FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Alternative Transportprotokolle im Einsatz: Laufzeittests mit dem Visualisation Interface Toolkit VISIT

Franz Petri, Thomas Eickermann

FZJ-ZAM-IB-2006-16

September 2006

(letzte Änderung: 14.09.2006)

Bericht erstellt für das D-Grid Integrationsprojekt (DGI)

D-Grid Integrationsprojekt (DGI)

Fachgebiet 3-3 – Alternative Transportprotokolle

Alternative Transportprotokolle im Einsatz: Laufzeittests mit dem Visualisation Interface Toolkit VISIT

Version 1.1, September 2006

Autoren

Franz Petri (ZAM, Forschungszentrum Jülich)

Thomas Eickermann (ZAM, Forschungszentrum Jülich)

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen 01AK800B gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

Inhaltsverzeichnis

1 Einleitung

Techniken des Grid Computing machen sich die Infrastruktur des Internet zu Nutze, um Daten global zur Speicherung und weiteren Verarbeitung zu verteilen. Heutige Forschungsprojekte sind aufgrund der zu übertragenden Datenmengen auf hohe Übertragungsraten angewiesen. So liegt z.B. das ab 2007 vom LHC des CERN erzeugte Datenvolumen bei jährlich ca. 15 Petabyte [1]. Um solche Datenmengen über das Internet übertragen zu können, kommen Netzwerke mit Bandbreiten von 10 Gbit/s und mehr zum Einsatz.

Das im Internet verwendete Transportprotokol TCP ist in seiner zurzeit überwiegend verwendeten Implementierung (TCP-Reno) kaum in der Lage, solche Übertragungsraten mit wenigen oder sogar nur einem einzigen Datenstream zwischen einem Server und einem Client zu erreichen.

In globalen Grids macht sich nämlich bemerkbar, dass Paketlaufzeiten nach unten durch die Lichtgeschwindigkeit begrenzt sind. Ähnlich wie bei einer Wasserleitung lässt sich für eine Datenleitung berechnen, welche Menge an Daten die Leitung auf einmal aufnehmen kann, wenn man weiß, wie lange die Daten für ihren Weg vom Anfang zum Ende der Leitung benötigen und wieviel Datenpakete pro Sekunde den „Querschnitt“ der Leitung passieren. Die *Kapazität* (auch *Bandwidth Delay Product – BDP* genannt) einer Leitung ergibt sich also durch:

$$BDP := Bandwith * RTT^{1}$$

Zwei Aspekte der Standardimplementierung von TCP bereiten auf Leitungen mit großer Kapazität (auch *Long Fat Pipes* genannt) Probleme:

- Zu geringe Puffergrößen: Da TCP verloren gegangene Datenpakete erkennen und nochmals übertragen können muss, werden die Daten so lange in Puffern vorgehalten, bis deren korrekte Übertragung gesichert ist. Um den kompletten Inhalt einer „voll gefüllten“ Leitung mit einem „Querschnitt“ von 10 Gbit/s und einer „Länge“ von 100 ms aufnehmen und puffern zu können, braucht ein solcher Puffer immerhin die Größe von 10 Gbit/s * 0,1 s = 1Gbit = 125 Mbyte (Vgl. hierzu: Standardgröße für TCP Socket-Sendbuffer, z.B. unter Suse 10.1: 16 Kbyte).
- Zu konservative Reaktion auf Paketverlust: TCP interpretiert Paketverlust als Datenstau auf dem Übertragungsweg und reduziert deshalb bei Paketverlust die Sendeleistung drastisch. Danach versucht Standard-TCP durch vorsichtiges „Herantasten“ an die ursprüngliche Übertragungsrate die Sendeleistung wiederherzustellen. „So würde es beispielsweise bei einer Übertragungsrate von 10 GBit/s und einer round-trip time von 100 ms nach einem Paketverlust länger als eine Stunde dauern, bis die maximale Übertragungsrate wieder erreicht

1 RTT: Paketlaufzeit in beide Richtungen; i.d.R werden Daten gleichzeitig gesendet und empfangen (Duplex).

wird. Da davon auszugehen ist, dass auch in nicht überlasteten Netzen Paketverluste mit einer relativen Häufigkeit von 10^{-7} auftreten, kann die maximale Bandbreite nicht ausgenutzt werden.“ [2]

Der erste Punkt ist ohne großen Aufwand – sofern man auf den eingesetzen Systemen die dafür notwendigen Rechte besitzt - durch Erhöhung der Puffergrößen in den Griff zu bekommen. Für die Lösung des zweiten Problems bieten sich u.a. folgende Möglichkeiten an:

- Standard-TCP wird so modifiziert, dass es nach Paketverlust eine aggressivere Strategie zur Wiederherstellung der Sendeleistung verfolgt. Die Schnittstelle zu den Applikationen bleibt gleich, es sind also keine Veränderungen auf Applikationsebene notwendig. Problematisch ist hingegen, dass Eingriffe am Betriebssystem notwendig wären.
- Die Daten werden über das vorhandene Protokoll UDP versandt. Eine im Vergleich zu Standard-TCP aggressivere Strategie zur Wiederherstellung der Sendeleistung wird auf Benutzer- bzw. Applikationsebene entwickelt, ebenso die für einen zuverlässigen Datentransport notwendige Erkennung von Paketverlusten. Vorhandene Applikationen müssten angepasst werden, aber der Kernel bliebe unverändert.

Bei der Evaluierung vorhandener TCP- und UDP-basierter Alternativprotokolle ist hierbei neben Messungen zur reinen Übertragungsleistung auch eine Betrachtung folgender Aspekte notwendig, auf die hier eingegangen werden soll:

- Nachrichtenlaufzeiten: wie lange benötigt ein Protokoll, um Nachrichten verschiedener Größe an eine Applikation zu senden und diese nach Verarbeitung an den Sender zurück zu liefern.
- Implementier-, Portier- und Handhabbarkeit: Wie groß ist der Aufwand, ein neues Protokoll in eine bestehende Umgebung einzufügen, wie zuverlässig funktioniert die Kommunikation über ein neues Protokoll über Systemarchitekturgrenzen hinweg, und wie einfach ist die Handhabung eines neuen Protokolls oder einer API.

Um diese Aspekte zu untersuchen, wurde das UDP-basierte Protokoll UDT, entwicket vom National Center for Data Mining der University of Illinois Chicago [3], auf AIX und Solaris portiert sowie in das Visualisierungsinterface-Toolkit VISIT, entwickelt vom Forschungszentrum Jülich [4], integriert. Eine UDT-Version der in der VISIT-Suite enthaltenen Pingpong-Applikationen wurde erstellt. Mit Hilfe von TCP-, UDT und VISIT-/VTS-Pingpongs (`tcppp, udtpp, ccudtpp, visitpp, vtspp`) wurden Messungen zur Nachrichtenlaufzeit bei unterschiedlicher Nachrichtengröße durchgeführt, die hier vorgestellt werden sollen.

Ein Ziel des Projektes ist es, eine einheitliche, standardisierte Programmierschnittstelle (API) für die Einbindung der alternativen Transportprotokolle in Anwendungen bereit zu stellen. Ein solches

API wird zzt. in der GGF/OGF Research Group SAGA (Simple API for Grid Applications) spezifiziert. Das FZJ hat hierzu Use-Cases aus dem Bereich der Online-Visualisierung beigesteuert, und so dazu beigetragen, dass ein „Stream“ API für TCP-ähnliche Netzwerk-Verbindungen in das SAGA „Strawman API“ aufgenommen wurde [4].

Da das SAGA API sich noch in einem relativ frühen Entwicklungsstadium befindet, haben wir UDT zunächst in das VISIT-interne API VTS (VISIT Transport Schnittstelle) integriert. VTS wird von VISIT genutzt, um auf einheitliche Weise auf unterschiedliche Transportmechanismen zugreifen zu können. Auf diese Weise können wir bereits jetzt Erfahrungen mit UDT in realen Anwendungen sammeln. Eine Portierung auf SAGA oder ein anderes API sollte später mit geringem Aufwand möglich sein.

2 Alternative Transportprotokolle im Einsatz

2.1 Das Visualisierungs-Interface Toolkit VISIT

Bedingt durch die Leistungssteigerungen sowohl bei Supercomputern als auch bei den Übertragungswegen und den angeschlossenen Workstations hat sich der Bedarf für eine neue Arbeitsweise mit Simulationen auf Supercomputern entwickelt, weg vom text- und terminalbasierten Monitoring, hin zu mehr visueller Interaktivität. Stichworte sind hierbei „Online-Visualisation“, „Interactive Simulation“ oder „Computational Steering“. Das Forschungszentrum Jülich hat ein Toolkit names VISIT („VISualisation Interface Toolkit“) entwickelt, das als Schnittstelle zwischen Simulation und Visualisierung u.a. folgenden Ansprüchen an ein solches Werkzeug genügen soll:

- Übername des Verbindungsauf- und Abbaus zwischen Simulation und Visualisierung.
- Bevorzugung der Simulation (CPU-Zeit auf Supercomputern ist kostenintensiv): alle Operationen werden von der Simulation angestoßen und innerhalb festgelegter Zeit garantiert abgearbeitet oder verworfen. Verzögerungen oder Absturz der Visualisierung beeinflussen die Simulation nicht.
- Möglichkeit zur Übertragung einfacher Datentypen wie Strings, Integers und Floats sowie Integer- und Floatarrays.
- Byte-Order-Konvertierung.
- Einsatz von „Service-Names“ anstatt Host-Name und Port-Nummer zum Verbindungsaufbau.
- Offenheit der Programmierschnittstelle für den Einsatz unterschiedlicher Mechanismen zum Nachrichtenaustausch (z.B. MPI oder Myrinet) durch ein internes API (VTS) mit Socket-ähnlicher Semantik.

Insbesondere die Ausnutzung des letztgenannten Punktes erleichterte die Integration des UDP-basierten alternativen Transportprotokolles UDT in die VISIT-Suite.

2.2 Die VISIT Transport Programmierschnittstelle VTS

VTS stellt ein C-API mit einer Socket-ähnlichen Semantik zur Verfügung. Obwohl die Anforderungen der VISIT-Bibliothek ausschlaggebend für die Funktionalität von VTS waren, ist das API auch allgemeiner einsetzbar. VTS unterstützt derzeit die Protokolle TCP, FiFo (named pipes), Stdio, File-I/O, einen „Tunnel“, um eine beliebige Anzahl von VTS-Verbindungen durch eine Verbindung eines anderen Protokolls zu tunneln (z.B. durch `ssh`). Zusätzlich wurde jetzt eine UDT-Unterstützung implementiert. Jedes Protokoll ist in einem sog. Device implementiert. Das VTS API

stellt die folgenden Funktionen bereit:

VTS unterscheidet die Client- und Server-Seite einer Verbindung. Die beiden Funktionen zum Starten eines Clients bzw. Servers sind die einzigen, die vom verwendeten Device abhängen.

```
int vts_client(int device, …);
int vts_server(int device, …);
```

Typische Aufrufe für TCP und UDT sind:

```
int fd = vts_client(VTS_RAWTCP, char *host, int port, int flag,
                    int timeout, char *text);
int fd = vts_client(VTS_RAWUDT, char *host, int port, int flag,
                    int timeout, char *text);

int fd = vts_server(VTS_RAWTCP, int port, int flag, char *text);
nt fd = vts_server(VTS_RAWUDT, int port, int flag, char *text);
```

`vts_client` baut eine Verbindung zum Server auf, `vts_server` erzeugt einen Server, der auf Client-Verbindungen „horcht“ (im TCP-Fall im Zustand `listen`). Der Grund für die Device-Namen „`VTS_RAWTCP`“ und „`VTS_RAWUDT`“ ist, dass es weitere Varianten „`VTS_TCP`“ und „`VTS_UDT`“ gibt, die über zusätzliche Funktionalität, wie Passwort-Authentifizierung beim Verbindungsaufbau sowie sog. kollektive Verbindungen verfügen. Bei kollektiven Verbindungen werden mehrere Client-Verbindungen vom Server als eine einzige Verbindung betrachtet, bei der die Verbindungspartner durch einen „rank“ unterschieden werden.

Jede VTS Funktion, die potentiell blockieren kann, hat eine timeout-Parameter, mit dem die Maximalzeit in Millisekunden angegeben wird, nach der die Funktion spätestens zurückkehrt (`-1` für blockierend). Der String-Parameter `text` gibt einen Text an, der eventuellen Fehlermeldungen der Funktion vorangestellt wird.

```
int vts_accept(int fd, int timeout, char *text);
```

wird von einem Server genutzt, um eine Verbindung anzunehmen.

```
int vts_close(int fd, char *text);
```

baut eine Verbindung ab.

```
int vts_send(int fd, int rank, void *data, int size,
             int timeout, char *text);
int vts_recv(int fd, int *rank, void *data, int size,
             int timeout, char *text);
```

versucht, size Byte Daten zu senden bzw. zu empfangen. Die Funktionen kehren zurück, wenn entweder alle Daten geschrieben/gelesen wurden, oder `timeout` abgelaufen ist. Achtung: dies unterscheidet sich von der Socket-Semantik. Der Parameter `rank` ist nur für kollektive Server relevant, um den Client zu spezifizieren.

```
int vts_select(int n, vts_fd_set *rset, vts_fd_set *west,
               vts_fd_set *eset, int *rank, int timeout,
               char *text);
```

entspricht dem `select` des Socket-API.

Darüber hinaus gibt es weitere Funkionen, die hier nicht relevant sind. Eine detaillierte Beschreibung des API befindet sich im VISIT-Manual.

2.3 Aufbau einer Messumgebung

Folgende Vorgaben galt es bei der konkreten Einrichtung der Messumgebung zu erfüllen:

- Es soll möglich sein, lokal Paketverlust und Latenz zu simulieren, ohne dabei die eigentliche Messung durch eine hohe CPU-Last, hervorgerufen durch die Simulation, zu verfälschen.
- Der Einsatz von 10 Gbit/s-Netzwerktechnik soll nicht durch das zugrunde liegende Bussystem beschränkt werden.
- Das Betriebssystem soll TCP-Varianten flexibel einsetzen können.

2.3.1 Hardware

Es wurden drei Server beschafft, von denen jeweils einer als Datensender und einer als Datenempfänger fungiert. Der dritte Rechner sorgt als zwischengeschaltetes Gateway transparent für die

Erzeugung von Paketverlust und Latenz. Sender und Empfänger sind direkt (back-to-back) mit dem Gateway verbunden.

Jeder Server ist ausgestattet mit:

- zwei AMD Dual-Core Opterons 265, 1,8 GHz, 2 MB Cache.[6]
- 4x 512 MB DDR RAM ECC-R.
- Tyan Thunder K8WE (S2895) Motherboard mit NVIDIA nForce Professional-Chipsatz und zwei PCI-Express (16x) Steckplätzen.[7]

Das Ethernet-Testsystem soll im Endausbau bestehen aus:

- 1 Gbit/s Ethernet (on Board)
- 10 Gbit/s Ethernet: zwei Myriom 10G-PCIE-8A-R PCI-Express (8x) Netzwerkkarten [8];
  zwei Neterion XFrame E 10 GbE PCI-Express (4x) Netzwerkkarten [9].

Zum Zeitpunkt der Erstellung des Berichts war das Paar Myricom-Karten bestellt aber noch nicht geliefert. Dies beschränkte die 10 Gbit/s-Messungen auf die Karten der Firma Neterion.

2.3.2 Betriebssystem

Als Betriebssystem kommt auf allen drei Rechnern Suse Linux in der Version 10.1 zum Einsatz. Zum Zeitpunkt der Entstehung dieses Berichts bietet diese Linuxdistribution die aktuellste Kernelversion: 2.6.16. Diese Kernelversion bietet die Möglichkeit, sämtliche zu testenden TCP-Varianten mit Hilfe des Befehls `modprobe` als Kernelmodule während der Laufzeit zu laden und auszutauschen, ohne den Kernel patchen oder neu übersetzen zu müssen, was die Reproduzierbarkeit der Ergebnisse erhöht. Als Treiber für die eingesetzte Hardware werden ausschließlich die mit diesem Kernel mitgelieferten Treiber benutzt (Ausnahme `s2io` Version 2.0.14.5152 für Neterion Xframe E).

2.3.3 Anmerkungen zum Netzwerkemulator netem:

Um auf dem zwischengeschalteten Gateway Paketverlust und Latenz zu simulieren, kam der Netzwerkemulator netem zum Einsatz [10]. Dieser ist standardmäßig in den Kernel 2.6.16 integriert und verzögert und/oder verwirft auf der Netzwerkschicht transparent für die darüberliegenden Protokolle ausgehende Pakete. netem wird aktiviert mit:

```
tc qdisc add dev <dev> root netem delay <delay> limit <queue-length>
```

z.B.

```
tc qdisc add dev eth0 root netem delay 100ms limit 10000
```

Bei aktiviertem netem war der Datendurchsatz bei den Testläufen auf ca. 800 Mbit/s (bei 1 Gbit/s theoretischem Limit), bzw. 1 Gbit/s (bei 10 Gbit/s theoretischem Limit) selbst bei voll optimierten Puffergrößen und Queuelängen begrenzt. Da zum Zeitpunkt der Berichterstellung netem als Faktor für diese Begrenzung nicht ausgeschlossen werden konnte, wurde fürs Erste auf die Simulation von Paketverlust und Latenz mit netem zugunsten belastbarer Ergebnisse verzichtet.

2.3.4 Anmerkungen zu forcedeth

forcedeth ist ein „reverse engineered"-Treiber, der die beiden von den Nvidia nforce-Chipsätzen kontrollierten 1 Gbit/s-Onboard-Ethernetcontroller ansteuert. Unter der zum Zeitpunkt der Berichterstellung standardmäßig im Kernel aktivierten Version 0.49 tauchten während der Testläufe sporadisch unter hoher Last folgende Fehlermeldungen auf (Ausgabe mit `dmesg`):

```
eth0: too many iterations (6) in nv_nic_irq
```

Der Fehler korrespondierte teilweise mit auftretenden Paketverlusten während der Messungen. Eine aktuellere Version (0.56) des Treibers brachte keine Veränderung. Abhilfe schaffte das Laden von forcedeth mit folgender Option:

```
modprobe forcedeth optimization_mode=1,1
```

welche die standardmäßig deaktivierte Interrupt Coalescing-Fähigkeiten des Treibers aktiviert.

2.3.5 Systemeinstellungen

Um die in der Einleitung erwähnten Beschränkungen des Durchsatzes durch zu geringe Puffergrößen zu umgehen, wurden entsprechende Betriebssystemparameter gemäß den Empfehlung [11] optimiert.

2.3.5.1 Puffer-Größen

Die Puffergrößen wurden entsprechend des jeweiligen Bandwith-Delay-Products folgendermaßen angepasst:

- Für Tests mit 1 Gbit/s (BDP ca. 125 KByte):

```
echo 113664 > /proc/sys/net/core/rmem_default
echo 131071 > /proc/sys/net/core/rmem_max
echo 113664 > /proc/sys/net/core/wmem_default
echo 131071 > /proc/sys/net/core/wmem_max
echo 4096 87380 174760 > /proc/sys/net/ipv4/tcp_rmem
echo 4096 16384 131072 > /proc/sys/net/ipv4/tcp_wmem
```

Dies entspricht Suse 10.1-Standardeinstellungen.

- Für Tests mit 10 Gbit/s (BDP ca. 1,25 Mbyte):

```
echo 113664 > /proc/sys/net/core/rmem_default
echo 2097152 > /proc/sys/net/core/rmem_max
echo 113664 > /proc/sys/net/core/wmem_default
echo 2097152 > /proc/sys/net/core/wmem_max
echo 4096 2097152 2097152 > /proc/sys/net/ipv4/tcp_rmem
echo 4096 16384 2097152 > /proc/sys/net/ipv4/tcp_wmem
```

2.3.5.2 Queue-Längen

Auch die tx- und rx-Queue-Längen wurden angepasst:

- Für Tests mit 1 Gbit/s wurden wieder die Betriebssystemstandards übernommen:

```
echo 1000 > /proc/sys/net/core/netdev_max_backlog
ifconfig eth2 txqueuelen 1000
```

- Für Tests mit 10 Gbit/s (BDP ca. 1,25 Mbyte):

```
echo 10000 > /proc/sys/net/core/netdev_max_backlog
ifconfig eth2 txqueuelen 10000
```

2.4 Durchführung und Ergebnis der Messungen

Für die Messungen kamen die Pingpong-Utilities, die in der VISIT-Suite enthalten sind, zum Einsatz. Vereinfachend gesprochen allozieren diese nach Programmstart einmal als Client- und einmal als Serverapplikation Arbeitsspeicher für eine Nachricht, deren Größe vom Benutzer festgelegt wird. Der Client sendet sodann die Nachricht (Ping) und wartet auf die Antwort (Pong) vom Server, der die gesamte Nachricht nach erfolgreichem Empfang zurücksendet.

Nach einer festen Anzahl von „Aufwärmläufen“ solcher PingPongs, die dafür sorgen, dass alle Puffer und Queues gefüllt sind und das Congestion Window offen ist, beginnt die eigentliche Zeitmessung. Hierfür wird auf der Clientseite die Differenz zwischen dem Zeitpunkt des ersten `send()`-Befehls zum Senden der Nachricht und dem Zeitpunkt der Beendigung des letzten erfolgreichen `recv()`-Befehls zum Empfang des Nachrichtenechos gemessen und ausgegeben. Das Messergebnis wird durch den Faktor 0.5 korrigiert und entspricht so der einfachen durchschnittlichen Laufzeit der Nachricht. Die Genauigkeit der Messung liegt im Bereich von Nanosekunden.

Dank der Integration von UDT in VISIT konnten VISIT-/VTS-Pingpongs sowohl mit TCP als auch mit UDT durchgeführt werden.

2.4.1 TCP

Mit folgenden TCP-Varianten wurden Messungen durchgeführt:

`TCP RENO, TCP BIC, TCP CUBIC, TCP HIGHSPEED, TCP HTCP, TCP HYBLA, TCP SCALABLE, TCP VEGAS, TCP WESTWOOD`.

Die Varianten werden aktiviert mit Hilfe des Befehls

```
modprobe <tcp_variante>
```

wobei `<tcp_variante>` durch `tcp_bic, tcp_cubic, tcp_highspeed, tcp_htcp, tcp_hybla, tcp_scalable, tcp_vegas` oder `tcp_westwood` zu ersetzen ist.

Der Pingpongserver wird gestartet mit

```
./tcppp –S
```

Der Pingpongclient wird gestartet mit:

```
./tcppp –F pp.ini <serveradresse>
```

2.4.1.1 Messergebnisse 1 Gbit/s:

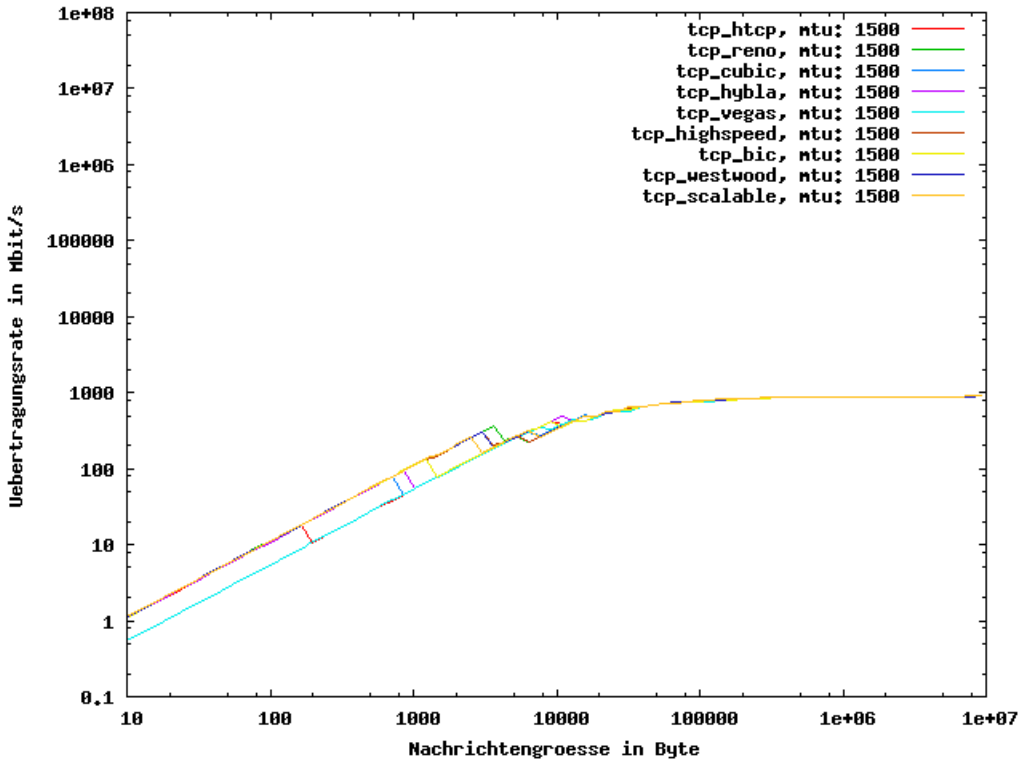

Abbildung 1: TCP-Varianten bei 1Gbit/s und MTU 1500

2.4.1.2 Messergebnisse 10 Gbit/s:

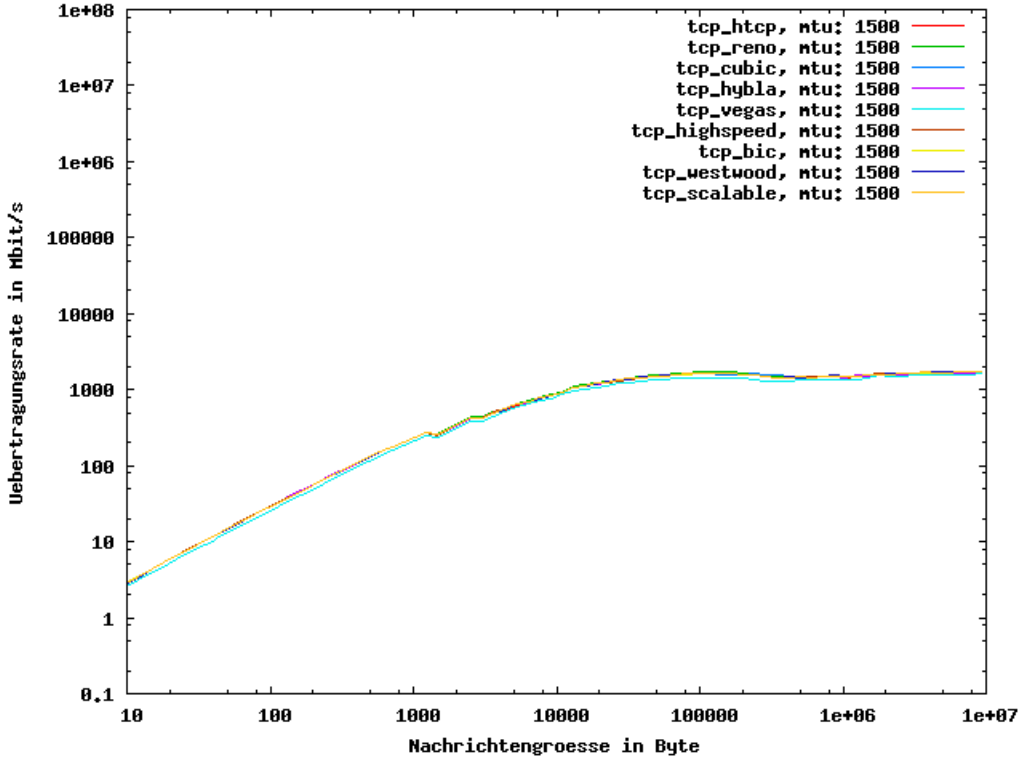

Abbildung 2: TCP-Varianten bei 10 Gbit/s und MTU 1500

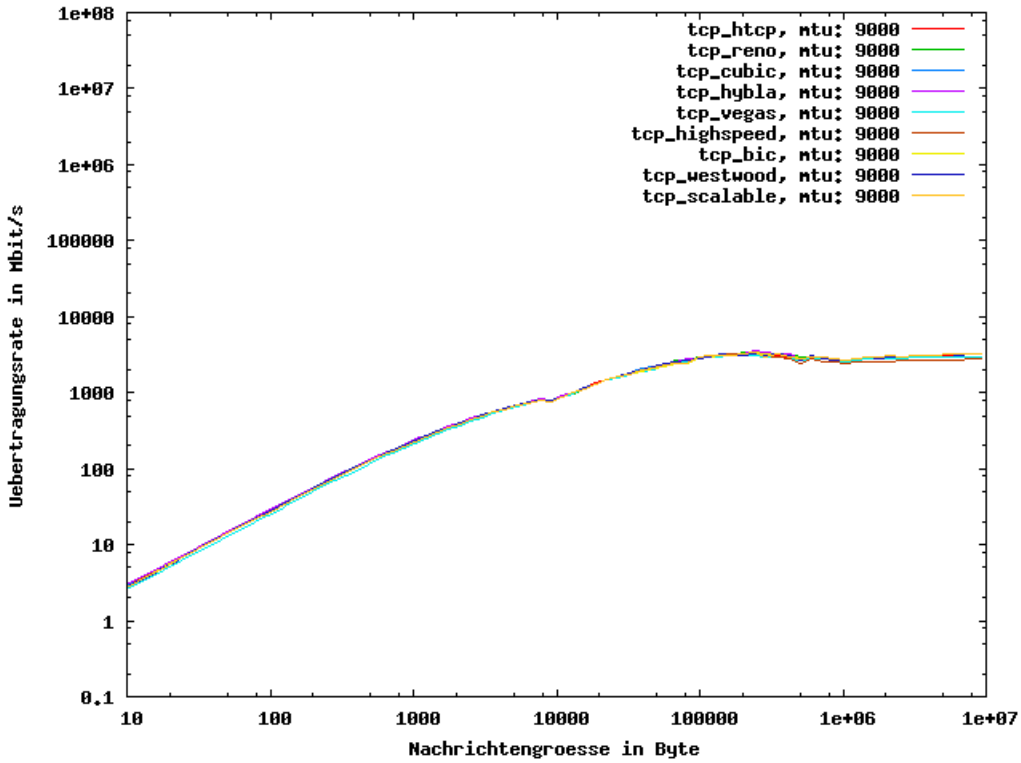

Abbildung 3: TCP-Varianten bei 10 Gbit/s und MTU 9000 (Jumbo Frames)

2.4.2 UDT

Wird UDT ohne weitere Veränderung übersetzt mit

```
make -e arch=AMD64
```

so handelt es sich bei der eingebauten Congestion Control um einen für UDT neu entwickelten Al-

gorithmus.

Der Pingpongserver wird gestartet mit

```
./udtpp –S
```

Der Pingpongclient wird gestartet mit

```
./udtpp –F pp.ini <serveradresse>
```

Unkommentiert man vor Ausführung von `make` in `src/Makefile` folgenden Eintrag

```
#CCFLAGS += -DCUSTOM_CC
```

so kann man mit Hilfe von UDT im Userspace - über UDP - Algorithmen zur Congestion Control aus folgenden TCP-Varianten nutzen:

`TCP RENO, TCP BIC, TCP HIGHSPEED, TCP SCALABLE, TCP VEGAS, TCP WESTWOOD, TCP FAST.`

Die jeweilige Congestion Control ist in UDT über eine Klasse `CCC` (Custom Congestion Control) realisiert. In der UDT-Pingpong-Implementierung mit Custom Congestion Controll (`ccudtpp`) wird die jeweilige Variante über einen Kommandozeilenparameter aktiviert:

Start des Pingpongservers mit

```
./ccudtpp –S –P <congestioncontrol>
```

Start des Pingpongclients mit

```
./ccudtpp –F pp.ini <serveradresse>
```

2.4.2.1 Messergebnisse 1 Gbit/s:

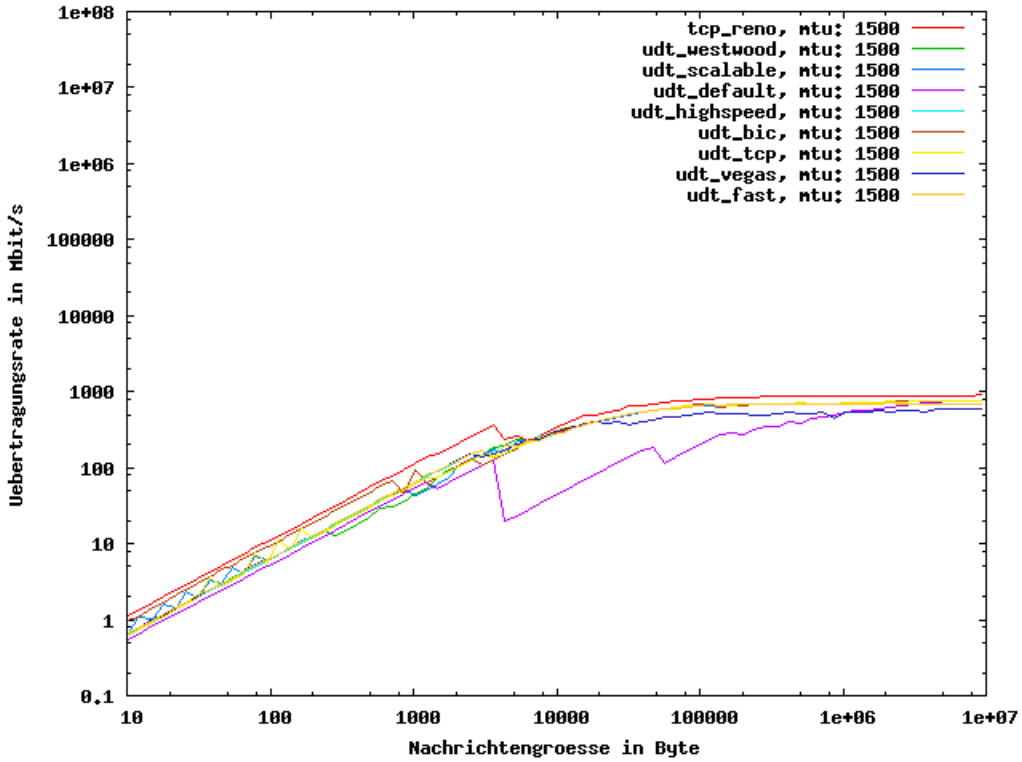

Abbildung 4: UDT mit verschiedenen Congestion-Control-Varianten bei 1 Gbit/s und MTU 1500

2.4.2.2 Messergebnisse 10 Gbit/s:

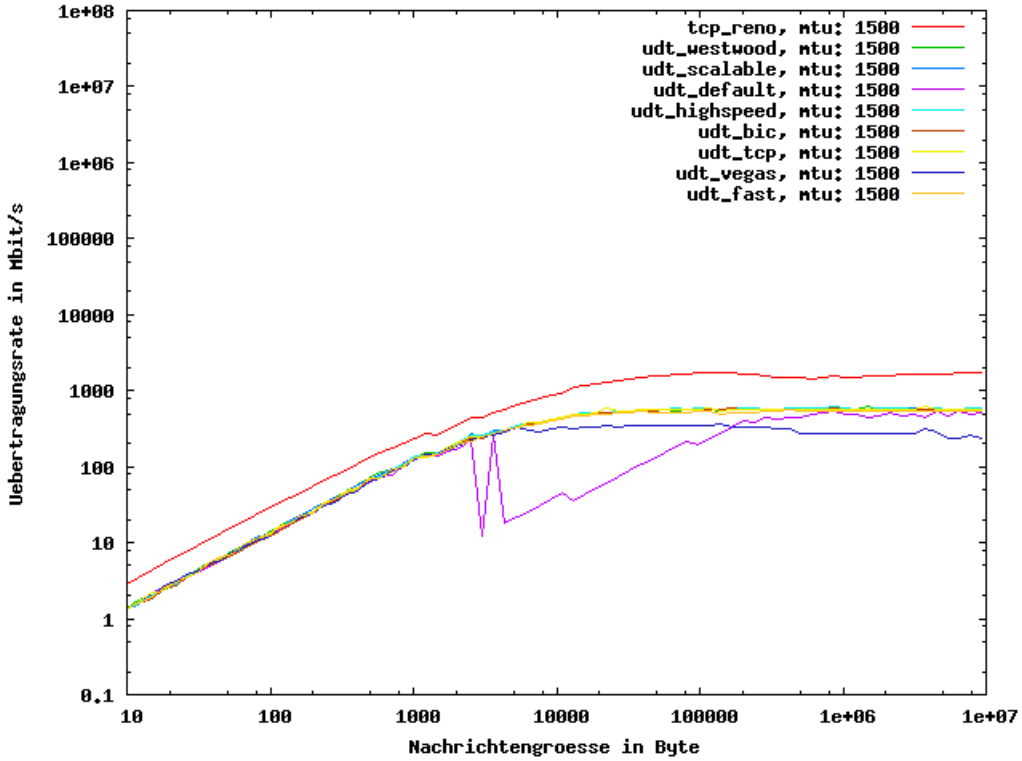

Abbildung 5: UDT mit verschiedenen Congestion-Control-Varianten bei 10 Gbit/s und MTU 1500

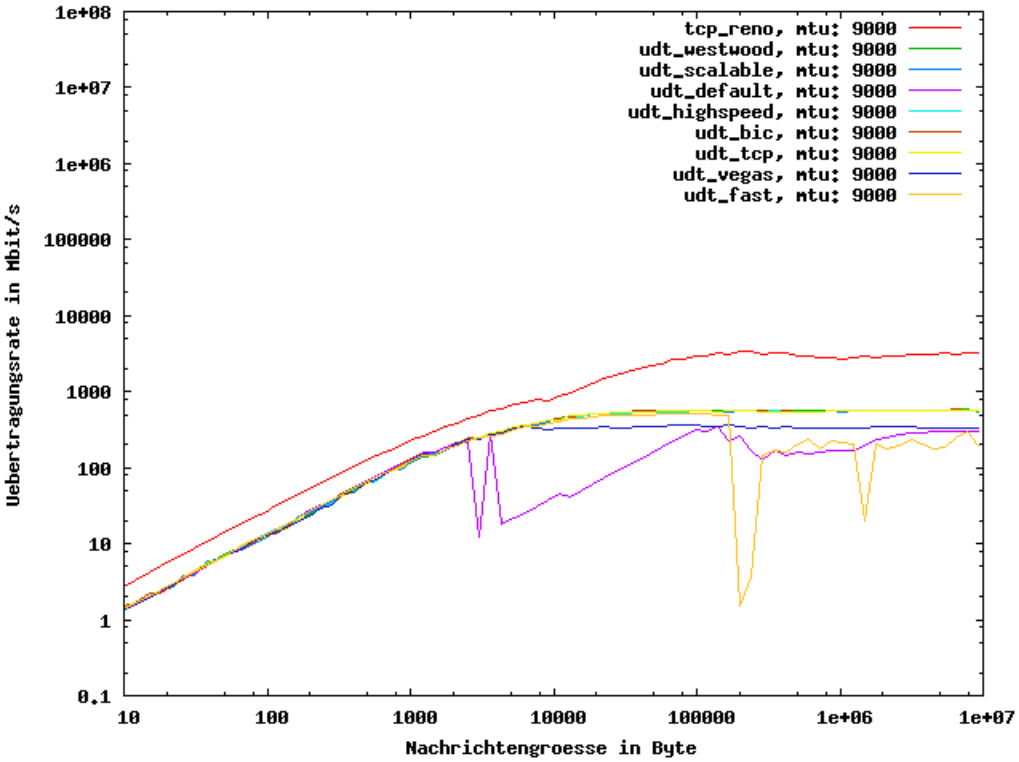

Abbildung 6: UDT mit verschiedenen Congestion-Control-Varianten bei 10 Gbit/s und MTU 9000

2.4.3 VISIT mit TCP

Der VISIT-Pingpong-Server wird gestartet mit

```
./visitpp -S -a 192.168.168.1
```

Der VISIT-Pingpongclient wird gestartet mit

```
./visitpp –F pp.ini
```

2.4.3.1 Messergebnisse 1 Gbit/s:

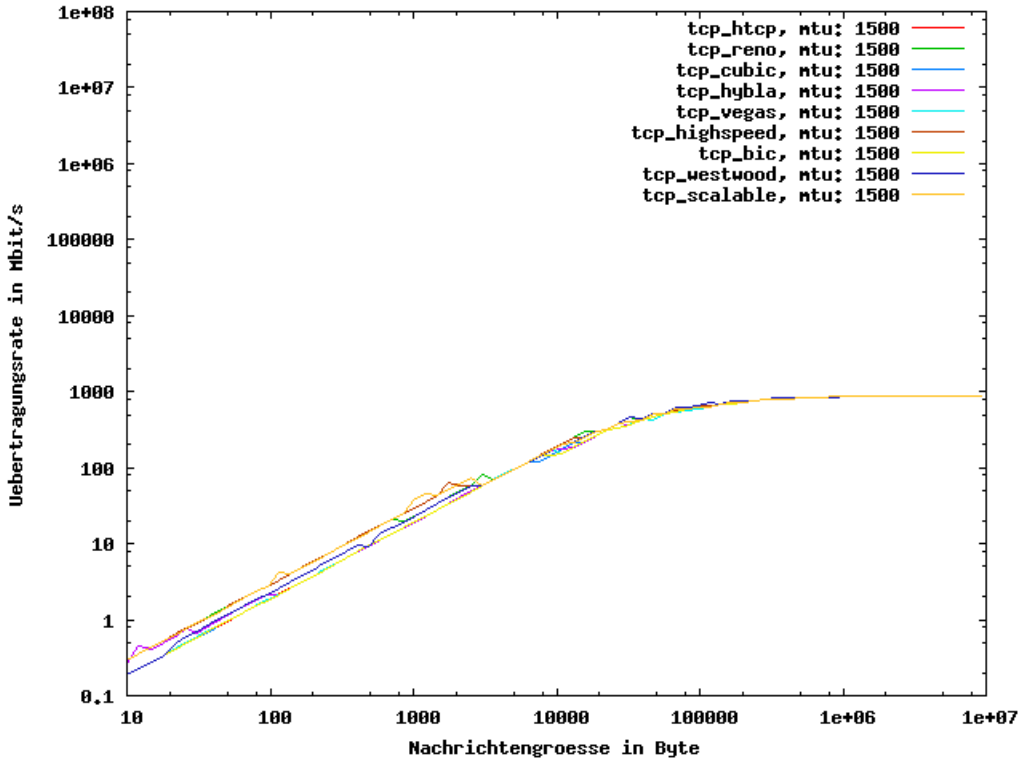

Abbildung 7: VISIT-Pingpongs mit TCP bei 1 Gbit/s und MTU 1500

2.4.3.2 Messergebnisse 10 Gbit/s:

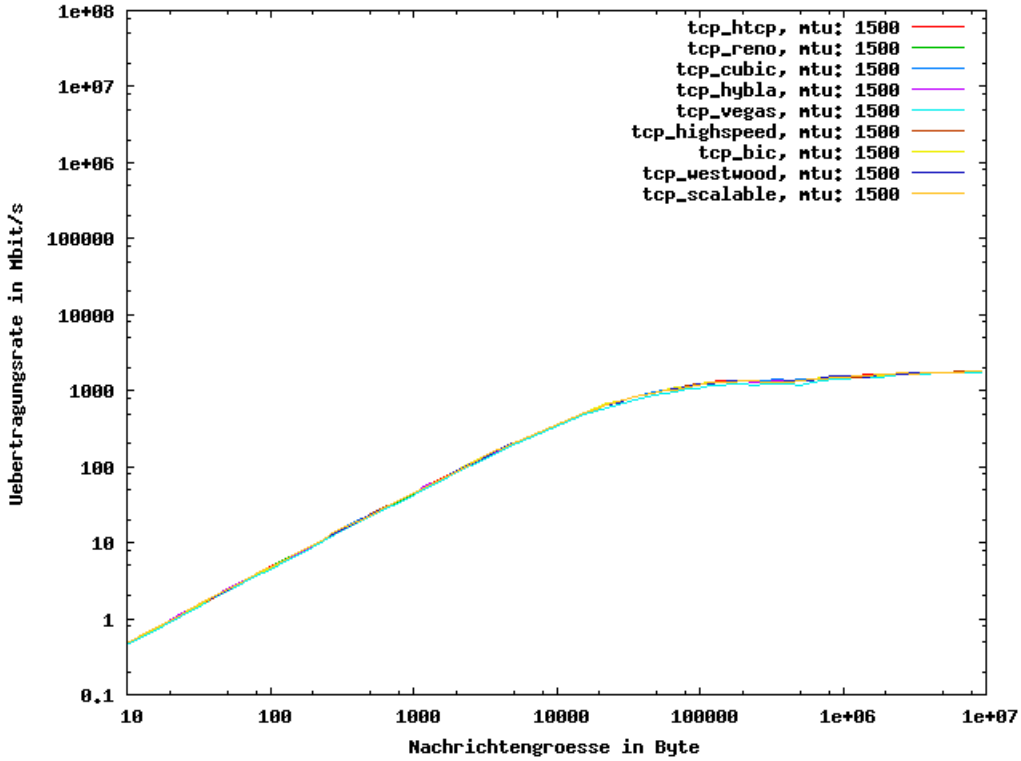

Abbildung 8: VISIT-Pingpongs mit TCP bei 10 Gbit/s und MTU 1500

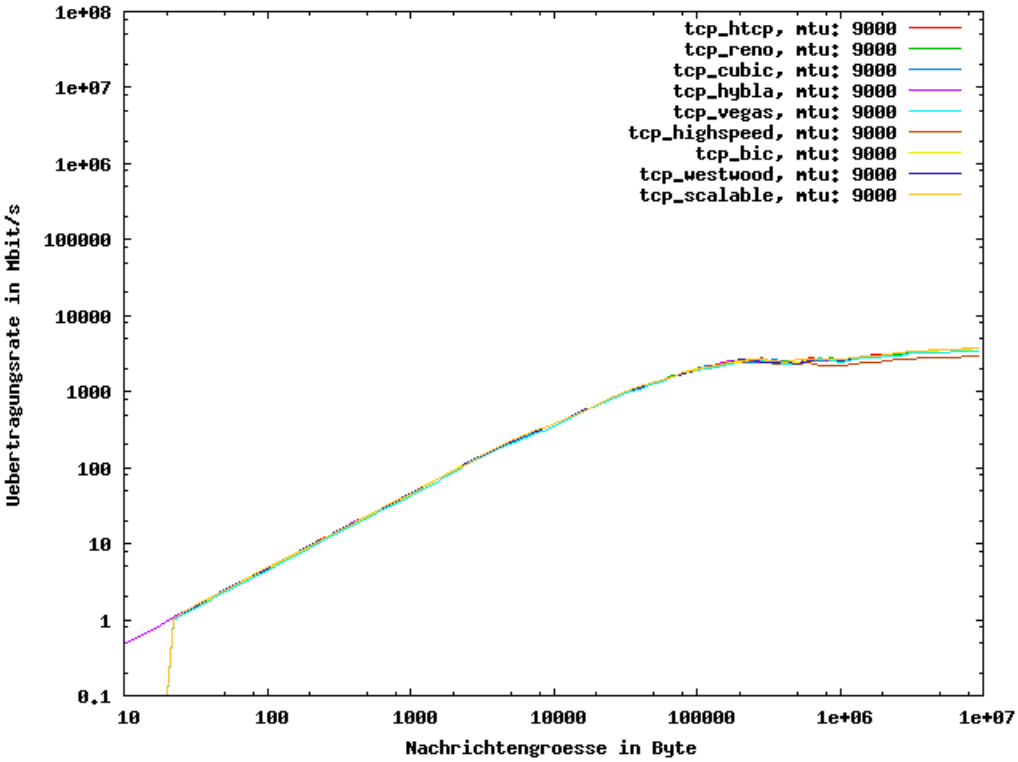

Abbildung 9: VISIT-Pingpongs mit TCP bei 10 Gbit/s und MTU 9000

2.4.4 VISIT mit UDT

Zum Zeitpunkt der Berichterstellung ist in VISIT die Aktivierung von Congestion Control-Varianten nicht möglich. Die Messungen wurden mit der UDT-Standard-Congestion-Control durchgeführt.

Start des Servers mit

```
./visitpp -S -a 192.168.168.1 -l UDT -r UDT
```

Start des Clients mit

```
./visitpp –F pp.ini
```

2.4.4.1 Messergebnisse 1 Gbit/s:

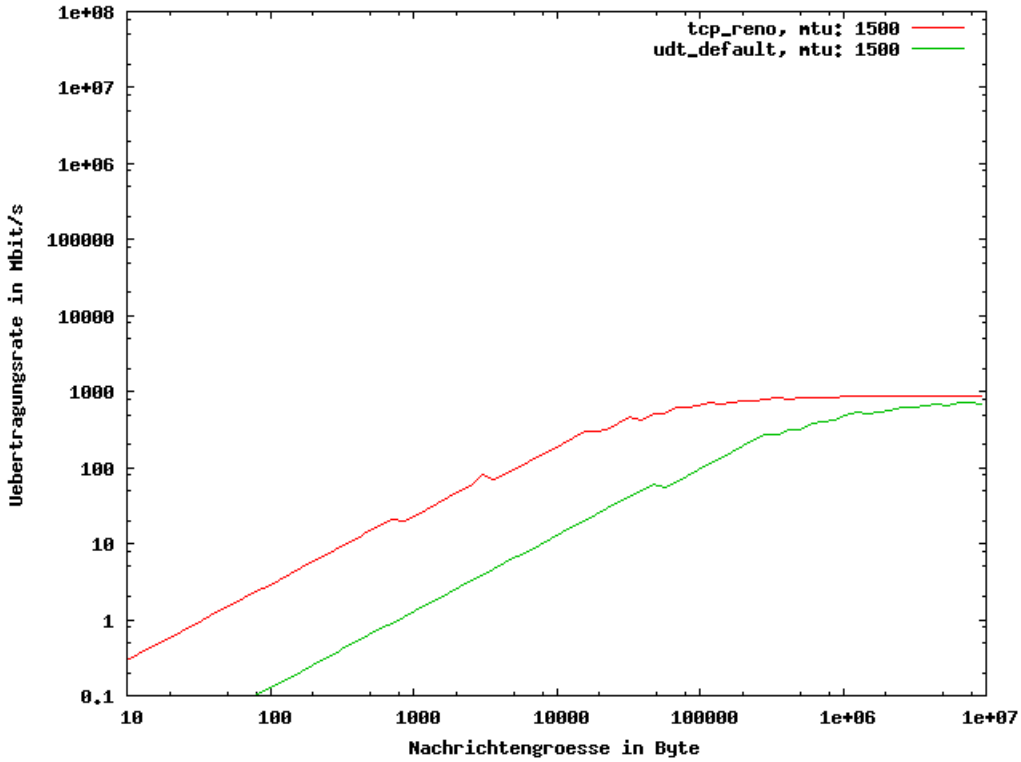

Abbildung 10: VISIT-Pingpongs mit UDT bei 1 Gbit/s und MTU 1500

2.4.4.2 Messergebnisse 10 Gbit/s:

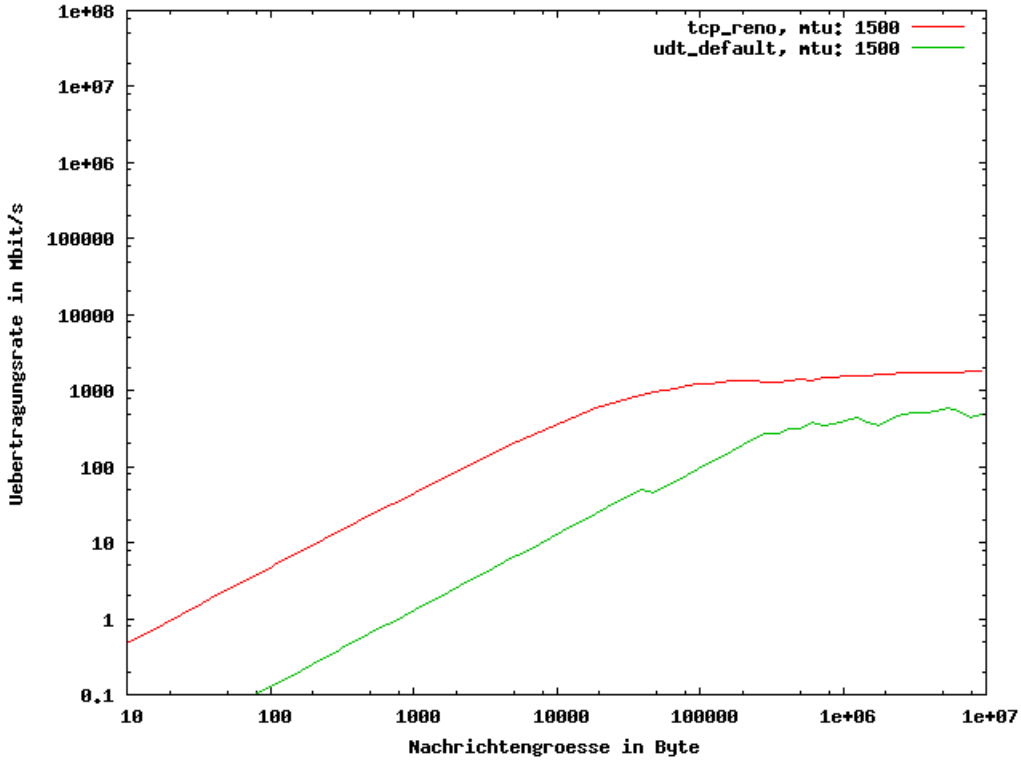

Abbildung 11: VISIT-Pingpongs mit UDT bei 10 Gbit/s und MTU 1500

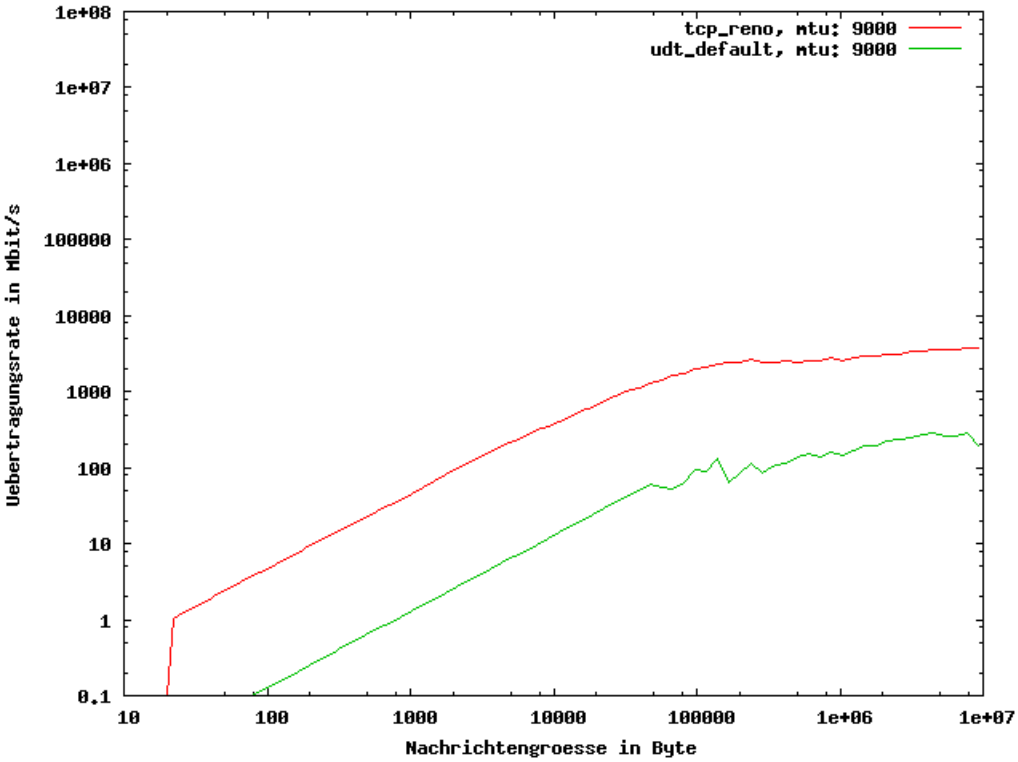

Abbildung 12: VISIT-Pingpongs mit UDT bei 10 Gbit/s und MTU 9000

2.4.5 VTS mit TCP

Start des Servers mit

```
./vtspp -S
```

Start des Clients mit

```
./vtspp –F pp.ini –H 192.168.168.1
```

2.4.5.1 Messergebnisse 1 Gbit/s:

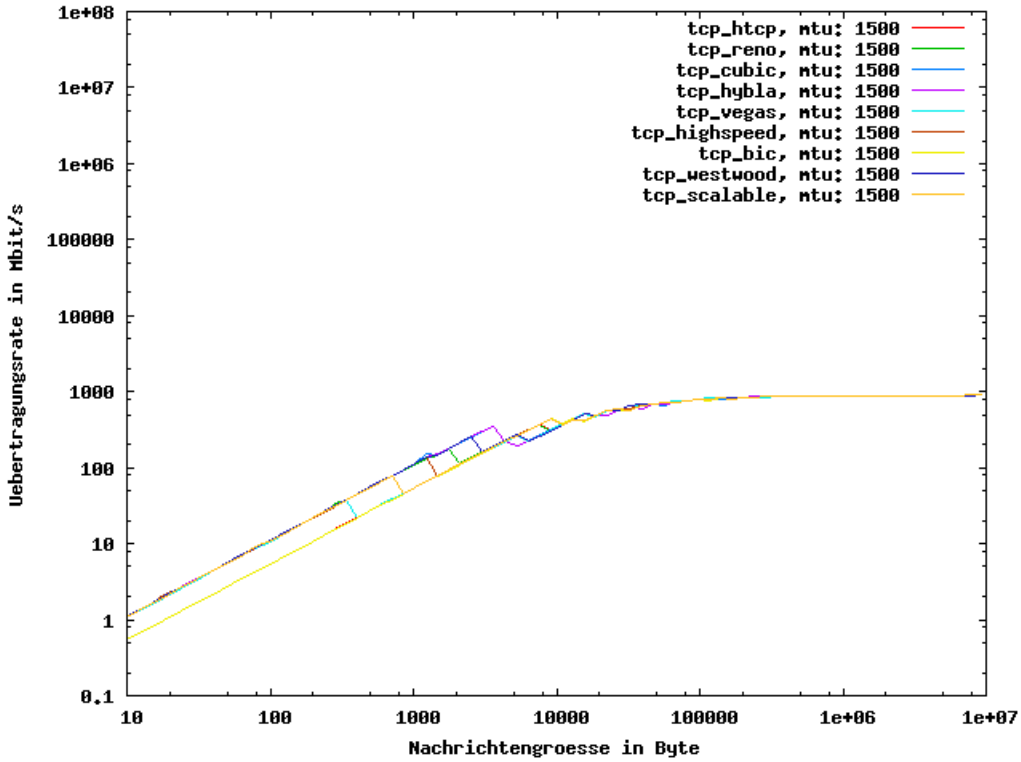

Abbildung 13: VTS-Pingpongs mit TCP bei 1 Gbit/s und MTU 1500

2.4.5.2 Messergebnisse 10 Gbit/s:

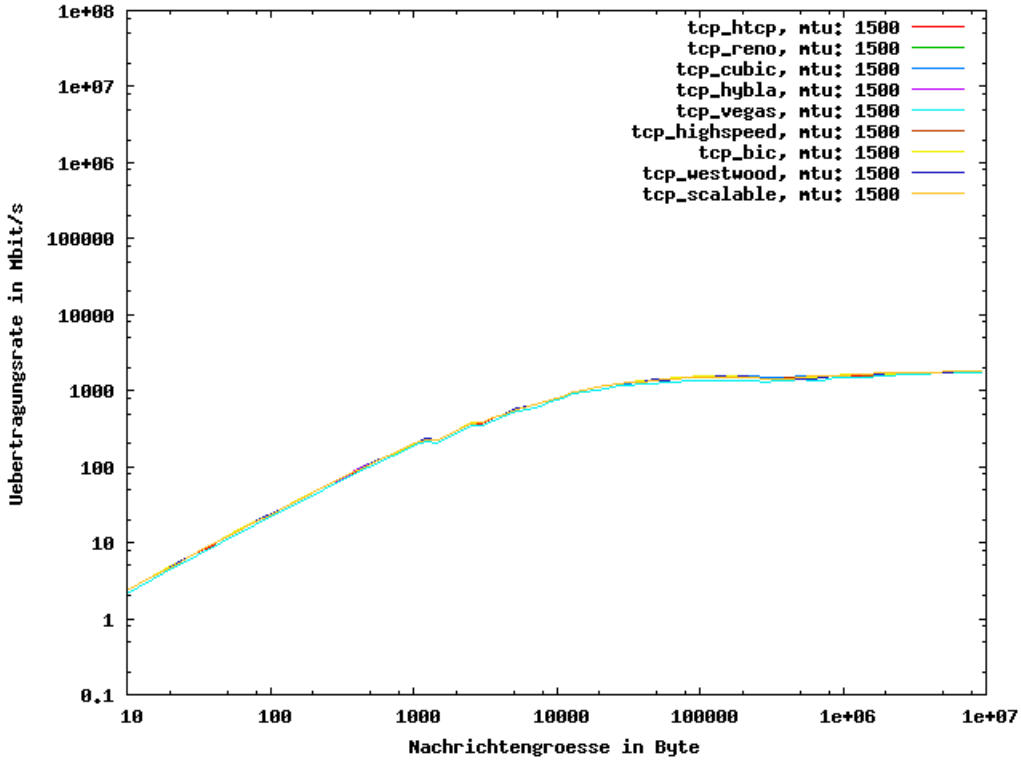

Abbildung 14: VTS-Pingpongs mit TCP bei 10 Gbit/s und MTU 1500

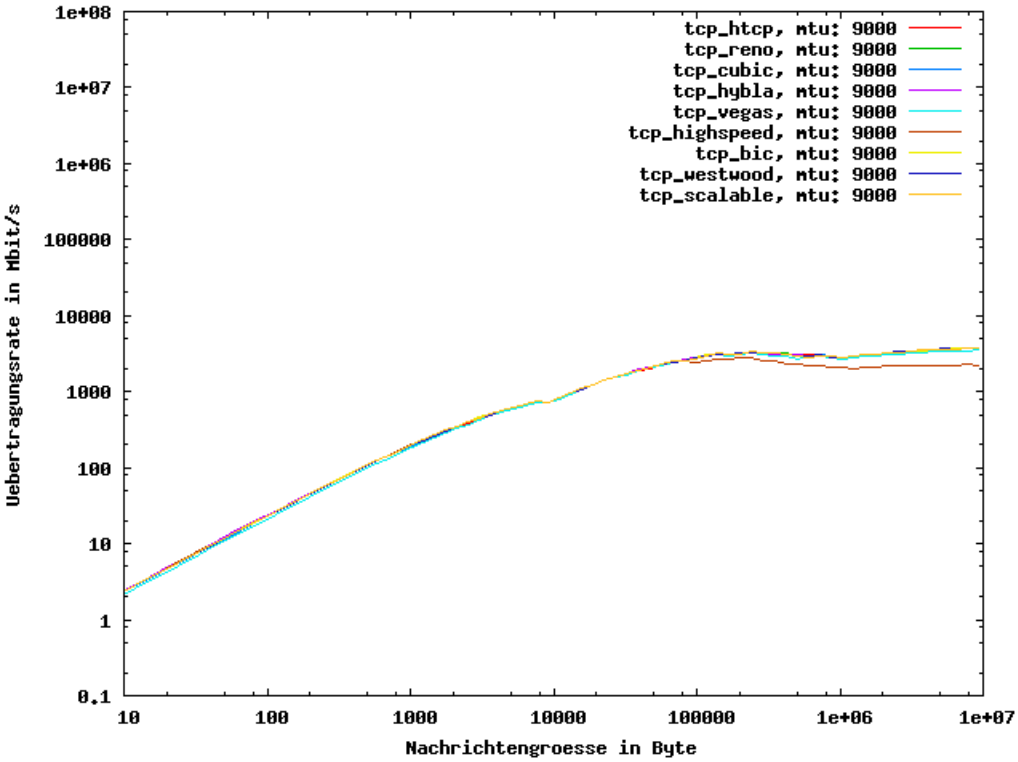

Abbildung 15: VTS-Pingpongs mit TCP bei 10 Gbit/s und MTU 9000

2.4.6 VTS mit UDT

Zum Zeitpunkt der Berichterstellung ist in VTS die Aktivierung von Congestion Control-Varianten nicht möglich. Die Messungen wurden mit der UDT-Standard-Congestion-Control durchgeführt.

Start des Servers mit

```
./vtspp -S -l UDT
```

Start des Clients mit

```
./vtspp –F pp.ini –H 192.168.168.1 –l UDT
```

2.4.6.1 Messergebnisse 1 Gbit/s:

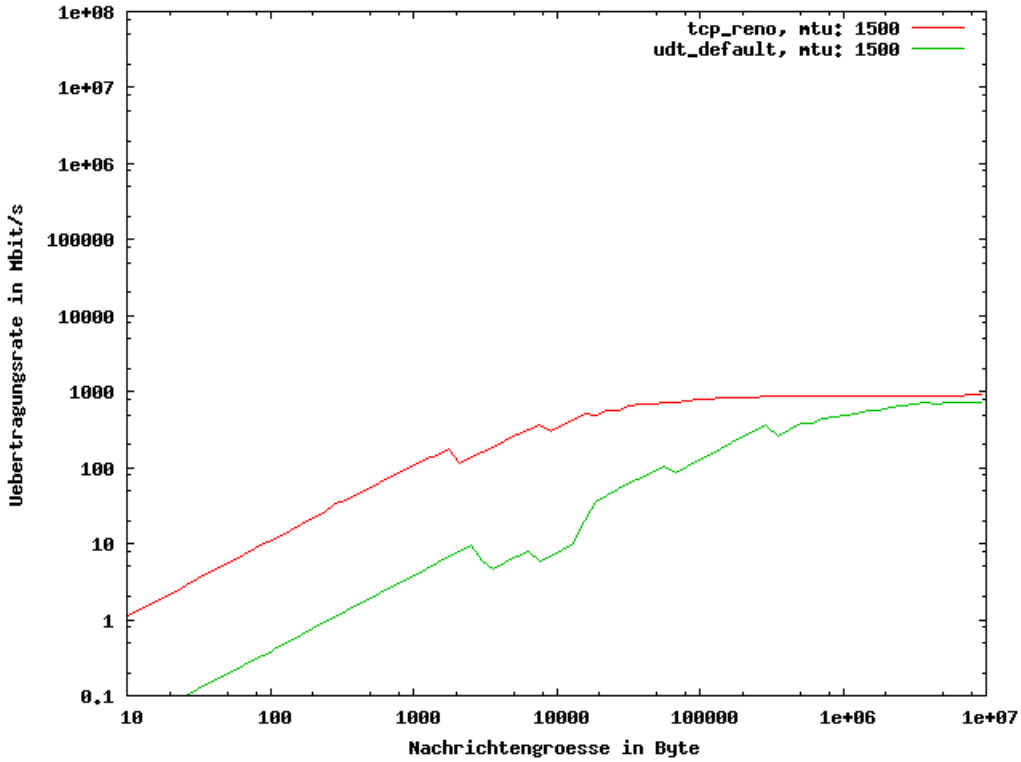

Abbildung 16: VTS-Pingpongs mit UDT bei 1 Gbit/s und MTU 1500

2.4.6.2 Messergebnisse 10 Gbit/s:

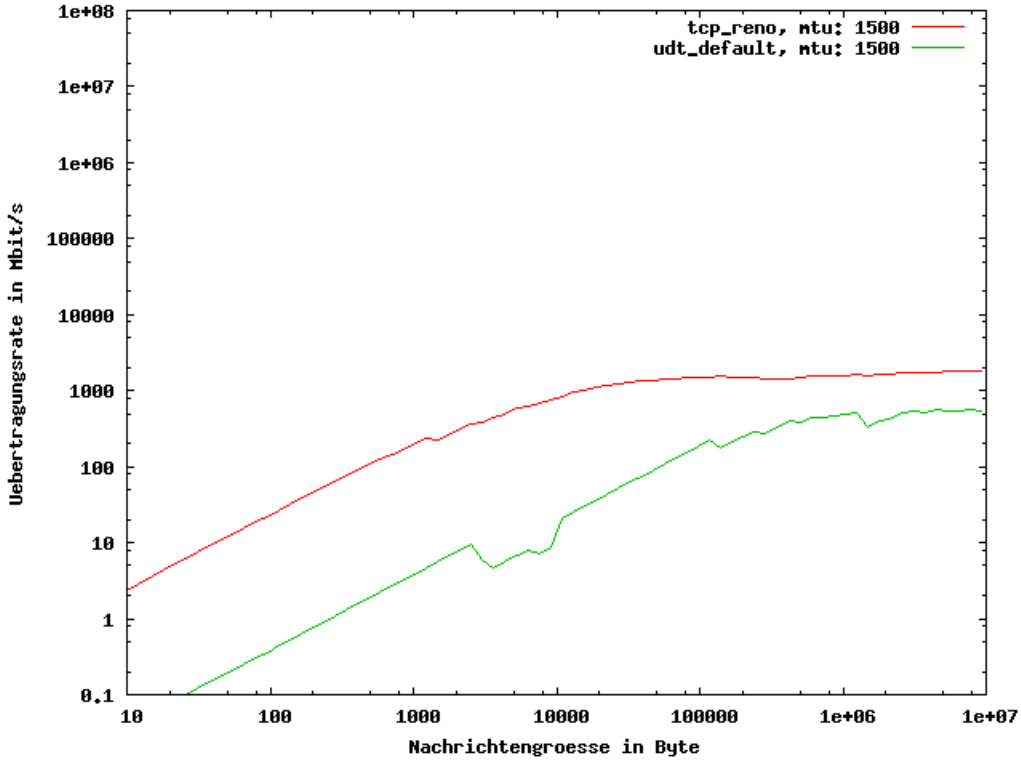

Abbildung 17: VTS-Pingpongs mit UDT bei 10 Gbit/s und MTU 1500

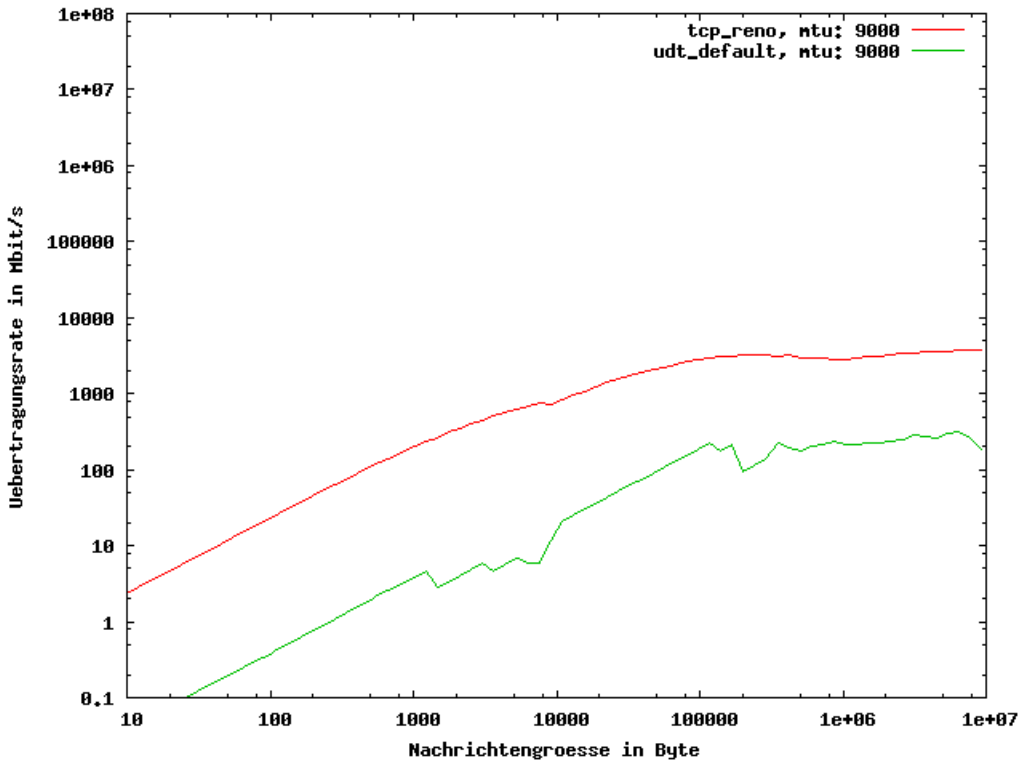

Abbildung 18: VTS-Pingpongs mit UDT bei 10 Gbit/s und MTU 9000

3 Zusammenfassung und Ausblick

Limitierender Faktor für den UDT-Durchsatz scheint die CPU-Leistung zu sein.

Der Performanzunterschied hinsichtlich Nachrichtenlaufzeit zwischen den verschiedenen TCP-Varianten scheint bei einer RTT, die höchstens der einer nationalen Grid-Infrastruktur entspricht, sowie in ungestörter Umgebung vernachlässigbar. Ob der gemessene Nachteil von UDT gegenüber TCP (Vgl. Abbildung 19, Abbildung 20, Abbildung 21) durch die Tatsache ausgeglichen wird, dass UDT komplett im Userspace läuft und somit keine Anpassungen am Betriebssystem notwendig sind, muss dem Einsatzzweck entsprechend abgewogen werden.

Insbesondere UDT befindet sich momentan noch in einem frühen Entwicklungsstadium und ist nicht ohne Fehler, von denen einige im Rahmen der Berichterstellung im Austausch mit dem Autor von UDT beseitigt werden konnten.

Abbildung 19: TCP Reno und TCP Reno über UDT bei 1 Gbit/s und MTU 1500

Abbildung 20: TCP Reno und TCP Reno über UDT bei 10 Gbit/s und MTU 1500

Abbildung 21: TCP Reno und TCP Reno über UDT bei 10 Gbit/s und MTU 9000

Ebenso zu beobachten bleibt der Emulator netem. Kann netem durch Weiterentwicklung als Flaschenhals eliminiert werden, so müssen neue Tests mit Paketverlust und größerem BDP (durch höhere RTT) zeigen, ob sich die speziell für diese Szenarien entwickelten TCP-Varianten bzw. UDT

bezüglich der Performanz von Standard-TCP absetzen können.

Die Möglichkeiten, den Durchsatz in einer 10 Gbit/s-Umgebung durch weiteres Parameter-, System- und Treibertuning, sind sicher noch nicht voll ausgeschöpft. Insbesondere wenn das Testsystem durch die Lieferung der beiden Myricom-Adapter vervollständigt ist, stehen hier noch weitere Messungen an.

Sobald eine Testumgebung mit zumindest 1 Gbit/s für AIX und Solaris zu Verfügung steht, folgen auch auf diesen Plattformen weitere Messungen.

4 Quellenverzeichnis

[1] http://lcg.web.cern.ch/LCG/overview.html

[2] http://www.d-grid.de/fileadmin/user_upload/documents/DGI-FG3-3/FG3-3_Analyse-TCP.pdf

[3] http://udt.sourceforge.net/

[4] https://forge.gridforum.org/sf/go/doc/12186?nav=1

[5] http://www.fz-juelich.de/zam/files/docs/ib/ib-00/ib-2000-16.pdf

[6] http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796,00.html

[7] http://www.tyan.com/products/html/thunderk8we.html

[8] http://www.myri.com/Myri-10G/NIC/10G-PCIE-8A-R.html

[9] http://www.neterion.com/products/xframeE.html

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

[11] http://www.psc.edu/networking/projects/tcptune/