UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Simulation fehlerbehafteter Verbindungen in virtuellen Netzen

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von

David Müller

Erstgutachter:	Prof. Dr. Christoph Steigner, Institut für Informatik
Zweitgutachter:	Frank Bohdanowicz, Institut für Informatik

Koblenz, im März 2012

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Das Erstellen virtueller Netzwerke hat den Vorteil, dass Routingalgorithmen oder auch verteilte Software sehr einfach, übersichtlich und kostengünstig getestet werden können. Zu den Routingalgorithmen gehört auch das Optimized Link State Routing und der Babel Algorithmus für mobile Ad-Hoc Netze. Folgende Ausarbeitung beschäftigt sich mit den beiden Algorithmen im Zusammenhang mit fehlerbehafteten Verbindungen in solchen Netzwerken. Normalerweise sind die virtuellen Verbindungen, welche in einem virtuellen Netzwerk erstellt werden, nahe an den optimalen Bedingungen. Oftmals sollen die Routingalgorithmen aber nicht unter optimalen Bedingungen getestet werden, um ein möglichst realistisches Verhalten zu simulieren. Aus diesem Grund sollen sie mithilfe des Netzwerktools Traffic Control unter nicht optimalen Bedingungen getestet werden. Traffic Control stellt einige Werkzeuge zur Verfügung, um Verbindungen hinsichtlich ihrer Verzögerung oder des Paketverlustes zu manipulieren. In dieser Arbeit werden einige Szenarien erstellt, in denen dies getestet wird. Zusätzlich zu den Routingalgorithmen werden noch Staukontrollverfahren, die dazu dienen Datenstau in Netzwerkverbindungen zu verhindern, auf ihre Arbeitsweise untersucht. Mit Traffic Control ist es auch möglich eine gestaute Verbindung zu simulieren.

Abstract

The advantage of virtual networks is to test routing-algorithms or distributed software in an easy-to-use, cheap and well-arranged way. Optimized Link State Routing (OLSR) and Babel are two modern routing-algorithms for mobile ad-hoc networks that distinguish faulty links from good link connections in their routing process. Typically virtual networks provide nearly perfect link connections without any network traffic disruptions. The Traffic Control (tc) tool is able to manipulate connections in terms of delay and packet loss. In this bachelor thesis virtual network scenarios are created with VNUML and Qemu and manipulated with tc in order to analyze the behavior of ad-hoc routing-algorithms like Babel and OLSR. The behavior of well-known TCP congestion control algorithms, e.g., TCP Vegas and TCP Reno are analyzed, too.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

In diesem ersten Kapitel wird der Aufbau der gesamten Arbeit beschrieben und es beinhaltet einen Abschnitt zur Motivation für diese Bachelorarbeit.

1.1 Aufbau

Der Aufbau dieser Arbeit besteht aus den folgenden Themengebieten. Nach der Motivation im ersten Kapitel folgt die Erläuterung der Grundlagen, über die Informationen vorhanden sein müssen, bevor der Rest der Ausarbeitung verstanden werden kann.
Zu den Grundlagen gehört die Bedienung von *Virtual User Mode Linux*[1] (VNUML) und *Quick Emulator*[2] (QEMU). Es wird außerdem ein Verständnis für die Funktionen und Werkzeuge von Traffic Control vermittelt. Zusätzlich gibt es eine Erläuterung zu dem Kernelmodul tcp_probe. Im darauf folgenden dritten Kapitel wird ein detailliertes Verständnis für *Optimized Link State Routing Protocols* (OLSR) [16] entwickelt und es wird ein Einblick in das Routingverfahren von Babel [4] gegeben.
Das vierte Kapitel beschäftigt sich mit den Staukontrollverfahren, der TCP Congestion Control. Es werden zwei Verfahren betrachtet. Zum einen TCP Reno [14] und zum anderen TCP Vegas [15]. Daraufhin folgt ein Kapitel, welches sich mit der Implementation eines eigens angefertigten Skripts zur Kontrolle von Traffic Control widmet. Im sechsten und letzten Kapitel werden praktische Szenarien

[1] http://neweb.dit.upm.es/vnumlwiki/index.php/Main_Page
[2] http://wiki.qemu.org/Main_Page

mit VNUML und QEMU erstellt, um die theoretischen Ansätze aus den vorherigen Kapiteln zu testen und zu analysieren.

1.2 Motivation

Virtual Network User Mode Linux[3] (VNUML) ist eine Software zum Aufbau virtueller Linux Rechner-Netzwerke. VNUML emuliert auf Basis von User Mode Linux[4] (UML), einer Virtualisierung des Linux-Kernels, mehrere Linux Systeme, die über Bridging Software zu einem Netzwerk verbunden werden. VNUML Netzwerke sind in erster Linie dazu geeignet, das Zusammenspiel verteilter Software, wie zum Beispiel Netzwerkprotokolle, in einer optimalen und homogenen Umgebung zu untersuchen. Mögliche Probleme oder Störungen der Netzwerkverbindungen werden von VNUML nicht berücksichtigt.

[3] http://neweb.dit.upm.es/vnumlwiki/index.php/Main_Page
[4] http://user-mode-linux.sourceforge.net/

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen, die zum Verständnis dieser Ausarbeitung von Nöten sind, beschrieben. Zu den Grundlagen zählen die Tools VNUML, Traffic Control und QEMU. Darüber hinaus wird die Funktion des Kernelmoduls[1] tcp_probe näher erläutert.

2.1 Virtual Network User Mode Linux

Virtual User Mode Linux[2] (VNUML) ist ein Tool zur Simulation von Netzwerken, welches auf User Mode Linux basiert. Mit VNUML ist es möglich, die von User Mode Linux erzeugten virtuellen Maschinen (VM) zu Netzen zusammenzufassen. Wird von kleineren Einschränkungen bedingt durch die Virtualisierung abgesehen, verhält sich jede VM wie ein realer Rechner ([11], S.5). Das ermöglicht die Konstruktion und auch Simulation von großen und auch kleinen Netzwerktopologien. Ein Vorteil ist, dass alles auf einem Hostsystem simuliert wird und keine echte Hardware zur Verfügung stehen muss. Die virtuellen Maschinen können fast beliebig konfiguriert werden und in einer Simulation verschiedene Rollen einnehmen (z.B. Router, Webserver, Client, etc). Jede dieser virtuellen Maschinen verfügt über ein eigenes Dateisystem und einen echten Systemkernel. Es ist damit auch möglich ohne große Mühe Programme auf den virtualisierten Maschinen zu installieren. VNUML eignet sich somit sehr gut zum Testen von verteilter Software oder Netzwerkprotokollen ([17], S.2 ff). Ein weiterer Vorteil ist, dass nur eine

[1] http://www.kernel.org/
[2] http://neweb.dit.upm.es/vnumlwiki/index.php/Main_Page

virtuelle Maschine erstellt werden muss, von der alle anderen geklont werden können.

2.2 Die Virtualisierungssoftware QEMU

QEMU ist ein Tool, das zum Virtualisieren und Emulieren von Betriebssystemen dient. Die Abkürzung QEMU steht für *Quick EMUlator*. Der größte Unterschied im Vergleich zu VNUML besteht darin, dass QEMU nur virtuelle Maschinen simuliert, diese aber nicht zu einem Netzwerk zusammenfasst. Ein weiterer Punkt indem sich QEMU von VNUML unterscheidet ist, dass QEMU zwei verschiedene Betriebsmodi für die virtuellen Maschinen anbietet. Zusätzlich zu der *User Mode Emulation*, wie sie auch bei VNUML verfügbar ist, bietet dieses Tool eine *Full System Emulation* an [2]. Mit jener Option ist es möglich einen kompletten Linux Kern zu emulieren. Das ist auch der Grund, warum QEMU zusätzlich zu VNUML in dieser Arbeit eingesetzt wird (siehe Kap. 6.3).

2.3 Traffic Control

Dieser Abschnitt beschäftigt sich mit dem Tool 'Traffic Control' (*TC*). TC ist Teil des *iproute2* Paketes, welches zur Kontrolle des TCP/IP Netzwerkverkehrs in Linux dient. Dieses ist Bestandteil des Linux Kernels und bietet Mechanismen, die entscheiden wie IP-Pakete empfangen und gesendet werden. Um zu verstehen

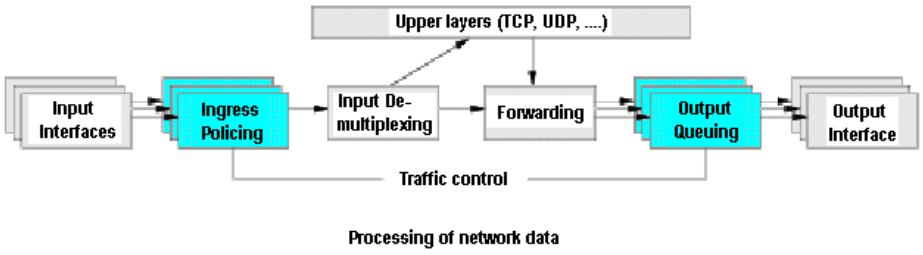

Abbildung 2.1: Aufbau der Infrastruktur [1]

wie TC funktioniert, muss zunächst ein Blick auf die Infrastruktur der Hardware geworfen werden und darauf, wie einkommende und ausgehende IP-Pakete behandelt werden. Wie Abbildung 2.1 zeigt, werden einkommende Pakete aus

dem Netzwerk vom *Input Interface* angenommen. Im Schritt der *Ingress Policing* können dann unerwünschte Pakete, wie beispielsweise zu schnell ankommende Pakete, aussortiert werden.
All das geschieht im Mechanismus einer Warteschlange, der sogenannten *ingress Qdisc* (siehe Kap. 2.3.1). Von dieser Warteschlange aus kommen die IP-Pakete nun zum *Input Demultiplexing* und werden entweder direkt an die Output Warteschlange weitergeleitet oder, falls das Paket auf einer anderen Schicht weiterverarbeitet werden soll, an höhere Protokollschichten weitergeleitet. Meistens werden die IP-Pakete nur dann direkt weitergeleitet, wenn die Linux Maschine als Router oder als Bridge fungiert. IP-Pakete sollten nur dann an höhere Schichten geschickt werden, wenn es sich um ein Hostsystem handelt.
Beim Schritt des *Forwarding* wird schließlich die Entscheidung getroffen, an welches *Output Interface* das Paket weitergeleitet wird. Ist diese Entscheidung gefällt, werden die Pakete beim *Output Queuing* in die Warteschlange des entsprechenden *Output Interfaces* gesteckt (*egress Qdisc*) und diese entscheidet letztlich, welches IP-Paket auch tatsächlich an das Interface gesendet wird, um von dort ins Netzwerk entlassen zu werden. Es existiert je eine Warteschlange pro *Input* und pro *Output Interface* ([10], S.2 ff).
Mit TC ist es nun möglich, diese Warteschlangen zu verändern. So können zum Beispiel mehrere Qdiscs auf ein Interface konfiguriert werden. Bei der Konfiguration einer Qdisc muss beachtet werden, dass es für den Linux Kernel[3] nur eine einzige Qdisc geben darf. Diese wird als **root Qdisc** bezeichnet.

2.3.1 Queuing Disciplines

'Queuing Disciplines' (*Qdiscs*) sind Warteschlangen mit Algorithmen, die kontrollieren, wann IP-Pakete gesendet werden sollen. Als Beispiel für die einfachste Form einer *Qdisc*, wäre die FIFO (First In-First Out) Queue zu nennen.
Wie in Kapitel 2.3 erwähnt, gibt es pro Interface eine Qdisc. Es wird in folgende Formen von Qdiscs unterschieden:

classful Qdisc: Bei dieser Art der Qdisc können sogenannte Klassen (siehe Kap. 2.3.2) erstellt werden, welche dann wieder Qdiscs zugeordnet werden können. So können beispielsweise Hierarchieschemen aufgebaut werden. Weiterhin können hier Filter (siehe Kap. 2.3.3) aufgesetzt werden.

[3] http://www.kernel.org/mirrors/

classless Qdisc: Diese Sorte Qdisc kann im Gegensatz zu einer classful Qdisc keine Klassen enthalten. Es gibt keine Möglichkeit diese zu erstellen. IP-Pakete werden nur an die Interfaces geschickt.

Bei der *ingress Qdisc* einer Linux Maschine handelt es sich um eine *classless Qdisc*. Hier kann nur ein Filter angewendet werden, um den einkommenden Verkehr zu regeln. Bei der *egress Qdisc* handelt es sich um eine *classful Qdisc*. Die meisten Manipulationen, die mit TC durchgeführt werden, finden an dieser Qdisc statt. Der volle Umfang des Tools kann also nur an der *egress Qdisc* ausgeschöpft werden, da hier Klassen erzeugt werden können. Ab diesem Punkt der Ausarbeitung wird, soweit nicht anders erwähnt, mit dem Begriff der **root Qdisc** die *classful Qdisc* am *Ouput Interface* beschrieben [7]. Abschließend werden hier noch die zwölf verschiedenen Qdiscs aufgelistet, welche aktuell vom Linux-Kernel unterstützt werden [1]:

1. Class Based Queue (CBQ)
2. Token Bucket Flow (TBF)
3. Clark-Shenker-Zhang (CSZ)
4. First In First Out (FIFO)
5. Packets First In First Out (PFIFO)
6. Priority Traffic Equalizer (TEQL)
7. Stochastic Fair Queuing (SFQ)
8. Asynchronous Transfer Mode (ATM)
9. Random Early Detection (RED)
10. Generalized RED (GRED)
11. Hierarchical Token Bucket (HTB)
12. Hierarchical Fair Service Curve (HFSC)

2.3.2 Class

Eine *Class* kann als eine Art Aufsatz für eine *Qdisc* gesehen werden, auf welchen eine weitere beliebige *Qdisc* aufgesetzt werden kann und mit deren Hilfe Pakete klassifiziert werden können. Typischerweise wird allen Qdiscs mit Klassen das simple FIFO Queuing Verhalten zugewiesen. Dies kann allerdings mit TC konfiguriert werden. So kann mit Hilfe von *Classes* ein Hierarchieschema von QDiscs mit Klassen erstellt werden, das beliebig tief erweitert werden kann (siehe Abb. 2.2).

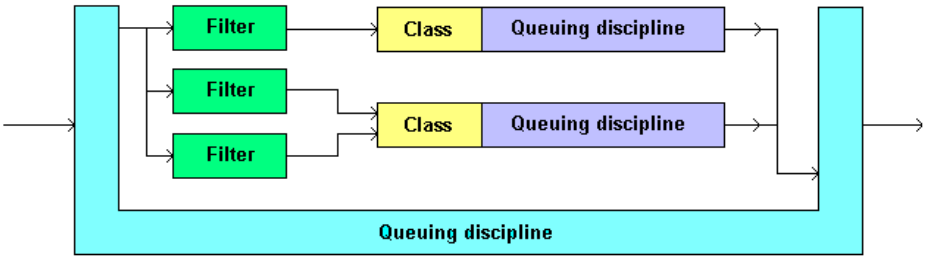

Abbildung 2.2: Beispiel für ein Hierarchieschema [1]

2.3.3 Filter

Mit Filtern können die IP-Pakete anhand von *Classifier* und eventuell dem *Policier* an die unterschiedlichen Qdiscs mit Klassen geleitet werden. Mehrere Filter können auf dieselbe Klasse verweisen. Alle Pakete, die nicht gefiltert wurden, werden vom *default* Filter an eine Qdisc geschickt. Der *Classifier* sorgt dafür, dass Pakete anhand von verschiedenen Parameter unterschieden und somit an verschieden Qdiscs geschickt werden.
Das sogenannte *classifying* kann auf unterschiedliche Arten ablaufen. Beispielsweise können Pakete markiert werden, wenn sie vom Interface angenommen werden. Mittels dieser Markierung kann der Filter entscheiden, in welcher Qdisc das Paket eingeordnet werden muss.
Der *Policier* ist dafür zuständig, Pakete zu behandeln, die über- oder unterhalb einer bestimmten Rate ankommen. Es ist ein Basiselement, um die Nutzung der Bandbreite zu regulieren. Allerdings kann mit Hilfe des *Policing* niemals eine Verzögerung (delay) von Paketen erreicht werden. Es kann nämlich nur eine Aktion

auf Pakete ausgeführt werden, die den gegebenen Kriterien nicht entsprechen (z.B. das Paket verwerfen) ([7], S.66 f).

2.3.4 Netem

Netem ist ein Linux Kernelmodul, welches zu Netzwerkemulationen genutzt wird. Es ist standartmäßig ab der Kernelversion 2.6.7 enthalten. Netem ist als Warteschlangendisziplin (Qdisc) implementiert, die von TC konfiguriert werden muss. Es ermöglicht es das Verzögern, Verwerfen und Duplizieren von TCP-Paketen. Zusätzlich ist es möglich, gezielt Bitfehler in einzelne Pakete einzufügen ([9], S.14).

2.4 Das Kernelmodul TCP_Probe

TCP_Probe ist ein Kernelmodul, dass es ermöglicht TCP spezifische Daten auszulesen. Mit ihn ist es möglich für jedes versendete IP-Paket Werte auslesen und diese in eine Datei schreiben. Es werden der Reihenfolge nach folgende Werte ausgelesen [6]:

1. Time seconds
2. Sender address:port
3. Receiver address:port
4. Bytes in packet
5. Next send sequence Number
6. Unacknowledged sequence Number
7. Congestion window
8. Slow start threshold
9. Send window

Eine Beispielzeile könnte folgendermaßen aussehen:

```
7.681036328 10.0.0.1:49721 10.0.0.4:5001 32
0xbf2b623b 0xbf2b56d3 3 37 14480 2
```

So können später Daten zur Übertragung von TCP Paketen gesammelt und ausgewertet werden. Damit ist das Kapitel der Grundlagen abgeschlossen.

Kapitel 3

Routingverfahren

In diesem Kapitel werden die Routingalgorithmen *Optimized Link State Routing* (OLSR) [16] und *Babel* [4] vorgestellt und analysiert. Diese werden dann schließlich in den Anwendungsbeispielen (siehe Kap. 6) implementiert.

3.1 Grundlagen

In diesem Abschnitt werden die wichtigsten Begriffe für die folgenden Kapitel erklärt:

Knoten: Als Knoten wird ein Teilnehmer im Netzwerk bezeichnet.

Netztopologie: Der Begriff Netztopologie steht im Folgenden für die Struktur eines Netzwerks, bestehend aus mehreren Knoten.

3.2 Optimized Link State Routing

Bei dem *Optimized Link State Routing* (OLSR) handelt es sich um ein proaktives Routingprotokoll, bei dem allen Routern die gesamte Netztopologie bekannt ist. Es wurde für mobile Ad-Hoc Netze entwickelt, lässt sich aber auch auf die Beispielszenarien anwenden. Proaktives Routing bedeutet, dass Informationen über die Netztopologie permanent aktualisiert werden und so Pfade zwischen zwei Knoten im Netz bekannt sind, noch bevor ein solcher Pfad überhaupt für eine Übertragung genutzt wird. Dies hat den Vorteil, dass nicht auf die Bestimmung des Pfades gewartet werden muss, wenn Daten gesendet werden sollen.

Nachteil des proaktiven Routing ist das sogenannte fluten (*flooding*). Beim Fluten sendet ein Knoten im Netz Informationen über seine bekannte Netztopologie an all seine Nachbarknoten. Die Knoten, die diese Nachrichten erhalten und noch nicht über die Topologie informiert worden sind, senden die Informationsnachrichten ebenfalls an alle Nachbarknoten, außer an den Knoten, von dem die ursprüngliche Nachricht ausging. So ist irgendwann jeder Knoten im Netz über die Topologie informiert. Durch dieses Verfahren entsteht jedoch ein immenser Netzwerkverkehr. OLSR versucht diesen Überschuss an teilweise redundantem Verkehr zu minimieren, indem nur ausgewählte Knoten, sogenannte Multipoint Relays (MPRs), die Topologieinformationen weiterschicken. Ein MPR ist ein direkter Nachbar eines Knoten. Jeder Knoten besitzt eine Liste von MPRs. Diese Liste wird das *Multipoint Relay Set* des Knotens genannt. Alle direkten Nachbarn, die sich nicht im *Multipoint Relay Set* befinden, empfangen und verarbeiten die Broadcastnachrichten zwar, aber schicken diese nicht weiter. Nur Knoten in den *Multipoint Relay Sets* schicken Topologieinformationen weiter. Durch den Einsatz dieser MPRs wird der Netzwerkverkehr signifikant reduziert, je größer das Netz wird. Bei kleinen Netzen, wie denen in den Beispielszenarien (siehe Kap. 6), kann der Vorteil noch nicht so klar erkannt werden, da fast alle Knoten MPRs sind. Um auf Änderungen an der Netztopologie schnellstmöglich zu reagieren, muss das Zeitintervall für Kontrollnachrichten (siehe Kap. 3.2.2) möglichst klein gewählt werden [16].

3.2.1 Verbindungsqualität

Zur Berechnung von Routen mit dem OLSR Verfahren wird bei einer Verbindung zwischen zwei Knoten (*Link)* auf die Qualität dieser geachtet. Das bedeutet, dass eine Route über zwei hervorragende Links einer Route über einen sehr schlechten Link bevorzugt wird. Dazu wurde in OLSR ein Verfahren angewendet, um gute von schlechten Links zu unterscheiden. Es wird der Paketverlust anhand der periodisch verschickten HELLO Nachrichten (siehe Kap. 3.2.2), die jeder Nachbar an 'uns' sendet, gemessen. Gehen beispielsweise 3 von 10 Paketen auf dem Weg verloren, haben wir einen Paketverlust von 30% und auf der anderen Seite eine erfolgreiche Übertragung eines Paketes in 70% der Fälle. Die Wahrscheinlichkeit einer erfolgreichen Übertragung eines Paketes, zwischen zwei direkt benachbarten Knoten, wird *Link Quality* (LQ) genannt und entspricht einem Wert zwischen

0% und 100%. Die *Neighbor Link Quality* (NLQ) beschreibt die entsprechende Link Quality, die ein Nachbarknoten hat. Die NLQ beschreibt also die LQ in die entgegengesetzte Übertragungsrichtung.
Die Wahrscheinlichkeit für eine erfolgreichen Packet Round Trip, das Senden eines Paketes und das Empfangen des Acknowledge Paketes, beträgt damit $LQ * NLQ$. Aus diesem Wert kann berechnet werden, wie viele Übertragungsversuche im Schnitt notwendig sind, um ein Paket erfolgreich zu übertragen. Es sind $\frac{1}{LQ*NLQ}$ Versuche. Dieser Wert wird auch der *Expected Transmission Count* (ETX) genannt. Um den ETX einer Route zu bestimmen, werden alle ETX der einzelnen Links dieser Route addiert [16].

3.2.2 Bestimmung der Topologieinformationen

Das grundlegende Instrument zur Bestimmung der Nachbar- und Linkinformationen zu diesen Nachbarn ist das periodische Verschicken der *HELLO* Nachrichten. Im folgenden Abschnitt wird die Funktionsweise dieser Nachrichten erklärt. Vorab müssen noch die Begriffe des *Link Sets* und *Neighbor Sets* erläutert werden [16].

Link Set: Ähnlich wie das MPR Set besitzt jeder Knoten ein Link Set, in dem *Link Tupel* gespeichert werden. In einem Link Tupel (LLocalIfaceAddr, LNeighborIfaceAddr, LSYMTime, LASYMTime, LTime) werden folgende Informationen gespeichert:

- Interface Adresse des eigenen Knotens (LLocalIfaceAddr).
- Interface Adresse des Nachbarknotens (LNeighborIfaceAddr).
- Zeit wie lange der Link als symmetrisch (beide Seiten können Daten senden und empfangen) gilt (LSYMTime).
- Zeit wie lange der Link als asymmetrisch (nur hörend) gilt (LASYMTime).
- Zeit wie lange der Link noch gültig ist (LTime).

Neighbor Set: Im Neighbor Set werden wiederum *Neighbor Tupel* (NNeighborMainAddr, NStatus, NWillingness) gespeichert. In diesen Tupeln sind folgende Parameter spezifiziert:

- Die Adresse des Nachbarknoten (NNeighborMainAddr).
- Die Information, ob der Knoten symmetrisch oder asymmetrisch ist (NStatus).
- Ein Integerwert zwischen 0 und 7, der ein Indikator für die Bereitschaft zur Übertragung von Daten darstellt (NWillingness).

Alle erzeugten *HELLO* Nachrichten werden per Broadcast verschickt und dürfen niemals von anderen Knoten weitergeleitet werden. Die Nachrichten sind für die Erfüllung der drei folgenden grundlegenden Aufgaben, zur Bestimmung der Topologieinformationen, verantwortlich.

1. link sensing: Die *Link Sets* werden mit Daten gefüllt. Dazu wird, sobald eine *HELLO* Nachricht empfangen wurde, das Link Set nach folgendem Schema aktualisiert.

1. Exisitiert kein *Link Tupel* mit LNeighborIfaceAddr = Quelladresse, dann erstelle ein Tupel mit
   - LNeighborIfaceAddr = Quelladresse
   - LLocalIfaceAddr = Adresse des Interfaces, das die Nachricht empfangen hat.
   - LSYMTime = aktuelle Zeit minus eins.
   - LTime = aktuelle Zeit plus Gültigkeitsdauer.
2. Anpassen der Gültigkeitsdauern (siehe [16]).

2. neighbor detection: Das *Neighbor Set* wird mit Daten gefüllt. Der Eintrag eines Neighbor Tupels basiert auf den Daten der Link Tupel. Änderungen werden hier in Abhängigkeit von den Änderungen im Link Set vergenommen, da es eine Verbindung zwischen den Link und Neighbor Sets gibt. Ein Knoten ist der Nachbar eines Knoten genau dann, wenn es mindestens einen Link zwischen diesen beiden Knoten gibt. Wird also ein Tupel in das Link Set eingefügt, so muss, falls noch keines existiert, auch ein Tupel für das Neighbor Set mit folgenden Parametern erzeugt werden:

- NNeighborMainAddr = LNeighborIfaceAddr(aus dem Link Tupel).

Auf die Bestimmung der anderen Parameter wird in dieser Ausarbeitung nicht weiter eingegangen (siehe dazu [16]). Zuletzt gilt, dass sobald ein Eintrag aus dem Link Set gelöscht wird, der entsprechende Eintrag aus dem Neighbor Set gelöscht werden muss, solange es keinen anderen Link mehr zwischen diesen beiden Knoten gibt.

3. **MPR selection signaling:** Das *MPR Set* wird mit Daten gefüllt. Die genaue Beschreibung der Bestimmung des MPR kann nachgeschlagen werden, da diese den Rahmen sprengen würde ([5], S.390 ff).

3.3 BABEL

Im folgenden Abschnitt werden die Eigenschaften und Methoden des Babel Routing Protokolls erläutert.
Bei dem Babel Routingalgorithmus handelt es sich um einen Distanzvektorroutingalgorithmus, der Schleifenfreiheit zusichert, nachdem er vollständig konvergiert ist. Babel basiert auf dem Bellmann-Ford Algorithmus (siehe Kap. 3.3.1).
Distanzvektor Routing bedeutet, dass die einzelnen Knoten ihre gesamte Routinginformationen mit ihren Nachbarn teilen und so die Routingtabellen aufstellen. Durch dieses ständige Austauschen der Routinginformationen organisieren sich Distanzvektoralgorithmen selbst. Auch Babel ist ein proaktives Protokoll, dass den Pfad von einem Start zu einem Zielknoten schon kennt, bevor eine Übertragung nötig ist.
Es kann sein, dass Routingschleifen auftreten während der Babel Algorithmus konvergiert. Babel ist jedoch dazu in der Lage die Dauer und die Frequenz, in welcher diese auftreten, sehr stark zu limitieren. Wird beispielsweise eine Änderung an der Netztopologie festgestellt, so rekonvergiert Babel sehr schnell zu einer neuen, oft noch nicht optimalen, Konfiguaration, welche jedoch schon die Schleifenfreiheit bewahrt. Dieser Vorgang findet noch ohne jeglichen Paketaustausch statt. Innerhalb der nächsten Minuten konvergiert Babel mit Hilfe von *Sequenced Distance-Vector Routing* (siehe [12]) dann zu einer optimalen Konfiguartion.
Wie bei OLSR, auch wird bei Babel auf die Qualität einer Verbindung zwischen zwei Knoten geachtet. Eine längere Route auf der weniger Pakete verloren gehen, wird einer kürzeren mit hohem Paketverlust vorgezogen. Die Kosten zwi-

schen den zwei Knoten A und B bezeichnen wir nun als $C(A, B)$. Die Metrik einer Route zwischen zwei Punkten ist die Summer aller Kosten der Verbindungen zwischen den Knoten auf dieser Route.
Ziel des Babel Algorithmus ist es, für jeden Knoten X einen Baum mit den niedrigsten Metriken zu allen erreichbaren Knoten zu errechnen. Die Kosten für eine Verbindung zwischen zwei Knoten werden ähnlich wie bei OLSR (siehe Kap. 3.2.1) berechnet. Ein Knoten X broadcastet regelmäßig *Hello* Nachrichten und schickt an jeden Knoten eine *IHU* 'I heard you' Nachricht zurück, von dem er ein Hello empfangen hat. So werden schließlich die Kosten ermittelt [4].

3.3.1 Bellmann-Ford Algorithmus

In diesem Absatz wird nun die Funktionsweise des Bellmann-Ford Algorithmus, auf welchem das Babel Verfahren basiert, beschrieben. Der Bellmann-Ford Algorithmus dient dazu, den günstigsten Weg von einem Startknoten S (der Sender) in einem Graphen (repräsentiert das Netzwerk) zu einem anderen Knoten A (der Empfänger) zu finden. Dazu besitzt jeder Knoten X zwei Datensätze:

- die Distanz $D(S)$ zu einem anderen Knoten S.
- und den *Next Hop* $NH(S)$ Router auf der Route zu S.

Zu Beginn des Algorithmus sind $D(S) = 0$, $D(A) = undefiniert$ und $NH(A) = undefiniert$.
Nun senden alle Knoten X periodisch *Route Update* Nachrichten an alle Nachbarn mit dem Inhalt $D(X)$. Jeder Nachbarknoten Y, der nun dieses Route Update empfängt, überprüft, ob X der Next Hop Router ist. Ist dies der Fall so wird $NH(Y) = X$ und $D(Y) = C(X, Y) + D(X)$ gesetzt. Ist dies allerdings nicht der Fall, so vergleicht Y den Wert $C(X, Y) + D(X)$ mit $D(Y)$. Ist der berechnete Wert kleiner, gibt es also eine neue, billigere Route und $NH(Y)$ wird auf X und $D(Y)$ auf $C(X, Y) + D(X)$ gesetzt [3].

3.3.2 Schleifenfreiheit

Babel führt Routenänderungen allerdings nur dann durch, wenn der Algorithmus sicher ist, dass durch das Setzen der neuen Route keine Routingschleife entstehen kann. Um diese Eigenschaft zu gewährleisten, überprüft Babel vor der Än-

derung nach, ob die Routenänderung bestimmte Bedingungen, sogenannte *Feasibility Conditions* erfüllen. Tut sie dies nicht, so wird die Änderung ignoriert. Die Erklärung für die Bedingung der Schleifenfreiheit soll anhand eines kleinen Beipiels erläutert werden.
Zunächst ist festzustellen, dass nur eine Schleife auftreten kann, wenn ein Router eine neue Route akzeptiert, deren Metrik größer als die Metrik der aktuellen Route ist. Angenommen ein Knoten A besitzt eine Eigenschaft *feasibility distance* $FD(A)$, welche die kleinste, jemals bekannte, Metrik zu einem beliebigen Nachbarn speichert. Bekommt dieser Knoten A nun eine Routenänderung von einem Nachbar B geschickt, so wird überprüft, ob die Metrik $D(B)$ echt kleiner als die *feasibility distance* von A ist ($D(B) < FD(A)$). Anhand dieser Bedingung ist nun eine Schleifenfreiheit garantiert, da nur Routen mit kleiner Metriken akzeptiert werden. Leisten alle Router dieser Eigenschaft folge, so sind die Metriken keines Routers ansteigend, was impliziert, dass keine Schleifen entstehen können [4, 3].

3.3.3 Verhinderung von Starvation

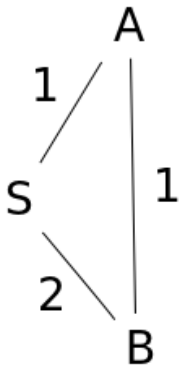

Abbildung 3.1: Starvation in Babel

Starvation ist im Routing ein Begriff dafür, dass ein Router Pakete losschicken möchte, es ihm aber nicht erlaubt ist. Dies ist bei Babel der Fall, wenn die Route mit der besten Metrik wegfällt. Gegeben sei folgende Situation (siehe Abb. 3.1).
Router A und Router B haben jeweils die direkte Route zu S gewählt. Daraus folgt:

$D(A) = 1, FD(A) = 1, D(B) = 2, FD(B) = 2$
Angenommen die Verbindung zwischen A und S fällt nun aus. Dann erfüllt die Route von A über B nicht die Bedingungen aus dem Kapitel der Schleifenfreiheit (siehe Kap. 3.3.2). A kann also keine Pakete mehr versenden. Dieser Sachverhalt nennt sich *Starvation*. Babel vermeidet dies, indem es Sequenznummer für die Routen vergibt. Ein Route besitzt nun also nicht mehr nur eine Metrik, sondern auch eine, sich nicht verringernde, Sequenznummer. Diese Nummer wird von keinem Router verändert, außer von der Quelle, wo sie entstanden ist. Das Tupel (s, m) wobei s eine Sequenznummer und m eine Metrik ist.
Das Update einer Route ist also möglich, wenn die Route eine größere Sequenznummer und eine kleinere Metrik als die alte Route besitzt. A mit $FD(A) = (s, m)$ akzeptiert ein Update (s', m') wenn folgendes gilt :
$s' > s \vee (s = s' \wedge m' < m)$.
Wenn S nun beispielsweise die Sequenznummer 137 hätte, würde das bedeuten:
$FD(A) = (137, 1)$
$D(B) = (137, 2)$
$FD(B) = (137, 2)$
Nach dem Wegfall der Route von A nach S würde S seine Sequenznummer auf 138 erhöhen und diese allen Nachbarn mitteilen. In diesem Fall gibt es nur noch den Nachbarn B. Es kommt zu folgenden Werten:
$FD(A) = (137, 1)$
$D(B) = (138, 2)$
$FD(B) = (138, 2)$
Nun ist ein Update der Routen wieder möglich. Auf die Protokolloperation von Babel wird nun nicht weiter eingegangen, da dies den Rahmen der Arbeit sprengen würde. Diese Information befindet sich unter [4].

Kapitel 4

TCP Congestion Control

In diesem Kapitel werden zwei TCP Verfahren zur Staukontrolle (*Congestion Control*) bei überlasteten Verbindungen beschrieben. Sie sollen unterbinden, dass zu viele Daten auf einmal ins Netzwerk injeziert werden und verhindern so eine Überlastung der Router und Verbindungen. Denn je mehr eine Verbindung in einem Netzwerk mit Paketen belastet wird, desto größer ist der Paketverlust auf dieser Verbindung beziehungsweise desto mehr Pakete werden von den Routern verworfen, da diese nur eine bestimmte Anzahl an Paketen empfangen können. Dieser Paketverlust kann mit TC simuliert werden (siehe Kap.2.3). Im folgenden Kapitel werden zwei Algorithmen zur Staukontrolle betrachtet.

- TCP Reno (reaktiv)
- TCP Vegas (proaktiv)

Die Idee bei der Staukontrolle ist, dass die Absender eines Datenpakets die vom Empfänger des Pakets zurückgeschickten *Acknowledge-Pakete* (ACK) evaluieren und erst dann ein neues Paket losschicken, wenn die Ankunft eines anderen Pakets bestätigt wurde. Eine Problemstellung bei der Staukontrolle ist allerdings die Abschätzung der Netzwerkkapazität zu Beginn der Übertragung. Wird diese überschätzt, so führt dies zu Paketverlust, da zu viele Pakete ins Netzwerk geschickt werden und ein Router nur ein bestimmte Anzahl an Paketen verarbeiten kann. Eine Unterschätzung führt zur einer Verschwendung der Bandbreite des Netzwerks, da dieses nicht optimal ausgelastet wird [14]. Es gibt also einen Punkt, an dem das Netzwerk optimal ausgelastet ist (siehe Abb. 4.1).

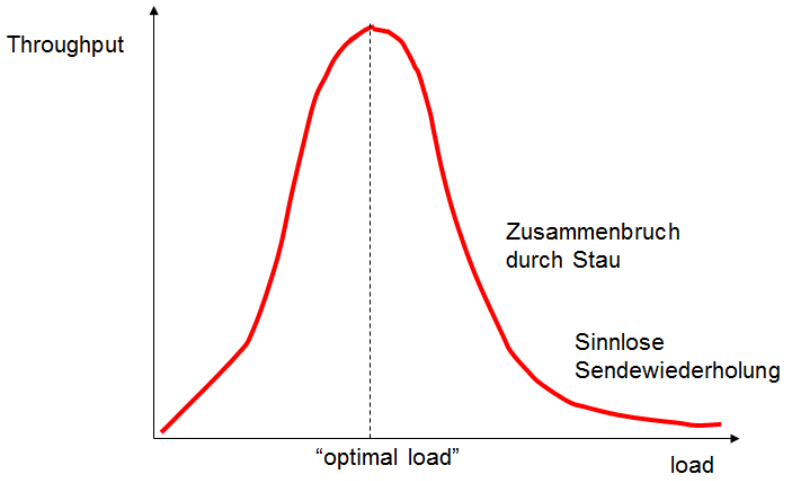

Abbildung 4.1: Optimal Load, [13] Folie 194

4.1 TCP Reno

TCP Reno ist ein *reaktiver* Staukontrollmechanismus, der dazu dient Datenstau zu verhindern. Das bedeutet, dass auf bestimmte Ereignisse erst dann reagiert wird, wenn diese auch eingetreten sind. Bei TCP Reno wird die *current window size* (cwnd) in typischen Situation zyklisch geändert ([13], Folie 265 ff). Zu Beginn wird die WS kontinuierlich vergrößert bis Paketverlust auftritt. Dies wird in zwei Phasen unterteilt, deren Funktionsweisen in den folgenden Kapiteln erläutert werden.

1. Slow Start Phase
2. Congestion Avoidance Phase

4.1.1 Congestion Window

Zum besseren Verständnis der nachfolgenden Kapitel, wird im Vorfeld der Begriff *Congestion Window* (cwnd) erläutert. Das cwnd beschreibt die Anzahl der Segmente, die ein Sender in ein Netzwerk schicken darf. Es ist eine Art der Flusskontrolle durch den Sender. Die Größe dieses Fensters wird in Byte angegeben. Im Folgenden wird die Größe aber der Einfachheit halber in natürlichen Zahlen beginnend mit 1 beschrieben. Beträgt dieser Wert beispielsweise 8, so dürfen 8

Pakete auf einmal verschickt werden. Diese Fenstergröße ist von Bedeutung, da sie im Laufe des Reno-Algorithmus immerzu geändert wird.

4.1.2 Slow Start

Der folgende Abschnitt beschreibt die erste Phase des Algorithmus von TCP Reno, die *Slow Start Phase*. Sie läuft nach folgendem Schema ab:
Es wird eine Rate, mit welcher neue Pakete ins Netzwerk entlassen werden sollen ermittelt. Diese Ermittlung basiert auf dem Wert, der die Rate der ankommenden ACK Paket beschreibt, die der Empfänger des ursprünglichen Pakets zurückschickt. Beim Slow Start wird für jedes empfangene ACK Paket die cwnd um eins erhöht. Dieser Vorgang führt zu einem exponentiellem Wachstum der cwnd. Dies wird in der folgenden Abbildung grafisch dargestellt (siehe Abb. 4.2). Zu Beginn wird ein Paket ($cwnd = 1$) übertragen und auf den Empfang des zugehörigen ACK Pakets gewartet. Kommt dieses an, so wird das cwnd um eins erhöht ($cwnd = 2$). Nun werden zwei Pakete verschickt und da jetzt auch zwei ACK Pakete empfangen werden, wird das cwnd um zwei erhöht ($cwnd = 4$). Dies führt innerhalb einer *Round Trip Time* (siehe Kap. 4.2) zu einer Verdopplung des cwnd ([14], S.285 ff).

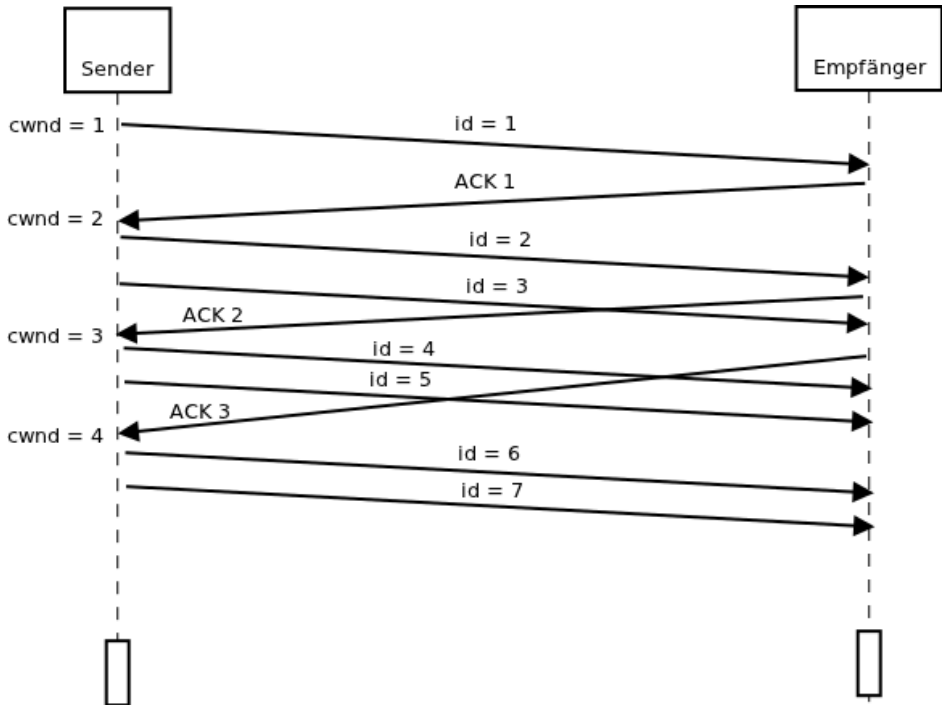

Abbildung 4.2: Slow Start

Ab einem bestimmten Punkt ist nun aber die maximale Kapazität der Verbindung oder des Empfängers erreicht und es kommt zu Paketverlust oder es wird der vorher festgelegte Grenzwert *Slow Start Threshold* (siehe Kap. 4.1.4) erreicht. Tritt Paketverlust durch duplizierte ACK Pakete (siehe Abb. 4.3) ein, weiß der Sender, dass die cwnd zu groß geworden ist und verringert werden muss (siehe Kap. 4.1.4). Eine Methode, um mit diesem Paketverlust umzugehen, ist die der *Congestion Avoidance*. Wird der Grenzwert *ssthresh* überschritten, wechselt TCP Reno in die *Congestion Avoidance* Phase.

4.1.3 Congestion Avoidance

Es wird nun der Algorithmus von Congestion Avoidance beschrieben, welcher in der zweiten Phase von TCP Reno verwendet wird. Die Annahme, die bei diesem Algorithmus getroffen wird, ist, dass die Wahrscheinlichkeit des aufgetretenen Paketverlustes durch einen Schaden an der Verbindung verursacht wurde minimal klein ist ($\ll 1$). Daraus folgt ein Hinweis darauf, dass auf der Verbindung

zwischen den beiden Teilnehmern ein Stau entstanden sein muss. Ein Paketverlust kann auf zwei verschiedene Weisen erkannt werden:

1. Ein Timeout, welcher durch das Ausbleiben eines ACK Pakets auftritt.
2. Der Empfang von duplizierten ACK Paketen.

Duplizierte ACK Pakete treten bei TCP dann auf, wenn der Empfänger ein Paket empfängt, dessen fortlaufende Nummer nicht der Nummer des letzten empfangenen Pakets plus eins entspricht. Tritt dieser Fall ein, so speichert der Empfänger für jedes weitere empfangene Pakete dessen Inhalt in einem Puffer. Er schickt aber nicht den normalen ACK Inhalt (Nummer des empfangenen Pakets) zurück, sondern die fortlaufende Nummer des letzten korrekt empfangen Pakets (siehe Abb. 4.3). Der Sender reagiert auf diese empfangenen duplizierten ACK Pake-

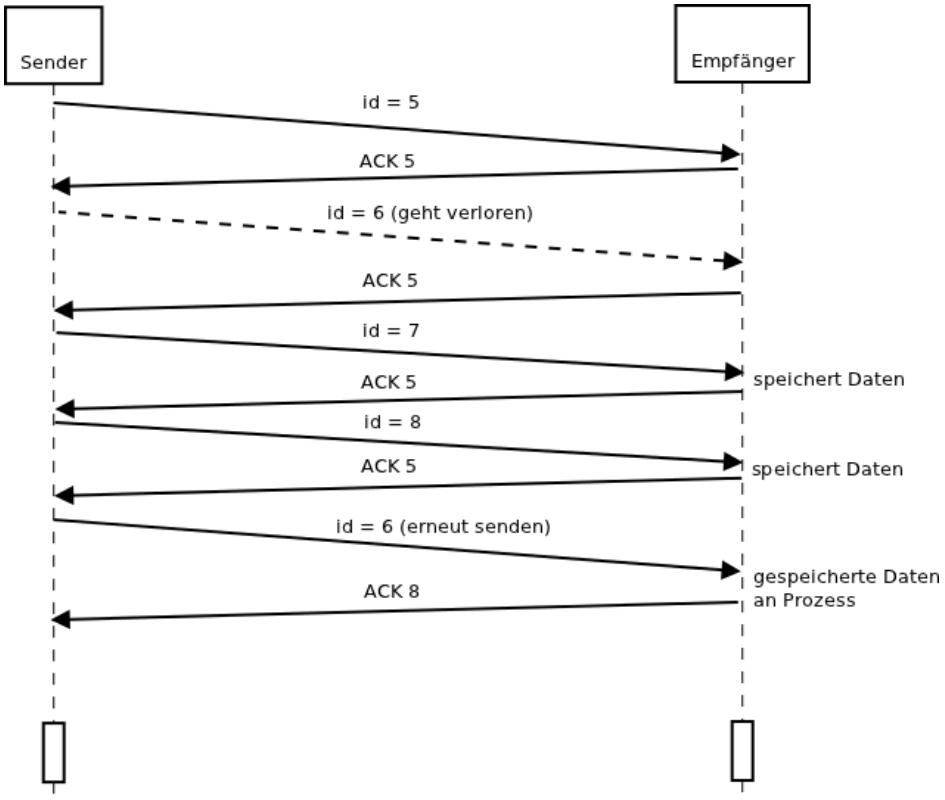

Abbildung 4.3: Duplicate Acknowledge

te, indem er das verlorenen Paket erneut versendet, nachdem der *retransmission*

Timer abgelaufen ist. Wird das fehlende Paket jetzt empfangen, so gibt der Empfänger alle vorher im Puffer gespeicherte Daten an den entsprechenden Prozess weiter. Zusätzlich sendet er das ACK Paket mit dem Inhalt des höchsten zuletzt empfangenen Pakets (im Beispiel Paket 8). Die Reihenfolge ist nun wieder korrekt und es kann normal weitergearbeitet werden.
Der eigentliche Algorithmus von Congestion Avoidance bewirkt, dass das cwnd nicht mehr exponentiell, wie in der Slow Start Phase, sondern nur noch linear ansteigt. Dieser lineare Anstieg vermeidet weiteren Stau auf der Verbindung. Exponentielles Wachstum wurde dadurch erreicht, dass pro empfangenes ACK Paket, das cwnd um eins erhöht wurde. Bei Congestion Avoidance wird das cwnd jetzt pro empfangenem ACK Paket nur noch um $\frac{1}{cwnd}$ erhöht ([14], S.310 ff). Dies hat zu Folge, dass sich das cwnd innerhalb einer RTT (siehe Kap. 4.2) genau um den Faktor eins erhöht (siehe Abb. 4.4).

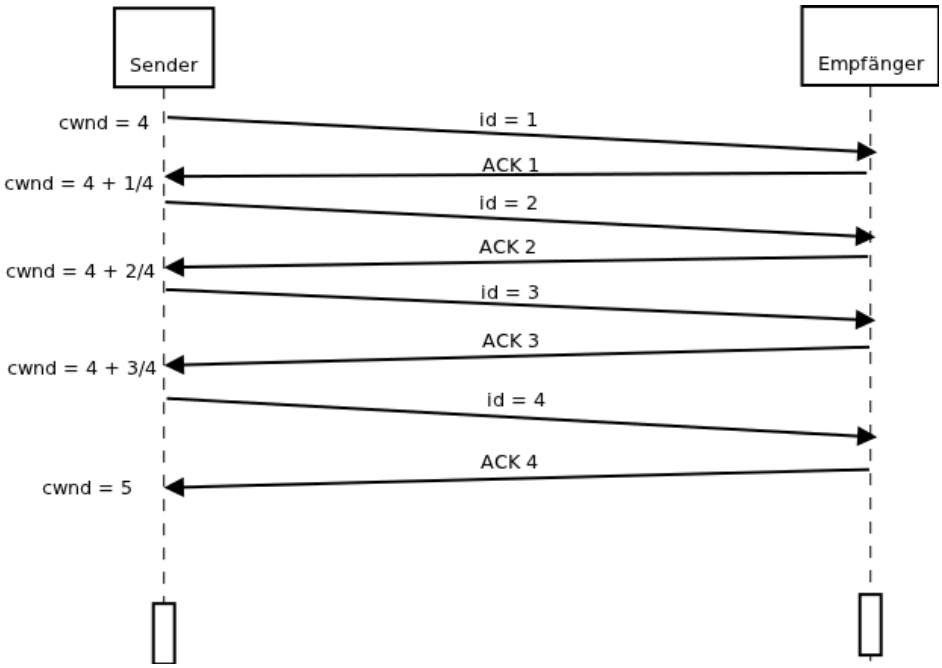

Abbildung 4.4: Congestion Avoidance

4.1.4 Slow Start Threshold

Dieser Abschnitt beschreibt, was man unter dem *Slow Start Threshold* (ssthresh) versteht und wofür er gebraucht wird.

Bei TCP Reno ist es so, dass sich der Algoithmus zu Beginn in der Slow Start Phase befindet. Sobald das cwnd jedoch einen bestimmten Grenzwert erreicht, den ssthresh, wird von der Slow Start in die Congestion Avoidance Phase gewechselt. Dieser Grenzwert ist von Beginn an festgelegt. Zusammenfassend kann die Aktualisierung der cwnd also wie folgt beschrieben [15] werden:

$$cwnd = \begin{cases} cwnd + 1 & \text{if } cwnd < ssthresh \\ cwnd + \frac{1}{cwnd} & \text{if } cwnd \geq ssthresh \end{cases}$$

Kommt es nun in der Slow Start oder der Congestion Avoidance Phase zu Paketverlust, welcher durch das Ablaufen des *retransmission Timers* registriert wurde, so werden die Werte der cwnd und des sstresh wie folgt aktualisiert:

$$sshtreh = \frac{cwnd}{2}, \quad cwnd = 1$$

4.1.5 Fast Retransmit and Fast Recovery

In diesem Abschnitt werden verbessernde Veränderungen am Congestion Avoidance Mechanismus beschrieben. Um diese zu erläutern, sei nochmals wiederholt, dass TCP gezwungen ist ein dupliziertes ACK Paket zu schicken, wenn ein empfangenes Paket ein ID besitzt, welche nicht der korrekten Reihenfolge entspricht. Diese Vorgehensweise (siehe Abb. 4.3) hat den Zweck, den Sender wissen zu lassen, dass ein Paket außerhalb der Reihenfolge empfangen wurde und welches Paket erwartet wird. Allerdings weiß der Empfänger dieses Pakets nicht, ob beim Sendevorgang nur die Reihenfolge der Pakete im Netzwerk durcheinander geraten ist oder ob das Paket tatsächlich verloren gegangen ist. Der sogenannte *Fast Recovery* Algorithmus nimmt nun an, dass nur die Paketreihenfolge durcheinander geraten ist und wartet eine kleine Anzahl weiterer duplizierter ACK Pakete ab. Sollte nach drei weiteren Paketen kein neu produziertes ACK Paket empfangen worden sein, so ist dies ein starker Indikator dafür, dass das Paket verloren gegangen ist und nicht nur die Reihenfolge durcheinander geraten ist [18]. Tritt dieser Fall ein, so wird das verloren gegangene Paket sofort erneut gesendet ohne auf das Ablaufen des *retransmission Timer* zu warten (siehe Abb. 4.3). In dieser Abbildung werden allerdings nur zwei duplizierte ACK Pakete (ACK 6)

empfangen. Als nächstes wird dann Congestion Avoidance ausgeführt und nicht wieder bei Slow Start begonnen. Diese Eigenschaft nennt sich *Fast Recovery* ([14], S.312 ff). Abschließend werden noch die Werte der cwnd und des sstresh aktualisiert:

$$sstresh = \frac{cwnd}{2}, \quad cwnd = sstresh$$

4.2 TCP Vegas

Bei TCP Reno wird die cwnd solange erhört, bis Paketverlust aufgrund von Stau auftritt. In einer solchen Situation wird die cwnd verringert, sodass auch der Datendurchsatz gedrosselt wird. Dieser Verlust an Bandbreite kann nicht verhindert werden, da TCP Reno nur anhand von Paketverlust in der Lage ist, Datenstau zu erkennen. Mit der Erkenntnis ,dass TCP Reno erst Paketverlust erzeugen muss, um Datenstau zu erkennen, entstand die Idee für TCP Vegas.
Im Gegensatz zu TCP Reno handelt es sich bei TCP Vegas um ein *proaktives* Verfahren zu Staukontrolle. Das bedeutet, dass schon bevor bestimmte Ereignisse eintreten Maßnahmen ergriffen werden, um einen eventuellen Datenstau zu verhindern. Durch einen Alogrithmus, der die cwnd so kontrolliert, dass Paketverlust vermieden wird, kann auch der Verlust an Bandbreite vermieden werden. Die Grundidee von TCP Vegas ist die angemessene Kontrolle der *window size* bevor es zu Paketverlust kommt.
Um diese Kontrolle zu gewährleisten, beobachtet TCP Vegas die Paketumlaufzeiten (RTT, *Round Trip Time*). Die RTT gibt an, wie lange ein Paket im Netzwerk benötigt, um vom Start zu dem Zielhost zu gelangen.
Wird die Paketumlaufzeit größer, so weiß TCP Vegas, dass ein möglicher Paketstau im Netzwerk herrscht und verringert daher die cwnd. Analog wird die cwnd bei sinkender Umlaufzeit vergrößert. In einer idealen Situation konvergiert die cwnd so gegen einen bestimmten Wert.
Im Detail berechnet sich dies so:
Zunächst wird der sogenannte *Expected Value* berechnet ([13], Folie 302 ff). Dieser gibt das Verhältnis von der cwnd zu der niedrigsten gemessenen RTT (BaseRTT) an.
$Expected = \frac{cwnd}{BaseRTT}$

Dann wird das Verhältnis zur wirklichen RTT, dem *Actual Value* ermittelt.
$Actual = \frac{cwnd}{RTT}$
Und es wird die Differenz dieser beiden Werte gebildet.
$Diff = (Expected - Actual) * BaseRTT = (\frac{cwnd}{BaseRTT} - \frac{cwnd}{RTT}) * BaseRTT$
Diese Differenz ist größer oder gleich null, da die RTT niemals kleiner, sondern höchstens gleich der BaseRTT werden kann. Das cwnd wird dann nach folgenden Kriterien aktualisiert [15].

$$cwnd = \begin{cases} cwnd + 1 & \text{if Diff} < \alpha \\ cwnd - 1 & \text{if Diff} > \beta \\ cwnd & \text{otherwise} \end{cases}$$

Es sollte noch erwähnt werden, dass α und β Konstanten sind, die vor der Benutzung von TCP Vegas festgelegt werden. Es gilt $0 < \alpha \leq \beta$.

Kapitel 5

Skript für Traffic Control

Um eine einfache und schnelle Benutzung für *Traffic Control* (TC) mit *Virtual User Mode Linux*[1] (VNUML) Szenarien bereitzustellen, wurde im Rahmen dieser Bachelorarbeit ein Skript entwickelt, dass eine Oberfläche zur Einstellung von TC Befehlen liefert. Das Skript wurde in Python geschrieben und in diesem Abschnitt wird die Programmierung und die Benutzung der wichtigsten Funktionen beschrieben. Ein Listing des gesamten Quellcodes befindet sich im Anhang (siehe A.1). Mit dem Package *TkInter* wird von Python ein einfaches Interface zur Gestaltung von GUI Applikationen zur Verfügung gestellt.
Im nächsten Abschnitt wird nun die Funktionalität des Quellcodes beschrieben. Es werden folgende Teilaufgaben bearbeitet.

1. Parse die VNUML Datei und speichere alle verfügbaren Daten.
2. Erstelle eine entsprechende Oberfläche.
3. Führe die vom Benutzer angegebenen Befehle aus.

5.1 Parsen der VNUML Datei

Zur Ausführung von Schritt eins wurde die Klasse 'IF' erstellt (Zeile 36-67). Diese durchsucht mithilfe eines Document Object Models (DOM) Parsers die per Argument im Skriptaufruf übergebene VNUML Datei auf folgende XML Elementen: (Für ein Beispiel der richtigen Benutzung des Skripts siehe Kap.6)

[1] http://neweb.dit.upm.es/vnumlwiki/index.php/Main_Page

```
<vm name="VM1">
    <if id="1" net="a">
        <ipv4>10.0.0.1</ipv4>
    </if>
</vm>
```

Der Parser durchsucht diese 'vm' Elemente, die in einer korrekten und vollständigen VNUML Datei vorhanden sind und speichert zu jedem Interface einer 'vm' die folgenden Informationen:

- Den Namen des Interface mit dem Postfix '-eID' wobei ID für das ID Attribut des 'if' Tags steht. Dies wird so gespeichert, da VNUML die Interfaces auf dem Host später so benennt, wenn diese automatisch erstellt werden. So kann dann direkt auf das Interface zugegriffen werden und das Postfix muss nicht immer mit angegeben werden.
- Die IP Adressen des jeweiligen Interface.

5.2 Erstellen der Oberfläche

Im zweiten Arbeitsschritt wird nun die Oberfläche anhand der gespeicherten Daten erstellt. Dies passiert in der Methode *create* (Zeile 69-122).
Die Oberfläche ist so aufgebaut, dass drei verschieden Frames existieren. Eine Kopfzeile, ein Hauptframe und eine Fußleiste. In der Kopfzeile wird mit verschiedenen Labels eine tabellenartige Struktur erstellt, die der Übersicht dient (siehe Abb. 5.1). Nun werden die gespeicherten Informationen zu den verscheiden In-

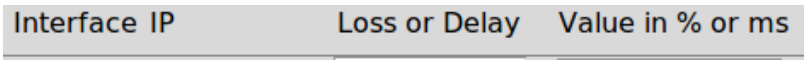

Abbildung 5.1: Kopfzeile

terfaces dynamisch in diese Struktur eingefügt (Zeile 99-121). Die Interfaces werden bezüglich ihrer 'vm' gruppierte und in farbige Blöcke unterteilt. Dies dient der Orientierung. So entsteht das Hauptframe (siehe Abb. 5.2).
Zuletzt wird noch die Fußleiste mit zwei Buttons erstellt (siehe Abb. 5.3).

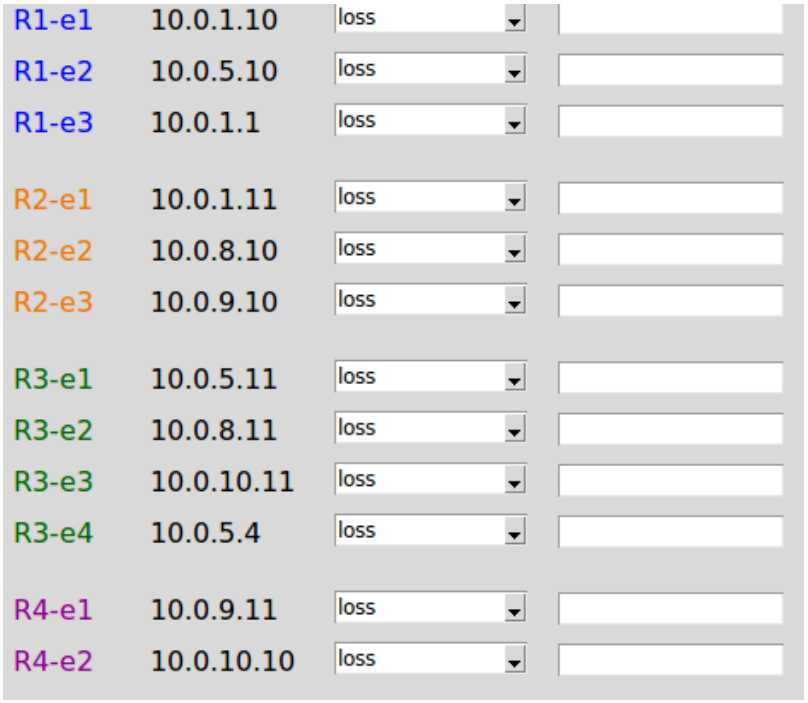

Abbildung 5.2: Hauptframe

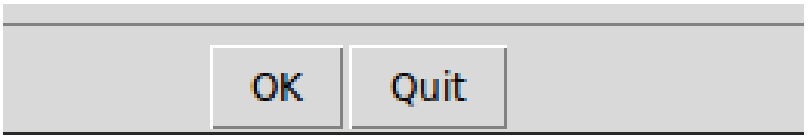

Abbildung 5.3: Fußleiste

5.3 Ausführung

Nun kann der Benutzer die gewünschten Einstellungen an den verschiedenen Interfaces vornehmen. Für die Beeinflussung eines Interface mit TC stehen folgende Möglichkeiten zur Verfügung (siehe Kap. 2.3.4):

- loss
- delay
- corruption
- duplicate
- loss und delay

- loss und corruption
- delay und corruption
- delay und duplicate
- loss, delay und corruption

Mehrere Argumente werden durch ein Komma getrennt. Wurden nun alle gewünschten Werte eingetragen, muss dies mit einem Klick auf 'OK' bestätigt werden. Durch diese Aktion wird im Quellcode die Funktion *okclick()* aufgerufen (Zeile 164-199). Diese Funktion ruft zunächst eine weitere Funktion *cleantc()* auf, welche dafür sorgt, dass alle Einstellungen auf jedem Interface zurückgesetzt werden (Zeile 125-156). Dazu wird die Ausgabe von,

```
tc qdisc show | grep netem
```

die jedes mit netem bearbeitete Interface ausgibt, mit Hilfe von regulären Ausdrücken überprüft. Es wird aus jeder Zeile der Name des zugehörigen Interface und die entsprechende Veränderung ausgelesen.
Schlussendlich wird dann ein interaktiver Befehl erstellt, der diese Einstellungen löscht (Zeile 156). Nun überprüft die Funktion 'okclick', in welchen Textfeldern eine Eingabe vorgenommen wurde, da nur diese behandelt werden müssen. Es wird ein Befehl bezüglich der Einstellungen des Benutzers erstellt. Wurde beispielsweise ein 'loss' Wert registriert, so wird Zeile 175 aufgerufen und für dieses Interface mit TC und netem der zugehörige Wert eingetragen.

Kapitel 6

Traffic Control mit VNUML

In diesem Kapitel soll die Funktionsweise von *Traffic Control* (TC) in *Virtual User Mode Linux*[1] (VNUML) und *Quick Emulator*[2] (QEMU) anhand von drei konstruierten Beispielszenarien analysiert und dokumentiert werden.

6.1 Szenario 1

Das erste Szenario dient dazu, die Funktion des Paketverlusts von TC so zu nutzen, dass der IP-Verkehr vom Optimized Link State Routing (OLSR) (siehe Kap. 3.2) auf entsprechende Routen umgeleitet wird. Dazu wird ein Netz aus vier Routern und zwei 'Hosts' erstellt (siehe Abb. 6.2). Mit dem Befehl

```
vnumlparser -t sceanrio1.xml -Z
```

kann VNUML gestartet werden, nachdem es richtig konfiguriert wurde. Ein Listing der Szenariodatei befindet sich im Anhang (siehe A.4).

6.1.1 Konfiguration

Bevor das Szenario selbst analysiert wird, wird die Konfiguration des Hostsystems und aller Einstellungen, die gemacht wurden beschrieben. Es werden folgende Punkte erläutert:

1. Installation von OLSR auf den virtuellen Maschinen.

[1] http://neweb.dit.upm.es/vnumlwiki/index.php/Main_Page
[2] http://wiki.qemu.org/Main_Page

2. Konfiguration von olsrd.conf.

Zur Installtion von OLSR auf den virtuellen Maschinen muss zunächst eine implementierte Version des Routingverfahrens heruntergeladen werden[3]. In diesem Szenario wurde die Version 0.6.0 verwendet. Danach muss die entpackte Version auf das Filesystem von VNUML kopiert und dort installiert werden. Dazu wird das Filesystem zunächst mit dem Befehl

```
mount -o loop root_fs_tutorial mntpoint/
```

gemountet. 'mntpoint/' ist ein leerer Ordner, der als Mountpoint für das Filesystem dient. Jetzt kann der entpackte olsrd-0.6.0 Ordner per

```
cp -R ~/olsrd-0.6.0 mntpoint/home
```

in das home-Verzeichnis der virtuellen Maschinen installiert werden. Die einfachste Variante die Version zu installieren ist, zuerst das root-Verzeichnis mit

```
chroot mntpoint/
```

zu wechseln. Danach muss mit

```
cd /home/olsrd-0.6.0
```

in das OLSR Verzeichnis gewechselt werden. Hier kann nun nach der Installationsleitung aus der beiliegenden Readme Datei vorgegangen werden (siehe A.5). Der nächste Schritt der Konfiguration ist die Bearbeitung der olsrd.conf Datei. Hier werden alle Run-Time Einstellungen für OLSR angegeben. Sie befindet sich nach der Installation im Ordner '/etc'. Eine genau Beschreibung der Datei befindet sich unter[4]. Im Anhang ist die Konfigurationsdatei, die für dieses Szenario verwendet wurde aufgelistet (siehe A.6). Die wichtigsten Einstellungen werden im Folgenden kurz erläutert.

DebugLevel 2: Diese Zeile bewirkt, dass OLSR nicht im Hintergrund läuft, sondern es eine Standartausgabe gibt.

IpVersion 4: Es wird mit IPv4 gearbeitet.

[3]http://www.olsr.org/?q=download
[4]http://www.olsr.org/docs/olsrd.conf.5.html

LinkQualityLevel 2: Bewirkt, dass die Link Quality bei der Erstellung der Routingtabellen berücksichtigt wird.

LinkQualityAlgorithm: Die Wahl des Link Quality Algorithmus, welcher gewählt werden muss, sobald LinkQualityLevel auf zwei gesetzt wurde. Standart und empfehlenswert ist 'ext ff'.

Interface configuration: Hier müssen die Namen aller Interfaces eingetragen werden, welche gesonderte Einstellungen erhalten sollen.

Mode Teilt OLSR mit, um welche Art von Interfaces es sich handelt. In diesem Szenario sind es kabelgebundene Verbindungen, somit wir der Parameter auf 'ether' gesetzt.

HelloInterval 2.0 Alle zwei Sekunden wird eine HELLO-Nachricht (siehe Kap. 3.2.2) verschickt.

HelloValidityTime 6.0 eine HELLO-Nachricht ist sechs Sekunden gültig.

OLSR ist jetzt fertig konfiguriert sowie installiert und kann nun verwendet werden. Auf einer laufenden virtuellen Maschine kann der Routingalgorithmus mit

```
olsrd
```

gestartet werden. Daraufhin folgt die Debugausgabe.

6.1.2 Analyse

Vorab sei erwähnt, dass man auch das eigens erstellte Skript aus Kapitel 5 benutzen kann, um die Verlustraten an den Interfaces einzustellen. Das Skript kann mit

```
python tc-skript ../VNUML/scenario1.xml
```

gestartet werden. Man kann dann die Prozentwerte an den einzelnen Interfaces einstellen und mit 'OK' bestätigen (siehe Abb. 6.1). So ist eine schnellere Bedienung möglich.

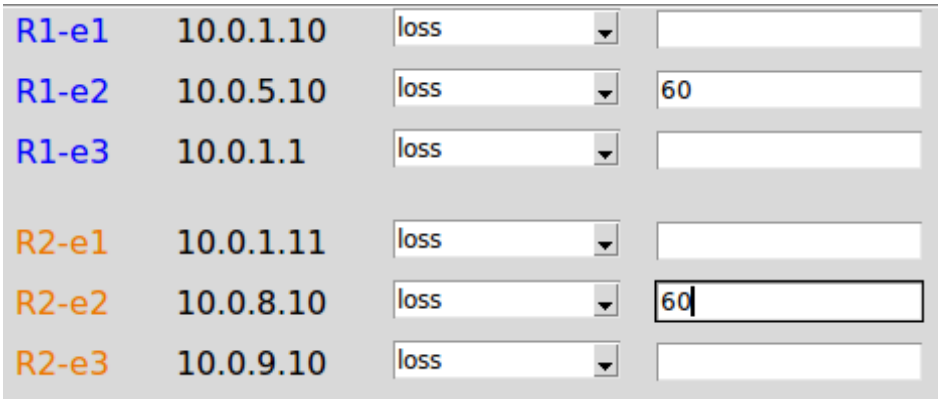

Abbildung 6.1: Skript

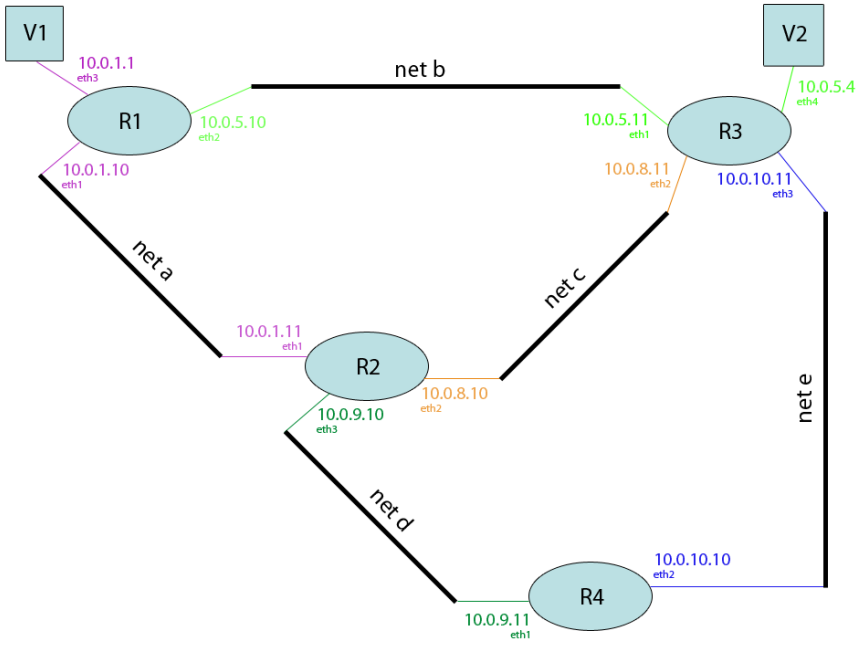

Abbildung 6.2: Szenario 1

Das Szenario besteht aus vier Routern, fünf Netzen und zwei 'Hosts'. Im Folgenden wird die Bezeichnung *Host* für V1 und V2 verwendet, obwohl es sich im Szenario um eine einfache Schnittstelle handelt. Aus Verständnisgründen ist die Bezeichnung *Host* allerdings einfacher. An Router R1 befindet sich der Host V1.

Von diesem Host wird nun der Weg eines IP-Paketes zu Host V2 verfolgt, der sich an Router R3 befindet. Der kürzeste Weg für ein Datenpaket führt von R1 aus, über Interface eth2 (10.0.5.10), direkt zu R3 an Interface eth1 (10.0.5.11). Dieses Interface liegt nun direkt an Router R3, mit dem auch Host V2 verbunden ist. Das Datenpaket hat somit sein Ziel (*Destination*) erreicht. Die folgende Abbildung 6.3 zeigt die Routingtabelle des Routers R1.

```
R1:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.1.11       0.0.0.0         UG    0      0        0 eth1
10.0.1.0        0.0.0.0         255.255.255.0   U     0      0        0 eth1
10.0.1.0        0.0.0.0         255.255.255.0   U     0      0        0 eth3
10.0.1.11       0.0.0.0         255.255.255.255 UH    2      0        0 eth1
10.0.5.0        0.0.0.0         255.255.255.0   U     0      0        0 eth2
10.0.5.4        0.0.0.0         255.255.255.255 UH    2      0        0 eth2
10.0.5.11       10.0.5.4        255.255.255.255 UGH   2      0        0 eth2
10.0.8.10       10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
10.0.8.11       10.0.5.4        255.255.255.255 UGH   2      0        0 eth2
10.0.9.10       10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
10.0.9.11       10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
10.0.10.10      10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
10.0.10.11      10.0.5.4        255.255.255.255 UGH   2      0        0 eth2
192.168.0.100   0.0.0.0         255.255.255.252 U     0      0        0 eth0
R1:~#
```

Abbildung 6.3: Routingtabelle von R1 ohne TC

Die markierte Zeile zeigt, dass Pakete, welche die Zieladresse (*Destination*) 10.0.5.4 haben, über das Interface eth2 geschickt werden. Eine Verfolgung der Route mittels *traceroute* ergibt dann folgende Ausgabe (siehe Abb. 6.4)

```
R1:~# traceroute -n 10.0.5.4
traceroute to 10.0.5.4 (10.0.5.4), 30 hops max, 60 byte packets
 1  10.0.5.4  0.140 ms  0.052 ms  0.048 ms
R1:~#
```

Abbildung 6.4: Traceroute ohne TC

Ein Datenpaket von V1 nach V2 benutzt also tatsächlich den intuitiv schnellsten Pfad zum Ziel. Jetzt wird überprüft, was geschieht, wenn die Leitung (*Link*) zwischen R1 (10.0.5.10) und R3 (10.0.5.11) fehlerhaft ist. Dazu wird mit TC ein Paketverlust von 60% auf diese Leitung gelegt. Dies kann auf dem Hostsystem, auf dem das Szenario simuliert wird, mit folgendem Befehl realisiert werden:

```
tc qdisc add dev R1-e2 root netem loss 60%
```

Kurz nachdem dieser Befehl ausgeführt wurde, registriert OLSR diese Änderung.

```
       *** olsr.org - 0.6.0 (2011-10-20 17:16:19 on dave-laptop) ***

--- 11:03:35.620769 ------------------------------------------------- LINKS

IP address       hyst         LQ       ETX
10.0.1.11        0.000  1.000/1.000    1.000
10.0.1.11        0.000  1.000/1.000    1.000
10.0.5.4         0.000  0.345/1.000    2.897
10.0.5.11        0.000  0.525/1.000    1.902

--- 11:03:35.621096 ---------------------- TWO-HOP NEIGHBORS

IP addr (2-hop)  IP addr (1-hop)  Total cost
10.0.9.11        10.0.5.11        2.677
                 10.0.1.11        2.000
10.0.1.11        10.0.5.11        2.677
10.0.5.11        10.0.1.11        2.000

--- 11:03:35.621481 ------------------------------------------------ TOPOLOGY

Source IP addr  Dest IP addr          LQ        ETX
10.0.1.10       10.0.1.11       1.000/1.000    1.000
10.0.1.10       10.0.5.11       0.596/1.000    1.677
10.0.1.11       10.0.1.10       1.000/1.000    1.000
10.0.1.11       10.0.5.11       1.000/1.000    1.000
10.0.1.11       10.0.9.11       1.000/1.000    1.000
10.0.5.11       10.0.1.10       1.000/0.596    1.677
10.0.5.11       10.0.1.11       1.000/1.000    1.000
10.0.5.11       10.0.9.11       1.000/1.000    1.000
10.0.9.11       10.0.1.11       1.000/1.000    1.000
10.0.9.11       10.0.5.11       1.000/1.000    1.000
```

Abbildung 6.5: OLSR von R1-e2 mit 60% Loss

In Abbildung 6.5 ist ein Ausschnitt aus der Debugausgabe von OLSR auf Router R1 zu sehen. Man kann an den markierten Zeile erkennen, dass sich der ETX Wert (siehe Kap. 3.2.1) des Links zur Zieladresse (10.0.5.4) von 1.000 auf 2.897 verschlechtert hat. OLSR hat den simulierten Paketverlust also registriert. Es folgt eine Änderung in der Routingtabelle von Router R1 (siehe Abb. 6.6).

```
R1:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.1.11       0.0.0.0         UG    0      0        0 eth1
10.0.1.0        0.0.0.0         255.255.255.0   U     0      0        0 eth1
10.0.1.0        0.0.0.0         255.255.255.0   U     0      0        0 eth3
10.0.1.11       0.0.0.0         255.255.255.255 UH    2      0        0 eth1
10.0.5.0        0.0.0.0         255.255.255.0   U     0      0        0 eth2
10.0.5.4        10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
10.0.5.11       10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
10.0.8.10       10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
10.0.8.11       10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
10.0.9.10       10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
10.0.9.11       10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
10.0.10.10      10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
10.0.10.11      10.0.1.11       255.255.255.255 UGH   2      0        0 eth1
192.168.0.100   0.0.0.0         255.255.255.252 U     0      0        0 eth0
R1:~#
```

Abbildung 6.6: Routingtabelle von R1 mit 60% Loss

Wenn nun die Routingtabelle mit der vorherigen Routingtabelle (siehe Abb. 6.3) verglichen wird, ist zu sehen, dass sich der Eintrag mit der Zieladresse 10.0.5.4 verändert hat. Die Route führt jetzt nicht mehr über das Standartgateway, sondern über 10.0.1.11 und Interface eth1.
Im Hinblick auf das Szenario, kann erkannt werden, dass von R1 nicht mehr der direkte Weg zu R3 gewählt wird, der jetzt über die schlechte Verbindung führt. Stattdessen wird ein Umweg über R2 als besser eingestuft. Dies kann auch mit einem Traceroute gezeigt werden (siehe Abb. 6.7).

```
R1:~# traceroute -n 10.0.5.4
traceroute to 10.0.5.4 (10.0.5.4), 30 hops max, 60 byte packets
 1  10.0.1.11  0.204 ms  0.157 ms  0.162 ms
 2  10.0.5.4  0.339 ms  0.250 ms  0.271 ms
R1:~#
```

Abbildung 6.7: Traceroute mit 60% Loss an R1-e2

Das Ziel wird jetzt über 2 Hops (10.0.1.11 und 10.0.5.4) erreicht.
Im nächsten Schritt dieses Szenarios wird nicht nur über R2, sondern auch über R4 gesprungen, um von R1 zu R3 zu gelangen. Dazu wird auf dem Hostsystem das Interface R2-e2 mit folgendem Befehl manipuliert:

```
tc qdisc add dev R2-e2 root netem loss 60%
```

```
R2:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.8.11       0.0.0.0         UG    0      0        0 eth2
10.0.1.0        0.0.0.0         255.255.255.0   U     0      0        0 eth1
10.0.1.1        10.0.1.10       255.255.255.255 UGH   2      0        0 eth1
10.0.1.10       0.0.0.0         255.255.255.255 UH    2      0        0 eth1
10.0.5.4        10.0.8.11       255.255.255.255 UGH   2      0        0 eth2
10.0.5.10       10.0.1.10       255.255.255.255 UGH   2      0        0 eth1
10.0.5.11       10.0.8.11       255.255.255.255 UGH   2      0        0 eth2
10.0.8.0        0.0.0.0         255.255.255.0   U     0      0        0 eth2
10.0.8.11       0.0.0.0         255.255.255.255 UH    2      0        0 eth2
10.0.9.0        0.0.0.0         255.255.255.0   U     0      0        0 eth3
10.0.9.11       0.0.0.0         255.255.255.255 UH    2      0        0 eth3
10.0.10.10      10.0.9.11       255.255.255.255 UGH   2      0        0 eth3
10.0.10.11      10.0.8.11       255.255.255.255 UGH   2      0        0 eth2
192.168.0.104   0.0.0.0         255.255.255.252 U     0      0        0 eth0
R2:~#
```

Abbildung 6.8: Routingtabelle von R2 ohne Paketverlust

Interface eth2 von Router R2 wird jetzt auch mit einem Paketverlust von 60% belegt.

```
R2:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.8.11       0.0.0.0         UG    0      0        0 eth2
10.0.1.0        0.0.0.0         255.255.255.0   U     0      0        0 eth1
10.0.1.1        10.0.1.10       255.255.255.255 UGH   2      0        0 eth1
10.0.1.10       0.0.0.0         255.255.255.255 UH    2      0        0 eth1
10.0.5.4        10.0.9.11       255.255.255.255 UGH   2      0        0 eth3
10.0.5.10       10.0.1.10       255.255.255.255 UGH   2      0        0 eth1
10.0.5.11       10.0.9.11       255.255.255.255 UGH   2      0        0 eth3
10.0.8.0        0.0.0.0         255.255.255.0   U     0      0        0 eth2
10.0.8.11       10.0.9.11       255.255.255.255 UGH   2      0        0 eth3
10.0.9.0        0.0.0.0         255.255.255.0   U     0      0        0 eth3
10.0.9.11       0.0.0.0         255.255.255.255 UH    2      0        0 eth3
10.0.10.10      10.0.9.11       255.255.255.255 UGH   2      0        0 eth3
10.0.10.11      10.0.9.11       255.255.255.255 UGH   2      0        0 eth3
192.168.0.104   0.0.0.0         255.255.255.252 U     0      0        0 eth0
R2:~#
```

Abbildung 6.9: Routingtabelle von R2 mit 60% Loss

An der Routingtabelle von R2 ohne Paketverlust (siehe Abb. 6.8) kann abgelesen werden, dass bei der Zieladresse 10.0.5.4 der direkte Weg über eth2 und 10.0.8.11 gewählt wird. Nachdem an diesem Interface jedoch auch ein Verlust von 60% anliegt, wird von OLSR die Route über eth3 und 10.0.9.11 eingetragen (siehe Abb. 6.9). Ein Traceroute mit Verlusten an R1-e1 und R2-e2 ergibt nun folgende

Ausgabe (siehe Abb. 6.10).

```
traceroute to 10.0.5.4 (10.0.5.4), 30 hops max, 60 byte packets
 1  10.0.1.11  0.247 ms  0.157 ms  0.153 ms
 2  10.0.9.11  0.386 ms  0.299 ms  0.820 ms
 3  10.0.5.4  1.518 ms  1.220 ms  0.854 ms
```

Abbildung 6.10: Traceroute mit 60% Loss an R1-e2 und R2-e2

Zusammenfassend kann man zu Szenario 1 sagen, dass OLSR auf Paketverlust reagiert und an den Routern gegebenenfalls bessere Routen, die nicht überlastet sind, einträgt.

6.2 Szenario 2

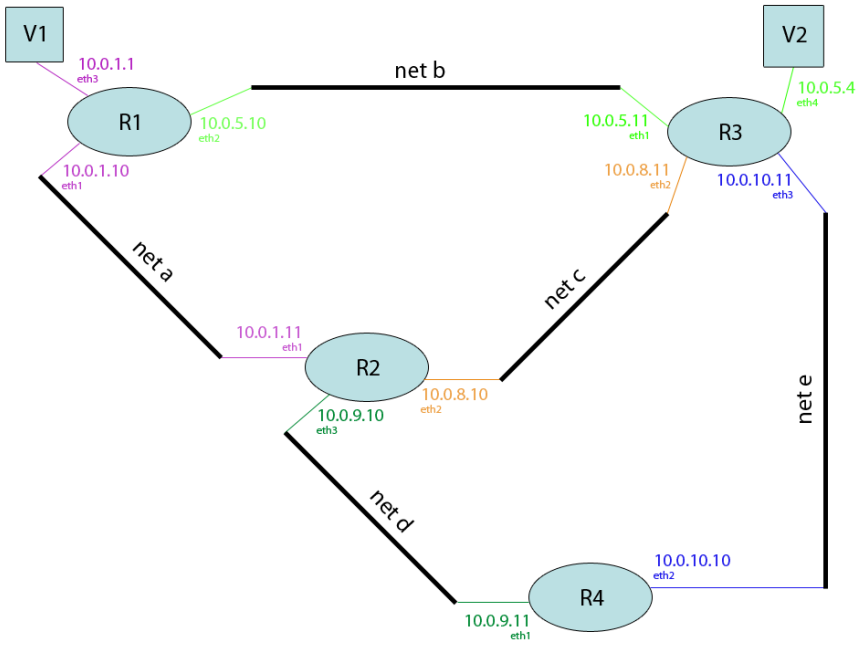

Abbildung 6.11: Szenario 2

Das gleiche Szenario (siehe Abb. 6.11) wird nun noch mit dem Distanzvektor Protokoll Babel (siehe Kap. 3.3) durchgeführt. Dazu muss auf den virtuellen Maschi-

nen lediglich eine implementierte Version von Babel anstelle von OLSR installiert werden.
Im Zuge dieser Arbeit wurde die Version 1.3.0[5] benutzt. Diese Version muss nun analog zu der Anleitung aus Kapitel 6.1 entpackt und in das Filesystem der virtuellen Maschinen installiert werden. Ist dieser Schritt erledigt, kann Babel dann mit diesem Befehl ausgeführt werden:

```
babeld eth1 eth2
```

Es muss der Name von jedem Interface angegeben werden, welches die virtuelle Maschine besitzt und das Teil des Netzwerks ist.
Nun können mithilfe des Skripts wieder die gleichen Manipulationen wie in Szenario 1 vorgenommen werden. Es ist zu beobachten, dass es auch bei der Ausgabe von Babel zu den gleichen Ergebnissen kommt wie in Szenario1 kommt (siehe Kap. 6.1.2). Die Routen der einzelnen Router werden nach Registrierung der Verluste auf bessere Leitungen umgeleitet.

6.3 Szenario 3

Dieses Szenario dient dazu, die Funktion des Paketverlusts und der Verzögerung von Traffic Control (*TC*) so zu nutzen, dass eine Analyse der Staukontrolle durch TCP Reno und TCP Vegas ermöglicht wird. Dazu wird mit dem QEMU Tool (siehe Kap. 2.2) ein Netz aus vier Hosts und einer virtuellen Brücke auf dem Hostsystem erstellt (siehe Abb. 6.12).

[5]http://www.pps.jussieu.fr/ jch/software/files/

6.3.1 Konfiguration

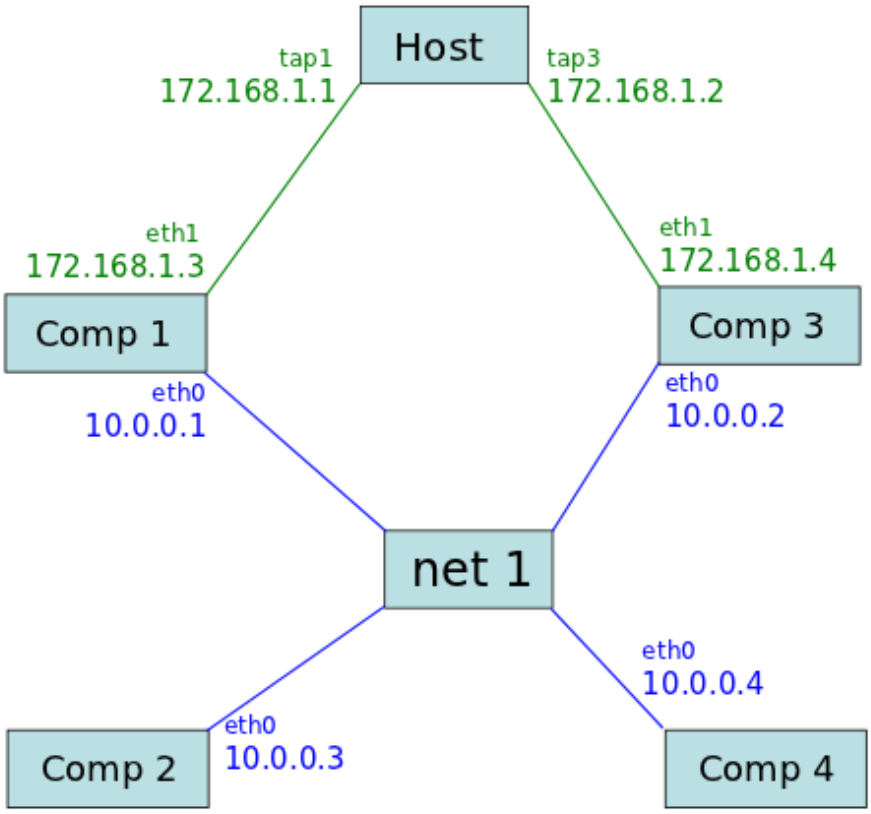

Abbildung 6.12: Szenario 3

Vorab ist zu sagen, dass alle folgenden Befehle als SuperUser (root) ausgeführt werden müssen und für die Messung der cwnd und des ssthresh das Kernelmodul tcp_probe eingebunden werden muss. Diese Einbindung ist mit User Mode Linux nicht möglich. Daher konnte zur Simulation dieses Szenarios kein VNUML benutzt werden. Es wurde stattdessen die *Full System Emulation* Option von QEMU (siehe Kap. 2.2) benutzt, um die vier Maschinen zu emulieren.
Zuerst müssen aber die Netzwerkbrücke und die Netzwerkschnittstellen auf dem Hostsystem erstellt werden, damit die simulierten Maschinen später Teil eines Netzwerkes sind. Für die Netzwerkinterfaces werden TUN/TAP-Gerät erzeugt. Es müssen sechs TAP Ethernet-Geräte erstellt werden, da zwei der virtuellen Maschinen noch mit dem Hostsystem verbunden werden sollen. Dies dient später

dem Datenaustausch zwischen Host und Virtueller Maschine (VM). Ein TUN/TAP Gerät mit Namen 'tap1' wird mit folgendem Befehl erstellt

```
tunctl -t tap1 -f /dev/net/tun
```

Jetzt muss noch die Netzwerkbrücke erstellt werden und dieser müssen die gerade generierten TUN/TAP Geräte hinzugefügt werden. Das geschieht durch die Befehle

```
brctl addbr net1
brctl addif net1 tap1
```

Es wird eine Brücke mit Namen 'net1' erstellt und das Interface 'tap1' wird zur Brücke 'net1' hinzugefügt. Es müssen allerdings nur vier der sechs erstellten Interfaces zur Brücke hinzugefügt werden, da die anderen beiden zur Kommunikation mit dem Host dienen (siehe Abb. 6.12). Alle an die Brücke angeschlossenen Interfaces können später im virtuellen Netzwerk miteinander kommunizieren. Die Interfaces zur Hostkommunikation werden wie folgt konfiguriert

```
ifconfig tap1 172.168.1.1 netmask 255.255.255.248
```

Dieser Befehl ordnet dem Interface eine IP-Adresse zu, damit es später angesprochen werden kann.
Jetzt können die VMs erstellt und gestartet werden. Für die Maschinen wurde der Kernel[6] 2.6.38.2 benutzt. Als Dateisystem steht ein Overlayimage für jede Maschine zur Verfügung. Ein solches kann mit dem Befehl

```
qemu-img create -b rootfs_tutorial_forqemu.qcow2
 -f qcow2 comp1.ovl
```

erzeugt werden. Die Datei[7] *rootfs_tutorial_forqemu.qcow2* muss dazu heruntergeladen werden.
Stehen Kernel und Dateisystem zur Verfügung, können die Maschinen mithilfe von QEMU gestartet werden. Dies ist mit

```
qemu -hda comp1.ovl -append "root=/dev/sda console=tty0"
-kernel bzImage-2.6.38.2 -daemonize -net nic,
```

[6] http://git.uni-koblenz.de/vm-ressources/vm-ressources/trees/master/kernels/2.6.38.2
[7] http://git.uni-koblenz.de/vm-ressources/vm-ressources/trees/master/images

```
model=e1000,macaddr=52:54:00:12:53:01,vlan=0 -net tap,
vlan=0,ifname=tap1,script=no -net nic,model=e1000,
macaddr=52:54:00:12:53:02,vlan=1 -net tap,vlan=1,
ifname=tap2,script=no
```

möglich.
Eine nähere Erläuterung des Befehls folgt im weiteren Verlauf.
Sind nun alle Interfaces konfiguriert und zugeordnet, werden die VM's mit obigem Befehl gestartet. Dabei sind in diesem Beispiel einer Maschine zwei Interfaces zugeordnet. Zum einen 'tap1' mit der MAC Adresse '52:54:00:12:53:01' und zum anderen 'tap2' mit der MAC Adresse '52:54:00:12:53:02'. Ein Skript, dass die Erstellung, Konfiguration und das Starten übernimmt befindet sich im Anhang der Arbeit (siehe A.2). Sind nun alle Maschinen gestartet, muss beim einloggen der Benutzer 'root' und das Passwort 'xxxx' benutzt werden. Für weitere Informationen zur Konfiguration von QEMU siehe [8].
Abschließend muss auf jeder virtuellen Maschine jedes Interface einzeln konfiguriert werden. Dazu wird nach dem Einloggen folgender Befehl ausgeführt

```
ifconfig eth0 172.168.1.3
ifconfig eth1 10.0.0.1
```

Dies waren die Befehle für die Maschine Comp1. Analog muss dies für Comp2, Comp3 und Comp4 durchgeführt werden. Um diese Befehl nicht jedes Mal nach dem Start einer Maschine ausführen zu müssen, kann man die Datei /etc/network/interfaces anpassen. Ein Beispiel für Comp1 befindet sich im Anhang. So wird die Konfiguration automatisch geladen. Ebenso kann man einen Hostname vergeben. Dazu ist der gewünschte Name nur in die Datei /etc/hostname zu schreiben. Nun ist dieses Szenario vollständig konfiguriert und es können Tests der Staukontrolle vorgenommen werden. Dazu müssen mit Hilfe von tcp_probe (siehe Kap. 2.4) alle Werte ausgelesen werden. Mit

```
modprobe tcp_probe port=5001
```

wird das Kernelmodul eingebunden. Dies muss auf beiden Maschinen gemacht werden, die mit dem Host verbunden sind. So können die Messergebnisse später übertragen werden.

6.3.2 Analyse

In diesem Abschnitt werden die Ergebnisse von verschiedenen Tests bezüglich TCP Reno und TCP Vegas ausgewertet und analysiert. Zu Beginn ist zu sagen, dass es sich jeweils um stichprobenartige Ergebnisse handelt und so keine vollkommene Korrektheit gewährleistet werden kann. Es soll aber beobachtet werden, ob ein Zusammenhang zwischen den theoretischen Überlegungen aus Kapitel 4 und den praktischen Messergebnissen zu sehen ist. Dazu wurden folgende Test durchgeführt:

1. TCP Reno:

- Es wird beobachtet, wie sich TCP Reno auf einer Netzwerkleitung ohne Verlust und ohne Verzögerung verhält.
- Es wird beobachtet, wie sich TCP Reno auf einer Netzwerkleitung mit Paketverlust verhält.

2. TCP Vegas:

- Es wird beobachtet, wie sich TCP Vegas auf einer Netzwerkleitung ohne Verlust und ohne Verzögerung verhält.
- Es wird beobachtet, wie sich TCP Vegas auf einer Netzwerkleitung mit Verzögerung verhält.

3. TCP Reno vs TCP Reno: Es wird der Fairness Aspekt beobachtet, wenn zwei Host über eine Leitung mit der TCP Reno als Staukontrollmechanismus Pakete senden.

4. TCP Vegas vs TCP Vegas: Es wird der Fairness Aspekt beobachtet, wenn zwei Host über eine Leitung mit der TCP Vegas als Staukontrollmechanismus Pakete senden.

5. TCP Reno vs TCP Vegas: Es wird beobachtet, wie sich die Staukontrolle verhält, wenn ein Host mit TCP Reno und einer mit TCP Vegas als Staukontrollmechanismus Pakete sendet.

Vorab wird nun kurz beschrieben, wie diese Tests durchgeführt wurden. Abschließen werden dann noch die Ergebnisse analysiert. Um die Durchführung zu

beschreiben, wird dies beispielhaft an einem Testszenario dargestellt. Das Testszenario, welches beschrieben wird, ist TCP Reno mit Paketverlust.
Allgemein ist zu sagen, dass ein Test folgendermaßen abläuft:

1. Es werden ein Sender und ein Empfänger definiert.
2. Der Sender schickt Daten per TCP an den Empfänger. Dieser Vorgang wird mit dem Tool *iperf* durchgeführt, welches in der Lage ist, TCP Verkehr zu erzeugen.
3. Der Sender speichert alle Daten des Sendervorgangs, die mit tcpprobe (siehe Kap. 2.4) gesammelt werden können.
4. Die gespeicherten Daten werden von der Sender Maschine auf den Host übertragen und dort ausgewertet.

In dem Beispiel legen wir nun Comp3 als Empfänger fest. Auf diesem muss nun ein iperf Server gestartet werden. Das geschieht mit

```
iperf -s
```

Comp3 ist nun bereit, Daten zu empfangen. Jetzt definieren wir Comp1 als Sendermaschine. Wie bereits erwähnt, werden Daten mithilfe von tcp_probe gespeichert. Damit dies möglich ist, muss erst das entsprechende Kernelmodul eingebunden werden. Das ist mit

```
modprobe tcp_probe port=5001
```

zu tun. Es sei erwähnt, dass iperf standardmäßig Daten über den Port 5001 verschickt. Nun müssen wir Zugriff auf die Datei gewähren, in welche die gesammelten Daten geschrieben werden sollen.

```
chmod 444 /proc/net/tcpprobe
```

Jetzt ist der Sender fast bereit Daten zu verschicken. Es wird nur noch ein Prozess gestartet, welcher die gesammelten Daten in eine separate Datei schreibt. Zusätzlich muss aber auch noch eine Verlustrate auf das Interface gelegt werden, da das Ziel unserer Messung die Beobachtung von TCP Reno auf einer verstopften Leitung ist.

```
tc qdisc add dev eth1 root netem loss 3%
cat /proc/net/tcpprobe > reno_loss.out & TCPCAP=$!
```

In der ersten Zeile wird eine Verlustrate von drei Prozent auf das Interface 'eth1' gelegt. Über dieses Interface sendet Comp1 Daten in das Netzwerk. Mit dem zweiten Befehl wird im Hintergrund ein Prozess gestartet, der die Daten aus '/proc/net/tcpprobe' in die Daten 'reno_loss.out' schreibt, sobald dieser wieder beendet wird. Daraufhin werden die Daten verschickt.

```
iperf -c 10.0.0.3 -i 1 -n 20971520 -Z reno
```

Dieser Befehl verschickt zwanzig Megabyte (20971520 Byte) Daten mit dem Stauprotokoll TCP Reno an die Zieladresse '10.0.0.3' und gibt jede Sekunde (-i 1) eine Ausgabe über den Datendurchsatz auf dem Bildschirm aus. Ist die Übertragung erfolgreich beendet, so wird mit

```
kill $TCPCAP
```

der gesamte Messvorgang abgeschlossen. Es stehen jetzt alle Ergebnisse in der Datei 'reno_loss.out'. Diese wird nun mit

```
scp reno_loss.out Benutzer@172.168.1.1:
```

in das Homeverzeichnis des Hosts übertragen. Hier können die Daten mithilfe von *gnuplot* visualisiert werden. Ein Skript das bei der Visualisierung hilft, befindet sich im Anhang (siehe A.3). Mit dieser Vorgehensweise sind folgende Ausgaben entstanden.

6.3.2.1 TCP Reno

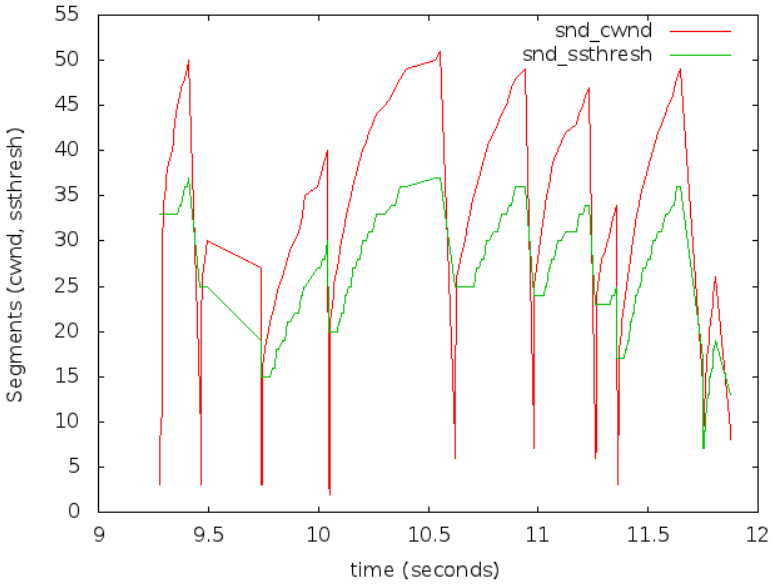

Abbildung 6.13: TCP Reno

Das erste Messergebnis stellt Abbildung 6.13 dar, welche die cwnd und den ssthresh einer Datenübertragung mit TCP Reno auf einer freien Leitung darstellt. Es ist zu erkennen, dass die Phase des Slow Starts bis zum ssthresh andauert und das die cwnd in dieser Phase sehr schnell wächst. Weiterhin der Übergang zu Congestion Avoidance und der damit lineare Anstieg zu erkennen. Abschließen ist noch anzumerken, dass bei Paketverlust die cwnd wieder auf null gesetzt wird und der ssthresh halbiert wird, so wie es im theoretischen Teil beschrieben ist. Bei dieser stichprobenartigen Messung entspricht der theoretische Teil also der praktischen Messung.

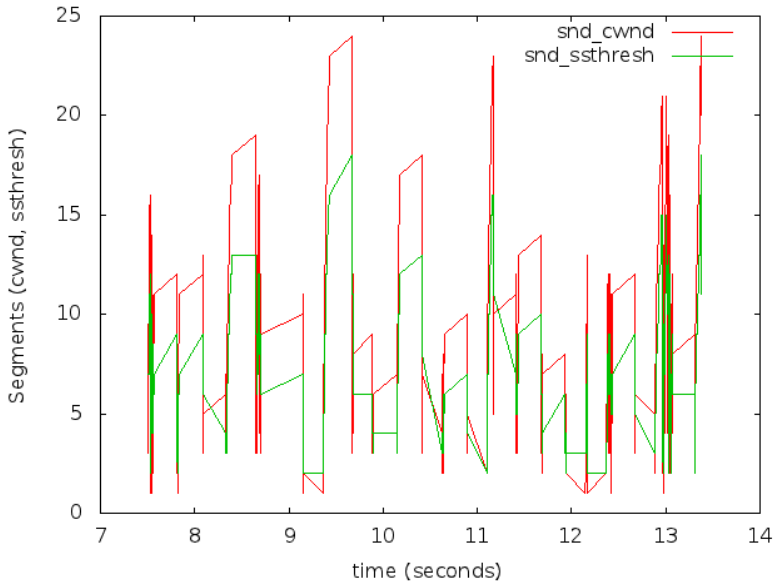

Abbildung 6.14: TCP Reno mit Verlustrate von 3%

Bei der nächsten Messung wurde mit Traffic Control (*TC*) ein Paketverlust von drei Prozent auf der Leitung angelegt (siehe Abb. 6.14). Es ist deutlich zu erkennen, dass die cwnd häufiger zurück gesetzt wird und wieder in der Slow Start Phase begonnen wird. Daraus lässt sich schließen, dass es häufiger zu Paketverlust gekommen ist, was den Einstellungen dieses Szenarios entspricht. In diesem Fall decken sich theoretische Überlegungen und praktische Ergebnisse also wieder. Was bei diesem Versuch auch zu erkennen ist, dass die cwnd nur bis zu einer Segmentgröße von 25 ansteigt, wohingegen beim Test ohne Paketverlust eine Segmentgröße von 50 gegeben war.

6.3.2.2 TCP Vegas

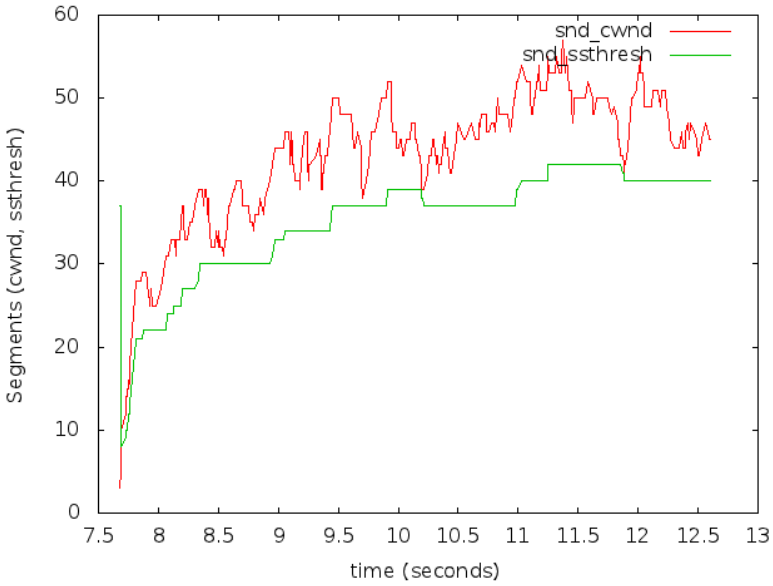

Abbildung 6.15: TCP Vegas

In diesem Unterkapitel werden die Messergebnisse zu TCP Vegas analysiert. Zunächst wurden Daten über *iperf* mit TCP Vegas übertragen. Es ist zu sagen, dass dazu vorher eine Einstellung in der virtuellen Maschine vorzunehmen ist. TCP Vegas muss dort als Staukontrolle registriert werden. Dies ist mit

```
sysctl -w net.ipv4.tcp_congestion_control=vegas
```

möglich. Das Ergebnis der Messung ist in Abbildung 6.15 zu sehen. Hier ist sehr gut zu erkennen, dass es im Gegensatz zu TCP Reno keine Slow Start Phase gibt, sondern die cwnd angemessen kontrolliert wird. Die cwnd und der ssthresh konvergieren, wie es die Theorie besagt, gegen eine bestimmte Größe. Die cwnd steigt zu Beginn der Übertragung kontinuierlich an und pendelt sich dann bei einer Größe von ca. fünfzig Segmenten ein.

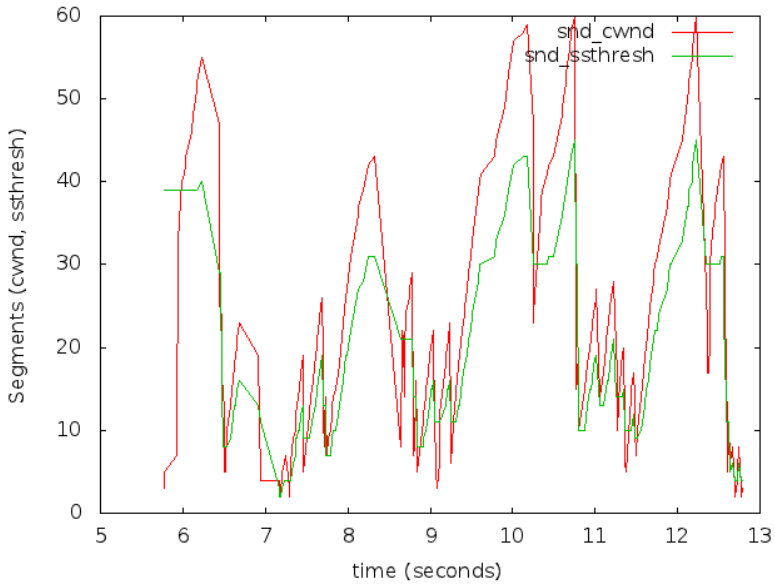

Abbildung 6.16: TCP Vegas mit 10 ms Verzögerung

Es wurde nun eine Verlustrate von zwei Prozent und einen Verzögerung von zehn Millisekunden auf der Leitung angelegt. Das Resultat zeigt Abbildung 6.16. Es sind jetzt größere Schwankungen zu erkennen, als wenn keine Verzögerung vorhanden wäre. Die Schwankungen haben einen größeren Ausschlag (von ca. 10 Segmenten auf ca. 40 Segmente) und die Frequenz, mit der die cwnd steigt, ist wesentlich höher.

6.3.2.3 TCP Reno vs TCP Vegas

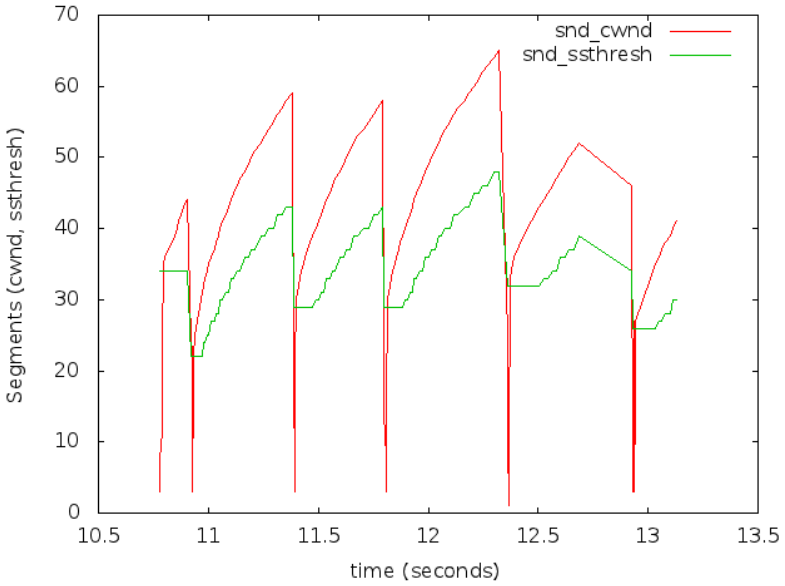

Abbildung 6.17: Comp1 TCP Reno vs Vegas

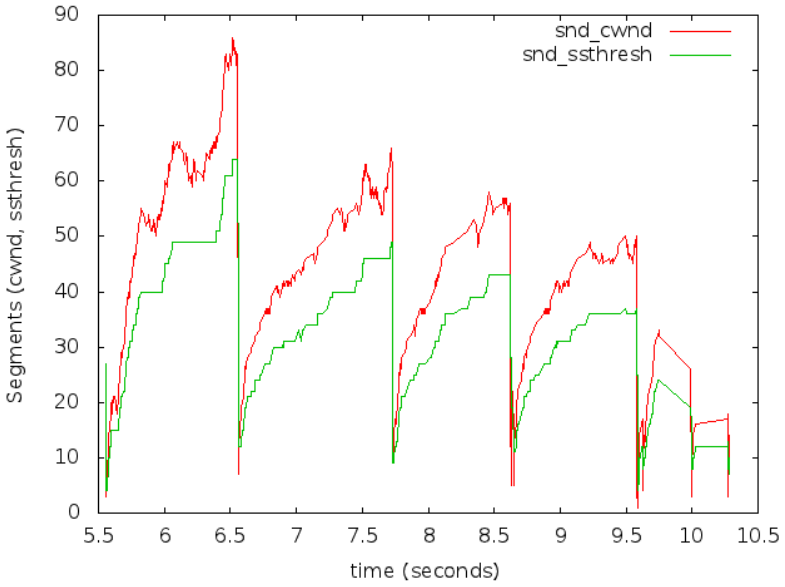

Abbildung 6.18: Comp3 TCP Reno vs Vegas

In diesem Abschnitt wird untersucht, wie sich TCP Reno und TCP Vegas verhalten, wenn die beiden Protokolle zur gleichen Zeit auf zwei unterschiedlichen Hosts eingesetzt werden. Dazu wurden von Comp1 nach Comp2 Daten mit *iperf* und TCP Reno übertragen und zeitgleich Daten mit TCP Vegas von Comp3 nach Comp4. Bei diesen Messergebnissen ist zu erwähnen, dass fast jedes Ergebnis von dem des anderen abgewichen ist und somit keine wirkliche Aussagekraft besteht. Es ist auf den ausgewählten Daten allerdings eine Tendenz zu erkennen. TCP Reno (siehe Abb. 6.17) wird fast normal durchgeführt. Slow Start und Congestion Avoidance sind eindeutig zu erkennen. Bei TCP Vegas (siehe Abb. 6.18) ist zu sehen, dass die cwnd mit zunehmender Zeit im gesamten kleiner wird. Wie aus der allgemeinen Literatur bekannt, ist dies dadurch zu erklären, dass TCP Reno aufgrund seines reaktiven Verhaltens Bandbreite aggresiver einfordert als TCP Vegas. Eine Erklärung für Einknicke nach unten konnte nicht gefunden werden. Hier weichen die praktischen Ergebnisse von den theoretischen ab.

6.3.2.4 TCP Vegas vs TCP Vegas

Dieser Absatz versucht die Messergebnisse zu erläutern, wenn TCP Vegas gleichzeitig von zwei Hosts auf einer Leitung verwendet wird. Wie in Abbildung 6.19 zu erkennen, wendet Comp1 ganz normal TCP Vegas an und die cwnd pendelt sich bei einer Segmentgröße von 50 bis 60 ein. Auf Comp3 (siehe Abb. 6.20) erhöht sich die cwnd stetig. Es handelt sich auch um linears Wachstum.

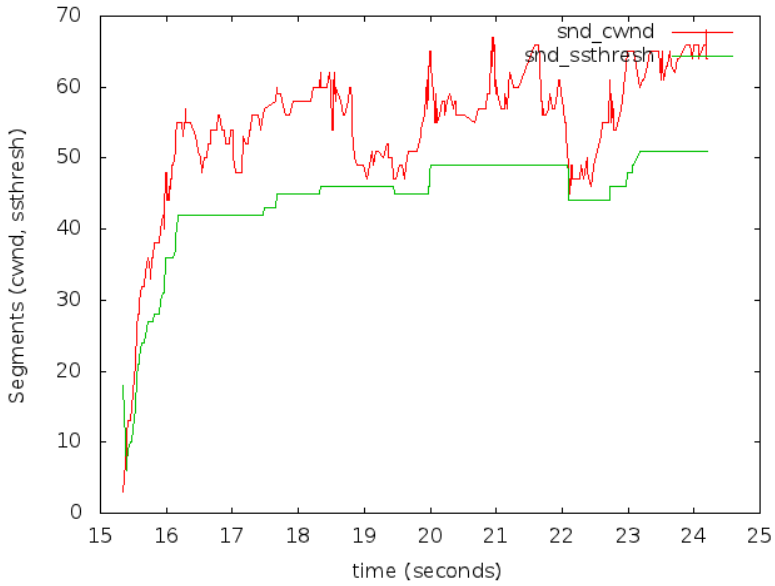

Abbildung 6.19: Comp1 TCP Vegas vs TCP Vegas

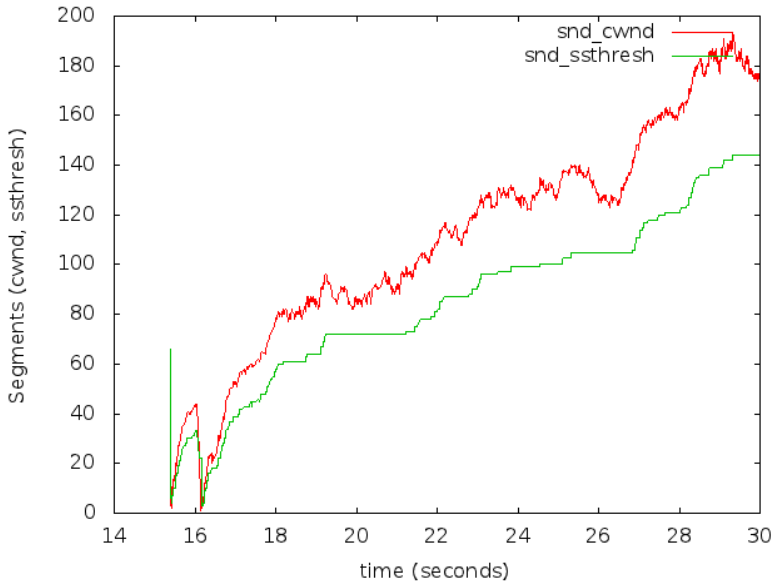

Abbildung 6.20: Comp3 TCP Vegas vs TCP Vegas

6.3.2.5 TCP Reno vs TCP Reno

Im letzten Abschnitt wird untersucht, wie sich TCP Reno verhält, wenn zwei Hosts Daten über eine Leitung übertragen. Die Messung von Comp1 ist in Abbildung 6.21 und die von Comp3 in Abbildung 6.22 dargestellt. Es sind wieder Slow Start und Congestion Avoidance Phasen zu erkennen. Außerdem scheinen sich beide Hosts die Leitung zu teilen, da die Segmentgrößen in etwa im gleichen Bereich liegen.

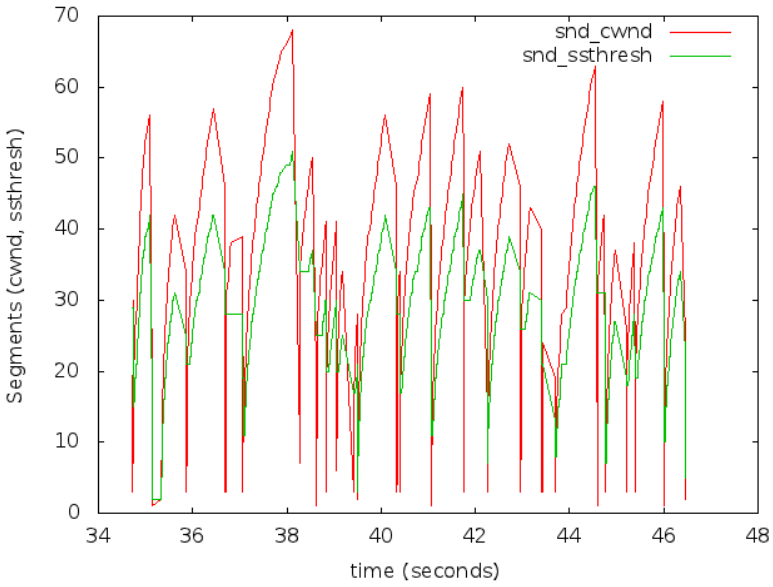

Abbildung 6.21: Comp1 TCP Reno vs TCP Reno

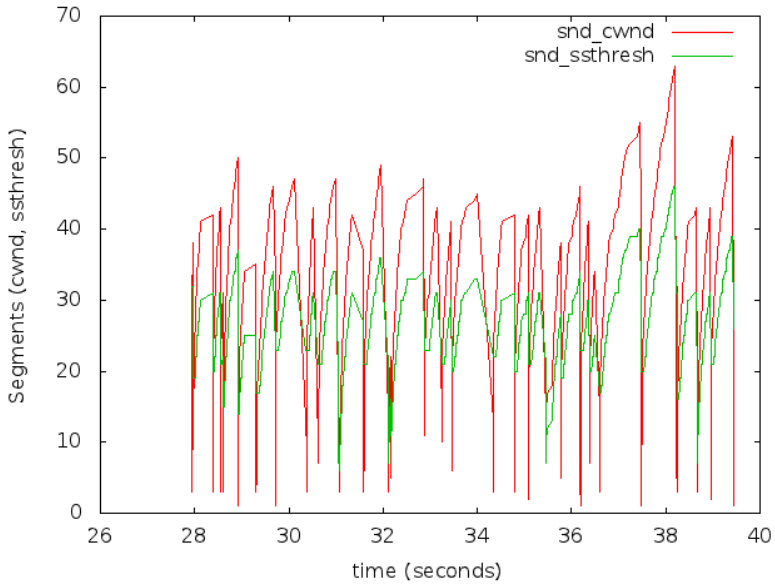

Abbildung 6.22: Comp3 TCP Reno vs TCP Reno

Kapitel 7

Fazit

In der vorliegenden Bachelorarbeit wurden virtuelle Netzwerkverbindungen mit Hilfe von Traffic Control (TC) manipuliert. Die virtuellen Netzwerke wurden mit *Virtual User Mode Linux* (VNUML) und *Quick EMUlator* (QEMU) erstellt.
Die Manipulation mit TC in Form von Paketverlust oder Paketverzögerung diente dazu, die Routingverfahren *Optimized Link State Routing* (OLSR) und *Babel* zu testen. Die Routingalgorithmen reagierten auf die gezielte Manipulation von Netzwerkverbindungen, welche mit TC simuliert wurde, indem sie die Einträge in den Routingtabellen der verschiedenen Router veränderten.
Es war zu erkennen, dass eine Manipulation der Leitung möglich ist und diese auch registriert wurde. Es wäre sinnvoll, diese Tests nun einmal in einem großen virtuellen Netzwerk durchzuführen, sodass beispielsweise die Vorteile der *Multipoint Relays* (MPR) von OLSR zum Vorschein kommen können. Dies war aufgrund der kleinen Szenariogrößen nicht möglich.
Die Tests der Routingalgorithmen waren aber nicht einziges Anwendungsgebiet von TC. Es wurde zusätzlich noch dazu eingesetzt, die Mechanismen der *Congestion Control* von TCP zu testen. TCP Reno und TCP Vegas waren hier die Anwendungsbeispiele. TCP Reno reagiert auf Paketverlust und TCP Vegas auf Paketverzögerungen. Diese wurden mit TC in einem QEMU Szenario getestet. Ergebnis dieser Testszenarien waren geplottete Graphen, bei denen die markanten Eigenschaften der Staukontrollverfahren beobachtet werden konnten.
Eine geeignete Erweiterung wäre es, eine gesamte Testreihe anzulegen. Damit könnten die stichprobenartigen Ergebnisse untermauert werden.
Weiterhin wäre es sinnvoll, in zukünftigen Szenarien nur noch QEMU zu benut-

zen, da die Möglichkeiten der Benutzung größer sind und es zukünftig keine Unterstützung mehr für VNUML geben wird.
Ein weiteres Problem bei der Benutzung von VNUML war, dass *User Mode Linux* nicht in der Lage war, das Kernelmodul tcp_probe einzubinden, welches zum Messen der Werte für die Staukontrolle zuständig war. Dies war auch der Hauptgrund für die Benutzung von QEMU.

Anhang A

Anhang

Es folgt das Python Skript zur Analyse einer VNUML Datei und Erstellung einer graphischen Oberfläche. Mit diesem GUI können Interfaces mit Traffic Control manipuliert werden.

A.1 TC-Skript

```
1  #!/usr/bin/python3
2  from Tkinter import *
3  import ttk
4  import xml.dom.minidom
5  import sys
6  import os
7  import pickle
8  import re
9
10 IF_X = 80
11 ABSTAND_Y = 30
12 LOSSCOL_X = 190
13 DELAYCOL_X = 320
14 TIMERCOL_X = 500
15 CHECK_X = 190
16
17 datei = open(sys.argv[1], "r")
18 dom = xml.dom.minidom.parse(datei)
19 datei.close()
20
21 lossnames = []
22 root = Tk()
23 root.geometry("900x700")
24 combovalues=["loss", "delay", "corrupt", "duplicate",  "
     loss,delay", "loss,corrupt", "delay,corrupt", "delay,
     dupli", "loss,delay,corrupt"]
```

```
25
26 losslist = {}
27 delaylist = {}
28 dellosslist = {}
29 tclist = {}
30 comboif = {}
31 entryif = {}
32 timerif = {}
33 vms = {}
34 colors = {1:"blue", 2:"DarkOrange2", 3:"dark green", 4:"
      DarkMagenta", 5:"SkyBlue3"}
35
36 class IF:
37     Router = 0
38     if_list = []
39     ip_list = {}
40
41     def readip(self,knoten):
42         ip = knoten.getElementsByTagName("ipv4")
43         print(ip[0].firstChild.data)
44
45
46     def readif(self,knoten):
47         for elem in knoten.getElementsByTagName("if"):
48             self.if_list.append(knoten.getAttribute('name')
                    + "-e" + elem.getAttribute('id'))
49             self.ip_list[self.if_list[-1]] = elem.
                  getElementsByTagName("ipv4")[0].firstChild.
                  data
50             vms[self.if_list[-1]] = self.Router
51
52     def dokument(self,start):
53         for elem in start.getElementsByTagName("vm"):
54             self.Router += 1
55             IF.readif(elem)
56
57     def showIF(self):
58         for IF in self.if_list:
59             print(IF)
60
61     def showIP(self):
62         print(self.ip_list)
63
64     def showSpecIP(self,IF):
65         return self.ip_list[IF]
66     def showSpecIF(self,i):
67         self.if_list[i]
68
69 def create():
70     losslist = {}
71     delaylist = {}
72     headlines = Frame(root, height=30, width=500)
73     mainframe = Frame(root, height=600,width=500)
74     buttonframe = Frame(root, height=100, width=500)
```

```
75      separator = Frame(height=2, bd=1, relief=SUNKEN, width
            =500)
76      separator2 = Frame(height=2, bd=1, relief=SUNKEN, width
            =500)
77
78      lblif = Label(headlines, text="Interface" ,font="bold")
79      lblip = Label(headlines, text="IP",font="bold")
80      loss = Label(headlines, text="Loss or Delay",font="bold
            ")
81      delay = Label(headlines, text="Value in % or ms",font="
            bold")
82      #timer = Label(headlines, text="Timer [(v:t),(v:t)]",
            font="bold")
83      okbutton = Button(buttonframe, text="OK", command=
            okclick)
84      quitbutton = Button(buttonframe, text="Quit", command=
            quitclick)
85
86      lblif.place(x=0,y=0)
87      lblip.place(x=IF_X,y=0)
88      loss.place(x=LOSSCOL_X, y=0)
89      delay.place(x=DELAYCOL_X, y=0)
90      #timer.place(x=TIMERCOL_X, y=0)
91      headlines.pack(padx=5, pady=0, fill="both", side=TOP)
92      separator.pack(fill=X, padx=0, pady=0, side=TOP)
93      mainframe.pack(padx=5, pady=0, fill="both", side=TOP)
94      buttonframe.pack(side=BOTTOM)
95      separator2.pack(fill=X, padx=5, pady=5, side=BOTTOM)
96      okbutton.pack(side=LEFT)
97      quitbutton.pack(side=LEFT)
98      j=0
99      for i in IF.if_list:
100         ABSTAND_IF = ((vms[i]-1)*15)
101         labelname_id = "lab%d" % j
102         labelname_ip = "lab_ip%d" % j
103         comboname = "combo%d" % j
104         entryname = "entry%d" % j
105         #timername = "timer%d" % j
106
107         lab_id = Label(mainframe, text=i, name=labelname_id
                , fg=colors[vms[i]],font="bold")
108         lab_ip = Label(mainframe, text=IF.showSpecIP(i),
                name=labelname_ip,font="bold")
109         combo = ttk.Combobox(mainframe, width="12", values
                =combovalues, name=comboname)
110         entry = Entry(mainframe, width=16, name=entryname)
111
112         lab_ip.place(x=IF_X, y = j*ABSTAND_Y+ABSTAND_IF)
113         lab_id.place(x=0, y = j*ABSTAND_Y+ABSTAND_IF)
114         combo.place(x=LOSSCOL_X,y=j*ABSTAND_Y+ABSTAND_IF)
115         entry.place(x=DELAYCOL_X, y= j * ABSTAND_Y+
                ABSTAND_IF)
116
117         combo.set(combovalues[0])
118         comboif[i] = str(combo)
119         entryif[str(entry)] = i
```

```
120             #timerif[str(timer)] = i
121             j+=1
122         root.mainloop()
123
124
125     def cleantc():
126         ausgabe = os.popen("tc qdisc show | grep netem")
127         IF = re.compile('dev\s[a-zA-Z0-9-_]+\s')
128         loss = re.compile('loss \d+\.?\d?%')
129         delay = re.compile('delay \d+.\d[a-z]+')
130         corruption = re.compile('corrupt \d+\.?\d?%')
131         duplication = re.compile('duplicate \d+\.?\d?%')
132
133         for zeile in ausgabe:
134             interface = IF.search(zeile)
135             v1 = ""
136             v2 = ""
137             v3 = ""
138             v4 = ""
139             if interface:
140                 delayval = delay.search(zeile)
141                 if delayval:
142                     v1 = delayval.group()
143
144                 lossval = loss.search(zeile)
145                 if lossval:
146                     v2 = lossval.group()
147
148                 corrval = corruption.search(zeile)
149                 if corrval:
150                     v3 = corrval.group()
151
152                 dupval = duplication.search(zeile)
153                 if dupval:
154                     v4 = dupval.group()
155
156                 os.system("tc qdisc del dev %s root netem %s %s
                        %s %s" % (interface.group()[4:interface.end
                    ()-1],v1,v2,v3,v4))
157
158
159     def quitclick():
160         cleantc()
161         root.destroy()
162
163
164     def okclick():
165         cleantc()
166         frames = root.winfo_children()
167         mainframe = frames[1]
168         for widget in mainframe.winfo_children():
169             if widget.winfo_class() == "Entry":
170                 interface = entryif[str(widget)]
171                 combo = widget.nametowidget(comboif[interface])
172                 combovalue = combo.get()
```

```
173                 if widget.get() != "":
174                     if combovalue == "loss":
175                         os.system("tc qdisc add dev %s root
                               netem loss %s%%" % (interface,
                               widget.get()))
176
177                     if combovalue == "delay":
178                         os.system("tc qdisc add dev %s root
                               netem delay %sms" % (interface,
                               widget.get()))
179
180                     if combovalue == "corrupt":
181                         os.system("tc qdisc add dev %s root
                               netem corrupt %s%%" % (interface,
                               widget.get()))
182
183                     if combovalue == "duplicate":
184                         os.system("tc qdisc add dev %s root
                               netem duplicate %s%%" % (interface,
                               widget.get()))
185
186                     if combovalue == "loss,delay":
187                         os.system("tc qdisc add dev %s root
                               netem loss %s%% delay %sms" % (
                               interface,widget.get().split(',')
                               [0],widget.get().split(',')[1] ))
188
189                     if combovalue == "delay,dupli":
190                         os.system("tc qdisc add dev %s root
                               netem delay %sms duplicate %s%%" % (
                               interface,widget.get().split(',')
                               [0],widget.get().split(',')[1] ))
191
192                     if combovalue == "loss,corrupt":
193                         os.system("tc qdisc add dev %s root
                               netem loss %s%% corrupt %s%%" % (
                               interface,widget.get().split(',')
                               [0],widget.get().split(',')[1] ))
194
195                     if combovalue == "delay,corrupt":
196                         os.system("tc qdisc add dev %s root
                               netem delay %sms corrupt %s%%" % (
                               interface,widget.get().split(',')
                               [0],widget.get().split(',')[1] ))
197
198                     if combovalue == "loss,delay,corrupt":
199                         os.system("tc qdisc add dev %s root
                               netem loss %s%% delay %sms corrupt %
                               s%%" % (interface,widget.get().split
                               (',')[0],widget.get().split(',')[1]
                               ,widget.get().split(',')[2]))
200 IF = IF()
201 IF.dokument(dom)
202 create()
```

A.2 setup.sh

Es folgt das Shell Skript zur Konfiguration der Hostmaschine für das QEMU Szeanrio und das Starten der virtuellen QEMU Maschinen.

```
1  tunctl -t tap1 -f /dev/net/tun
2  tunctl -t tap2 -f /dev/net/tun
3  tunctl -t tap3 -f /dev/net/tun
4  tunctl -t tap4 -f /dev/net/tun
5  tunctl -t tap5 -f /dev/net/tun
6  tunctl -t tap6 -f /dev/net/tun
7
8  ifconfig tap1 172.168.1.1 netmask 255.255.255.248
9  ifconfig tap3 172.168.1.2 netmask 255.255.255.248
10
11 brctl addbr net1
12 brctl addif net1 tap2
13 brctl addif net1 tap4
14 brctl addif net1 tap5
15 brctl addif net1 tap6
16
17 ifconfig net1 promisc up
18 ifconfig tap1 promisc up
19 ifconfig tap2 promisc up
20 ifconfig tap3 promisc up
21 ifconfig tap4 promisc up
22 ifconfig tap5 promisc up
23 ifconfig tap6 promisc up
24
25 sudo qemu -hda comp1.ovl -append "root=/dev/sda console=
      tty0" -kernel bzImage-2.6.38.2 -daemonize -net nic,model
      =e1000,macaddr=52:54:00:12:53:01,vlan=0 -net tap,vlan=0,
      ifname=tap1,script=no -net nic,model=e1000,macaddr
      =52:54:00:12:53:02,vlan=1 -net tap,vlan=1,ifname=tap2,
      script=no
26 sudo qemu -hda comp2.ovl -append "root=/dev/sda console=
      tty0" -kernel bzImage-2.6.38.2 -daemonize -net nic,model
      =e1000,macaddr=52:54:00:12:53:03,vlan=0 -net tap,vlan=0,
      ifname=tap3,script=no -net nic,model=e1000,macaddr
      =52:54:00:12:53:04,vlan=1 -net tap,vlan=1,ifname=tap4,
      script=no
27 sudo qemu -hda comp3.ovl -append "root=/dev/sda console=
      tty0" -kernel bzImage-2.6.38.2 -daemonize -net nic,model
      =e1000,macaddr=52:54:00:12:53:05,vlan=0 -net tap,vlan=0,
      ifname=tap5,script=no
28 sudo qemu -hda comp4.ovl -append "root=/dev/sda console=
      tty0" -kernel bzImage-2.6.38.2 -daemonize -net nic,model
      =e1000,macaddr=52:54:00:12:53:06,vlan=0 -net tap,vlan=0,
      ifname=tap6,script=no
```

interfaces

Hier ist eine Beispielkonfiguration einer interface Datei einer virtuellen Maschine dargestellt. Die Datei befindet sich in einer virtuellen QEMU Maschine im Verzeichnis /etc/network/

```
1  iface lo inet loopback
2  auto lo
3
4  auto eth0 eth1
5  iface eth0 inet static
6  address 172.168.1.3
7  netmask 255.255.255.248
8
9  iface eth1 inet static
10 address 10.0.0.1
11 netmask 255.255.255.0
```

A.3 plot.sh

Es folgt ein Shell Skript zur Erstellung einer graphischen Ausgabe von gemessenen Daten mit tcp_probe.

```
1 #!/bin/sh
2 gnuplot -persist << EOF
3 show timestamp
4 set xlabel "time (seconds)"
5 set ylabel "Segments (cwnd, ssthresh)"
6 plot "$1" using 1:7 title "snd_cwnd" with lines, \
7      "$1" using 1:8 title "snd_ssthresh" with lines
8 EOF
```

A.4 Szeanrio1

Hierbei handelt es sich um das erste VNUML Szenario.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vnuml SYSTEM "/usr/share/xml/vnuml/vnuml.dtd">

<vnuml>
  <global>
    <version>1.8</version>
    <simulation_name>scenario1</simulation_name>
    <ssh_version>2</ssh_version>
    <ssh_key>/root/.ssh/id_rsa.pub</ssh_key>
    <automac/>
    <vm_mgmt type="private" network="192.168.0.0" mask="24"
        offset="100" >
       <host_mapping/>
    </vm_mgmt>
    <vm_defaults>
       <filesystem type="cow">/usr/local/share/vnuml/
          filesystems/root_fs_tutorial</filesystem>
       <kernel>/usr/local/share/vnuml/kernels/linux</kernel
          >
       <console id="0">xterm</console>
       <forwarding type="ip" />
    </vm_defaults>
  </global>

  <net name="a" mode="virtual_bridge" />
  <net name="b" mode="virtual_bridge" />
  <net name="c" mode="virtual_bridge" />
  <net name="d" mode="virtual_bridge" />
  <net name="e" mode="virtual_bridge" />

  <vm name="R1">
    <if id="1" net="a">
      <ipv4>10.0.1.10</ipv4>
    </if>
    <if id="2" net="b">
      <ipv4>10.0.5.10</ipv4>
    </if>
    <if id="3" net="a">
      <ipv4>10.0.1.1</ipv4>
    </if>
    <route type="ipv4" gw="10.0.1.11">default</route>
  </vm>

  <vm name="R2">
    <if id="1" net="a">
      <ipv4>10.0.1.11</ipv4>
    </if>
    <if id="2" net="c">
      <ipv4>10.0.8.10</ipv4>
    </if>
    <if id="3" net="d">
```

```
49        <ipv4>10.0.9.10</ipv4>
50      </if>
51      <route type="ipv4" gw="10.0.8.11">default</route>
52    </vm>
53
54    <vm name="R3">
55      <if id="1" net="b">
56        <ipv4>10.0.5.11</ipv4>
57      </if>
58      <if id="2" net="c">
59        <ipv4>10.0.8.11</ipv4>
60      </if>
61      <if id="3" net="e">
62        <ipv4>10.0.10.11</ipv4>
63      </if>
64      <if id="4" net="b">
65        <ipv4>10.0.5.4</ipv4>
66      </if>
67     <route type="ipv4" gw="10.0.5.10">default</route>
68    </vm>
69
70    <vm name="R4">
71      <if id="1" net="d">
72        <ipv4>10.0.9.11</ipv4>
73      </if>
74      <if id="2" net="e">
75        <ipv4>10.0.10.10</ipv4>
76      </if>
77      <route type="ipv4" gw="10.0.10.10">default</route>
78    </vm>
79
80  </vnuml>
```

A.5 Readme OLSR

Hierbei handelt es sich um den Ausschnitt der Installationsanleitung von OLSR für Linux.

```
=========
 * LINUX
=========

To build olsrd you need to have all the regular development tools
installed. This includes gcc, make, glibc, makedep etc.
To install to a directory different from /(/etc, /usr/bin) use
DESTDIR=targetdir. To use other compilers set CC=yourcompiler.

To build:
 make
To install(as root):
 make install
To delete object files run:
 make clean
Optionally, to clean all generated files:
 make uberclean

Before running olsrd you must edit the default configuration file
/etc/olsrd.conf adding at least what interfaces olsrd is to run on.
Options in the config file can also be overridden by command line
options. See the manual pages olsrd(8) and olsrd.conf(5) for details.
The binary is named 'olsrd' and is installed in (PREFIX)/usr/sbin.
You must have root privileges to run olsrd!
To run olsrd just type:
olsrd

If debug level is set to 0 olsrd will detach and run in the background,
if not it will keep running in your shell.
```

A.6 olsrd.conf

Hierbei handelt es sich um die Konfigurationsdatei für OLSR.

```
# OLSR.org routing daemon config file
# This file contains the usual options for an ETX based
# stationary network without fisheye
# (for other options see olsrd.conf.default.full)

DebugLevel  2

IpVersion 4

LinkQualityAlgorithm    "etx_ff"

LinkQualityFishEye  0

###################################
### Example plugin configurations ###
###################################
LoadPlugin "olsrd_txtinfo.so.0.1"
{
    PlParam     "Accept"   "127.0.0.1"
}

###############################################
### OLSRD default interface configuration ###
###############################################

InterfaceDefaults {
    Mode "ether"
    HelloInterval 2.0
    HelloValidityTime 6.0
}
```

Literaturverzeichnis

[1] L. Balliache. Differentiated service on linux howto - linux traffic control. http://opalsoft.net/qos/DS-21.htm, 2003.

[2] F. Bellard. Qemu manual. http://wiki.qemu.org/Manual. 19.3.2012.

[3] J. Chroboczek. Babel - a routing protocol for sparse networks. http://www.pps.jussieu.fr/~jch/software/babel/babel-funkfeuer.pdf, 2008.

[4] J. Chroboczek. *The Babel Routing Protocol*. http://www.rfc-editor.org/rfc/rfc6126.txt, April 2011.

[5] E. Grgori and M. Conti. *Networking 2002*. Springer, 2002.

[6] Linux Foundation. tcp testing. http://www.linuxfoundation.org/collaborate/workgroups/networking/tcptesting, 2009. 19.3.2012.

[7] B. Hubert. Linux advanced routing and traffic control howto. Technical report, Netherlabs BV, http://lartc.org/lartc.pdf, 2003. 19.3.2012.

[8] C. Israel. Netzwerkvirtualisierung mit qemu und vde-switches. Master's thesis, Universität Koblenz-Landau, 2011.

[9] A. Keller. tc packet filtering and netem. Technical report, Eidgenössische Technische Hochschule Zürich, Institut für Technische Informatik und Kommunikationsnetze, tcn.hypert.net/tcmanual.pdf, 2006. 19.3.2012.

[10] A. Keller. Trace control for netem. Technical report, Eidgenössische Technische Hochschule Zürich, Institut für Technische Informatik und Kommunikationsnetze, tcn.hypert.net/tcn.pdf, 2006. 19.3.2012.

[11] M. Monreal. Simulation mit vnuml. Technical report, Universität Koblenz-Landau, August 2007.

[12] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. Technical report, ACM SIGCOMM'94 Conference on Communications Architectures, 1994.

[13] C. Steigner. Grundlagen der rechnernetze foliensatz 17. https://userpages.uni-koblenz.de/~steigner/GdRN/GdRN-17.ppt, 2011. 19.3.2012.

[14] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[15] K. Kurata und G. Hasegawa und M. Murata. Fairness comparisons between tcp reno and tcp vegas for future deployment of tcp vegas. Technical report, INET Society Japan, http://www.isoc.org/inet2000/cdproceedings/2d/, 2000. 19.3.2012.

[16] T. Clausen und P. Jacquet. *Optimized Link State Routing Protocol*. http://www.rfc-editor.org/rfc/rfc3626.txt, Oktober 2003. 19.3.2012.

[17] A. Volk und T. Keupen. *Anleitung zur Installation des Netzwerk-Simulators VNUML*. Universität Koblenz-Landau, 2005.

[18] M. Allman und V. Paxson. *TCP Congestion Control*. http://www.rfc-editor.org/rfc/rfc5681.txt, September 2009.