden Prozesse nur standardmäßig mit **SCHED_OTHER** laufen, und so vom Kernel ausgesetzt werden. *Tcpreplay*

¹³pre-emptive: dt. unterbrechend

ist auch „ganz normal“ wie alle anderen Prozesse `SCHED_OTHER`, und konkurriert deswegen mit den auslastenden Prozessen um die CPU, was es langsamer macht.

Dazu sollte man erwähnen, dass *NetVCR* entwickelt wurde, als man noch den 2.4 Kernel benutzt hat. Diesen Kernel musste man extra patchen, damit er für Echtzeit- bzw. zeitempfindliche Anwendungen geeignet war. In [GAK02] wird beschrieben, wie und warum man unter Verwendung von Firmtimern, Pre-emptibility- und Low-latancy-Patch den 2.4 Linuxkernel für zeitempfindliche Anwendungen aufwerten kann. In neueren Kernen, wie 2.5 oder 2.6, sind diese Verbesserungen allerdings schon mit eingebaut [Die03], weswegen man nicht noch einmal extra patchen muss.

3.2.2 DAG-Karten

Momentan unterstützt *NetVCR* nur Paketdateien im ERF-Format. Das ist ein spezielles Format, welches von der DAG Software erzeugt wird, und an der University of Waikato in Neuseeland zum ersten Mal spezifiziert wurde. Sie haben dort spezielle und vor allem genauere Snifferhardware (sog. DAG Karten) entwickelt, wozu ERF das Ausgabeformat ist. Allerdings kann man mit deren Software [Dag01] problemlos *pcap*-Dateien und andere in das ERF-Format umwandeln, und sie so auch mit *NetVCR* benutzen. Für *pcap*-Dateien heißt es `bpfdag`. Die erforderliche Software ist im Ziparchiv von *NetVCR* bereits enthalten. Für die Konvertierung von *pcap* nach ERF heißt das Tool `dagconvert`. Genauere Informationen zu dem DAG-Projekt sind auf folgender Homepage zu finden. <http://dag.cs.waikato.ac.nz/>

3.3 Funktionenübersicht

Funktion	tcpreplay 1.0.1	tcpreplay 2.2.2	NetVCR
split traffic	-	X	-
rewrite ports	-	X	-
rewrite IP	-	X	-
rewrite MAC	-	X	-
fix checksums	-	X	-
loop packet data	X	X	-
maximum speed	X	X	X
multiple speed by factor	X	X	-
fixed packed rate (pps)	-	X	-
fixed rate (mbps)	X	X	X
set random IP	-	X	-
include/exclude packets	-	X	-
allowed packets	all	all	TCP,UDP,ICMP

Tabelle 1: Überblick der Programmoptionen

4 Testmessungen

Nachdem nun die Programme vorgestellt wurden, werden im Folgenden die Testläufe mit den einzelnen drei Programmen beschrieben und die Ergebnisse erläutert. Es wird untersucht, ob Pakete beim Sendevorgang verloren gehen, wie groß die maximale Bandbreite ist und wie genau die Timestamps beim Abspielen eingehalten werden. Bei auftretenden Fehlern wird auch versucht, die Gründe dafür zu erklären. Dazu wird erst einmal die gesamte Messumgebung vorgestellt, die Struktur der Messungen erklärt, und danach auf die drei unterschiedlichen Tests eingegangen.

4.1 Hardware Umgebung

Die Testrechner im Routerlab (Abbildung 6) waren mit einem Glasfaserkabel direkt miteinander verbunden. Damit wird die Komplexität des Weges reduziert und mögliche Fehlerquellen ausgeschlossen. Zur Steuerung sind beide Rechner an das Netz des Lehrstuhls angebunden, wo man ohne Probleme remote darauf zugreifen kann.

Der Senderechner ('MON') und der Empfängerrechner ('PF') waren in Sachen Hardware gleich ausgestattet. Für die Tests wichtige Komponenten waren:

- AMD Athlon XP 2400+ Prozessor
- 1 GB DDR RAM
- 64K L1 Cache / 256K L2 Cache
- 1Gbit Netgear Glasfaser Netzwerkkarte
- Standard PCI Bus

Der einzige Unterschied der beiden Computer lag im benutzten Betriebssystem. Beide hatten ein Debian Linux installiert, wobei 'MON' schon den neuen 2.6.5 Kernel benutzt, und 'PF' noch den alten 2.4.25. Es wurde so gelassen, da für spätere Tests die Funktion des Sende- und Zielrechners umgedreht wurde, um die Einflüsse des Betriebssystems zu untersuchen.

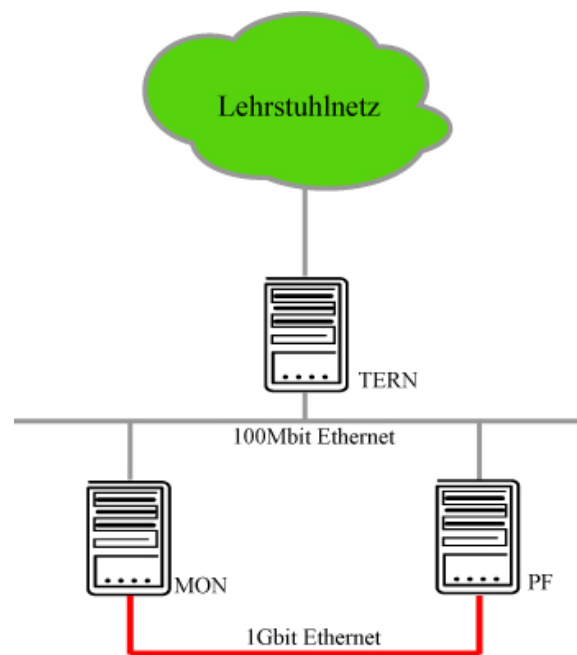


Abbildung 6: Aufbau der Testumgebung

Zuerst wurde angefangen, auf dem 100Mbit Interface der beiden Rechner zu testen. Da dies jedoch für die Programme viel zu langsam war, um irgendwelche Grenzen zu erkennen, wurden die Messungen verworfen, und nur noch auf dem Gigabit Link ausgewertet.

4.2 Software Umgebung

Um die Replayprogramme zu füttern, braucht man natürlich erst einmal die richtigen Pakettraces dazu. Diese wurden für die unterschiedlichen Tests synthetisch erzeugt, damit man sie gut und schnell auswerten kann. Außerdem bekommt man da keine Unregelmäßigkeiten, wie bei realem Netzverkehr. (Später wurden die Testergebnisse natürlich auch an einem realen Trace verifiziert.)

Es wurde ein Perlscript namens `edcap` von Vlad Manilici¹⁴ benutzt, welches einzelne Ethernetpakete im `pcap`-Format erzeugen kann. Der Vorteil hieran ist, dass man alle Werte nach Belieben festlegen kann, was allerdings die Kenntnis von Ether- IP- und TCP-Header erfordert.

Die anfängliche Überlegung war, reine Ethernetpakete ohne tiefere Protokolle zu schicken. Sozusagen sich den IP und TCP Header sparen, und somit die Laufzeit von `edcap` zu verringern. Allerdings wurden wegen der oben schon erwähnten Schwäche von `NetVCR`, nur TCP, UDP und ICMP Pakete abspielen zu können, in den Traces nur TCP Pakete erzeugt und die Idee mit den reinen Ethernetpaketen verworfen.

Die Anzahl der Pakete in den Traces bewegt sich in der Größenordnung von 100000 hoch bis zu 2Mio Paketen. Um sie immer noch alle einzigartig und somit unterscheidbar zu machen (z.B. für Verlust-Tests), reicht das ID-Feld im IP Header nicht mehr aus, da es nur ein 16-Bit Feld¹⁵ ist. Die ID der Pakete wurde deshalb ein Protokoll höher, im TCP Header versteckt. Da es eh kein realer Verkehr ist, wurde die TCP Sequenznummer bei jedem Paket um 1 erhöht. Bei einem 32-Bit Feld sind das über 4 Milliarden Werte, was völlig ausreichend ist.

Um die Ergebnisse aussagekräftig zu machen, ändert sich z.B. die Paketgröße oder das Zeitintervall innerhalb eines Traces nicht. Es wurden jedoch verschiedene Traces mit unterschiedlichen Ausgangseinstellungen generiert.

Weiterhin mussten am Empfängerrechner die ankommenden Pakete wieder aufgezeichnet werden. Hierfür wurde `tcpdump`¹⁶ verwendet. Allerdings hat `tcpdump` auch seine Fehler. Die Ascii Ausgabe ist sehr langsam, und auch die eingebaute Adresskonvertierung braucht viel Zeit. Deswegen werden beim „Einfangen“ der Pakete sehr viele übersehen. Man kann das umgehen, indem man die Daten von `tcpdump` nicht sofort auswerten lässt, sondern erst einmal in eine Binärdatei schreibt, um sie später mit einem erneuten `tcpdump` Aufruf auszulesen. Das bringt schon wesentlich weniger Verluste, aber beim Aufzeichnen großer Pakete (z.B. MTU Pakete) gehen immer noch zu viel verloren. Um diesen Effekt zu vermeiden, kann man die Pakete gleich nach dem TCP Header (54 Byte) abzuschneiden, und den Rest der Nutzlast zu verwerfen. Denn in der Nutzlast steht sowieso nur zufällig erzeugter Datenmüll und die Paket-ID ist im TCP Header gerettet. Die beiden Aufrufe sahen also so aus:

¹⁴V. Manilici: Mitarbeiter am Lehrstuhl Netzwerkarchitekturen TU München

¹⁵16-Bit Feld: entspricht $2^{16} = 65536$ verschiedenen Einträgen

¹⁶tcpdump: siehe <http://www.tcpdump.org/>

```
tcpdump -s 54 -i <iface> -w <file> (zum Aufzeichnen)
```

```
tcpdump -nr <file> (um die Daten anzuschauen)
```

Auf die Verlustraten von *tcpdump* wird im Abschnitt „Verlusttest“ noch einmal genauer eingegangen.

4.3 Tests mit maximaler Geschwindigkeit

Zuerst wurden die Testprogramme auf ihre maximale Geschwindigkeit (in pps¹⁷) und ihre maximale Bandbreite (Mbps¹⁸) hin untersucht. Dazu musste erst festgelegt werden, wie die Zeitdauer des Sendevorgangs am besten bestimmt werden konnte. Für *tcpreplay* bieten sich dabei drei Möglichkeiten an.

1. Man kann die eigenen Angaben von *tcpreplay* benutzen, die nach jeder erfolgreichen Beendigung des Programms auf `STDOUT` ausgegeben werden. Hier ist unter Anderem die Dauer in Sekunden schon in die Bandbreite umgerechnet.
2. Die von *tcpreplay* gesendeten Pakete können am Zielrechner mit *tcpdump* wieder aufgezeichnet werden. Dann wird der Timestamp des ersten und letzten Paketes betrachtet und die Differenz von beiden berechnet. Das ist dann die Zeit, die *tcpreplay* benötigt hat, die Pakete zu senden. Natürlich muss man sich bei dieser Möglichkeit vorher versichern, dass keine Pakete verloren gegangen sind.
3. Man kann im Testscript, welches *tcpreplay* immer wieder ausführt, direkt vor und nach jedem Aufruf mit `gettimeofday()`¹⁹ den genauen Timestamp speichern. Hier wird auch die Differenz von Beiden berechnet, was dann die Laufzeit des gesamten Programms ist.

Nach Vergleichen zwischen den 3 Meßmethoden konnte man sagen, dass es keinen Unterschied macht, wie man die Laufzeit/Sendezeit bestimmt. Alle 3 Werte waren im Rahmen der Messgenauigkeit identisch. Bei Version 3 wird zwar der Start-Up²⁰ des Programms mitgemessen, der aber im Bereich von 0,001s liegt und deswegen bei Messungen von mehreren 100000 Paketen nicht ins Gewicht fällt.

Methode 1 wäre die sinnvollste gewesen, aber da *NetVCR* keine eigenen Angaben macht, fällt sie weg. Damit alle drei Programme gleich gemessen werden, wurde Methode 3 gewählt, weil sie sich einfacher zu implementieren lies als Methode 2.

¹⁷pps: Pakete pro Sekunde

¹⁸Mbps: Megabits pro Sekunde

¹⁹`gettimeofday()`: Funktionsaufruf in C oder Perl, der den Timestamp bis auf 6 Stellen nach dem Komma angibt

²⁰Start-Up: Die Zeit, die das Programm braucht, um Initialisierungen etc vorzunehmen, bevor es mit der eigentlichen Funktion beginnt

Nach anfänglichen Probemessungen wurde festgestellt, dass das Protokoll für die Geschwindigkeit keine Auswirkungen hat. Es ist egal, ob ein 1514 Byte TCP oder UDP Paket geschickt wird. Deswegen wurden alle weiteren Messungen nur mit TCP Paketen gemacht.

Um einen guten Durchschnittswert zu bekommen, wurde 3000 mal 100000 Pakete verschiedener Größen abgeschickt, gemessen, und davon dann der Mittelwert bestimmt. Zuerst sollten immer 1 Mio. Pakete gesendet werden, was jedoch zu einem Problem mit dem Plattencache geführt hat. Darauf wird weiter unten im Kapitel noch eingegangen.

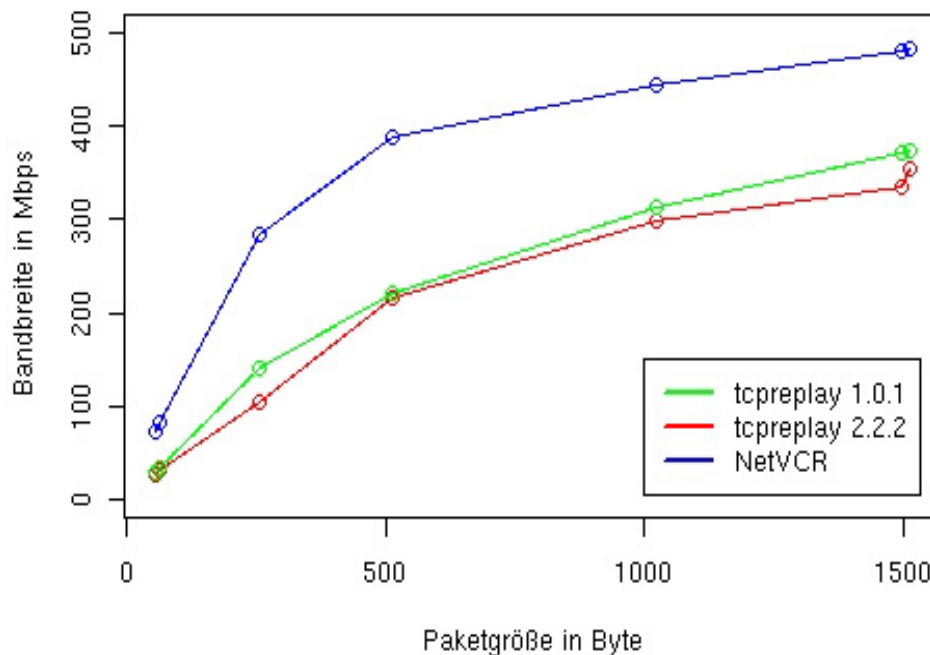


Abbildung 7: Vergleich der Bandbreiten alle drei Programme

Wie in Abbildung 7 zu sehen ist, wurden die Tests mit verschiedenen Paketgrößen durchgeführt, nämlich 54, 64, 256, 512, 1024, 1496 und 1514 Byte. Minimal große Pakete treten z.B. beim Verbindungsaufbau auf, maximale bei großen Downloads. Zum Beispiel ist 54 Byte ein TCP Paket ohne Payload, 64 Byte die Mindestgröße eines Paketes bei der Übertragung und 1514 Byte der MTU Wert bei Ethernet.

Es ist deutlich zu erkennen, dass bei kleiner Paketgröße auch die Bandbreite geringer ist. Das liegt daran, dass nicht genug Pakete in der Sekunde geschickt werden können, um den selben Durchsatz an Bytes zu erreichen wie bei den großen Paketen. Die unterschiedlichen Ergebnisse beider *tcpreplay* Versionen kann man damit erklären, dass in der neueren Version viel mehr Möglichkeiten überprüft werden müssen, d.h. die CPU für jedes Paket einfach mehr rechnen muss. Die alte Version ist noch sehr einfach gehalten, und somit auch ein bisschen schneller. Die maximal erreichten Bandbreiten für die Programme waren: *tcpreplay 1.0.1* - 373,14Mbps, *tcpreplay 2.2.2* - 355,26Mbps und *NetVCR* - 483,54Mbps.

In der folgenden Tabelle 2 sind noch einmal alle Meßwerte in Zahlen aufgelistet.

	Paketgröße in Byte						
	54	64	256	512	1024	1496	1514
tcpreplay 1.0.1	28,98	33,08	140,95	221,68	312,74	371,24	373,07
tcpreplay 2.2.2	25,99	30,33	104,50	215,89	298,09	334,02	355,26
NetVCR	71,33	82,34	284,57	389,29	443,79	481,62	483,54

Tabelle 2: Maximalbandbreiten in mbps

Bei kleinen Paketen kommen pro Sekunde wesentlich mehr auf die Leitung, allerdings, weil sie so klein sind, bringen sie nicht genug Bytes an Durchsatz. Für das alte *tcpreplay* und *NetVCR* schauen die Ergebnisse verhältnismäßig gleich aus.

Zwei Besonderheiten sind bei den Tests aufgefallen. Die erste ist, dass die Geschwindigkeit von *NetVCR* bei kleinen Paketgrößen stark schwankt. Teilweise dauerte es 0,1s länger, die 100000 54 Byte Pakete zu verschicken, als die mit 64 Byte. Deswegen wurde die Anzahl der Wiederholungen auch auf 3000 gesetzt, um trotzdem einen guten Durchschnittswert zu erreichen.

Als zweites fällt auf, dass der erste Messwert einer Reihe immer um mindestens die Hälfte kleiner ist, als der Durchschnitt. Die Vermutung war, dass beim ersten Durchlauf das Trace von der Platte eingelesen und dann im Festplattencache abgelegt worden ist. Alle weiteren Durchläufe konnten dann natürlich direkt aus dem Cache auf die Datei zugreifen, was wesentlich schneller geht. Um das zu bestätigen, wurde mit *hdparm* die Geschwindigkeit des Lesens von der Festplatte bestimmt. Auf 'MON' waren es 7,39MB/s (= 59,12Mbps). Dieser Wert entspricht ziemlich genau der Bandbreite, die der erste 1514 Byte Lauf gebracht hat, nämlich 54,89 Mbps.

Das selbe Phänomen kann man auch erreichen, wenn man die Paketanzahl in dem Tracefile sehr groß macht. Bei 1514 Byte Paketen reichen schon 500000+ Stück. Man kommt trotz wiederholtem Messen nur auf eine Bandbreite von ca. 54Mbps, also genau das Selbe wie bei den ersten Durchläufen im obigen Abschnitt. Das bestärkt natürlich die Vermutung, dass das Problem am Cache liegt. Er ist wahrscheinlich zu klein, d.h. es wird der Traceanfang mit dem Traceende im Cache überschrieben, und beim wiederholten abspielen stehen die ersten Pakete schon nicht mehr zur Verfügung.

Wenn man den Cache von Hand löscht, oder sicher mit anderen Daten füllt, kommen ebenfalls die schlechten Bandbreiten zustande.

4.4 Verlusttest

Nun wurde überprüft, wie verlässlich die Replayprogramme sind und ob sie Pakete verlieren oder übersehen. Wenn man sie mit maximaler Geschwindigkeit laufen lässt, ist die Wahrscheinlichkeit für Paketverlust intuitiv am größten. Allerdings sagen z.B. die Pro-

grammierer von *tcpreplay*, dass ihr Programm keine Pakete verliert, und das entweder am Kernel oder am Sniffer²¹ liegt [Tur04]. Genau das wird in diesem Abschnitt untersucht.

4.4.1 Vorgehensweise zur Bestimmung von Paketverlusten

Die Schwierigkeit liegt darin, zu bestimmen, wer Pakete verloren hat. Denn wenn sie nicht wieder aufgezeichnet werden, kann entweder der Sniffer oder das Sendeprogramm schuld am Verlust der Pakete sein. Man kann die Wahrscheinlichkeit, dass der Sender verliert, also nur eingrenzen, aber nicht sicher bestimmen. Folgendes Schema soll die Idee hinter dem Test erklären.

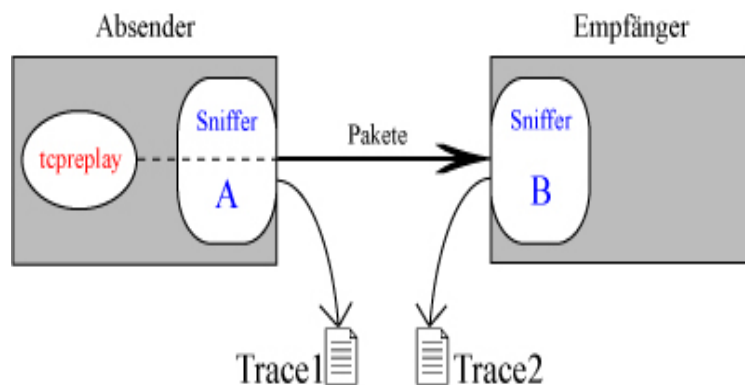


Abbildung 8: Schema für die theoretische Bestimmung von Paketverlusten

Es wird noch auf dem Senderechner aufgezeichnet, bevor die Pakete übers Netz gehen (A). Dann auf dem Zielrechner noch einmal (B). Danach werden die Pakete beider aufgezeichneten Traces anhand ihrer ID verglichen, um Verluste festzustellen. Wenn ein Paket in beiden Traces nicht vorkommt, kann es sein, dass beide Sniffer es übersehen haben, oder es besteht die Möglichkeit, dass es tatsächlich vom Sender nicht abgeschickt wurde. Denn alle Pakete, die bei wenigstens einem Trace vorhanden sind, wurden sicher geschickt, und nur von dem anderen Sniffer nicht erkannt.

Man kann den Verlust des Senders folgendermaßen rechnerisch eingrenzen:

- a, b = die zwei unabhängigen Dumps
- N = die Gesamtmenge von Paketen
- V = die Menge von Paketen, die weder bei a noch bei b angekommen sind
- K = die Menge von Paketen, die tatsächlich beim Sender verloren gegangen sind: $[K] < [V]$
- V_a = die Menge von Paketen, die nur bei a verlorengegangen ist
- V_b = die Menge von Paketen, die nur bei b verlorengegangen ist
- P_a = die Wahrscheinlichkeit, dass a irgendein Paket verliert
- P_b = die Wahrscheinlichkeit, dass b irgendein Paket verliert

²¹sniffer: Programm, was Netzverkehr aufzeichnen kann, z.B. *tcpdump*

P_v = die Wahrscheinlichkeit, eines gleichzeitigen Verlustes auf beiden Seiten:
 $P_v = P_a * P_b$

Aus den Messungen sind die Werte N , V , V_a und V_b bekannt. Damit wird versucht, K genauer einzugrenzen als $0 < K < V$.

Die Werte P_a und P_b kann man nicht genau bestimmen, sondern auch nur eingrenzen:

$$(1) \quad \frac{V_a}{N} < P_a < \frac{V+V_a}{N}$$

$$(2) \quad \frac{V_b}{N} < P_b < \frac{V+V_b}{N}$$

Daraus ergibt sich für $P_v = P_a * P_b$:

$$(3) \quad \frac{V_a * V_b}{N * N} < P_v < \frac{(V+V_a) * (V+V_b)}{N * N}$$

K kann man so berechnen:

$$(4) \quad K = V - P_v * N$$

Jetzt das eingegrenzte P_v einsetzen:

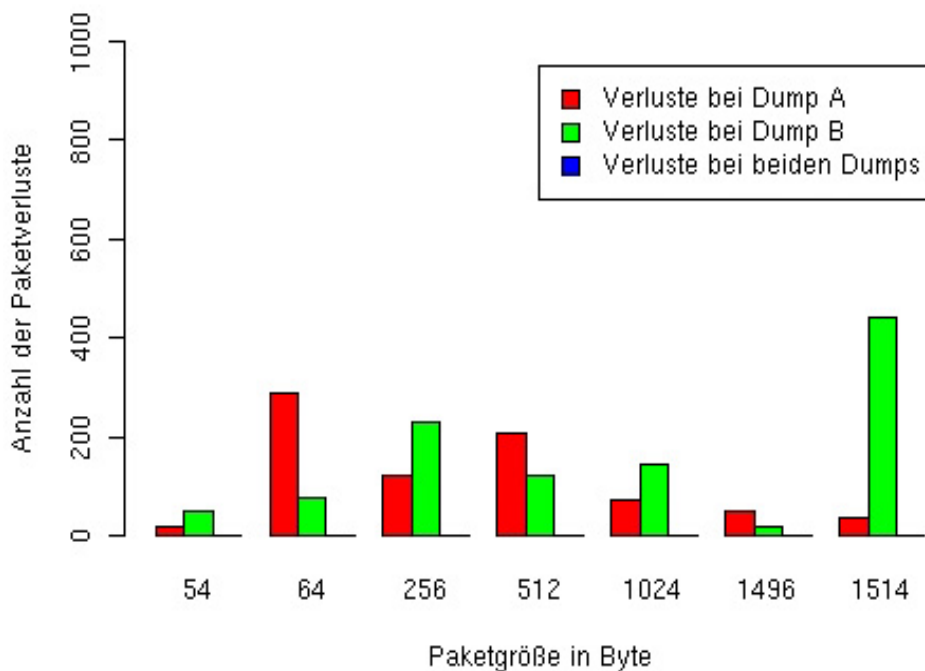
$$(5) \quad V - \frac{(V+V_a) * (V+V_b)}{N} < K < V - \frac{V_a * V_b}{N}$$

4.4.2 Ergebnisse

Wenn man die Testergebnisse aus Abbildung 9 betrachtet, fällt auf, dass zwar immer Pakete bei einem der Dumps verloren gehen, allerdings sind es nie die Selben. Die Testwerte sind die Durchschnittswerte von 20 Messungen und bei keiner einzelnen Messung fehlte einmal das gleiche Paket in beiden Dumps. Deswegen kann man mit ziemlich großer Sicherheit behaupten, dass *tcpreplay* keine Pakete verliert, sondern nur die Sniffer ab und zu welche übersehen. Man kann daraus für *tcpdump* eine durchschnittliche Verlustquote von 0,02% – 0,44% bestimmen. Auf die Fehler von *tcpdump* wird auch in anderen Arbeiten, z.B. in [Sch04] eingegangen.

Die Ergebnisse von *tcpreplay 2.2.2* und *1.0.1* waren fast identisch, weswegen hier stellvertretend *tcpreplay 2.2.2* gezeigt wird. Die Ergebnisse von *NetVCR* hingegen weichen sehr stark von Denen mit *tcpreplay* ab, wie es Abbildung 10 zeigt.

Der Sniffer auf dem Senderechner übersieht bei kleinen Paketgrößen fast alle Pakete (99,6%), während der auf dem Zielrechner die normale Verlustquote aufweist. Beim genaueren Betrachten stellt man fest, dass die gesniffen Pakete immer die ersten ca. 300 vom Anfang des Traces sind. Alle späteren werden auf dem Senderechner übersehen. Die Erklärung hierfür ist die Echtzeitimplementierung von *NetVCR*. Danach würde *tcpdump*, nach den ersten Paketen, die CPU entzogen werden, und erst wieder zugeteilt werden,

Abbildung 9: Paketverluste beim Test mit *tcpreplay*

wenn der Sendevorgang bereits beendet ist. Bei großen Paketen (über 1024 Byte) ist der Verlust des Sniffers auf dem Senderechner immer noch ca. die Hälfte.

Da allerdings der zweite Dump keine höheren Verluste zeigt als bei *tcpreplay*, kann man davon ausgehen, dass auch *NetVCR* keine Pakete verliert. Eine 100 prozentige Sicherheit gibt es nur, wenn man beispielsweise einen Router zwischen die Verbindung beider Rechner hängt, und dann direkt im Router abfragt, wie viele Pakete dort durch gekommen sind. Das würde hier jedoch zu weit führen, und wurde deshalb nicht ausprobiert.

4.5 Untersuchungen mit Inter-Timestamp Intervallen

Als letztes wurden die Abstände zwischen den einzelnen Paketen beim wieder einspielen untersucht. Das Ergebnis dieser Tests soll zeigen, wie gut die Replayprogramme vorgegebene Timestamps einhalten und Erklärungen finden, warum Paketabstände vom Originaltrace abweichen.

Der Versuchsaufbau für diese Untersuchungen ist recht einfach. Auf dem Zielrechner läuft *tcpdump*, was den gesamten gesendeten Netzverkehr wieder auffängt. Dann werden die Timestamps des Dumps extrahiert und mit denen des Originals verglichen. So kann man erkennen, in welchen Abständen die Pakete angekommen bzw. verschickt worden sind. D.h., es wird im Prinzip die Differenz zweier aufeinanderfolgender Timestamps betrachtet. Die einzigste Schwierigkeit war, dass *tcpdump* keine Pakete verlieren durfte, weil sich sonst

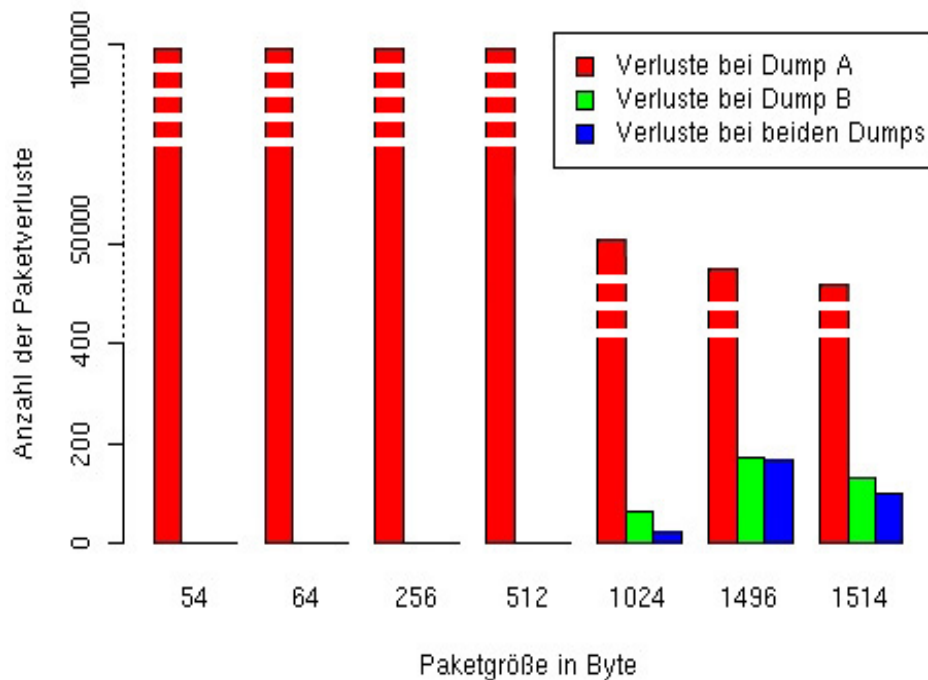


Abbildung 10: Paketverluste beim Test mit NetVCR

die beiden Timestamps vor und nach dem verlorenen Paket addieren, und das Ergebnis zu sehr aus der Reihe tanzt.

Man darf nicht die Timestamp Intervalle mit dem Begriff Inter-Paket-Gap verwechseln. Der Inter-Paket-Gap bezeichnet die Zeit zwischen Ende des ersten und Anfang des zweiten Paketes auf dem Draht, den man jedoch hier wegen mangelnder Messgenauigkeit nicht exakt bestimmen kann. Die Timestamps sind nicht so genau gesetzt, außerdem ist in jedem Timestamp Intervall die Zeit, die das Paket selbst braucht, mit eingerechnet.

4.5.1 Synthetisch generierte Traces

Das erste Ziel war es, den minimalsten Abstand zweier Timestamps zu bestimmen, welche die Replayprogramme schaffen. Dazu wurde ein Trace mit 100000 64 Byte Paketen mit maximaler Geschwindigkeit verschickt. *Tcp replay 1.0.1* erzielte einen Abstand von durchschnittlich $14\mu s$ und *2.2.2* hatte $15\mu s$. *NetVCR* war nicht klar zu bestimmen, da *tcpdump* manchmal zwei identische Timestamps aufeinander folgen hatte, was natürlich eine Differenz von 0 ergab. Die Vermutung ist, dass die Pakete zu schnell für *tcpdump* angekommen sind, und deswegen der augenscheinlich gleiche Timestamp verwendet wurde. Im Durchschnitt war der Abstand zweier Timestamps bei *NetVCR* $6\mu s$.

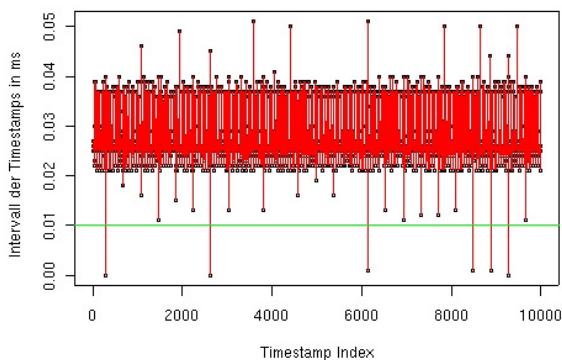
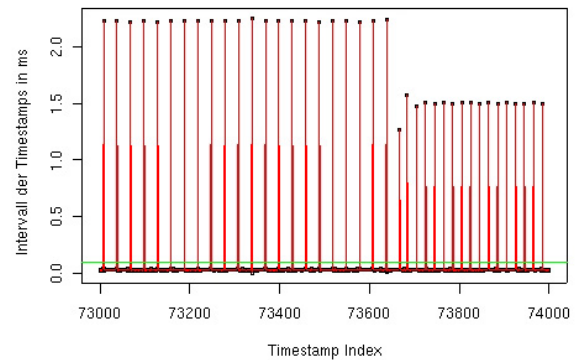
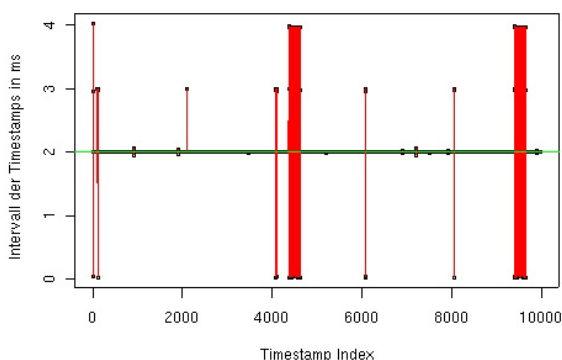
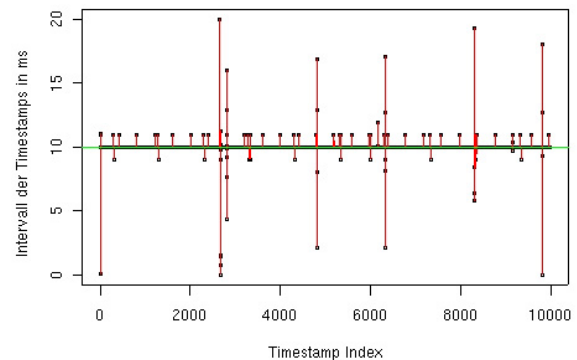
Als nächstes wurde ein Trace erzeugt, was deutlich zu kleine Timestamps hatte ($1\mu s$ Differenz), als dass sie eingehalten werden könnten. Theoretisch müssen die Pakete von

den Replayprogrammen auch hier in größtmöglicher Geschwindigkeit gesendet werden. Das ist jedoch nicht der Fall. Sobald man die Option „Maximalgeschwindigkeit“ nicht angibt, betrachten die Programme jeden Timestamp, was pro Paket ca. $10\mu s$ mehr an Zeit benötigt. Folgende Tabelle 3 zeigt die Ergebnisse.

	tcpreplay 1.0.1	tcpreplay 2.2.2	NetVCR
mit Maximalgeschwindigkeit	$14\mu s$	$15\mu s$	$6\mu s$
mit $1\mu s$ Timestamps	$23\mu s$	$25\mu s$	$17\mu s$

Tabelle 3: kleinstmögliche Timestamp Differenzen

Nun kamen Testtraces an die Reihe, die Timestamp-Differenzen von $0,1ms$ oder höher hatten. Damit lagen sie also deutlich über den schnellsten Möglichkeiten der Replayprogramme. Folgende Abbildungen zeigen als schwarze Linie die vorgegebenen Abstände und als rote die Werte nach dem Senden mit *tcpreplay*.

Abbildung 11: Timestamp Differenzen bei $10\mu s$ Abbildung 12: Timestamp Differenzen bei $100\mu s$ Abbildung 13: Timestamp Differenzen bei $2000\mu s$ Abbildung 14: Timestamp Differenzen bei $10000\mu s$

Es ist zu erkennen, dass die vorgegebenen Werte nicht eingehalten werden. Allerdings ist die Gesamtzeit, die zum Schicken benötigt wird, nicht kürzer oder länger als die Originalgesamtzeit.

Auf den ersten Blick hin ist es bei kleineren Abständen sehr viel ungenauer als bei größeren. In Abbildung 12 kann man sehen, wie viele kleine Differenzen (im Bereich der max. Geschwindigkeit) gefolgt werden von einer Großen (ca. $1 - 2ms$). Sozusagen, um die verlorene Zeit von vorher wieder auszugleichen. Diese großen Differenzen kommen immer in den selben Abständen. Eine Möglichkeit ist es, dieses Verhalten durch den Linuxschedulingtimer zu erklären. Dieser ist nicht genauer als $1ms$ [Kos98]. *Tcpreplay* schickt also das erste Paket, und stellt fest, dass es noch $100\mu s$ warten muss, bis das nächste zu schicken ist. Das nächste mal jedoch, als *tcpreplay* dran ist, sind schon ca. $1,5ms$ vergangen, also hätten schon viel mehr Pakete verschickt werden müssen. Jetzt sendet *tcpreplay* also so schnell wie möglich noch alle ausstehenden Pakete raus, um wieder mit der Zeit aufzuholen. Irgendwann erreicht es wieder das Paket, was jetzt wirklich an der Reihe war, und beschließt, wieder $100\mu s$ zu warten, bis das nächste kommt. Woraus dann wieder, wegen dem Timer, im Mittel $1,5ms$ werden...

Die Abbildungen 13 und 14 zeigen, dass bei einem Intervall $> 2ms$ in der Quelldatei die Ausgabe von *tcpreplay* sehr viel genauer wird. Allerdings sind immer noch Sprünge zu erkennen, wo ein Paket mit zu großem Abstand durch eines mit zu Kleinem wieder ausgeglichen wird.

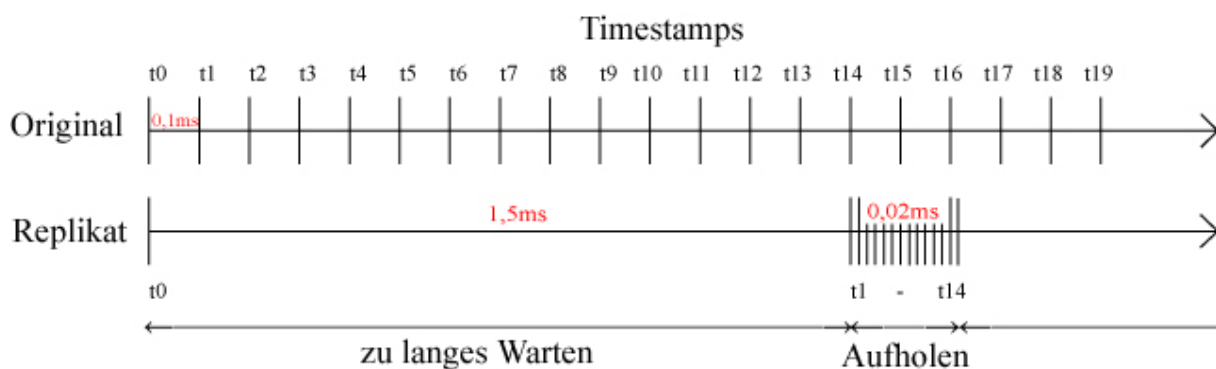


Abbildung 15: Schema zum „Nachschicken“ von *tcpreplay*

Man kann also behaupten, dass bei einem Paketabstand größer als $2ms$ das Trace halbwegs genau wiedergegeben wird. Je größer die Paketabstände, desto besser. Ein „schnelles“ Trace, d.h. Paketabstände deutlich unter $1ms$, kann wegen mangelnder Genauigkeit nur unzureichend repliziert werden.

Außer, dass die kleinsten Abstände bei *NetVCR* einige μs geringer sind, zeigt sich in der Verteilung das selbe Bild.

4.5.2 Realer Netzverkehr

Mit diesen Erkenntnissen wurde ein Trace abgespielt, was nicht mit extra 100% gleichen Paketabständen zusammen gesetzt war, sondern echt vom Lehrstuhlnetz gesniff wurde. Wenn man sich die direkten Timestamps von Original und Replay anschaut, ist fast kein

Unterschied zu erkennen. Dazu ist die Skala nicht fein genug (siehe Abbildung 16). Auch in der Häufigkeitsverteilung der einzelnen Paketabstände kann man nicht viel heraus lesen. Da im Endeffekt die Zeit genau gleich bleibt, kann man hier nichts erkennen, da die Verschiebungen innerhalb der Abstandstruktur nicht ersichtlich sind. (siehe Abbildung 17)

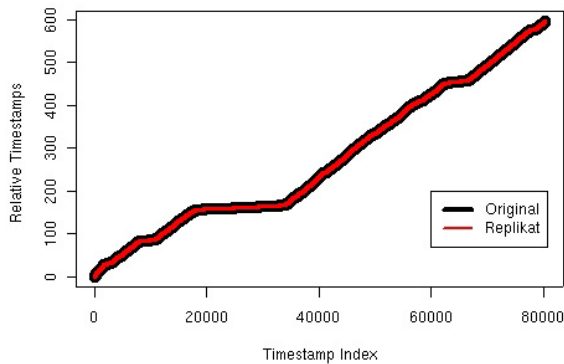


Abbildung 16: Vergleich der absoluten Timestamps von Original und Replikat

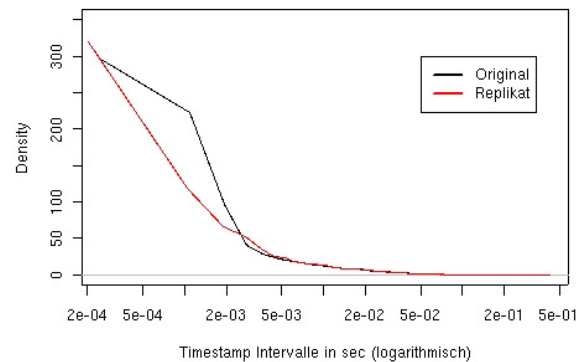


Abbildung 17: Häufigkeit der Timestamp Intervalle von Original und Replikat

Einen genauen Vergleich von Original und Replikat erhält man, wenn man die einzelnen Abstände gegeneinander ins Koordinatensystem einträgt. Die nächsten Abbildungen zeigen folgende Idee: auf der x-Achse werden die Originalabstände angetragen, und auf der y-Achse die Replayabstände (Abb.18). Wenn sie beide genau gleich wären, dann müsste der daraus resultierende Graph eine Winkelhalbierende des 1. Quadranten sein. Liegt der Wert jedoch unter der Winkelhalbierenden, war der Abstand im Replikat kleiner als im Original. Andersrum, liegt er über der Winkelhalbierenden, war der Abstand im Replikat größer. Man sieht, dass es keine reine Winkelhalbierende ist, allerdings sind die Werte zu fein.

Deshalb wurden in Abbildung 19 beide Raumachsen mit einer logarithmischen Skala versehen. Jetzt kann man deutlich die Abweichungen zur Halbierenden erkennen. Außerdem sind die Sprünge sichtbar, die während dem Senden passieren. Einmal sind die Abstandswerte im Bereich von $1\mu s$, das sind also für *tcpdump* zu schnell gesendete Pakete. Dann der Bereich von ca. $10\mu s - 100\mu s$, wo die Maximalwerte der Replayprogramme liegen, und zuletzt wieder ab ca. $1000\mu s (= 1ms)$, was die Timergenauigkeit ist.

In dem Graphen sind aber auch deutlich senkrechte Lücken zu erkennen, was darauf schließen lässt, dass die Quelldatei schon mit Fehlern behaftet ist. Benutzt man anstatt *tcpdump* extra zum Sniffen gebaute Hardware (z.B. eine DAG-Karte²²), dann verschwinden diese senkrechten Lücken. Also muss man feststellen, dass die Zeitbestimmung über Standard Hardware, wie sie bei *tcpdump* eingesetzt wird, zu ungenau ist und zu viele Fehler mit in die Messung einbringt. Folgende Abbildung 20 zeigt den selben Versuch, nur diesmal mit einem Trace, welches von einer DAG-Karte erzeugt wurde.

²²DAG-Karte: siehe oberes Kapitel 3.2.2

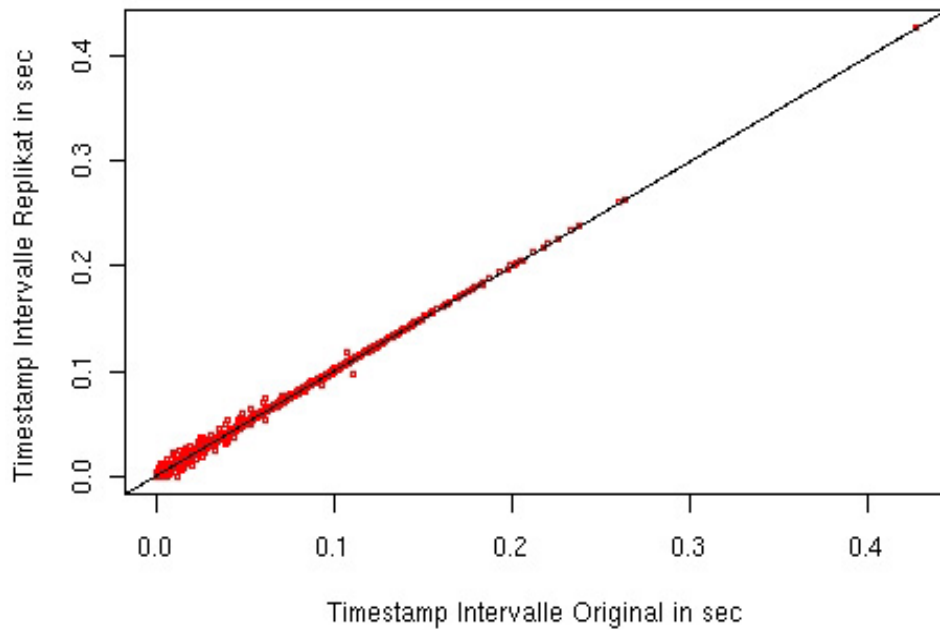


Abbildung 18: Intervall-Intervall Graph in großem Maßstab

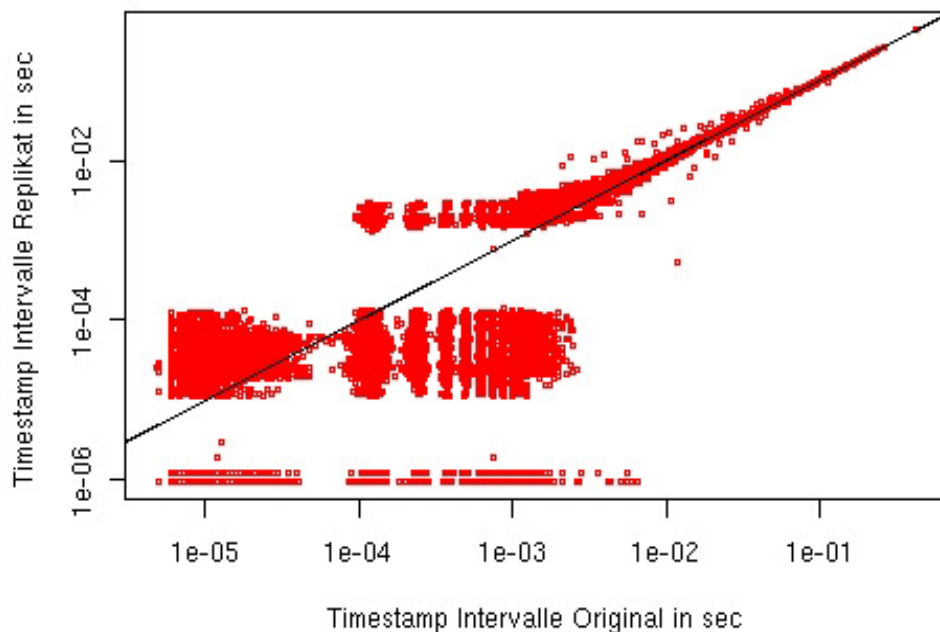


Abbildung 19: Intervall-Intervall Graph mit logarithmischen Achsen

Zusammenfassend kann man sagen, dass die Werte, die zum Rechnen zur Verfügung stehen, zu ungenau und mit zu vielen Fehlern behaftet sind, um konkrete Aussagen zu machen. Man kann die Eigenheiten der Replayprogramme erkennen, aber über deren Herkunft mit diesen Angaben nur spekulieren.

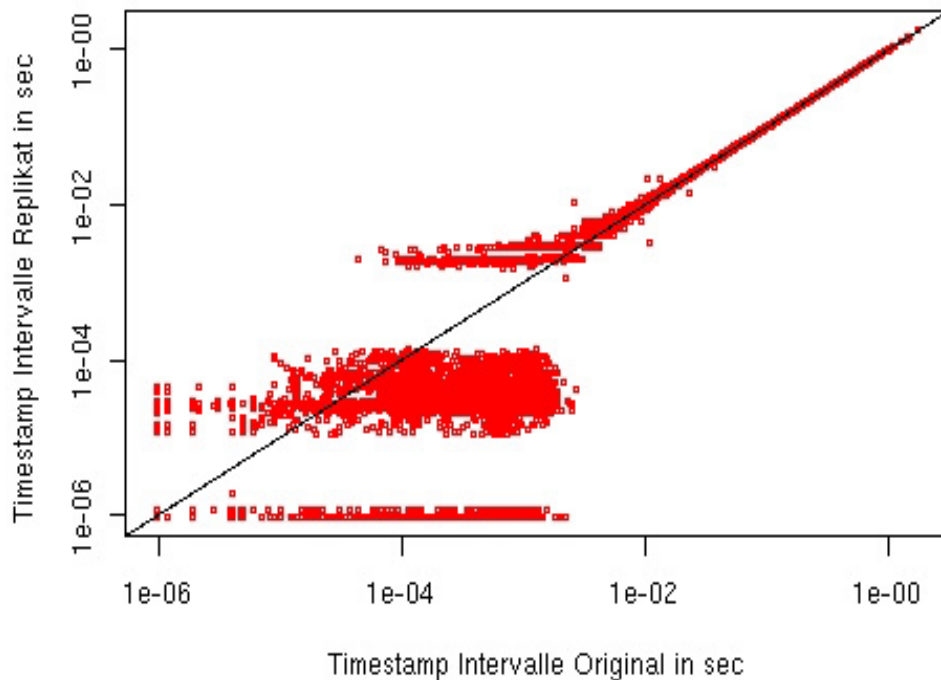


Abbildung 20: Intervall-Intervall Graph mit Quelle einer DAG-Karte

4.6 Weitere Einflüsse

Alle bisherigen Tests wurden in der gleichen Umgebung ausgeführt. Es gibt noch eine Vielzahl von Faktoren, die bis jetzt noch nicht berücksichtigt und variiert worden sind. Zum Beispiel können das verwendete Betriebssystem, der Hersteller der Netzwerkkarte (bzw. deren Treiber), die Auslastung der CPU, usw. eine große Rolle spielen. Zwei dieser Möglichkeiten wurden noch untersucht, und die Abweichungen zu den vorherigen Ergebnissen werden jetzt kurz erläutert.

4.6.1 CPU Auslastung

Zuerst wurde untersucht, wie sich die Auslastung des Rechners bzw. der CPU auf die Testergebnisse auswirkt. Die Tests wurden in zwei Szenarien durchgeführt. Zum Ersten lief nur ein Programm im Hintergrund, was eine hohe CPU Auslastung hervor ruft. Allerdings nur im Userspace, es waren also keine Systemaufrufe. Als Zweites wurden dazu noch 5 Prozesse gestartet, die einen Speicher Stresstest durchführten, und ständig 128MB vom Speicher lesen bzw. schreiben.

Benutzt wurde dafür das Programm `generate_load()`, was für solche Tests mit im *NetVCR* Ziparchiv enthalten ist.

Während *tcp replay 1.0.1* gleich bei hoher CPU Auslastung sehr stark an Geschwindigkeit einbüßt (mehr als 50%), bleiben *tcp replay 2.2.2* und *NetVCR* noch vergleichsweise kon-

stant, wenn man die Ergebnisse mit denen aus Kapitel 4.3 vergleicht. Wird allerdings der Speicher Stresstest dazu geschaltet, sinkt auch die Bandbreite von *tcpreplay 2.2.2* fast auf zwei Drittel zusammen. *NetVCR* hingegen behält die vorherigen Werte bei. Die Erklärung hierfür ist die Echtzeit Implementierung von *NetVCR*, die bereits in Kapitel 3.2.1 erläutert wird. Das Programm läuft mit höherer Priorität, deswegen bekommt `generate_load()` keine Rechenzeit, und somit nicht die Möglichkeit, Ressourcen zu verbrauchen. Tabelle 4 zeigt noch einmal die Ergebnisse aus Kapitel 4.3 ohne Auslastung, und die der zwei neuen Tests.

	Paketgröße in Byte						
	54	64	256	512	1024	1496	1514
ohne Hintergrundlast							
tcpreplay 1.0.1	28,98	33,08	140,95	221,68	312,74	371,24	373,07
tcpreplay 2.2.2	25,99	30,33	104,50	215,89	298,09	334,02	355,26
NetVCR	71,33	82,34	284,57	389,29	443,79	481,62	483,54
mit CPU Auslastung							
tcpreplay 1.0.1	13,94	16,30	70,21	110,78	155,48	186,89	187,33
tcpreplay 2.2.2	12,81	13,58	52,77	123,42	308,25	352,39	355,75
NetVCR	50,36	71,09	284,44	370,76	428,02	486,69	488,08
mit CPU und Speicher Auslastung							
tcpreplay 1.0.1	16,80	11,61	48,65	80,10	113,02	131,45	132,76
tcpreplay 2.2.2	11,58	10,95	37,28	93,19	205,34	247,79	262,98
NetVCR	58,38	65,21	212,11	324,08	423,30	478,37	480,50

Tabelle 4: Maximalbandbreiten in mbps unter Last

Die Paketverluste sind mit Hintergrundlast genauso wenig vorhanden wie ohne. Auch die Verteilung der Timestamp Intervalle sieht vergleichbar aus, allerdings sind noch deutlichere Sprünge zu erkennen.

Es gibt natürlich noch andere Arten von Hintergrundlast, die man testen kann. Dieser Abschnitt soll deswegen auch nur zeigen, daß es durchaus einen Einfluß auf die Qualität der Replayprogramme hat, ob sie allein auf dem System laufen, oder nicht. Alle verschiedenen Möglichkeiten zu testen, hätte hier zu weit geführt.

4.6.2 Betriebssystem

Es wurden keine anderen Betriebssysteme wie Windows oder Mac OS untersucht. Dagegen wurde der ältere 2.4 Linux Kernel getestet, der sich zu dem 2.6 schon um einiges unterscheidet [Die03]. Wie in der Einleitung zum Tests-Kapitel bereits erwähnt, hat der bis jetzt als Zielrechner eingesetzte Computer den alten Kernel installiert. Deswegen wurde lediglich die Funktion des Sende- und Zielrechners vertauscht, und die Tests noch einmal ausgeführt.

Der geringe, bzw. nicht vorhandene Paketverlust der Replayprogramme ändert sich nicht. Genauso tritt die Veränderung bei den Paketabständen wieder auf. Allerdings ändert sich etwas in der Maximalgeschwindigkeit der Programme, denn sie wird größer. Bei 1514 Byte Paketen erreichen alle Programme die 500Mbps Grenze, und werden nicht schneller. Eine mögliche Erklärung hierfür wäre, dass der PCI Bus nicht schnell genug ist, und deswegen die Daten nicht schneller an die Netzwerkkarte gelangen können. Andere externe Messungen, z.B. mit *iperf*, ergaben ungefähr das selbe obere Limit. Also ist hier die Grenze in der Hardware erreicht.

Verglichen mit den Ergebnissen auf dem 2.6 Kernel ist also dort das Bottleneck das Betriebssystem, bzw. das Scheduling. Denn die höchste Bandbreite von *tcpreplay* 2.2.2 betrug dort nicht mehr als 355,26Mbps.

Wenn man allerdings den 2.4 Kernel nicht patcht (siehe Kapitel 3.2.1), dann kann *NetVCR* seine Echtzeitprogrammierung nicht nutzen, und ist bei Hintergrundlast genauso langsam wie *tcpreplay*.

5 Zusammenfassung

Abschließend kann man sagen, dass bei einer Timing-Genauigkeit des Kernels von ca. 1ms keine genaueren Ergebnisse bei den Paketabständen zu erzielen sind. D.h. die Replayprogramme verändern den Abstand zwischen den Paketen enorm, weil sie versuchen, den Fehler des zu groben Timers wieder auszugleichen. Zudem fallen äußere Einflüsse, z.B. die Ungenauigkeit der Quelldaten, auch mit ins Gewicht, was man unter anderem in Abbildung (19 und 20) in Kapitel 4.5.2 schön sehen kann.

Erfreulich ist, dass die untersuchten Programme keine Pakete von sich aus verlieren, und eventuelle Verluste nur der Aufzeichnungssoftware zuzuschreiben ist.

Außerdem bleibt festzuhalten, dass *NetVCR* zwar schneller ist, allerdings weniger Benutzerkomfort hat als *tcpreplay*. Der Echtzeit Vorteil von *NetVCR* wird auch nur auf Kosten der anderen Prozesse erreicht und man könnte wahrscheinlich mit ein paar Änderungen im Sourcecode von *tcpreplay* genau die selbe Schedulingstrategie benutzen.

Natürlich gibt es noch andere Möglichkeiten, Pakete ins Netz einzuspielen. Generatoren-Software benötigt keine abgespeicherte Datei und erzeugt sich die Pakete direkt selbst. Allerdings gibt es davon bestimmt so viele Unterschiedliche, wie es Paketsniffer gibt, deswegen werden diese hier nicht gesondert betrachtet.

Auch gibt es spezielle Hardware, welche Pakete verschicken kann. Diese Pakete sind dann meistens genauer und schneller, allerdings kann man für gute Hardware sehr viel Geld bezahlen.

Abbildungsverzeichnis

1	Weg eines Paketes durch den Kernel	4
2	Aufbau eines Ethernetpaketes	7
3	Aufbau eines IP-Paketes	7
4	Aufbau eines TCP-Paketes	8
5	Aufbau eines UDP-Paketes	8
6	Aufbau der Testumgebung	13
7	Vergleich der Bandbreiten alle drei Programme	16
8	Schema für die theoretische Bestimmung von Paketverlusten	18
9	Paketverluste beim Test mit tcpreplay	20
10	Paketverluste beim Test mit NetVCR	21
11	Timestamp Differenzen bei $10\mu s$	22
12	Timestamp Differenzen bei $100\mu s$	22
13	Timestamp Differenzen bei $2000\mu s$	22
14	Timestamp Differenzen bei $10000\mu s$	22
15	Schema zum „Nachschicken“ von tcpreplay	23
16	Vergleich der absoluten Timestamps von Original und Replikat	24
17	Häufigkeit der Timestamp Intervalle von Original und Replikat	24
18	Intervall-Intervall Graph in großem Maßstab	25
19	Intervall-Intervall Graph mit logarithmischen Achsen	25
20	Intervall-Intervall Graph mit Quelle einer DAG-Karte	26

Tabellenverzeichnis

1	Überblick der Programmooptionen	12
2	Maximalbandbreiten in mbps	17
3	kleinstmögliche Timestamp Differenzen	22
4	Maximalbandbreiten in mbps unter Last	27

Literatur

- [HH04] Jennifer C. Hou und Guanghui He: „*Laboratory Assignment 3: Coordinated Congestion Control*“, <http://lion.cs.uiuc.edu/courses/cs397hou/project3.pdf>, Februar 2004
- [RC01] Alessandro Rubini und Jonathan Corbet: „*Linux Gerätetreiber; Kap.14 Netzwerk-Treiber*“, <http://www.oreilly.de/german/freebooks/linux-drive2ger/book1.html>, 2001
- [Sev03] Jonathan Sevy: „*Linux Network Stack Walkthrough*“, <http://edge.mcs.drexel.edu/GICL/people/sevy/network/>, September 2003
- [Sta04] Jonathan Stanton: „*Network II; Kernel Networking*“, <http://www.seas.gwu.edu/jstanton/courses/cs184/lectures/lec9-2004-osnet.pdf>, 2004
- [Wel00] Harald Welte: „*skb: Linux network buffers*“, <http://gnumonks.org/ftp/pub/doc/skb-doc.html>, Oktober 2000
- [Aip01] AIP Research Group: „*Linux Network Implementation*“, http://ants.iis.sinica.edu.tw/summer/Linux_Network_Implementation.ppt, August 2001
- [Win02] Jared Davis Winick: „*sendTo() and recvFrom() path in Linux for UDP*“, <http://www.eecs.umich.edu/jwinick/sendrecv.html>, Oktober 2002
- [New04] Herbert Newald: „*Ethernet 1*“, http://cis.technikum-wien.at/campus/lehre/documents/icss/icss_4/icss_4_syi/icss_4_syi_download/sb/icss_sysint_sb4_v11.pdf, Januar 2004
- [Str03] A. Strey, Universität Ulm: „*E Datenkommunikation*“, <http://www.informatik.uni-ulm.de/ni/Lehre/WS03/TechInf2/2003w-TI2-E2-4.pdf>, 2003
- [Tur04] Aaron Turner: „*Tcpreplay 2.x FAQ*“, <http://tcpreplay.sourceforge.net/FAQ.html>, Mai 2004
- [UBT03] Matt Undy, Matt Bing und Aaron Turner: „*Manual page of tcpreplay*“, <http://tcpreplay.sourceforge.net/tcpreplay.html>, Mai 2003
- [Fen] Wuchang Feng: „*TCPivo Homepage*“, <http://www.cse.ogi.edu/sysl/projects/tcpivo/>
- [FGW03] Wuchang Feng, Ashvin Goel, Jonathan Walpole, Abdelmajid Bezzaz and Wuchi Feng: „*TCPivo: A High Performance Packet Replay Engine*“, http://www.thefengs.com/wuchang/work/tcpivo/TCPivo_SIGCOMM_MoMeTools_2003.pdf, August 2003

- [Net02] NetVCR Entwickler: „*Readme-Datei zu NetVCR*“, NetVCR Ziparchiv, 2002
- [Uni02] Unix Manpages: „*Linux Programmer’s Manual: setscheduler(2)*“, http://unixhelp.ed.ac.uk/CGI/man-cgi?sched_setscheduler+2, Juni 2002
- [Uwi99] Universität Wien: „*General Programming Concepts: Writing and Debugging Programs; Threads Scheduling*“, http://www.unet.univie.ac.at/aix/aixprgpd/genprogc/threads_sched.htm, 1999
- [GAK02] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow und Jonathan Walpole: „*Supporting Time-sensitive Applications on a Commodity OS*“, <http://www.cse.ogi.edu/~jsnow/publications/osdi2002.pdf>, 2002
- [Die03] Oliver Diedrich: „*The Next Generation; Linux 2.6: Fit für die Zukunft*“, <http://www.heise.de/ct/03/24/194/>, Dezember 2003
- [Dag01] DAG Entwickler: „*DAG-Software*“, <http://dag.cs.waikato.ac.nz/dag/dag-soft.html>, 2001
- [Sch04] Fabian Schneider: „*Analyse der Leistung von BPF und libpcap in Gigabit-Ethernet Umgebungen*“, Juni 2004
- [Kos98] Rainer Koster: „*Design of a Real-Time Communication Service for Local Area Networks; Clock Granularity*“, <http://www.wagss.informatik.uni-kl.de/Projekte/Squirrel/da/node5.html>, April 1998
- [Ste94] W. Richard Stevens: „*TCP/IP Illustrated Vol. 1*“, Addison Wesley Verlag, 1994