Eine Untersuchung zum Verlust von TCP SYN und SYN/ACK Paketen bei Web-Zugriffen

Bakkalaureatsarbeit

Benjamin Kaser

Universität Innsbruck
Abt. NSG

Benjamin.Kaser@student.uibk.ac.at

Betreut durch Michael Welzl

Abgegeben am: 26.02.2008

Zusammenfassung Diese Arbeit ist eine Untersuchung über verlorene SYN und SYN/ACK Pakete beim Zugriff auf Webserver über das HTTP Protokoll. Für die Analyse wurden verschiedene Messungen auf unterschiedlichen Systemen durchgeführt und entsprechend ausgewertet. Das Ergebnis der Arbeit vermittelt den Eindruck, dass Optimierungen im TCP Protokoll vorgenommen werden sollten.

Inhaltsverzeichnis

1 Einleitung

Manchmal kommt es vor, dass der Seitenaufbau beim Aufruf einer HTML-Seite im Internetbrowser nicht so schnell erfolgt, wie das normalerweise der Fall ist. Dabei tritt manchmal das Phänomen auf, dass die Seite erst geladen wird, nachdem einige Sekunden Wartezeit verstrichen sind, oder nachdem der User den „Refresh Button“ geklickt hat. Ziel der Arbeit ist es herauszufinden, ob dafür verlorene SYN oder SYN/ACK Pakete beim Verbindungsaufbau verantwortlich sind.

Da dieses Verhalten nur sehr sporadisch auftritt, ist es fast unmöglich es direkt zu beobachten, bzw. nachzustellen. Daher muss ein anderer Ansatz gewählt werden:

Auf verschiedenen Clients und Servern wird der Netzwerkverkehr mit geeigneten Tools aufgezeichnet und statistisch ausgewertet.

2 Theoretischer Hintergrund zur Problematik

Um die Analyse durchführen zu können, bedarf es zunächst ein wenig Theorie zu TCP/IP. In den nachfolgenden Absätzen werden die wichtigsten Aspekte kurz behandelt.

2.1 Der Verbindungsaufbau in TCP

Der Verbindungsaufbau in TCP erfolgt über den 3-Way-Handshake, welcher in Abb.1 zu sehen ist. Dieser Mechanismus stellt sicher, dass sich Client und Server kennen und eine in Bezug auf Datenverlust sichere Verbindung zwischen beiden Teilnehmern besteht [11].

Der Client beginnt den Verbindungsaufbau damit, dass er ein SYN Paket schickt. Dieses hat eine Sequenznummer (seq), welche durch einen bestimmten Algorithmus zufällig vergeben wird. Außerdem ist in dem Paket bereits der Port eingetragen, auf dem die Verbindung aufgebaut werden.

Akzeptiert der Server den Verbindungsversuch, so antwortet er mit einem SYN/ACK Paket. Um sicherzustellen, dass Client und Server von derselben Verbindung sprechen, erhöht der Server die Sequenznummer um eins und sendet diese dann als Bestätigungsnummer (ack) zurück. Die Sequenznummer selbst wird neu berechnet.

Daraufhin sendet der Client ein ACK Paket als Bestätigung. Dieses enthält die um eins erhöhte Sequenznummer der SYN/ACK Pakets, damit wiederum sichergestellt ist, dass beide Teilnehmer von derselben Verbindung sprechen.

Erst nach dieser Prozedur steht die Verbindung und der Datenaustausch beginnt.

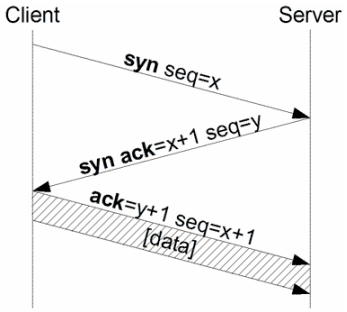

Abbildung 1. 3 Way Handshake TCP [1]

Unter bestimmten Umständen kann es jedoch schon beim Verbindungsaufbau zum Paketverlust kommen. Je nachdem, in welchem Stadium sich die Verbindung gerade

befindet, kann dies zu unterschiedlichen Situationen führen. Das TCP Protokoll sieht dafür Timer vor, welche ein erneutes Senden des verlorenen Paketes bewirken. Es gilt nun durch Messungen herauszufinden, wie diese Timer funktionieren, was sie bewirken und wie man das Verhalten von TCP verbessern kann und somit Wartezeiten für den User verkürzen kann.

2.2 Verlust von Synchronisationspaketen und deren Messung

Die Arbeit verfolgt den Ansatz, dass verlorene Synchronisationspakete für das Hängenbleiben des Browsers verantwortlich sind. Nachfolgend wird anhand einiger Schaubilder gezeigt, wo der Verlust auftreten kann und wie man ihn durch eine Messung nachweisen kann:

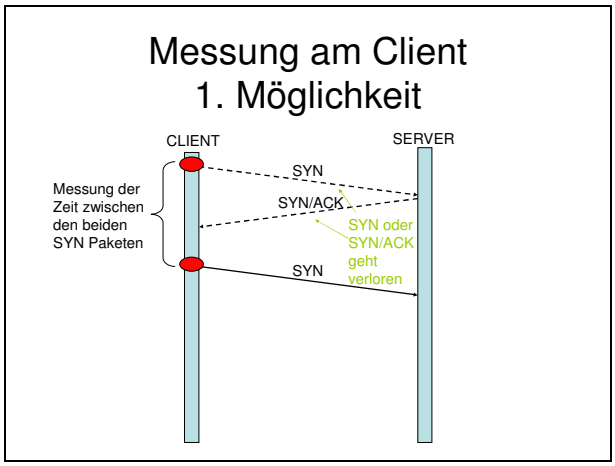

Abbildung 2. Verlust von SYN oder SYN/ACK, Messung am Client

Der Verlust eines SYN oder eines SYN/ACK Pakets lässt sich am Client einfach messen. Nach einer bestimmten Zeit wird ein erneutes SYN Paket geschickt, ohne dass in der Zwischenzeit ein SYN/ACK ankommt. In Abb. 2 ist dies grafisch verdeutlicht. Geht aber ein ACK Paket verloren, kann man das durch die Messung zweier SYN/ACK Pakete nachweisen. Siehe Abb. 3

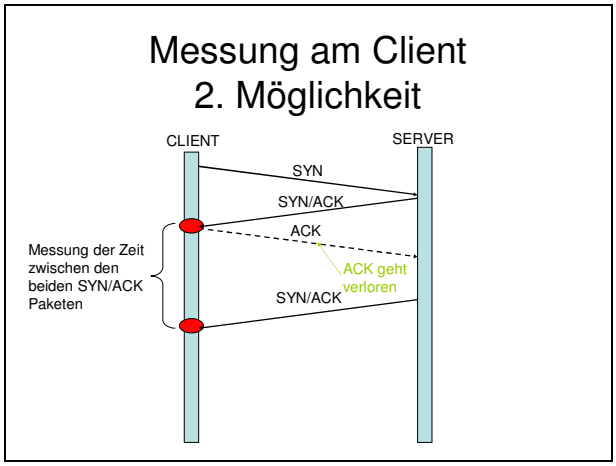

Abbildung 3. Verlust von ACK, Messung am Client

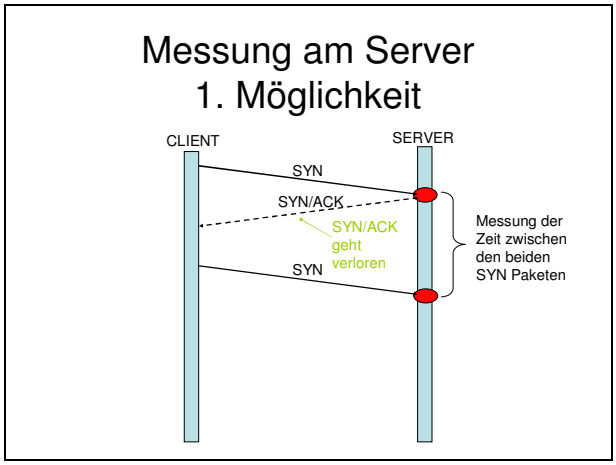

Abbildung 4. Verlust von SYN/ACK, Messung am Server

Den Verlust von SYN Paketen kann man am Server leider nicht messen, da ja beim Verlust des SYN Pakets keine Referenz zur Verfügung steht. Allerdings lässt sich der Verlust von SYN/ACK Paketen durch doppeltes SYN feststellen, sowie der Verlust von SYN/ACK und ACK Paketen durch doppelte SYN/ACK Pakete. Dies verdeutlichen Abb. 4 und Abb. 5.

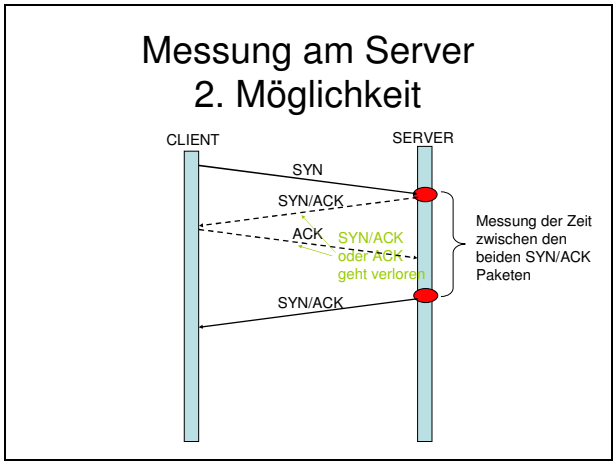

Abbildung 5. Verlust von SYN/ACK oder ACK, Messung am Server

In Abb.6 sind nochmals alle Messungen und deren Schlussfolgerungen aufgezeigt:

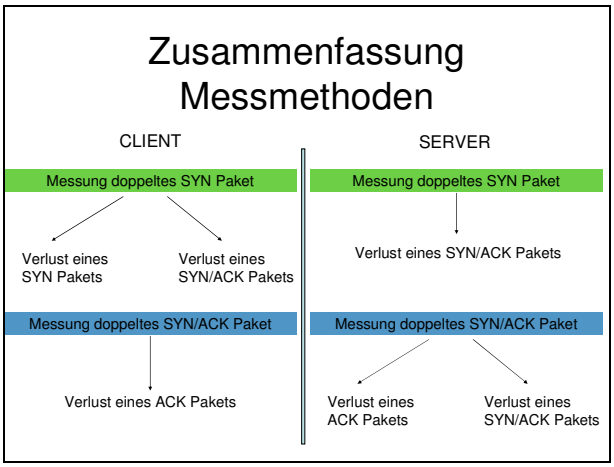

Abbildung 6. Zusammenfassung der Messmethoden

2.3 Timer in TCP

Um eine sinnvolle Implementierung einer Auswertesoftware vorzunehmen, war es wichtig über die theoretischen Hintergründe der Timer in TCP und deren konkrete Implementierung in verschiedenen Betriebssystemen bescheid zu wissen.

2.3.1 Retransmission Timer

Der wichtigste Timer in TCP ist der Retransmission Timer. Nach Ablauf dieses Timers werden unbestätigte Datenblöcke neu gesendet Der Wert für den Timer wird als RTO (Retransmission Timeout) bezeichnet. Dieser sollte auf einen Startwert von 3 Sekunden gesetzt werden [5,6]. Sobald das erste Paket bestätigt wird, errechnet er sich dynamisch durch einen Algorithmus der die Round Trip Time eines Pakets (Zeit zwischen Senden des Pakets und Erhalt der Bestätigung) berücksichtigt [5].

Falls nach erneuter Übertragung eines Pakets der Timer ein weiteres Mal abläuft, wird die RTO verdoppelt [5]. Je nachdem ob Synchronisationspakete (SYN und SYN/ACK) oder Daten übertragen werden, gibt es je nach Betriebssystem unterschiedliche Zähler, die darüber entscheiden, wie oft ein Paket erneut übertragen wird, bis die Verbindung als unterbrochen gilt [6]. Bei Windows Betriebssystemen geschieht dies bei Verbindungspaketen normalerweise zwei Mal, allerdings kann der Wert in der Registrierung angepasst werden [3,9]. Im Linux 2.6 Kernel ist standardmäßig ein Wert von 5 Wiederholungen für SYN und SYN/ACK eingestellt [8,9].

Da bei SYN Paketen auf der Senderseite und bei SYN/ACK Paketen auf der Empfängerseite noch keine RRT errechnet werden kann, wird der Startwert von 3 Sekunden für den Retransmission Timer verwendet.

2.3.2 Persistence Timer

Es kann vorkommen, dass das Empfangsfenster gerade auf 0 steht und das Paket, welches das Fenster wieder öffnen sollte verloren geht. Somit können keine weiteren Daten geschickt werden. Als Gegenmittel schickt der Persistence Timer in bestimmten Abständen kleine Pakete die die Fenstergröße abfragen [7,9].

2.3.3 Quiet Timer

Gibt die Zeit an, die verstreichen muss, bis ein TCP-Port wieder nach Abbau einer Verbindung freigegeben wird. Dieser Timer gibt zeitlich verzögerten Paketen die Chance der richtigen Verbindung und damit dem richtigen Verarbeitungsprozess zugeordnet zu werden, wenn diese bereits beendet wurde [7,10].

2.3.4 Keep Alive Timer und Idle Timer

Diese beiden Timer sind nicht Bestandteil der offiziellen RFC Anforderungen für eine richtige TCP/IP Implementierung, allerdings werden sie von bestimmten Betriebsystemen implementiert [6,7].

In regelmäßigen Abständen wird ein leeres Paket versendet um festzustellen, ob die Gegenseite noch anwesend ist. Antwortet die Gegenseite nicht mehr, wird nach Ablauf des Idle Timers die Verbindung abgebrochen.

3 Hilfsmittel und Programme zur Datenerfassung

Nachfolgend findet man eine Aufstellung der gesamten Hilfsmittel und Werkzeuge, die für diese Analyse verwendet wurden. Bei den selbst geschriebenen Programmen werden die wichtigsten Codefragmente genauer beschrieben.

3.1 Selbst entwickelte Programme

Nachfolgend das im Rahmen dieser Bakkalaureatsarbeit entwickelte Programm zur Auswertung der Tracefiles, sowie die Liste der Jobs, mit denen die Aufzeichnung durchgeführt wurde.

3.1.1 TSYN_Stat.c (mit libpcap Bibliothek) [Anhang A]

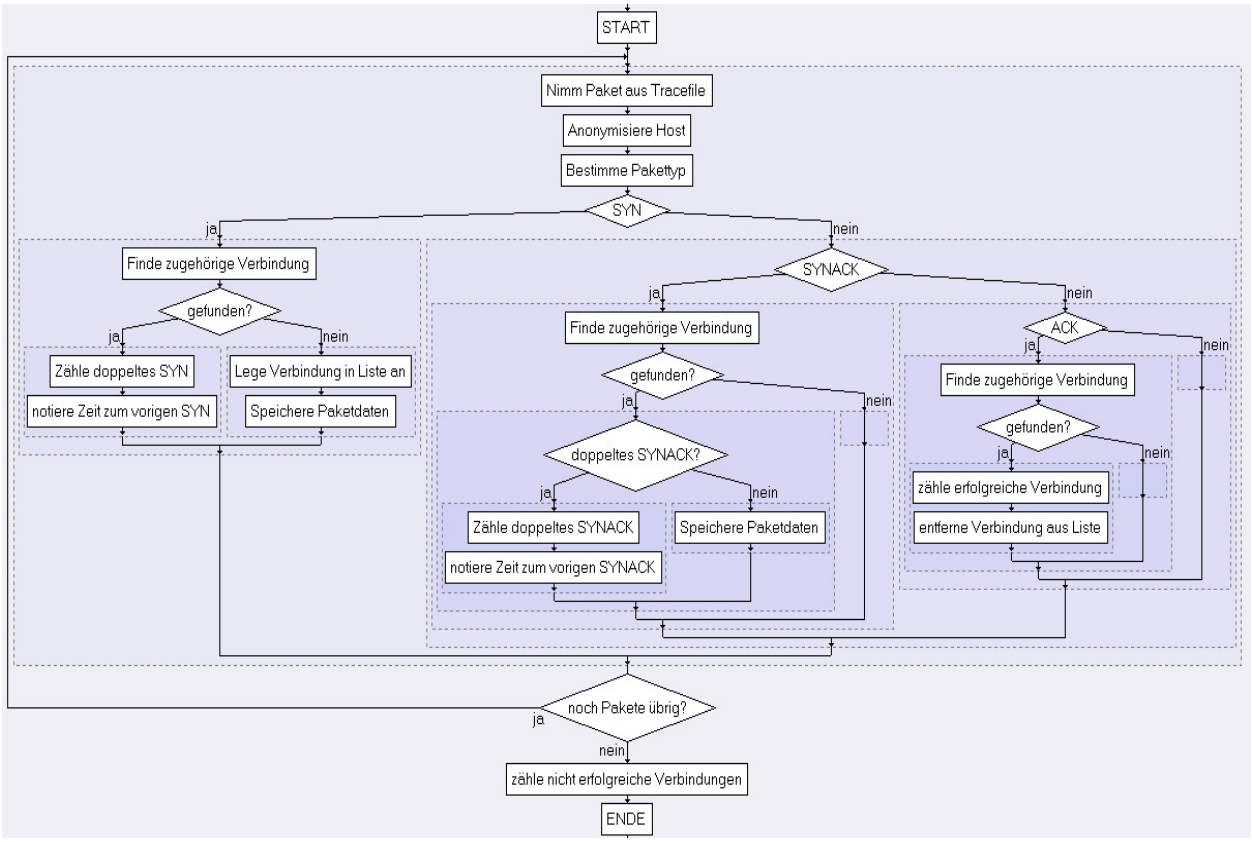

Abbildung 7. Flussdiagramm des Auswerteprogrammes TSYN_Stat.c

3.1.1.1 Funktion zum anonymisieren der IP-Adressen

Damit das ZID einverstanden war, auf ihren Servern eine Tracefile- Aufzeichnung durchzuführen, musste das Programm die User anonymisieren. Das geschah mit folgender Funktion. Alle IP-Adressen wurden in einer internen Liste gespeichert und nur der Index als Bezeichnung zurückgegeben.

```
int host_lookup_table(char *host)
{
    int index = 1;
    while (hosttable[index] != NULL) {
          if (strcmp(host,hosttable[index]) == 0) //if
ip is known return the index
                return index;
          else
                index++;
    }
    hosttable[index] =(char *) malloc
(sizeof(char)*16); // reservation of memory for the ip-
address
    strcpy(hosttable[index],host); //save the ip-
address
    return index;
}
```

3.1.1.2 Ausschnitt aus der Paketbearbeitungsfunktion (Abschnitt wird für jedes Paket im Tracefile durchgeführt)

In der Hauptfunktion wurde zunächst die Anonymisierung der IP-Adressen durchgeführt, und danach je nach Pakettyp entschieden, welche Funktion aufgerufen wird. Der Pakettyp selbst steht in jedem TCP Paket an der Stelle th_flags, welche in der Struktur definiert wurde.

```
shost = host_lookup_table(inet_ntoa(ip->ip_src)); //
cover the source-ip
dhost = host_lookup_table(inet_ntoa(ip->ip_dst)); //
cover the destination-ip
switch(tcp->th_flags) // different functions for SYN,
SYN/ACK and ACK packets
{
    case 2:     got_syn(count,shost,dhost,ntohs(tcp-
>th_sport),ntohs(tcp->th_dport),header-
>ts.tv_sec+(header->ts.tv_usec/1000000.0), ntohl(tcp-
>th_seq), ntohl(tcp->th_ack),tcp->th_win, ip->ip_ttl);
          break;
    case 18:got_SYN/ACK(count,shost,dhost,ntohs(tcp-
>th_sport),ntohs(tcp->th_dport),header-
>ts.tv_sec+(header->ts.tv_usec/1000000.0), ntohl(tcp-
>th_seq), ntohl(tcp->th_ack),tcp->th_win, ip->ip_ttl);
          break;
    case 16:got_ack(count,shost,dhost,ntohs(tcp-
>th_sport),ntohs(tcp->th_dport),header-
>ts.tv_sec+(header->ts.tv_usec/1000000.0), ntohl(tcp-
>th_seq), ntohl(tcp->th_ack),tcp->th_win, ip->ip_ttl);
```

```
            break;
      default:break;
}
```

3.1.1.3 Vorgehensweise beim Auffinden eines SYN Pakets

Wenn ein SYN Paket gefunden wurde, dann wurde zunächst überprüft, ob dasselbe SYN Paket bereits einmal angekommen ist (doppeltes SYN). Gleiches SYN Paket bedeutet hierbei: gleiche Hosts, gleiche Ports. Sollte das der Fall sein, wurde es gezählt und in einer separaten Liste die Zeitdifferenz zwischen beiden SYNs gespeichert. Ist es das erste SYN, werden alle Daten des Pakets zwischengespeichert um den vollständigen Verbindungsaufbau nachvollziehen zu können.

```
void got_syn(int count, int shost,int dhost,u_short
sport,u_short dport,double timestamp,u_int seqnr,u_int
acknr,u_short window,u_short ttl)
{
      int i = 0;
      int insert = 1;
      int double_syn = 0;

      for (i = 0 ; i < MAX_ENTRIES ; i++) {
            if ((matrix[i].host == shost) &&
(matrix[i].port == sport))
            {
                  got_double_syn(timestamp-
matrix[i].syn_time);
                  matrix[i].syn_time = timestamp;
                  matrix[i].sent_syns =
matrix[i].sent_syns + 1;
                  insert = 0;
                  double_syn = 1;
                  break;
            }
      }

      if (insert == 1)
      {
            for (i = 0 ; i < MAX_ENTRIES ; i++) {
                  if (matrix[i].host == 0)
                  {
                        matrix[i].host = shost;
                        matrix[i].port = sport;
                        matrix[i].seq_nr = seqnr;
                        matrix[i].ack_nr = acknr;
                        matrix[i].syn_time = timestamp;
                        matrix[i].SYN/ACK_time = 0;
```

```
                        matrix[i].sent_syns = 0;
                        matrix[i].sent_SYN/ACKs = 0;
                        break;
                }
        }
    }
    printline(shost,dhost,sport,dport,count,seqnr,acknr
,timestamp,"SYN",window,ttl,double_syn,0);
}
```

3.1.1.4 Vorgehensweise beim Auffinden eines SYN/ACK Pakets

Wenn ein SYN/ACK Paket gefunden wurde, wurde zunächst das passende SYN Paket dazu gesucht (gleicher Port, gleicher Host, richtige Sequenznummer). Dann wurden die Daten zum Verbindungsaufbau in der Liste ergänzt (z.B.: die Ack Nummer). Sollte ein gleicher SYN/ACK bereits einmal angekommen sein, wurde dies in der Liste gezählt und die Zeitdifferenz in einer separaten Liste gespeichert.

```
void got_SYN/ACK(int count, int shost,int dhost,u_short
sport,u_short dport,double timestamp,u_int seqnr,u_int
acknr,u_short window,u_short ttl)
{
    int i = 0;
    int double_SYN/ACK = 0;
    for (i = 0 ; i < MAX_ENTRIES ; i++) {
          if ((matrix[i].host == dhost) &&
(matrix[i].port == dport) && (matrix[i].seq_nr ==
(acknr-1)))
          {
                matrix[i].ack_nr = seqnr;
                matrix[i].sent_SYN/ACKs =
matrix[i].sent_SYN/ACKs + 1;
                if (matrix[i].sent_SYN/ACKs > 1)
                {
                      got_double_SYN/ACK(timestamp-
matrix[i].SYN/ACK_time);
                      double_SYN/ACK = 1;
                }
                matrix[i].SYN/ACK_time = timestamp;
                break;
          }
    }
    printline(shost,dhost,sport,dport,count,seqnr,acknr
,timestamp,"SYN/ACK",window,ttl,0,double_SYN/ACK);
}
```

3.1.1.5 Vorgehensweise beim Auffinden eines ACK Pakets

Wenn ein ACK Paket gefunden wurde, wurde zunächst der passende Verbindungsaufbau (SYN und SYN/ACK Paket) dazu gesucht (gleicher Port, gleicher

Host, richtige Sequenznummer, richtige Acknummer). Wenn der Verbindungsaufbau korrekt war, wurde der Listeneintrag wieder gelöscht und Platz für einen anderen Verbindungsaufbau geschaffen. Die Liste der erfolgreichen Verbindungen wurde dann ergänzt, je nachdem wie viele SYN und SYN/ACK benötigt wurden).

```
void got_ack(int count, int shost,int dhost,u_short
sport,u_short dport,double timestamp,u_int seqnr,u_int
acknr,u_short window,u_short ttl)
{
    int i = 0;

    for (i = 0 ; i < MAX_ENTRIES ; i++) {
          if
((matrix[i].host==shost)&&(matrix[i].port==sport)&&(mat
rix[i].seq_nr==(seqnr-1))&&(matrix[i].ack_nr==(acknr-
1)))
          {

    printline(shost,dhost,sport,dport,count,seqnr,acknr
,timestamp,"ACK",window,ttl,0,0);

                if (matrix[i].sent_syns < 9)

    syn_successful[matrix[i].sent_syns]++;
                else
                      syn_successful[9]++;

                if (matrix[i].sent_SYN/ACKs < 9)

    SYN/ACK_successful[matrix[i].sent_SYN/ACKs]++;
                else
                      SYN/ACK_successful[9]++;

                matrix[i].host=0;
                matrix[i].port = 0;
                matrix[i].seq_nr = 0;
                matrix[i].ack_nr = 0;
                matrix[i].syn_time = 0;
                matrix[i].SYN/ACK_time = 0;
                matrix[i].sent_syns = 0;
                matrix[i].sent_SYN/ACKs = 0;
                break;
          }
    }
}
```

3.1.2 Scripts zur Datenaufzeichnung

Hier findet sich eine kurze Übersicht über die verschiedenen Scripts die zur Aufzeichnung verwendet wurden. Hier sind auch die Filteroptionen von „tcpdump“ bzw. „snoop“ ersichtlich:

3.1.2.1 Script zur Aufzeichnung der Tracefiles des Windows Proxy Server:

```
set i=0

:begin

tcpdump.exe -c 100000 -w tcpdump.pcap ((src host
srvbx025 and dst port 80 and tcp[13] == 2) or (src host
srvbx025 and dst port 80 and tcp[13] == 16) or (dst
host srvbx025 and src port 80 and tcp[13] == 18 ))

rename tcpdump.pcap tcpdump%i%.pcap

set /a i=%i%+1

goto begin
```

Das Script führt „tcpdump“ in einer Endlosschleife mit folgenden Filtern aus:
SYN Pakete von SRVBX025 (Hostname Proxy, IP: 10.1.1.79) mit Zielport 80
SYN/ACK Pakete an SRVBX025 mit Quellport 80
ACK Pakete von SRVBX025 mit Zielport 80

Nach 100.000 Paketen wird ein neues File geschrieben und das alte umbenannt. Nachher werden die Files auf ein Linuxsystem kopiert und dort mit dem nächsten Script ausgewertet.

3.1.2.2 Script zum Erstellen der Statistiken aus den Tracefiles des Windows Proxy Server (wurde in Linux durchgeführt)

```
#!/bin/sh

i=0

rm -r ./stats

mkdir ./stats

while [ $i -le $1 ]

do

./TSYN_Stat.out $i > ./stats/packets$i.csv

i=$[$i+1]

done

exit 0
```

3.1.2.3 Script zur Aufzeichnung und Auswertung der Tracefiles auf den Linux Mailservern (hier nur eines der beiden Scripts, wobei sich nur die IP-Adresse ändert)

```
#!/bin/sh

i=0

IP=138.232.1.162

rm -r ./traces

mkdir ./traces

rm -r ./stats
```

```
mkdir ./stats
while [ true ]
do
tcpdump -i eth0 -w ./traces/tcpdump$i.pcap -s 70 -c
400000 "(tcp[13] == 2 and dst host $IP and dst port 80)
or (tcp[13] == 18 and src host $IP and src port 80) or
(tcp[13] == 16 and dst host $IP and dst port 80)"
./TSYN_Stat.out $i > ./stats/packets$i.csv
i=$[$i+1]
done
exit 0
# gcc -lpcap TSYN_Stat.c -o TSYN_Stat.out
```

Das Script läuft in einer Endlosschleife und führt zuerst mittels tcpdump die Aufzeichnung von 400.000 Paketen aus. Gleich im Anschluss werden mit dem Programm TSYN_Stat.c die Statistiken erzeugt und in dem Ordner stats gespeichert.
Die Filter von tcpdump sind folgende:
SYN Pakete an 138.232.1.162 (Webmailserver lwm1) mit Zielport 80
SYN/ACK Pakete von 138.232.1.162 mit Quellport 80
ACK Pakete an 138.232.1.162 mit Zielport 80

3.1.2.4 Script für die Auswertung der Tracefiles des LBNL/ICSI Enterprise Tracing Projects

```
#!/bin/sh
i=$1
mkdir ./traces
while [ $i -le $2 ]
do
tcpdump -r ./raw_traces/tcpdump$i.pcap -w
./traces/tcpdump$i.pcap "(tcp[13] == 2 and dst port 80)
or (tcp[13] == 18 and src port 80) or (tcp[13] == 16
and dst port 80)"
i=$[$i+1]
done
exit 0
```

Mit diesem Script wurden die Tracefiles gefiltert um nur die relevanten Pakete auszuwerten. Nach dieser Prozedur wurde dasselbe Script wie bei den Windows Proxy Server Files verwendet, um die Statistiken zu erzeugen.

3.1.2.5 Script zur Aufzeichnung der Tracefiles des Solaris Proxy Server:

```
#!/bin/sh
i=0
IP=xxx (-> hat ZID eingetragen)
rm -r ./raw_traces
mkdir ./raw_traces
while [ true ]
do
```

```
snoop -d ie0 -o ./raw_traces/snoop$i.sol -s 70 -c
400000 "(tcp[13] == 2 and src host $IP and dst port 80)
or (tcp[13] == 18 and dst host $IP and src port 80) or
(tcp[13] == 16 and dst host $IP and dst port 80)"
i=$[$i+1]
done
exit 0
```

Die Filteroptionen des snoop Befehls:
Alle SYN Pakete vom Internet Proxy Server mit Zielport 80
SYN/ACK Pakete an den Internet Proxy Server 138.232.1.162 mit Quellport 80
ACK Pakete vom Internet Proxy Server mit Zielport 80

Nachher werden die Files auf ein Linuxsystem kopiert und dort mit dem nächsten Script ausgewertet.

3.1.2.6 Script zur Auswertung der Tracefiles des Solaris Proxy Server (wurde auf Linux Rechner durchgeführt):

```
#!/bin/sh
i=0
rm -r ./traces
mkdir ./traces
rm -r ./stats
mkdir ./stats
while [ $i -le $1 ]
do
editcap -F libpcap ./raw_traces/snoop$i.sol
./traces/tcpdump$i.pcap
./TSYN_Stat.out $i > ./stats/packets$i.csv
i=$[$i+1]
done
exit 0
```

Mit dem Befehl editcap (welches dem Programm ethereal beiliegt) wurden die Tracefiles von snoop Format in das pcap Format umgewandelt. Somit war es möglich die umgewandelten Tracefiles mit dem Programm TSYN_Stat.c auszuwerten, ohne darin Änderungen vorzunehmen.

3.2 Verwendete Hilfsmittel

Diese Programme und Programmiersprachen wurden verwendet.

3.2.1 Ethereal

Dieses Programm ist in der Netzwerkanalyse weit verbreitet und dient dazu den Verkehr in einem Netzwerk mittels grafischer Oberfläche auszuwerten. Es verwendet dieselben Bibliotheken wie tcpdump und das im Rahmen der Bakkalaureatsarbeit erstellte Programm, nämlich die libpcap Bibliothek.

3.2.2 Sh Script

Diese Scriptsprache wurde auf den Linux Servern verwendet um die Aufzeichnung der Verkehr über einen job zu steuern.

3.2.3 Batch Script

Diese Scriptsprache wurde auf den Windows Servern verwendet um die Aufzeichnung des Verkehrs über einen Job zu steuern.

3.2.4 C mit Library libpcap

Diese Programmiersprache wurde verwendet um die Tracefiles statistisch auszuwerten und csv Dateien für die spätere Analyse in einer Tabellen-Kalkulationssoftware zu erzeugen.

3.2.5 Excel

Als Tabellenkalkulationsprogramm diente Microsoft Excel 2003. Hier wurden sämtliche Berechnungen an den Statistiken durchgeführt und außerdem alle Diagramme gezeichnet.

3.2.6 Tcpdump (Linux und Windows)

Dieses Netzwerkanalyseprogramm diente zur Aufzeichnung der Tracefiles auf allen Linux und Windowsservern. Die Software verwendet auch die libpcap Bibliothek. Das Programm ist kommandozeilenbasiert und man kann die Aufzeichnung durch entsprechende Parameter nach bestimmten Kriterien filtern.

3.2.7 Snoop (Solaris)

Dieses Netzwerkanalyseprogramm diente zur Aufzeichnung der Tracefiles auf den Solaris Proxyservern. Es ist ähnlich zu tcpdump, allerdings etwas umfangreicher und daher mächtiger.

4 Messungen

4.1 Webmail Server Universität IBK (Messung am Server)

Die erste Testmessung wurde an den beiden Webmailservern der Universität Innsbruck für einen Zeitraum von 5 Tagen durchgeführt um festzustellen, wie oft das Problem auftritt. Bei dieser Messung wurde festgestellt, dass das Problem mit verloren gegangenen SYN und SYN/ACK Paketen tatsächlich existiert. Daraufhin wurde das Programm zur Auswertung erweitert und verbessert, sowie der Messzeitraum auf 15 volle Tage ausgedehnt um eine breitere Messwertbasis zu haben. Die Art der Messung entspricht einer Messung am Server (siehe Kap 2.2)

4.1.1 Messaufbau

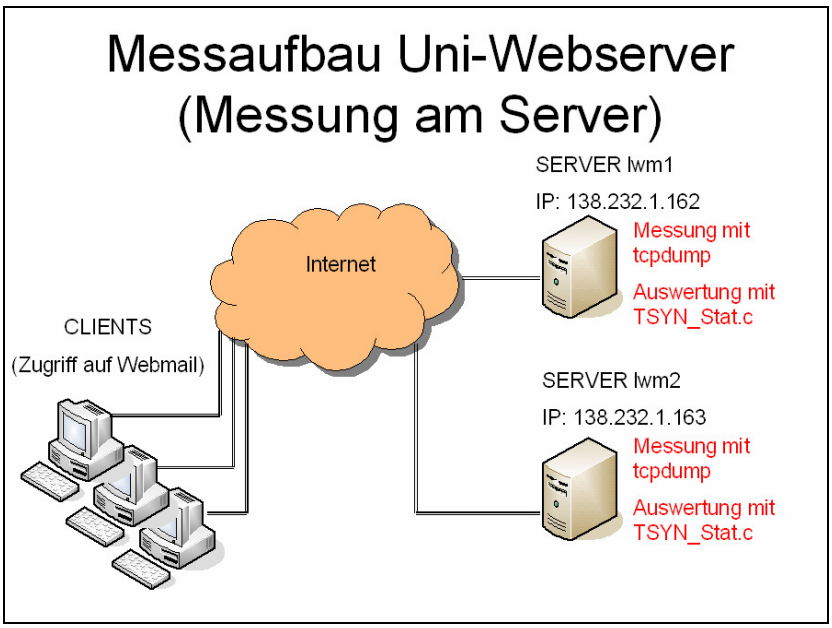

Abbildung 8. Messaufbau Webmail Server Uni IBK

4.1.2 Daten zur Messung

Betriebssystem des beobachteten Rechners: Linux
Beginn der Aufzeichnung: 06.12.2007
Ende der Aufzeichnung: 18.12.2007
Dauer: 12 Tage
Anzahl Pakete aufgezeichnet: 208.000.000
Verbindungsversuche erfolgreich: 1.390.410
Netzwerkverbindung: 1 Ethernet 10/100/1000 Mbit full duplex
Internetanbindung: 120 Mbit SDSL
Verwendete Messmethoden: Messung am Server 1 und 2 (siehe Kap. 2.2)

4.2 Internet Proxy Server Universität IBK (Messung am Client)

Um auch Messungen am Client vorzunehmen wurde einer der Internet Proxy Server der Uni Innsbruck verwendet. Hierbei wurde zur Aufzeichnung der Tracefiles snoop verwendet. Die Art der Messung entspricht einer Messung am Client (siehe Kap. 2.2).

4.2.1 Messaufbau

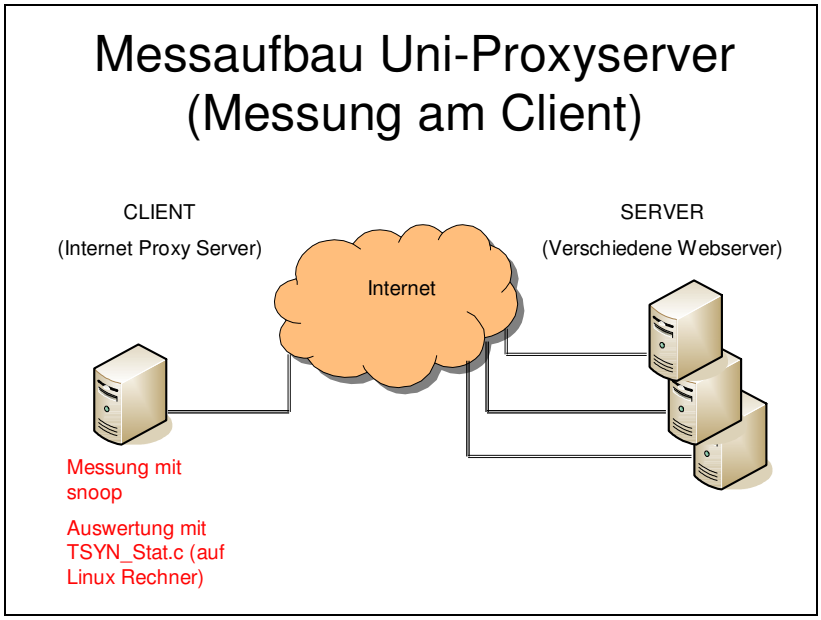

Abbildung 9. Messaufbau Uni Internet Proxyserver

4.2.2 Daten zur Messung

Betriebssystem des beobachteten Rechners: Solaris
Beginn der Aufzeichnung: 08.01.2008
Ende der Aufzeichnung: 10.01.2008
Dauer: ca. 2 Tage
Anzahl Pakete aufgezeichnet: 154.000.000
Verbindungsversuche erfolgreich: 1.708.442
Netzwerkverbindung: 4 Ethernet 10/100/1000 Mbit full duplex
Internetanbindung: 120 Mbit SDSL
Verwendete Messmethoden: Messung am Client 1 und 2 (siehe Kap. 2.2)

4.3 Internet Proxy Server Fa. Alupress AG Brixen (Messung am Client)

Da zunächst nicht klar war, ob eine Messung auf den Internet Proxy Servern der Universität Innsbruck möglich ist, wurde bei Firma Alupress eine Tracefile-Aufzeichnung auf dem Internet Proxy Server gestartet. Diese Messung entspricht einer Messung am Client (siehe Kap. 2.2). Bei Alupress surfen ca. 150 User über diesen Internet Proxy Server.

4.3.1 Messaufbau

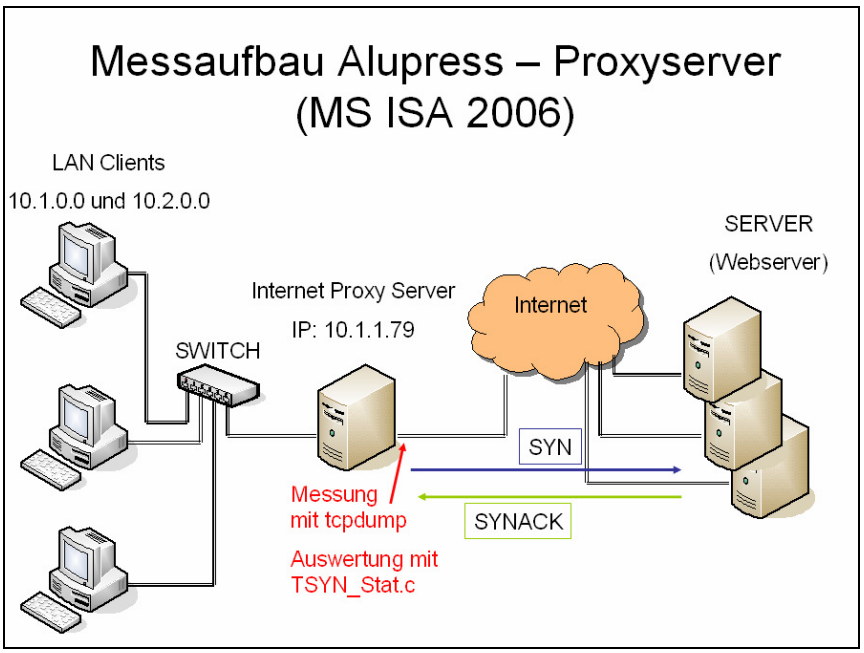

Abbildung 10. Messaufbau Alupress Proxyserver

4.3.2 Daten zur Messung

Betriebssystem des beobachteten Rechners: Microsoft Windows
Beginn der Aufzeichnung: 11.12.2007
Ende der Aufzeichnung: 19.12.2007
Dauer: 8 Tage
Anzahl Pakete aufgezeichnet: 1.700.000
Verbindungsversuche erfolgreich: 75.049
Netzwerkverbindung: 2 Ethernet 10/100/1000 Mbit full duplex
Internetanbindung: 4 Mbit SDSL
Verwendete Messmethoden: Messung am Client 1 und 2 (siehe Kap. 2.2)

4.4 Freie Logfiles vom LBNL/ICSI Enterprise Tracing Project [2] (Messung am Client, sowie am Server)

Diese Tracefiles wurden im Laufe eines groß angelegten Projekts in einer Firma aufgezeichnet, anonymisiert und in verschiedenen Arbeiten statistisch ausgewertet. Die Tracefiles stehen auf der Homepage frei zum Download bereit. Insgesamt sind es in etwa 11Gb an Dateien, welche durch einen speziellen Filter in tcpdump auf die für diese Arbeit relevanten http Daten reduziert wurden. Übrig geblieben sind ca. 450Mb an Tracefiles im pcap Format.

4.4.1 Messaufbau

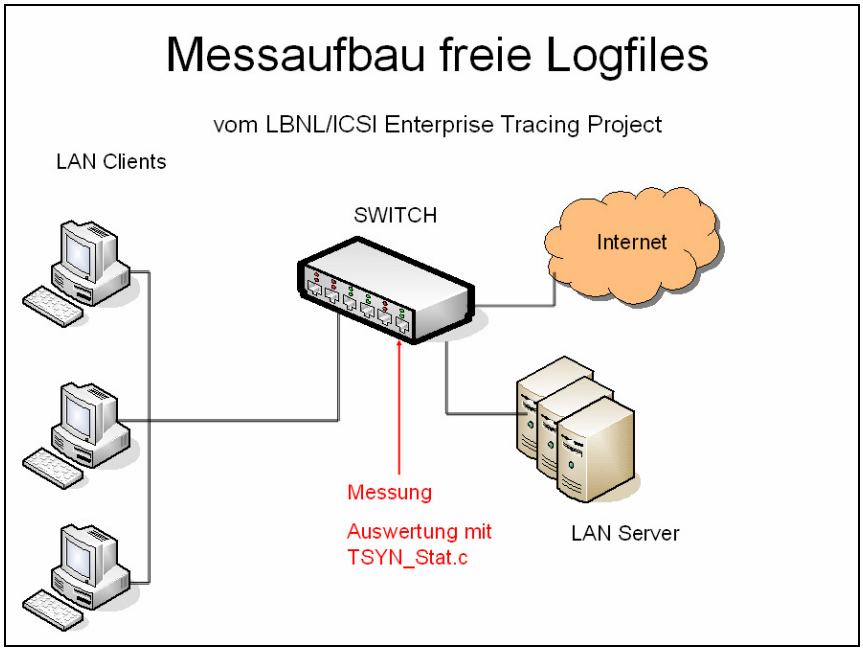

Abbildung 11. Messaufbau freie Logfiles vom LBNL/ICSI Enterprise Tracing Project

4.4.2 Daten zur Messung

Betriebssystem der beobachteten Rechner: verschiedene
Beginn der Aufzeichnung: 10.2004
Ende der Aufzeichnung: 01.2005
Dauer der Aufzeichnung: 3 Monate
Anzahl Pakete aufgezeichnet: ca. 5.000.000 zum HTTP Protokoll
Verbindungsversuche erfolgreich: 219.867
Verwendete Messmethoden: Mischung aller vier Messarten (siehe Kap 2.2)

5 Statistiken

In den nachfolgenden Diagrammen sind sämtliche Auswertungen der gemessenen Daten dargestellt. Um die Leserlichkeit zu verbessern wurden bestimmte Grafiken logarithmisch dargestellt, da oft sehr große mit kleinen Werten verglichen werden müssen.

5.1 Verteilungsfunktionen

Nachfolgend sind alle Verteilungsfunktionen der einzelnen Messungen dargestellt. Um die vom Betriebssystem automatisch gesendeten Pakete (durch den Retransmission Timer, siehe Kap. 2.3) besser identifizieren zu können wurde die Skalierung der X-Achse in 3 Sekundenblöcken gewählt.

Bei einigen Grafiken steht in der Beschreibung, dass sie ohne automatisch gesendete Pakete sind. Damit ist gemeint, dass alle Werte in Intervall in den Intervallen [0s-0,3s], [2,5s-3,5s], [5,5s-6,5s], [11,5s-12,5s] und [23,5s-24,5s] entfernt wurden. Die Intervalle wurden intuitiv nach Betrachten der Messwerte gewählt, d.h. bei der groben Analyse der Messwerte wurde festgestellt, dass eine Häufung der Paketverluste im Intervall von 0,5s vor und 0,5s nach dem eigentlichen Wert des RTO Timers aufgetreten ist. Es wird angenommen, dass diese Werte nur aufgrund von Ungenauigkeiten in der Abwicklung des Retransmission Timers nicht exakt dem RTO Timer entsprechen und daher automatisch versendete Pakete sind.

5.1.1 Webmail Webserver Universität IBK (Server)

5.1.1.1 Lwm 1

Die nachfolgenden Diagramme gehören zum ersten Webmailserver der Universität Innsbruck. Die Messung entspricht einer Messung am Server.

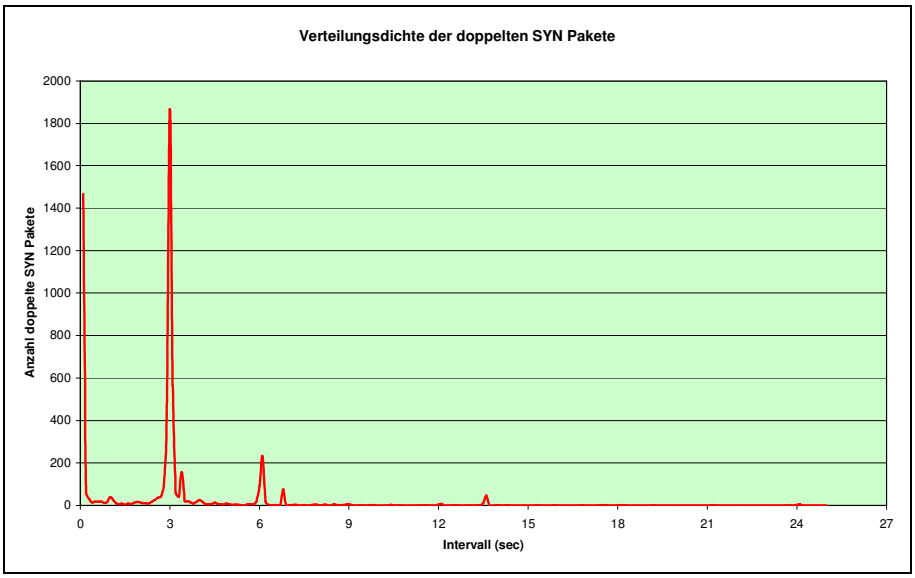

Abbildung 12. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket

Man sieht in bb.12 eine deutliche Spitze im Bereich um die 3 Sekunden. Diese Pakete sind wohl automatisch durch den Retransmission Timer geschickt worden. Auffallend ist die Spitze zwischen 0 und 0,3 Sekunden, welche nicht einfach zu erklären ist. Falsche Handhabung der User (z.B. Doppelklick, statt Einzelklick), oder

Fehlfunktionen im Betriebssystem könnten der Grund für den kurzen Abstand zum vorigen SYN darstellen.

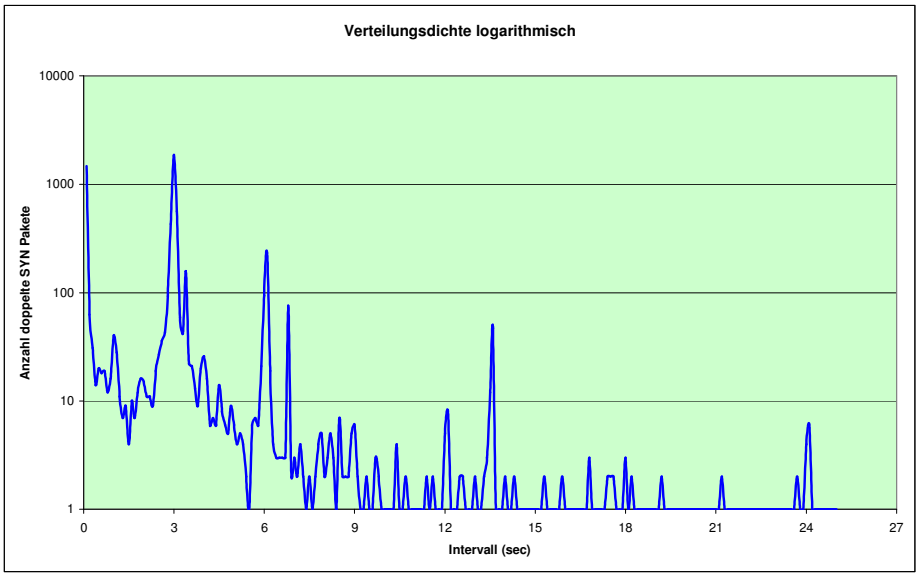

Abbildung 13. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket logarithmisch

In Abb. 13 sieht man durch die logarithmische Darstellung die von den Hauptwerten abweichenden doppelten SYN Pakete besser. Am Auffallensten sind hier die die Werte zwischen 1 und 2,5 Sekunden, zwischen 3,5 und 5 Sekunden, sowie bei ca. 7 und bei ca. 13,5 Sekunden. Diese doppelt geschickten SYN Pakete könnten durch den User (mit Klick auf den Refresh Button) ausgelöst worden sein.

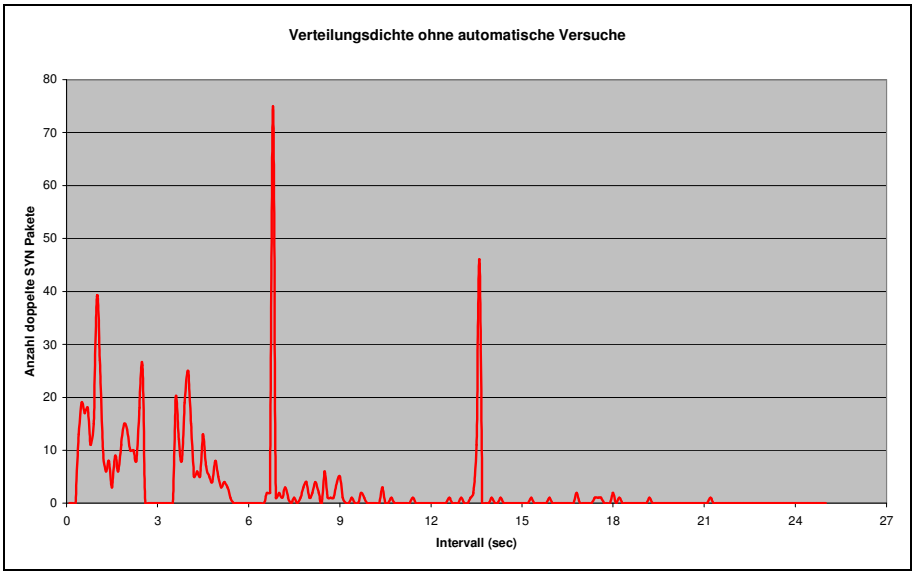

Abbildung 14. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket ohne die automatisch geschickten

In Abb. 14 sind die offensichtlich durch den RT Timer geschickten Pakete ausgeblendet um die Anomalien besser sichtbar zu machen. Hierbei sieht man die vorhin erwähnten Spitzen deutlich.

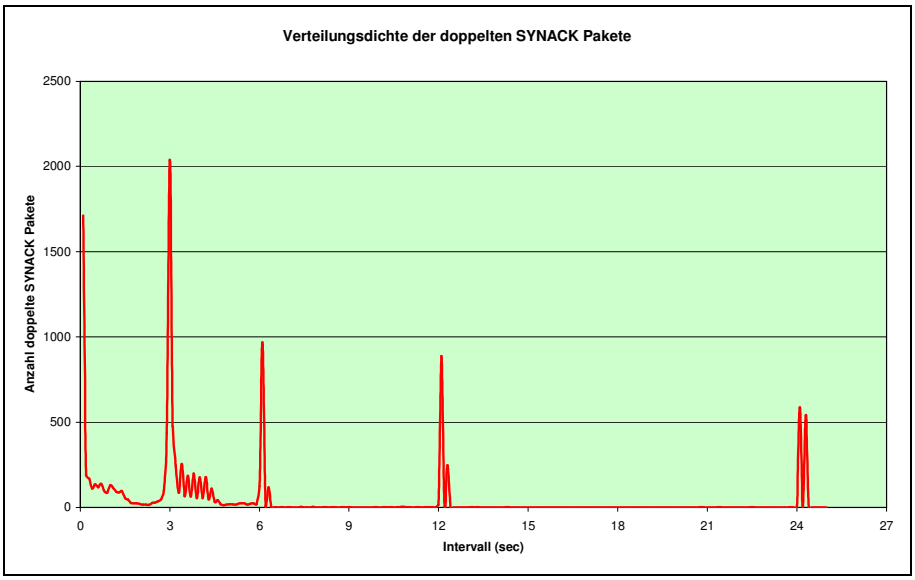

Abbildung 15. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket

In Abb. 15 sieht man die doppelt geschickten SYN/ACK Pakete und deren Abstand zum Vorgängerpaket aufgetragen. Auffallend ist hier, dass zwar ca. gleich viele SYN wie SYN/ACK Pakete nach 3 Sekunden neu geschickt wurden (etwa 2000), allerdings wesentlich mehr SYN/ACK Pakete bei 6, 12 und 24 Sekunden verschickt wurden.

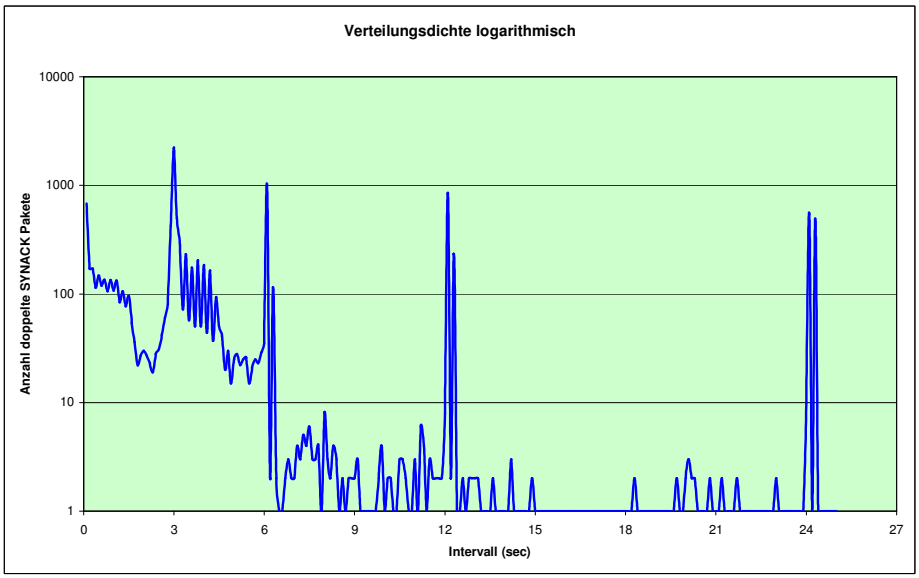

Abbildung 16. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket logarithmisch

Durch die logarithmische Darstellung in Abb. 16 sieht man deutlich die Anomalien zwischen 0,5 und ca. 1,5 Sekunden, sowie zwischen 3,5 und 5 Sekunden. Besonders die Werte im Intervall [3,5s-5s] deuten darauf hin, dass der Server das erneute Übertragen der verlorenen Pakete nicht zur rechten Zeit abwickeln konnte (etwa durch Überlast) und deshalb eine leichte Verzögerung verursacht.

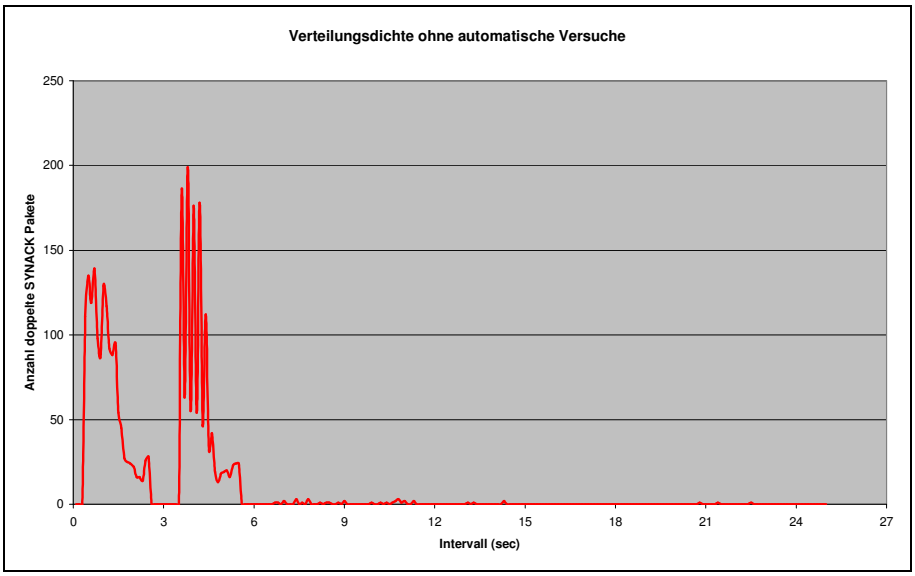

Abbildung 17. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket ohne die automatisch geschickten

In Abb.17 sind die Spitzen im Intervall [3,5s-5s] noch einmal deutlich sichtbar. Obwohl in der Bildunterschrift anders beschrieben ist aufgrund der Masse der mehrfach verschickten SYN/ACK Pakete davon auszugehen, dass es sich hier wohl doch um automatisch verschickte Pakete handelt, welche dann allerdings nicht zur korrekten Zeit verschickt wurden.

5.1.1.2 Lwm 2

Die nachfolgenden Diagramme gehören zum zweiten Webmailserver der Universität Innsbruck. Die Messung entspricht einer Messung am Server.

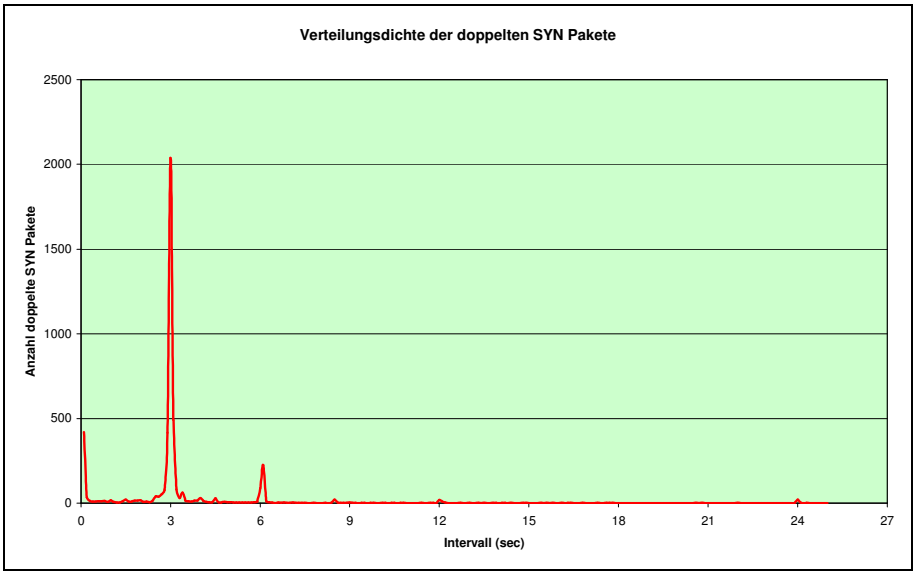

Abbildung 18. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket

In Abb. 18 sieht man ähnliche Werte wie in Abb.12. Allerdings fällt hier auf, dass die Anzahl der doppelten Pakete zwischen 0 und 0,3 Sekunden wesentlich geringer ausfällt als beim ersten Webmail Server.

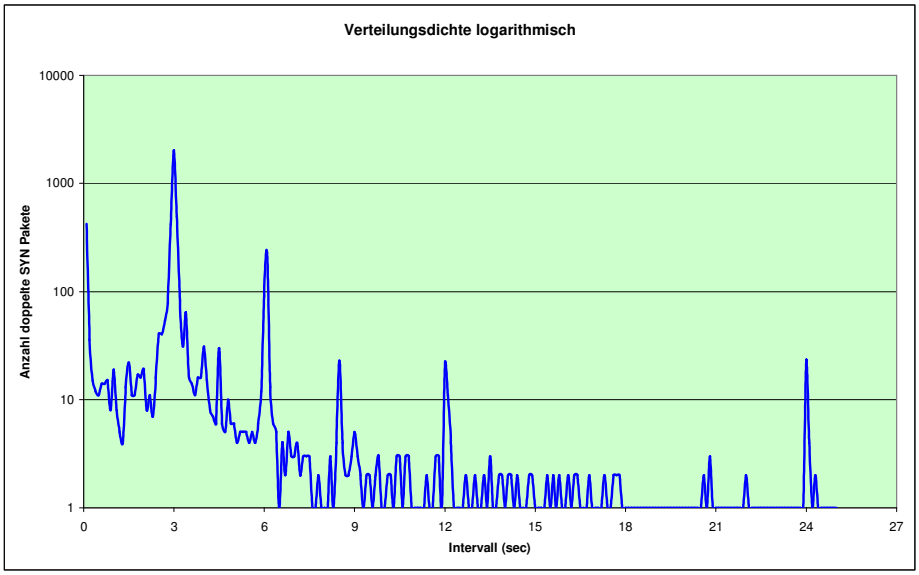

Abbildung 19. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket logarithmisch

Auch für Abb. 19 sieht man durch die logarithmische Darstellung die von den Hauptwerten abweichenden doppelten SYN Pakete besser. Am Auffallensten sind hier die die Werte zwischen 1 und 2,5 Sekunden, zwischen 3,5 und 5 Sekunden, sowie bei ca. 8 Sekunden. Diese doppelt geschickten SYN Pakete könnten durch den User (mit Klick auf den Refresh Button) ausgelöst worden sein.

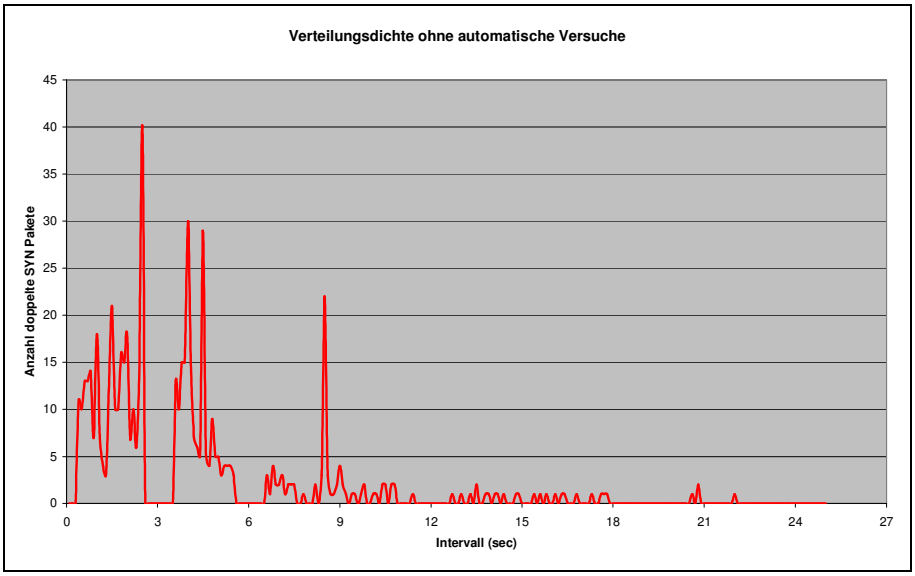

Abbildung 20. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket ohne die automatisch geschickten

Ein Ausblenden der vermutlich automatisch geschickten Pakete in Abb. 20 zeigt deutlich Spitzen bei 2,4 Sekunden, sowie zwischen 4 und 5. Auch der Grund für die Spitze bei 8 Sekunden mit über 20 Paketen ist unklar.

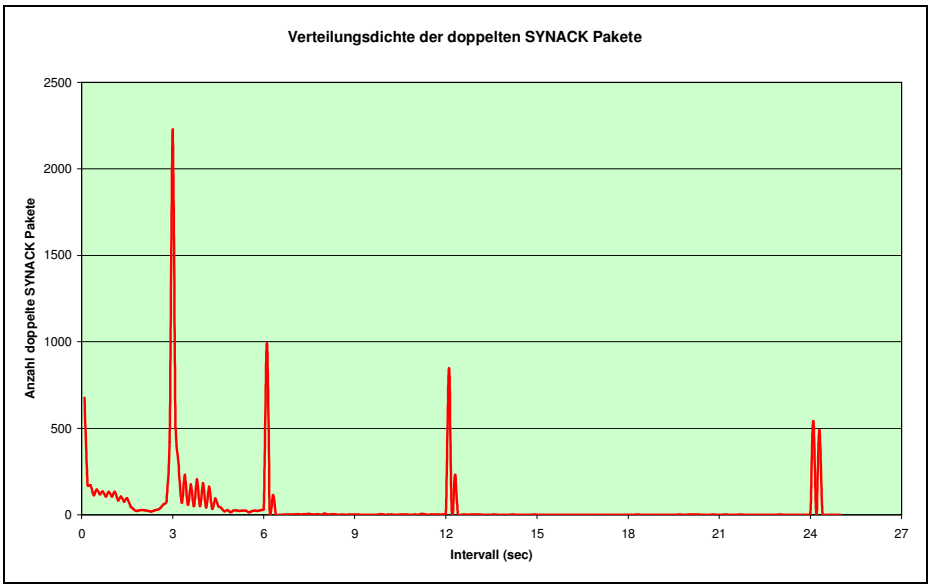

Abbildung 21. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket

Für die Betrachtung der SYN/ACK Pakete (Abb. 21) gilt ähnliches wie für den ersten Webmailserver. Insgesamt gibt es mehr SYN/ACK Pakete bei 6,12 und 24 Sekunden.

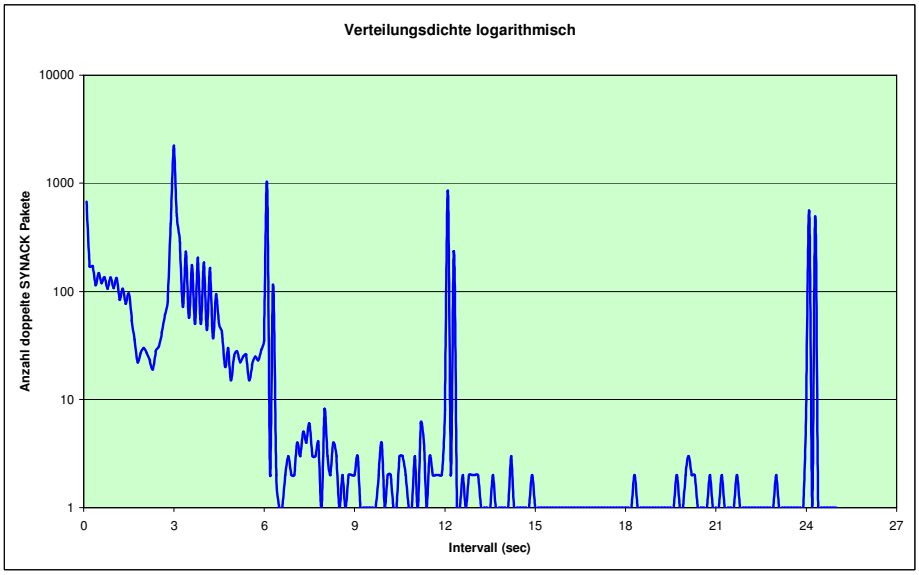

Abbildung 22. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket logarithmisch

In der logarithmischen Darstellung in Abb. 22 sieht man ähnlich wie beim ersten Webmail Server nach zwischen 3,5 und 5 Sekunden eine hohe Anzahl an doppelt verschickten SYN/ACK Paketen. Vermutlich handelt es sich hierbei ebenfalls um automatisch geschickte SYN/ACK Paketen, welche allerdings durch Überlast des Servers nicht innerhalb des drei Sekunden Intervalls verschickt werden konnten.

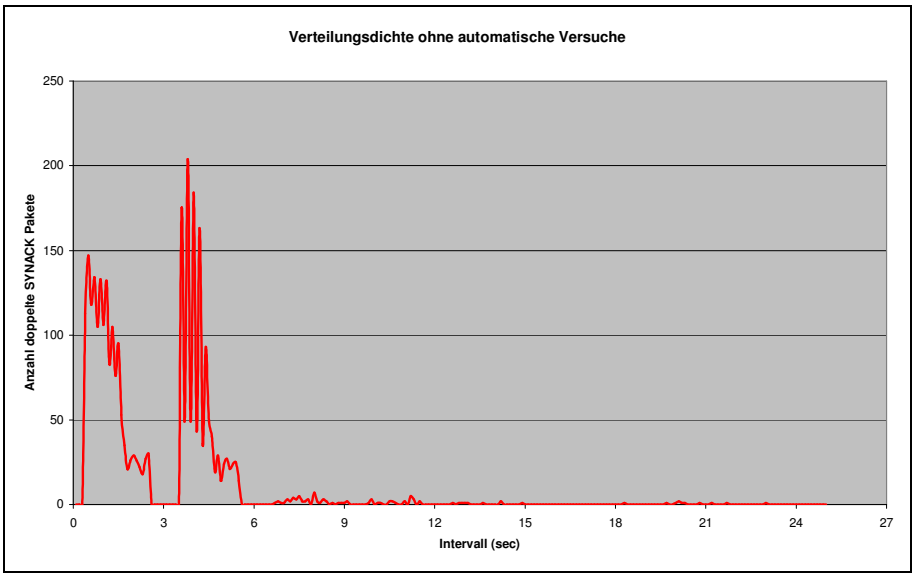

Abbildung 23. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket ohne die automatisch geschickten

In Abb.23 ist dieser Effekt noch einmal verdeutlicht dargestellt. Obwohl in der Bildunterschrift anders beschrieben, scheint es sich hierbei um automatisch geschickte SYN/ACK Pakete zu handeln, da es wesentlich mehr als zuvor registrierte SYN Pakete sind. Per Definition kann ein manuell ausgelöstes SYN/ACK nur dann manuell sein, wenn es auch ein zugehöriges manuelles SYN Paket gibt, da es der Client ist, der durch den Klick auf den Refresh Button ein erneutes Senden eines SYN Pakets auslöst.

5.1.2 Internet Proxy Server Fa. Alupress AG Brixen (Client)

Nachfolgend wird die Messung am Internet Proxy Server von Fa. Alupress betrachtet. Die Messung entspricht einer Messung am Client.

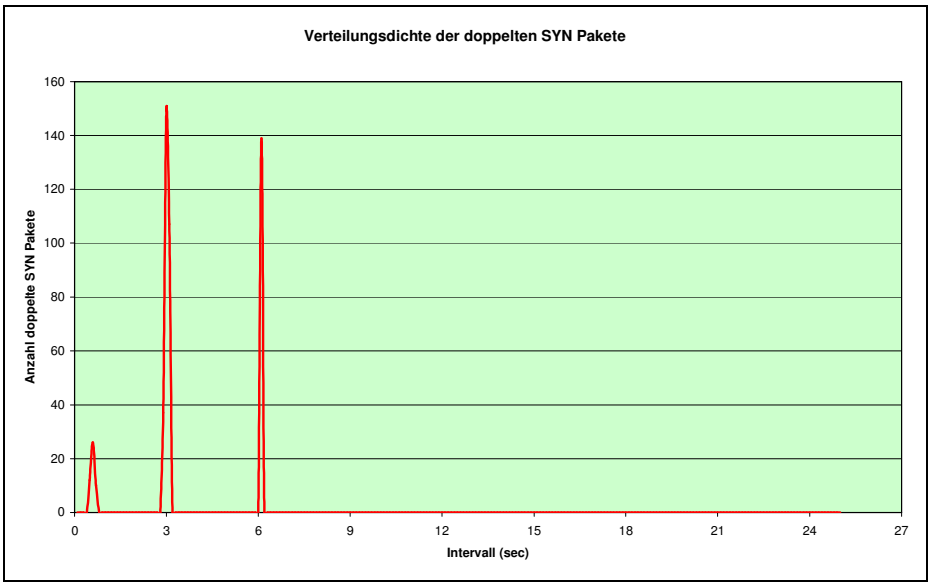

Abbildung 24. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket

In Abb. 24 gibt es nur einen Ausschlag, der manueller Natur sein könnte. Diese etwa 30 Pakete könnten etwa durch einen Doppelklick.

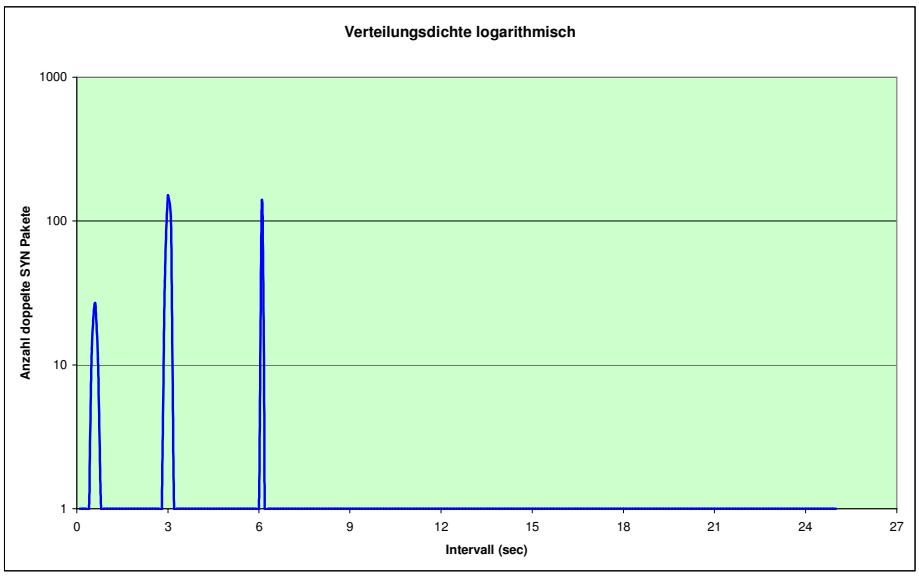

Abbildung 25. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket logarithmisch

Abb. 25 zeigt die Messung hier nochmals in logarithmischer Darstellung.

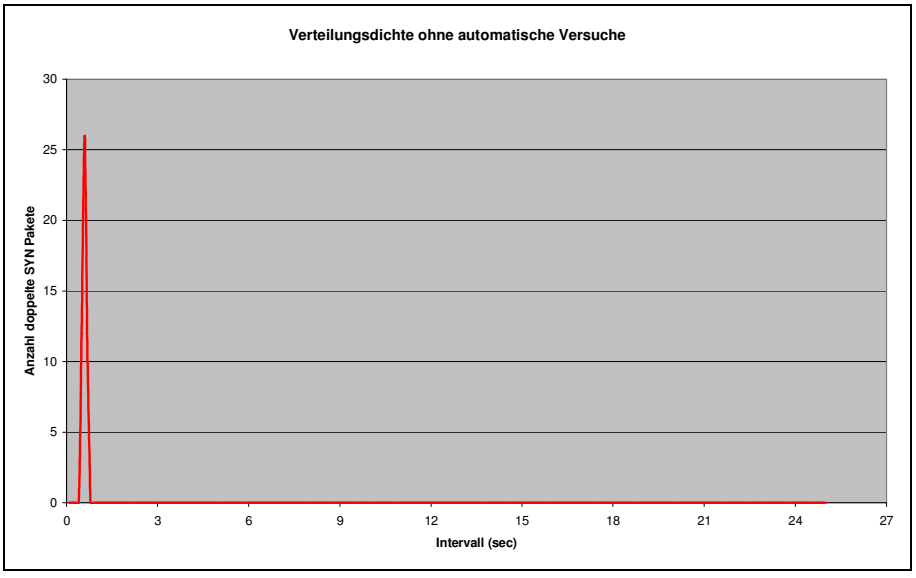

Abbildung 26. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket ohne die automatisch geschickten

In Abb. 26 wurden die automatischen Paketsendungen ausgeblendet. Auffallend ist, dass hier nur bei etwa 0,5s doppelt verschickte Pakete vorkommen. Der Grund hierfür könnten wiederum die User mit einem Doppelklick, oder einen Klick auf „Refresh“. Es kann sich aber auch um ein Verhalten des Microsoft ISA Server 2006 handeln.

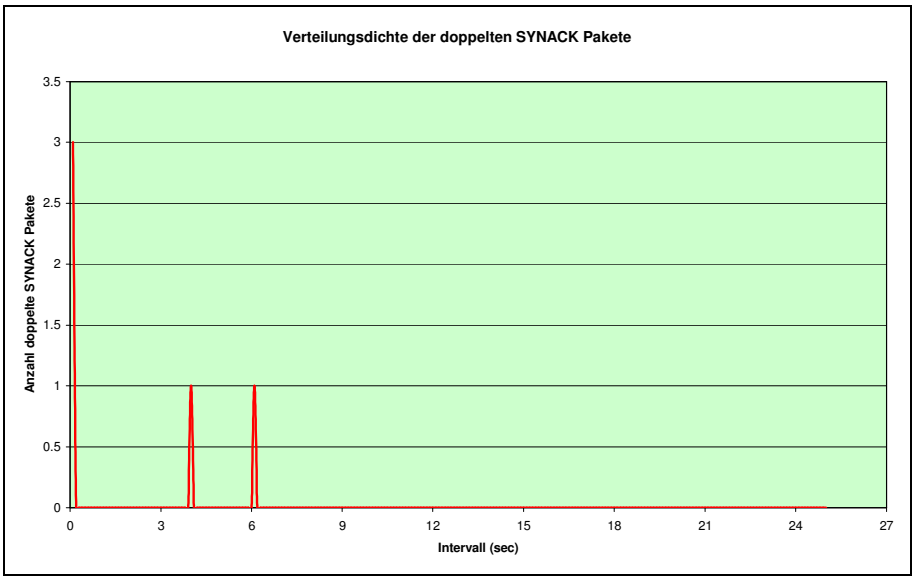

Abbildung 27. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket

In Abb. 27 sieht man die doppelt verschickten SYN/ACK Pakete. Auffallend ist hier, dass es nur sehr wenige sind. Das würde bedeuten, dass der Prozentsatz an verlorenen SYN Paketen deutlich höher liegt, als der der verlorenen SYN/ACK Paketen. Es kann sich aber bei den verschickten SYN Paketen aber auch um Blindgänger handeln, welche ihr Ziel nicht erreicht haben, weil es dieses gar nicht mehr gibt.

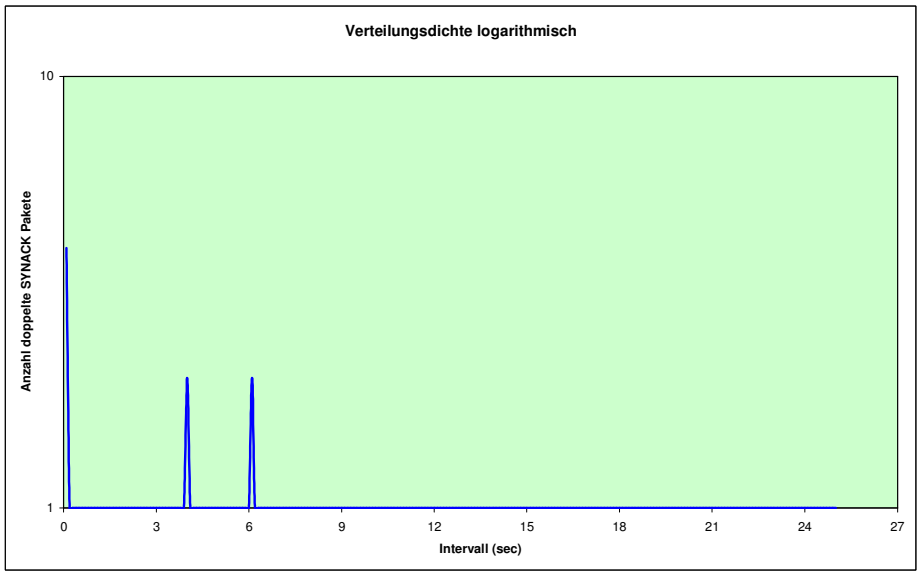

Abbildung 28. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket logarithmisch

Der Vollständigkeit halber ist in Abb. 28 die Verteilungsdichte der SYN/ACK Pakete nochmals logarithmisch dargestellt.

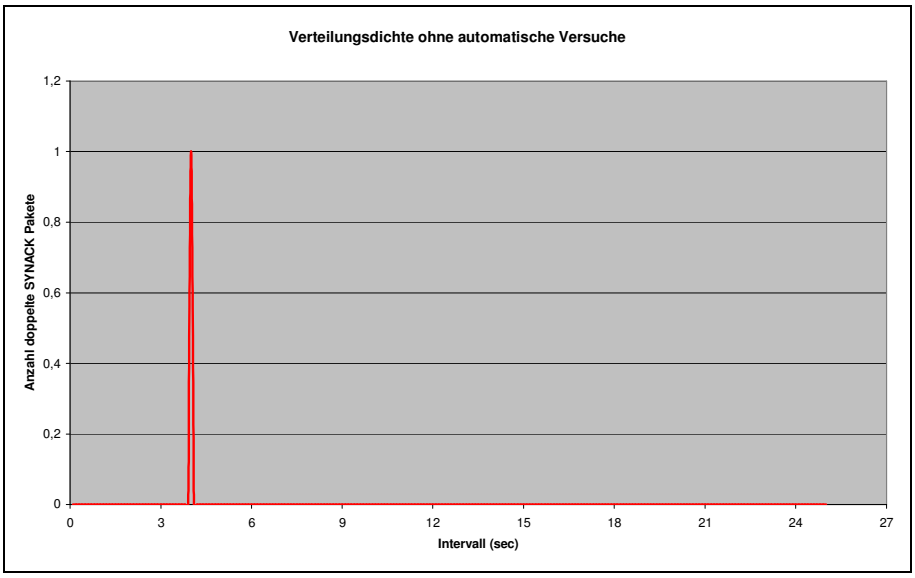

Abbildung 29. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket ohne die automatisch geschickten

In Abb.29 sieht man das einzige Paket, das aus dem automatischen Intervallen liegt. Da es allerdings kein zugehöriges SYN Paket gibt, muss man davon ausgehen, dass der Server hier wahrscheinlich nur verspätet reagiert hat und das Paket eigentlich automatisch durch den Retransmission Timer verschickt wurde.

5.1.3 Internet Proxy Server Universität IBK (Client)

Etwas umfangreicher sind die Messwerte der Aufzeichnung an einem der Uni Innsbruck Internet Proxy Server. Bei der Auswertung der Messungen fällt auf, dass auf diesem Server der RTO (siehe Kap. 2.3) auf 3,4s gesetzt ist. Das ist laut TCP/IP Spezifikationen [5,6] erlaubt, denn der Wert von 3s ist nur eine Empfehlung.

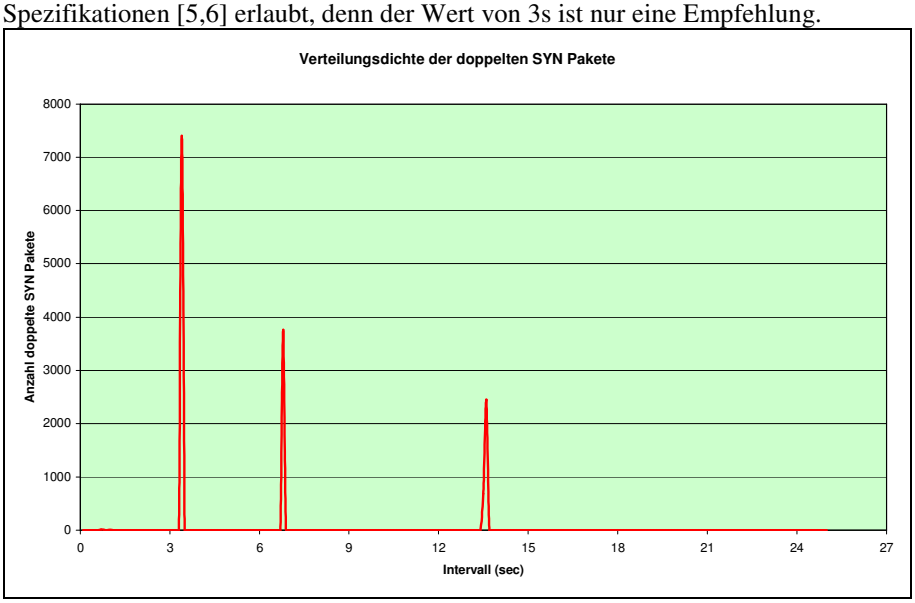

Abbildung 30. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket

Abb.30 verdeutlicht den RTO von 3,4s. Man sieht deutliche Ausschläge bei 3,4s und dessen Vielfachen 6,8s sowie 13,6s.

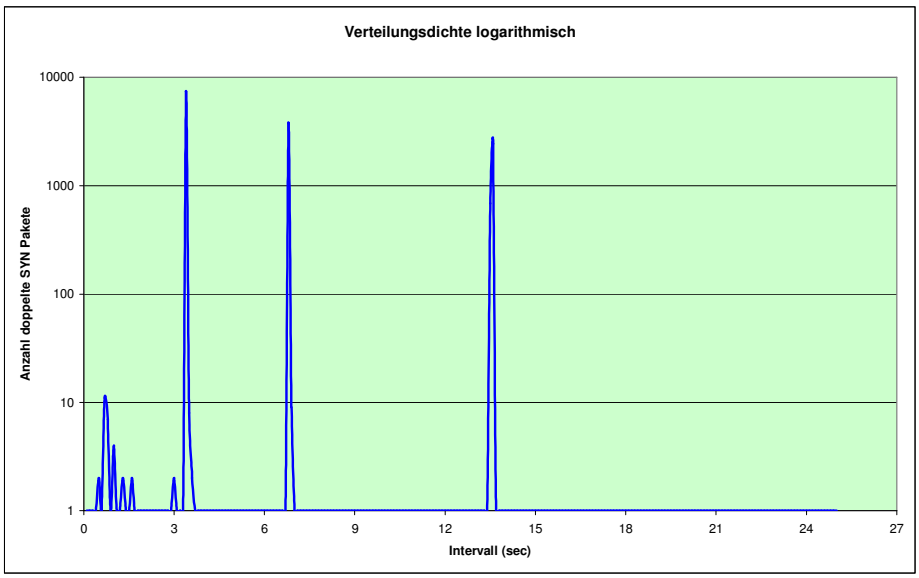

Abbildung 31. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket logarithmisch

In der logarithmisch skalierten Abb. 31 fällt auf, dass nur sehr wenige Verbindungen nicht im „automatischen Intervall“ zu finden sind, davon alle im Bereich zwischen 0,5s und 2s.

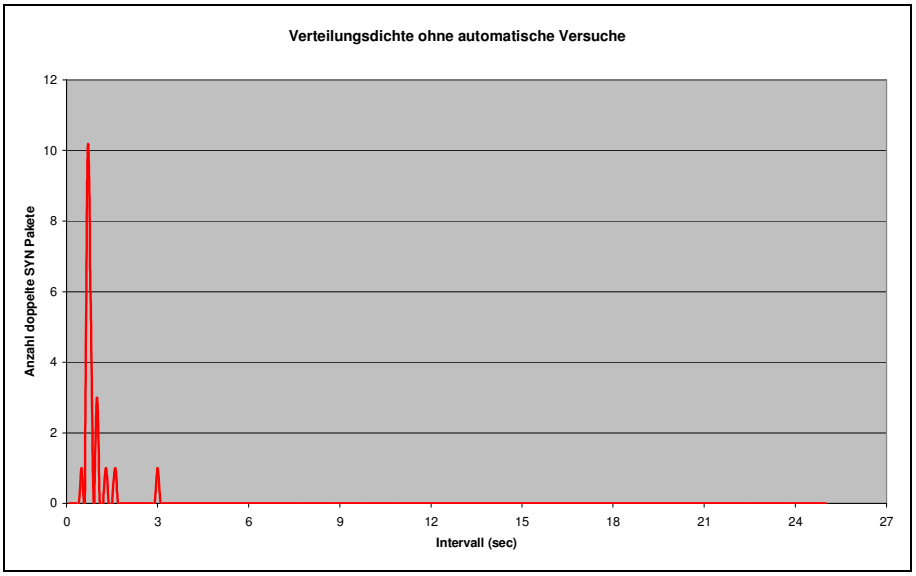

Abbildung 32. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket ohne die automatisch geschickten[1]

[1] Das Intervall für ein automatisch geschicktes Paket wurde hier an den RTO von 3,4s angepasst.

In Abb.32 sieht man dies noch einmal verdeutlicht, da die offensichtlich automatisch durch den Retransmission Timer neu geschickten Pakete herausgefiltert wurden.

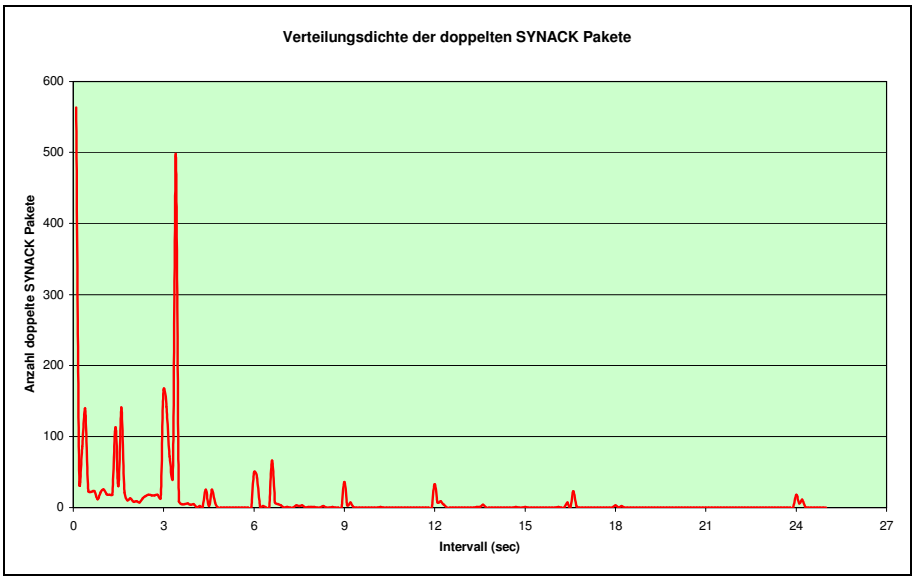

Abbildung 33. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket

Anders sieht es da schon bei den SYN/ACK Paketen aus, wie man in Abb. 33 sehen kann. Wesentlich mehr Spitzen sind hier zu sehen, besonders im an den Positionen bei 0 bis 0,2s und bei etwa 1,5s. Allerdings gibt es in Summe um ein Vielfaches mehr SYN Pakete als SYN/ACK Pakete. Dieses Phänomen konnte man schon auf dem Alupress Internet Proxy Server feststellen, was zu der Vermutung führt, dass dies durch die Messungen am Client begründet ist.

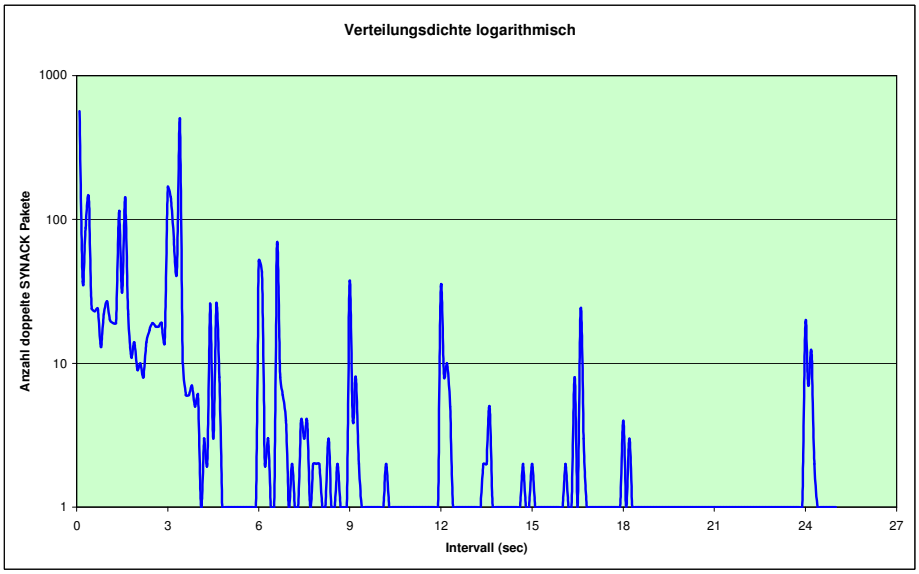

Abbildung 34. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket logarithmisch

In Abb.34 ist die Anzahl der mehrfachen SYN/ACK Pakete nochmals logarithmisch dargestellt. Auffallend wiederum die Spitzen bei 1,5s, 9s 16,5 und 18s. Da es hierbei keine zugehörigen SYN Pakete gibt, muss auch hierbei von einer fehlerhaften Einhaltung des Retransmission Timers ausgegangen werden.

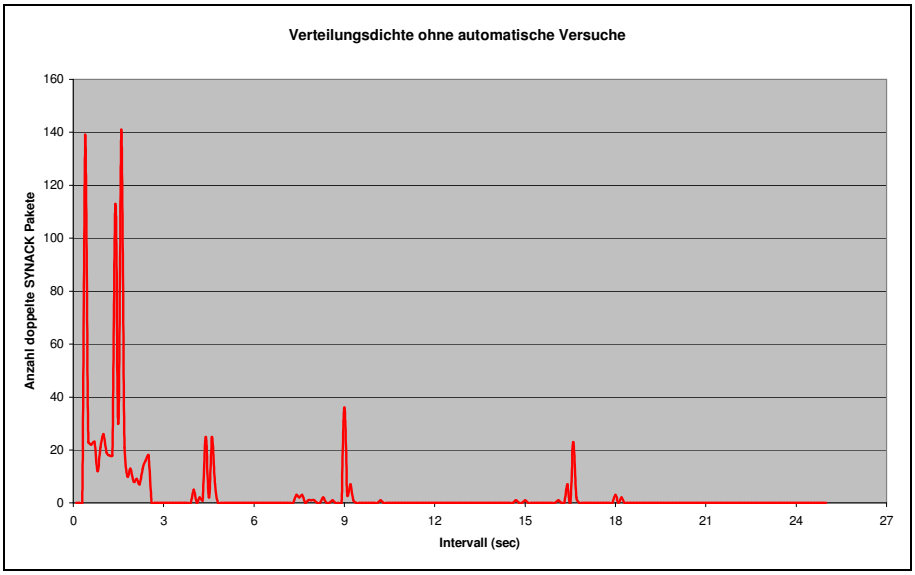

Abbildung 35. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket ohne die automatisch geschickten[2]

In Abb. 35 sieht man diese Spitzen noch einmal verdeutlicht. Ein Teil der Pakete in Intervall zwischen 0 und 2,5 Sekunden könnten manuell ausgelöst worden sein, da es hier im selben Intervall SYN Pakete gibt.

[2] Das Intervall für die automatisch geschickten Pakete wurde hierbei auf 2,5s bis 3,9s und deren Vielfache erweitert, da einerseits ein doppeltes SYNACK durch ein erneutes Übetragen des Empfängers ausgelöst werden könnte, andererseits der Sender durch ein erneutes SYN das Senden eines erneuten SYNACKs auslösen könnte. Da Sender und Empfänger unterschiedliche RTO verwenden, muss das Intervall dementsprechend vergrößert werden.

5.1.4 Freie Logfiles vom LBNL/ICSI Enterprise Tracing Project [2]

Die Tracefiles vom LBNL/ICSI Enterprise Tracing Project sind eine Mischung aus Messungen am Server und Messungen am Client, da hier der gesamte Verkehr im Netzwerk der Firma überwacht wurde.

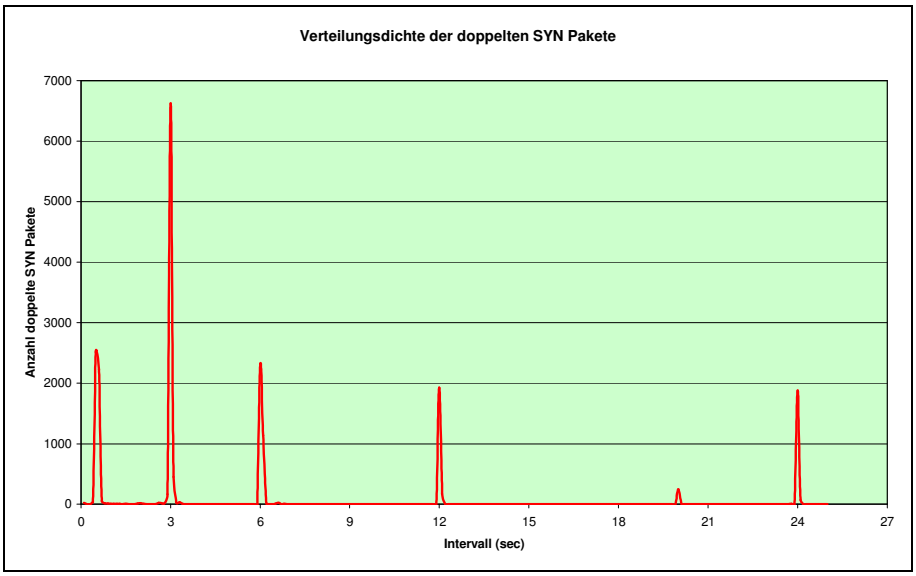

Abbildung 36. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket

Abb.36 zeigt die Verteilung der doppelt versendeten SYN Pakete. Sehr auffällig ist die Häufung der doppelten Pakete bei ca. 0,5s. Da hier derart viele SYN Pakete auftreten, wird es sich hier wohl nicht um manuell ausgelöste Verbindungsversuche handeln, sondern eher ein Programm bzw. ein geänderter RTO dafür verantwortlich sein. Wie man im logarithmischen Diagramm (Abb. 37) sehen kann gibt es bei etwa 20s noch einmal eine Spitze, deren Ursprung unklar ist.

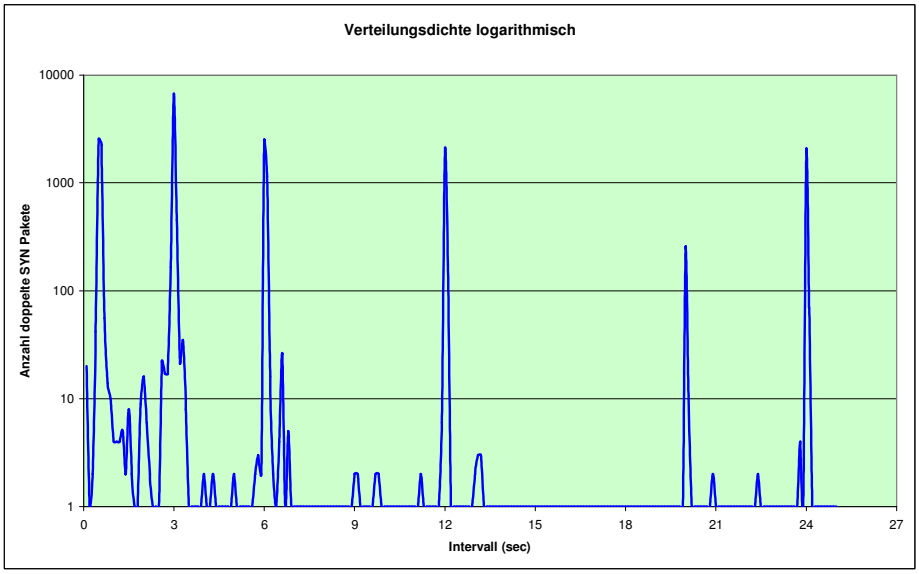

Abbildung 37. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket logarithmisch

Außerdem fällt auf, dass bei den Vielfachen des Standard RTO von 3s immer ca. gleich viele SYN Pakete verschickt wurden. Das könnte auf viele erfolglose Verbindungsversuche deuten.

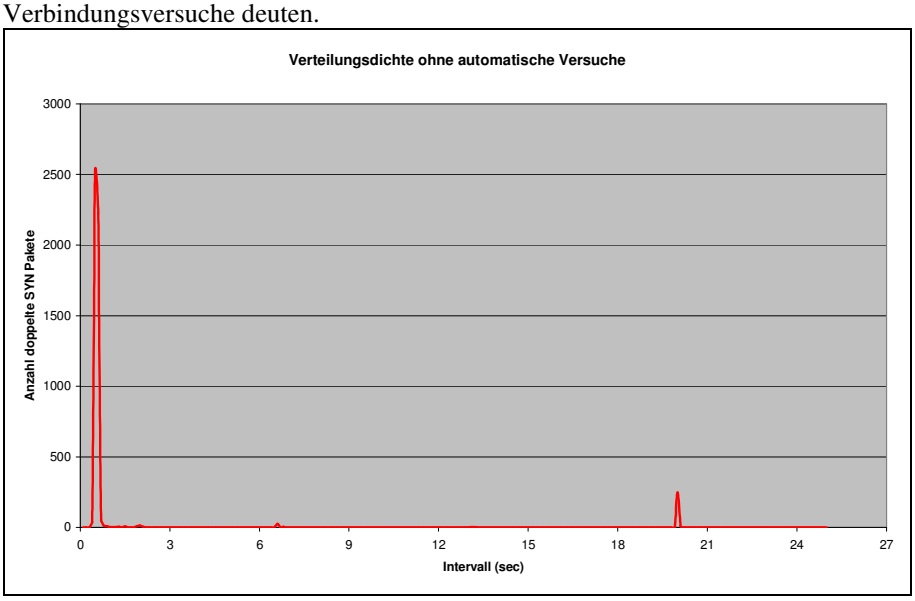

Abbildung 38. Doppelte SYN Pakete nach Abstand in Sekunden zum vorherigen SYN Paket ohne die automatisch geschickten

In Abb. 38 werden zur besseren Übersicht die offensichtlich automatischen Verbindungsversuche ausgeblendet. Allerdings deutet die Häufigkeit der SYN Pakete bei ca. 0,5s ebenfalls auf irgendeinen Automatismus hin.

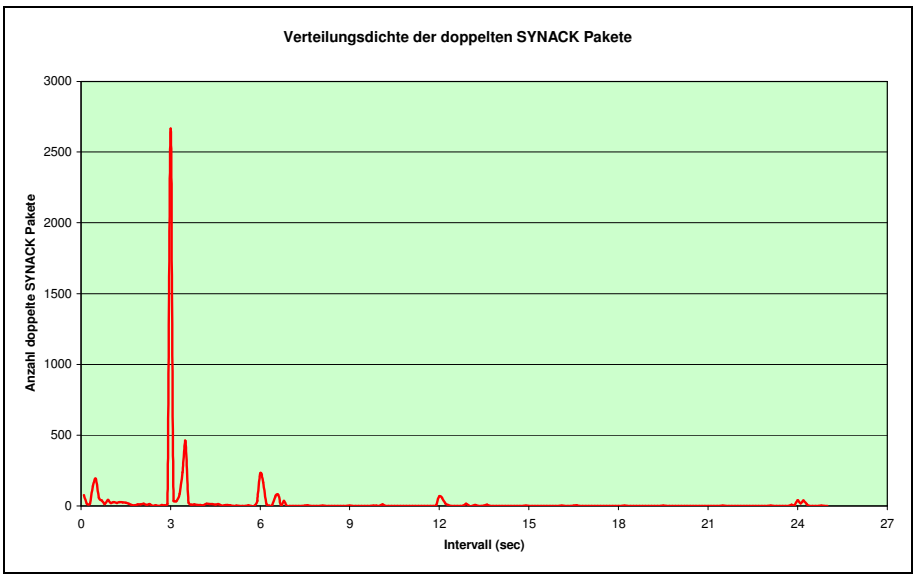

Abbildung 39. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket

Bei den SYN/ACK Paketen schaut es hingegen ganz anders aus. Nur bei 3s ist eine deutliche Spitze zu sehen. Außerdem gibt es bei 3,4s eine Spitze, was darauf hindeutet, dass es hier einen Server mit einer RTO von 3,4s im Netzwerk geben könnte, ähnlich dem Solaris Internet Proxy in der anderen Messung [siehe Kap. 5.1.3]. Außerdem unterstützt das häufige Auftreten von vielen SYN Paketen, aber

gleichzeitig kaum SYN/ACK Paketen die Theorie der vielen erfolglosen Versuche eines Verbindungsaufbaus. In Abb.40 kann man das nochmals mit logarithmischer Darstellung beobachten.

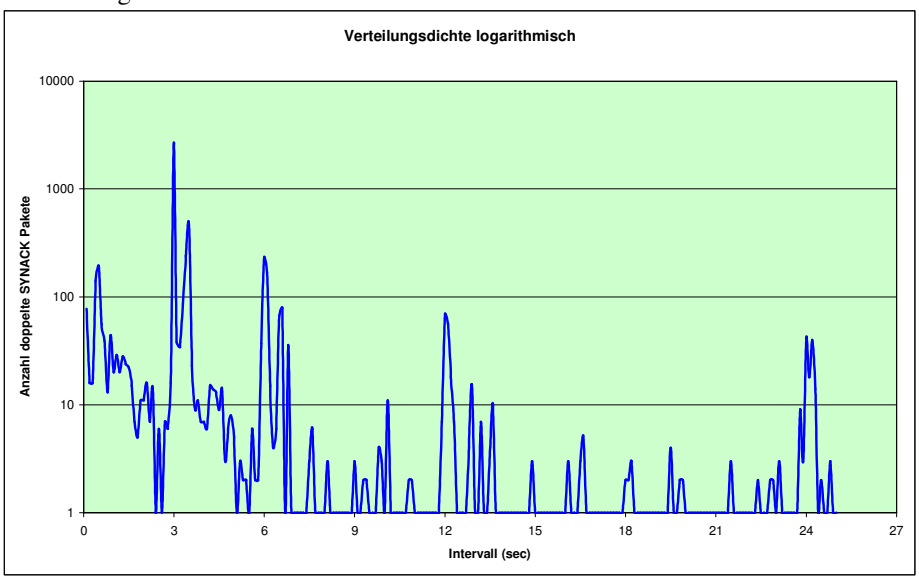

Abbildung 40. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket logarithmisch

In Abb.41 fällt die Spitze bei ca.0.7s auf. Da auch bei den SYN Paketen hier eine Häufung auftritt, könnte es eine Mischung von durch einen User ausgelösten Paketen (durch Klick auf Refresh oder Doppelklick) und durch eine Software ausgelösten Paketen sein. Außerdem gibt es eine Häufung von SYN/ACK Paketen bei ca. 6.8s. Dies deutet aber auf einen Server mit höherem RTO Timer hin, sodass es sich hier wahrscheinlich nicht um manuell ausgelöste Pakete handelt.

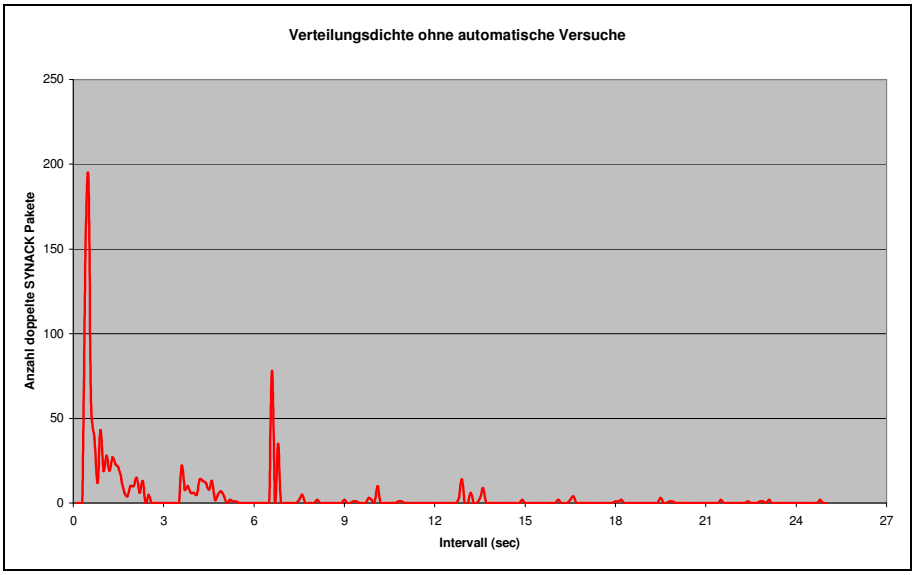

Abbildung 41. Doppelte SYN/ACK Pakete nach Abstand in Sekunden zum vorherigen SYN/ACK Paket ohne die automatisch geschickten

5.2 Anzahl erfolgreich aufgebaute Verbindungen

In den nachfolgenden Grafiken sind die Anzahl der erfolgreichen Verbindungen dargestellt. Auf der x-Achse sind die Anzahl an notwendigen SYN bzw. SYN/ACK Paketen dargestellt um eine erfolgreiche Verbindung zu ermöglichen. Auf der y-Achse ist die Anzahl an Verbindungen aufgetragen. Auch hierbei wurde wieder eine logarithmische Darstellung verwendet, um die Leserlichkeit zu erleichtern.

Um einen besseren Eindruck über die Werte zu erhalten, gibt es zu jeder Grafik mit Absolutwerten eine prozentuelle Darstellung.

Damit die Diskussion der Grafiken leichter fällt, werden nachfolgend einige Abkürzungen eingeführt. So bezeichnet V1, die Anzahl an erfolgreichen Verbindungen, welche bereits mit einem Synchronisationspaket zustande gekommen sind. Analog dazu bezeichnet V2 also die Anzahl an Verbindungen, welche zwei Synchronisationspakete für einen erfolgreichen Verbindungsaufbau benötigen, usw…

5.2.1 Webmail Webserver Universität IBK (Server)

5.2.1.1 Lwm 1

In den nachfolgenden Diagrammen ist die Anzahl an erfolgreichen Verbindungsversuchen mit dem ersten Webmail Server der Universität Innsbruck dargestellt. Die Y-Achse ist logarithmisch skaliert, da die Werte sehr weit gestreut sind.

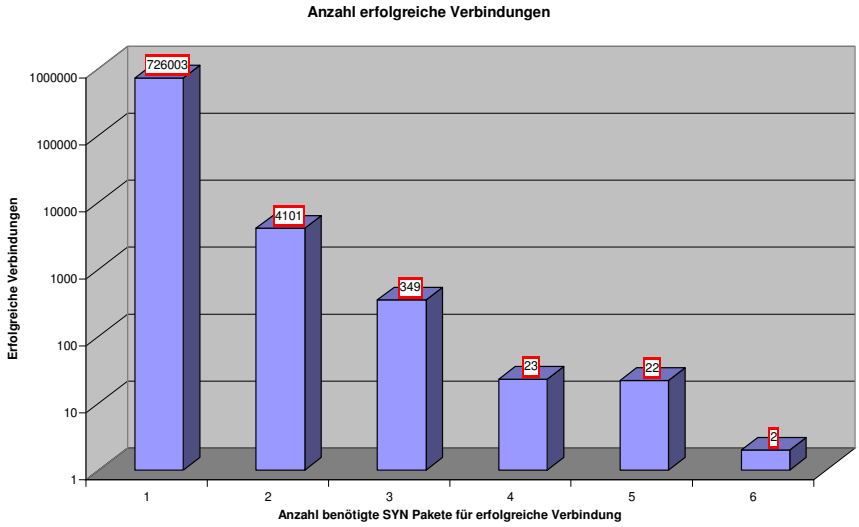

Abbildung 42. Erfolgreiche Verbindungen nach Anzahl geschickter SYN Pakete

In Abb. 42 sieht man, dass der Großteil der Verbindungsversuche bereits nach einem SYN Paket abgeschlossen ist. Es tritt selten auf, dass mehr als ein SYN Paket benötigt wird. Interessant ist, dass das Verhältnis von V1:V2 viel größer ist als z.B.: V2:V3.

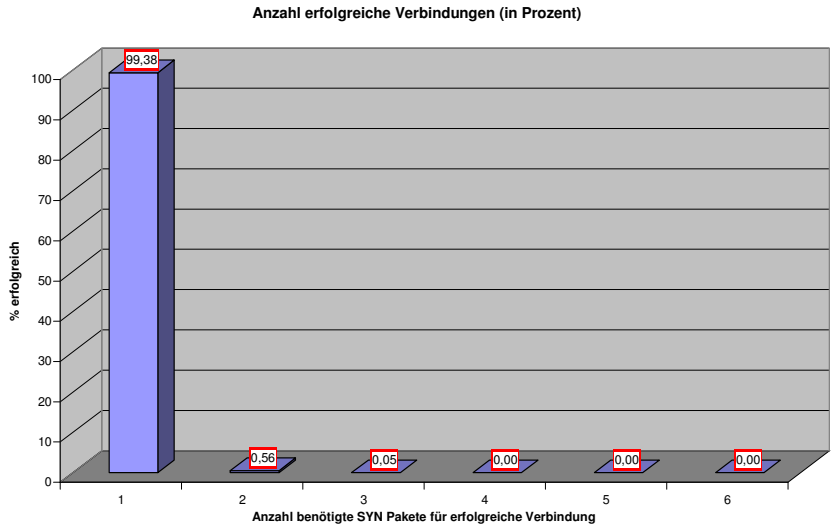

Abbildung 43. Erfolgreiche Verbindungen nach Anzahl geschickter SYN Pakete prozentuell

In Abb.43 ist die Messung prozentuell dargestellt. Dabei sieht man, dass 99,36% aller erfolgreichen Verbindungen bereits nach einem SYN Paket aufgebaut werden. Nur ein halber Prozent aller Verbindungen benötigt dafür zwei Pakete. Die Anzahl an Verbindungen mit mehr als zwei SYN Paketen bis zum erfolgreichen Verbindungsaufbau ist minimal.

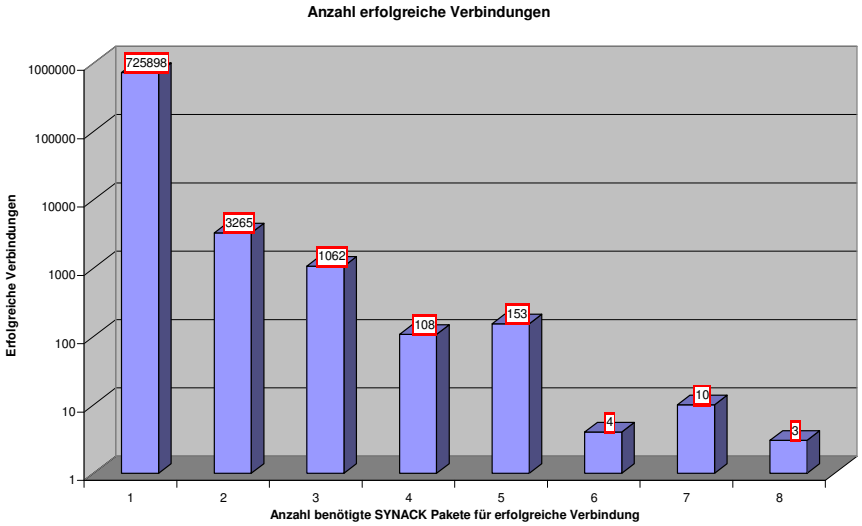

Abbildung 44. Erfolgreiche Verbindungen nach Anzahl geschickter SYN/ACK Pakete

Abb. 44 zeigt die Situation bei SYN/ACK Paketen. Auch hier benötigt der Großteil der Verbindungen nur ein SYN/ACK Paket um erfolgreich aufgebaut zu werden. Ebenfalls auffällig ist, dass das Verhältnis von V1:V2 viel größer ist, als V2:V3.

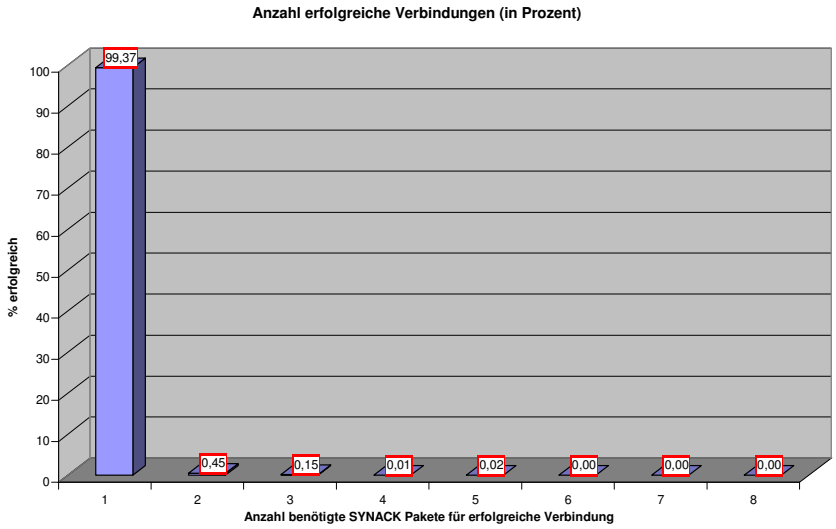

Abbildung 45. Erfolgreiche Verbindungen nach Anzahl geschickter SYN/ACK Pakete prozentuell

Prozentuell gesehen, schaut es bei den SYN/ACK Paketen in etwa gleich aus, wie bei den SYN Paketen. Nur etwa jede 230te Verbindung braucht mehr als ein SYN/ACK Paket um erfolgreich zu sein.

5.2.1.2 Lwm 2

In den nachfolgenden Diagrammen ist die Anzahl an erfolgreichen Verbindungsversuchen mit dem zweiten Webmail Server der Universität Innsbruck dargestellt.

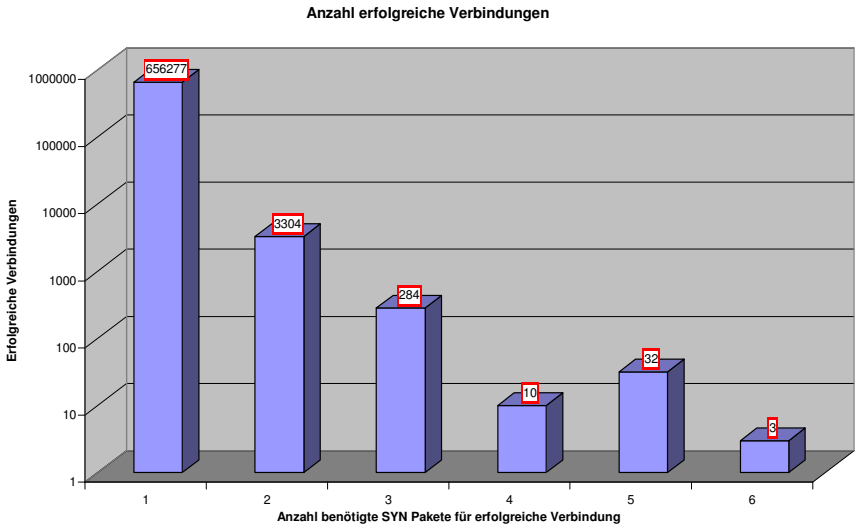

Abbildung 46. Erfolgreiche Verbindungen nach Anzahl geschickter SYN Pakete

Auch am zweiten Webmailserver ergibt sich ein ähnliches Bild wie am ersten Webmailserver. Der Großteil der Verbindungen steht nach einem SYN Paket. Das Verhältnis von V1:V2 ist ebenfalls viel größer als etwa V2:V3.

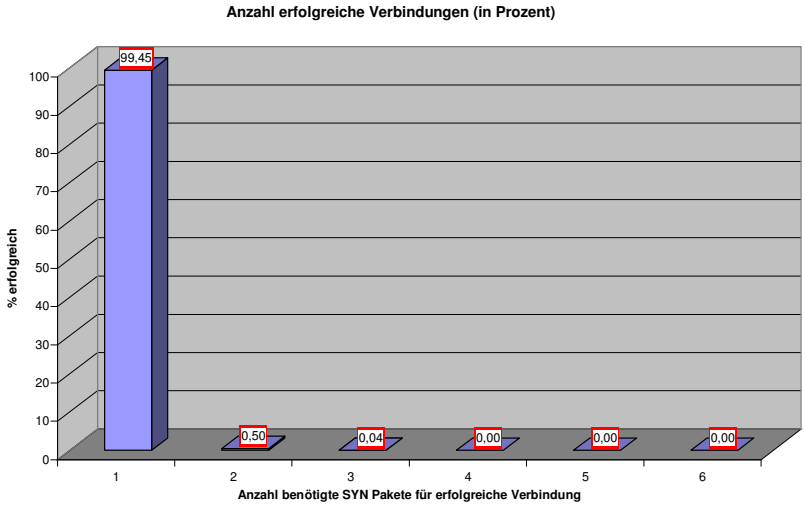

Abbildung 47. Erfolgreiche Verbindungen nach Anzahl geschickter SYN Pakete prozentuell

Prozentuell gesehen benötigt etwa eine von 200 Verbindungen mehr als ein SYN Paket für einen erfolgreichen Aufbau, wobei nur jede 2500te mehr als zwei Synchronisationspakete benötigt.

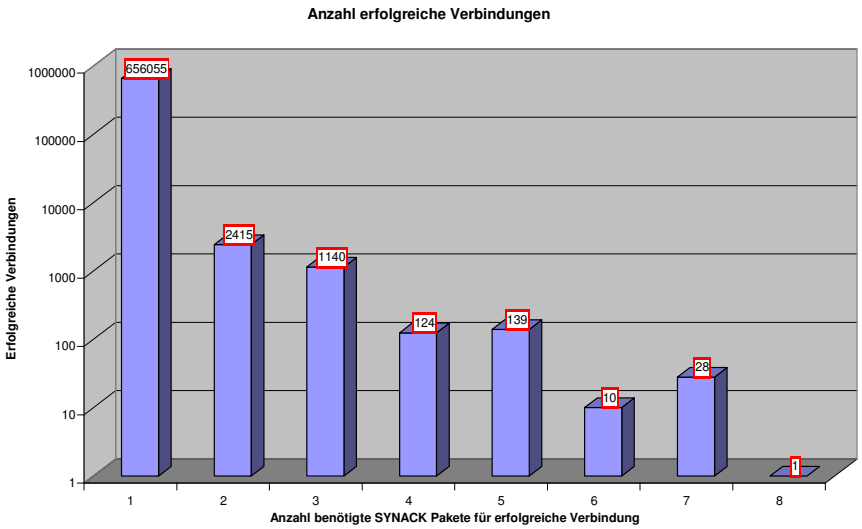

Abbildung 48. Erfolgreiche Verbindungen nach Anzahl geschickter SYN/ACK Pakete

Auch bei der Betrachtung der Verbindungsaufbauten nach SYN/ACK Paketen ergibt sich ein ähnliches Bild wie bei Webmailserver eins. Das Verhältnis von V1:V2 ist auch hier wesentlich größer als von V2:V3.

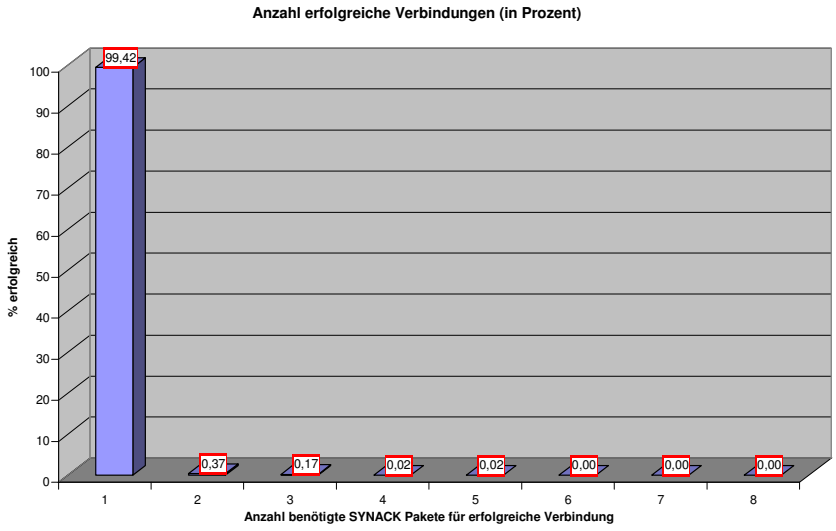

Abbildung 49. Erfolgreiche Verbindungen nach Anzahl geschickter SYN/ACK Pakete prozentuell

Prozentuell gesehen (Abb.49) gibt es keine Abweichungen zu den vorigen Messwerten. Etwa jede 230te Verbindung ist erst nach mehr als einem SYN/ACK erfolgreich. Auffallend: wesentlich mehr Verbindungen werden erst nach drei SYN/ACK Paketen aufgebaut als nach drei SYN Paketen.

5.2.2 Internet Proxy Server Fa. Alupress AG Brixen (Client)

Die Messung am Internet Proxy Server der Fa. Alupress Brixen ermöglicht, die Messung auch für die Clientseite zu analysieren.

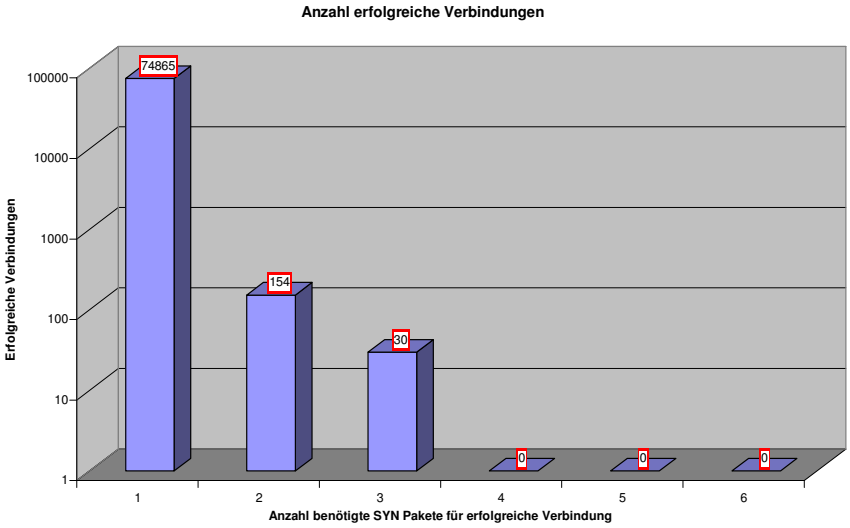

Abbildung 50. Erfolgreiche Verbindungen nach Anzahl geschickter SYN Pakete

Am Auffallendsten ist in Abb. 50, dass bei keiner erfolgreichen Verbindung mehr als drei SYN Pakete geschickt wurden. Das bestätigt die Standard Windows Einstellung von maximal zwei Wiederholungen bei Paketverlust beim Verbindungsaufbau. Zusätzlich ist auch hier das Verhältnis von V1:V2 viel größer als V2:V3.

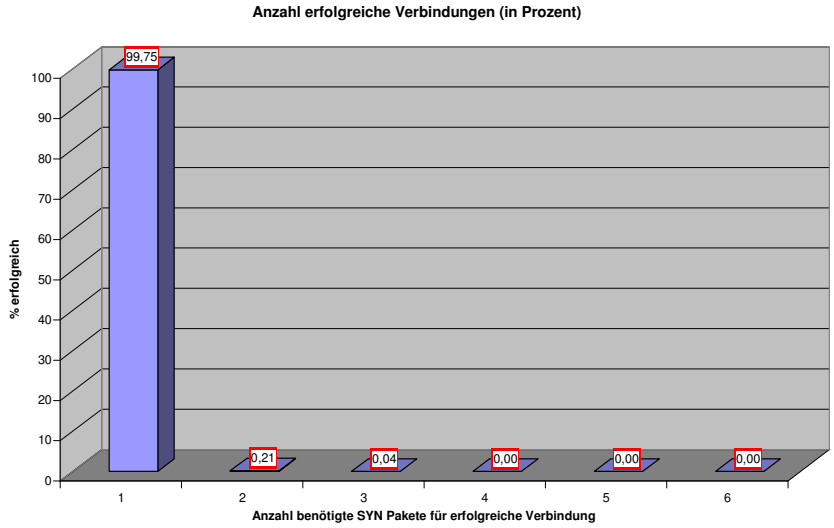

Abbildung 51. Erfolgreiche Verbindungen nach Anzahl geschickter SYN Pakete prozentuell

Nach relativer Häufigkeit in Prozent schaut es so aus, dass es hier weniger Verbindungsaufbau mit Paketverlust gibt. Nur etwa jede 470ste Verbindung braucht zwei SYN Pakete, nur jede 2500ste braucht drei Synchronisationspakete.

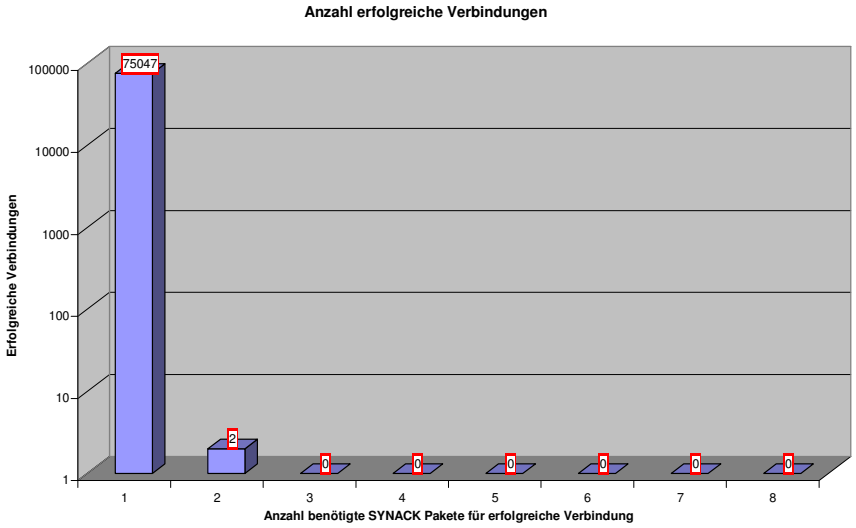

Abbildung 52. Erfolgreiche Verbindungen nach Anzahl geschickter SYN/ACK Pakete

Ein Blick auf die Anzahl der SYN/ACK Pakete in Abb. 52 zeigt, dass diese fast überhaupt nicht erneut gesendet werden. Das kann bedeuten, dass sie entweder alle auf dem Weg zum Internet Proxy verloren gehen, oder die gesendeten SYN Pakete gar nicht erst ankommen.

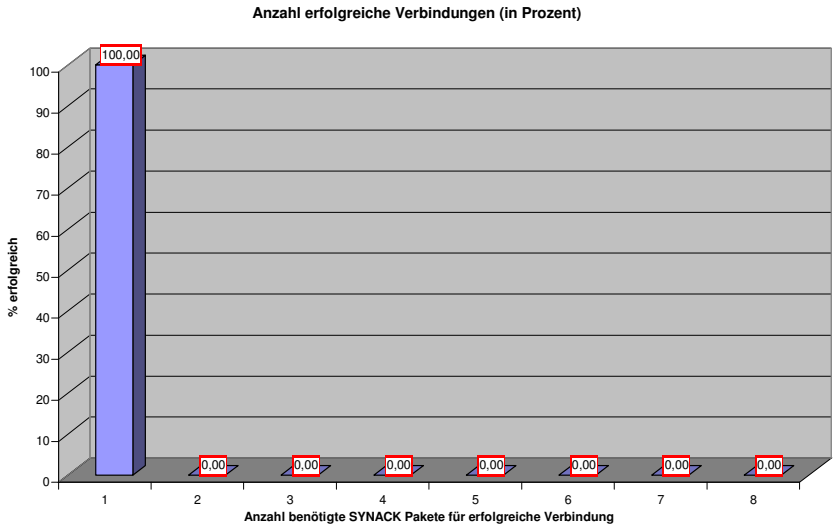

Abbildung 53. Erfolgreiche Verbindungen nach Anzahl geschickter SYN/ACK Pakete prozentuell

Prozentuell (Abb. 53) gesehen sind alle Verbindungen nach einem empfangenen SYN/ACK Paket erfolgreich aufgebaut. Die Anzahl an Verbindungen mit mehr empfangenen SYN/ACK Paketen ist so gering, dass sie aufgrund der Rundung der Prozentwerte in der Grafik nicht mehr sichtbar sind.

5.2.3 Internet Proxy Server Universität IBK (Client)

Die Daten des Internet Proxy Servers der Universität IBK sind zwar umfangreicher als die des Alupress Internet Proxy Servers, die Server verwenden aber auch ein anderes Betriebssystem.

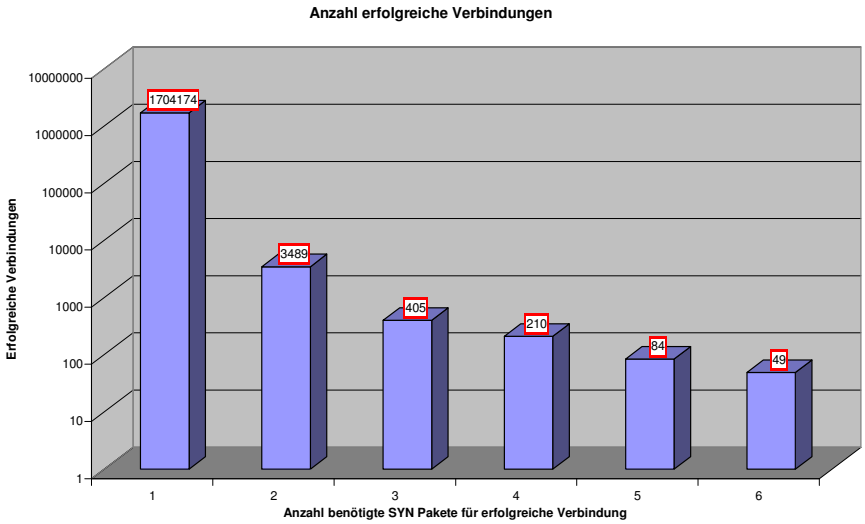

Abbildung 54. Erfolgreiche Verbindungen nach Anzahl geschickter SYN Pakete

In Abb. 54 sieht man, wie viele SYN Pakete hier für eine erfolgreiche Verbindung benötigt wurden. Wie bereits bei allen vorhergegangenen Messungen ist das

Verhältnis von V1:V2 wesentlich größer als V2:V3 und folgende. Außerdem lässt sich feststellen, dass das Solaris Betriebssystem erst nach mehreren Verbindungsversuchen aufgibt. Laut Messung sind es bis zu fünf Wiederholungen.

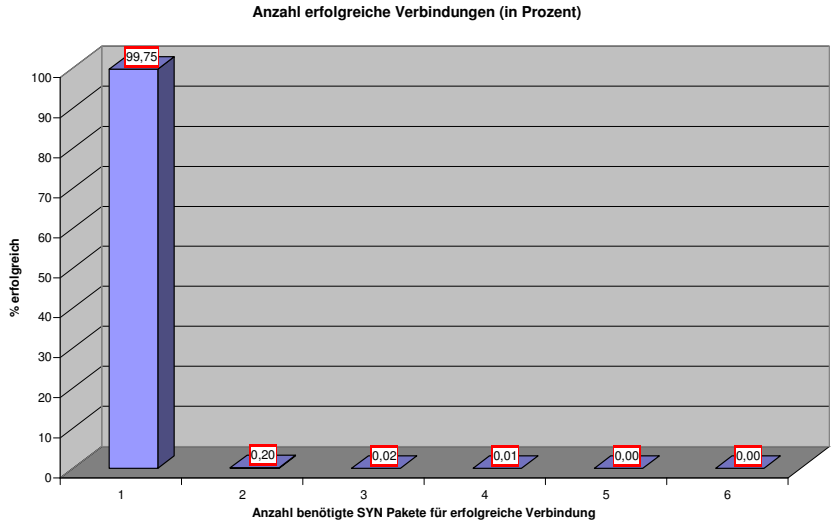

Abbildung 55. Erfolgreiche Verbindungen nach Anzahl geschickter SYN Pakete prozentuell

Prozentuell dargestellt ergibt sich in Abb.55 ein ähnliches Bild wie beim Alupress Proxy Server. Nur jede 500ste erfolgreiche Verbindung braucht zwei SYN Pakete, etwa jede 5000ste drei SYN Pakete.

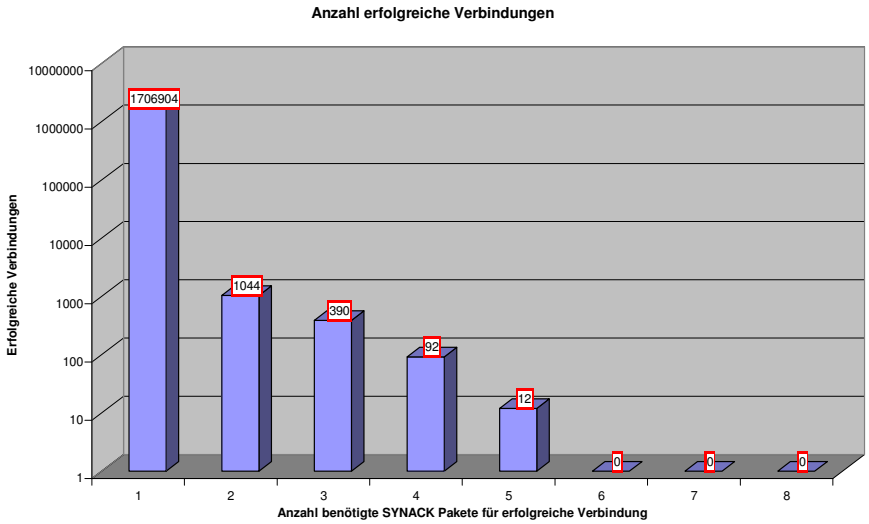

Abbildung 56. Erfolgreiche Verbindungen nach Anzahl geschickter SYN/ACK Pakete

In Abb.56 ist die Messung aus Sicht der benötigten SYN/ACK Pakete dargestellt. Es fällt auf, dass die pro erfolgreiche Verbindung weniger SYN/ACK als SYN Pakete benötigt werden.

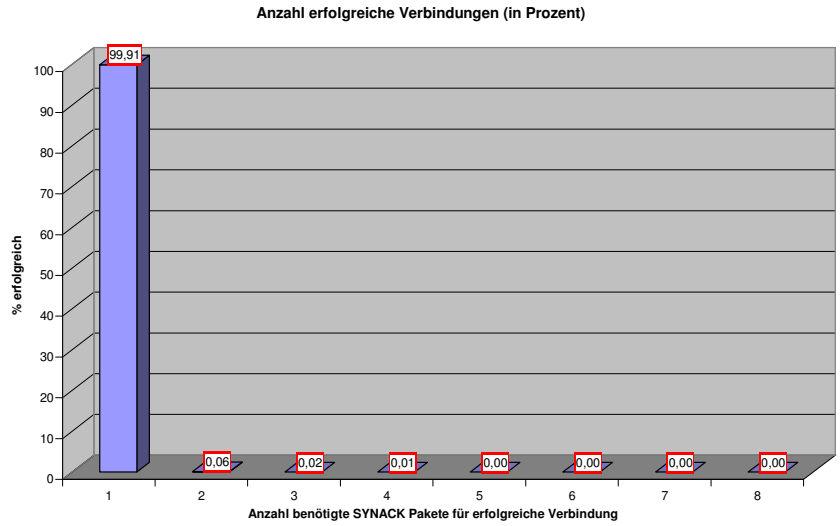

Abbildung 57. Erfolgreiche Verbindungen nach Anzahl geschickter SYN/ACK Pakete prozentuell

In Abb.57 ist dies noch einmal prozentuell dargestellt. Hier sieht man, dass kaum eine Verbindung mehr als SYN/ACK Paket benötigt. Da es sich hierbei aber um eine Messung am Client handelt, kann es sein, dass die Pakete unterwegs verloren gegangen sind und deshalb nicht gemessen wurden.

5.2.4 Freie Logfiles vom LBNL/ICSI Enterprise Tracing Project [2]

Zuletzt noch die Betrachtung der erfolgreichen Verbindungen beim LBNL/ICSI Tracing Project.

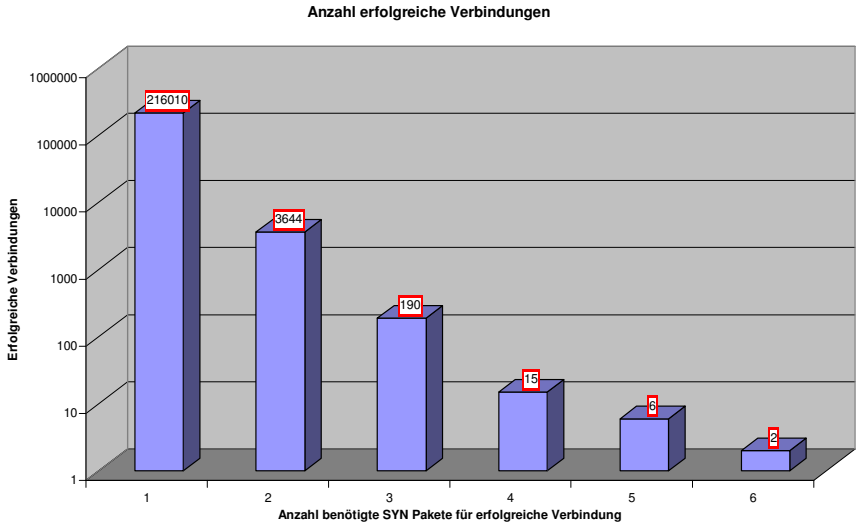

Abbildung 58. Erfolgreiche Verbindungen nach Anzahl geschickter SYN Pakete

Auffallend ist, wie bereits bei den Messungen zuvor, dass das Verhältnis V1:V2 größer ist als V2:V3. Der Fakt, dass bis zu fünf Wiederholungen auftreten lässt ahnen, dass hier Linux oder Solaris Rechner im Spiel waren, denn aus den vorherigen

Messungen und der Literatur ist ersichtlich, dass Windows Rechner normalerweise nur zwei Wiederholungen senden.

Prozentuell ist ein deutlicher Unterschied zu den vorhergehenden Messungen festzustellen (Abb.59): Es tritt hier Paketverlust viel häufiger auf. So benötigt bereits jede 57te Verbindung mehr als ein SYN Paket.

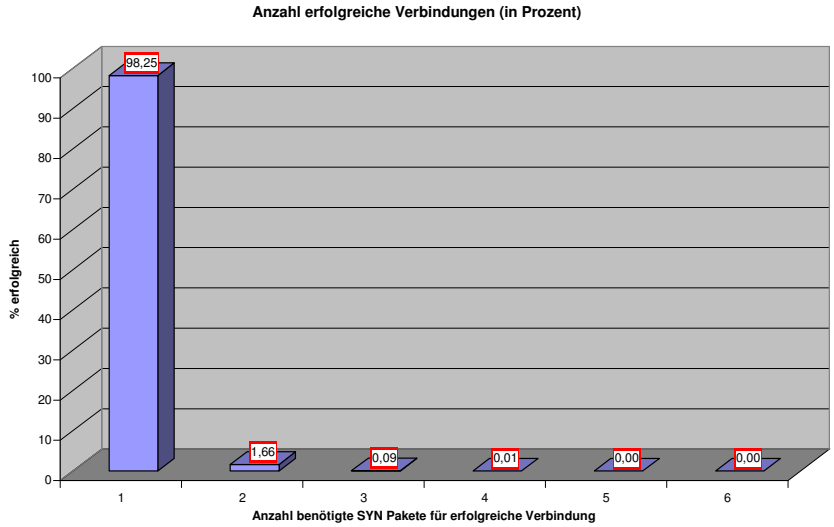

Abbildung 59. Erfolgreiche Verbindungen nach Anzahl geschickter SYN Pakete prozentuell

Eine ähnliche Situation kann man in Abb.60 bei den SYN/ACK Paketen feststellen. Hier gibt es sogar Verbindungen, die bis zu 7 Wiederholungen senden. Von der Anzahl her sind die Anzahl der SYN/ACK Pakete leicht über der Anzahl an SYN Paketen.

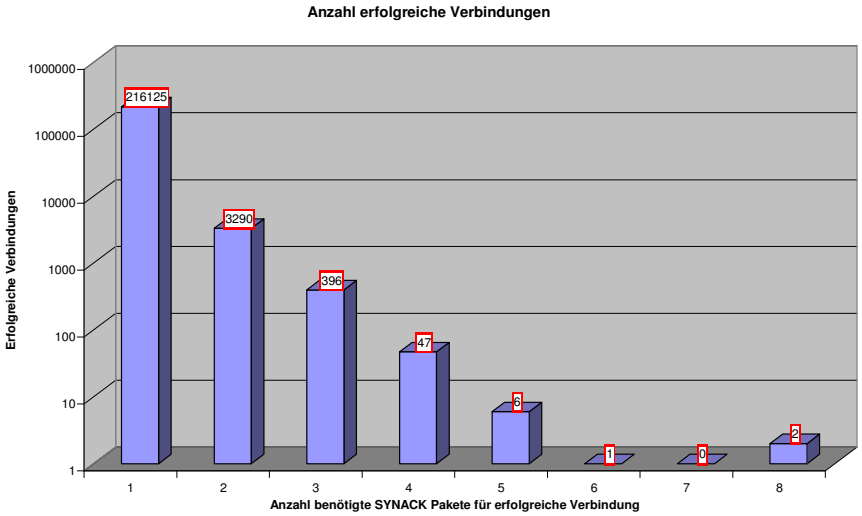

Abbildung 60. Erfolgreiche Verbindungen nach Anzahl geschickter SYN/ACK Pakete

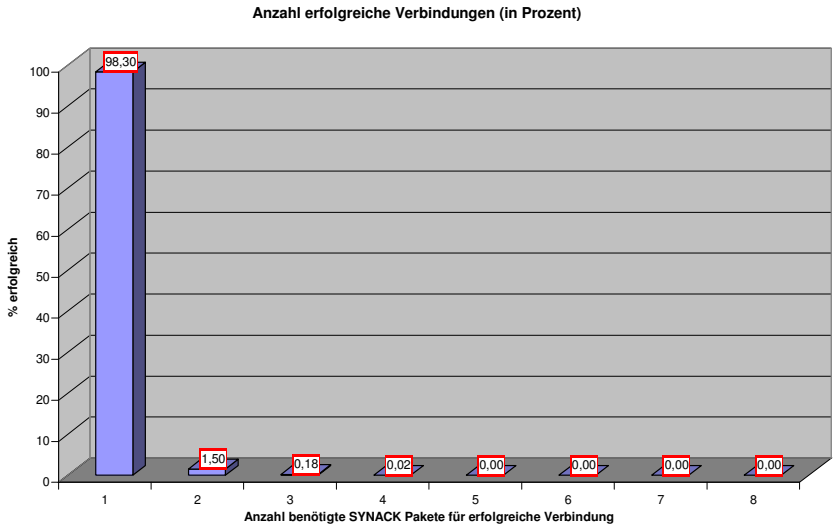

Abbildung 61. Erfolgreiche Verbindungen nach Anzahl geschickter SYN/ACK Pakete prozentuell

Prozentuell gesehen ist das Ergebnis dieser Messung ähnlich wie bei den SYN Paketen. Auch hier sind viele Verbindungen erst nach mehreren SYN/ACK Paketen erfolgreich aufgebaut; so braucht etwa jede 66ste Verbindung ein zweites SYN/ACK Paket.

6 Auswertung

Die Messungen und Statistiken sind in Tab.1 und Tab.2 zusammengefasst.

	Verbindungsversuche	Verbindungen erfolgreich	Verbindungen nicht erfolgreich	mehrfache SYNs	mehrfache SYNs bei erfolgreichen Verbindungen	mehrfache SYNs bei nicht erfolgreichen Verbindungen	mehrfache SYNs, die nicht im automatischen Intervall liegen	Wieviele SYNs pro erfolgreiche Verbindung?	Doppeltes SYN Paket bei jeder x ten Verbindung
ZID Webmail LWM1	737.188	730.523	6.665	5.968	5.162	806	690	1,007	142
ZID Webmail LWM2	665.829	659.913	5.916	4.838	4.064	774	577	1,006	162
Alupress Proxy Server	75.493	75.049	444	480	214	266	48	1,003	351
ZID Proxy Server	1.721.382	1.708.442	12.940	19.452	5.696	13.756	23	1,003	300
LBNL/ICSI	232.852	219.867	12.985	22.268	4.103	18.165	428	1,019	54
Summe	**3.432.744**	**3.393.794**	**38.950**	**53.006**	**19.239**	**33.767**	**1.766**	**1,006**	**176**

Tabelle 1. Auswertung doppelte SYN Pakete

	Verbindungsversuche	Verbindungen erfolgreich	Verbindungen nicht erfolgreich	mehrfache SYNACKs	mehrfache SYNACKs bei erfolgreichen Verbindungen	mehrfache SYNACKs bei nicht erfolgreichen Verbindungen	mehrfache SYNACKs, die nicht im automatischen Intervall liegen	Wieviele SYNACKs pro erfolgreiche Verbindung?	Doppeltes SYNACK Paket bei jeder x ten Verbindung
ZID Webmail LWM1	737.188	730.523	6.665	19.762	6.586	13.176	2.865	1,009	111
ZID Webmail LWM2	665.829	659.913	5.916	18.206	5.856	12.350	3.009	1,009	113
Alupress Proxy Server	75.493	75.049	444	314	2	312	1	1,000	37.525
ZID Proxy Server	1.721.382	1.708.442	12.940	7.725	2.148	5.577	889	1,001	795
LBNL/ICSI	232.852	219.867	12.985	10.579	4.266	6.313	1.046	1,019	52
Summe	**3.432.744**	**3.393.794**	**38.950**	**56.586**	**18.858**	**37.728**	**7.810**	**1,006**	**180**

Tabelle 2. Auswertung doppelte SYN/ACK Pakete

Die Ergebnisse der Messungen lassen sich in den nachfolgenden Punkten festhalten:

- Etwa bei jeder 180sten erfolgreichen Verbindung wird ein zusätzliches SYN und SYN/ACK Paket aufgrund von Paketverlust versendet. Man kann davon ausgehen, dass dies eine zusätzliche Wartezeit von mindestens drei Sekunden bedeutet.
- Die Tatsache, dass das Verhältnis von V1:V2 wesentlich größer ist als von V2:V3, spricht dafür, dass Paketverlust nicht ganz zufällig auftritt. Wenn der Paketverlust zufällig wäre, so müsste das Verhältnis konstant bleiben, da sich die Wahrscheinlichkeiten des Verlustes von einem, zwei und drei Paketen miteinander multiplizieren. Ein Grund dafür könnte eine zeitweise Überlastung des Netzes oder des Empfängers sein.
- In Tab.1 und Tab.2 lässt sich feststellen, dass mehr SYN/ACK Pakete als SYN Pakete außerhalb des automatischen Intervalls auftreten. Es müsste aber eine

durch den User manuell „refreshte“ Verbindung immer mit einem SYN Paket starten. Daher ergibt sich die Behauptung, dass diese SYN/ACK Pakete nicht alle manuell ausgelöst wurden, sondern aus anderen Gründen außerhalb der automatischen Intervalle liegen. Der Hauptgrund könnte eine hohe Auslastung der Server, welche die SYN/ACK Pakete schicken, sein. Hohe Auslastung könnte eine Verzögerung der Abarbeitung der Anfragen erklären. Dadurch könnten die Pakete aus den gewählten Intervallen herausfallen.

- Da sich anhand der Messung nicht feststellen lässt, ob ein Paket automatisch oder manuell wiederholt versendet wurde, sind die so genannten automatischen Intervalle intuitiv gewählt. Daher sind die Messergebnisse nur ungefähre Werte und nicht hundertprozentig korrekt. Da aber über 1700 SYN Pakete nicht in der Nähe des RTO Timers und dessen Vielfachen liegen ist davon auszugehen, dass zumindest einige davon manuell ausgelöst wurden und es somit bei Usern zu sichtbaren Wartezeiten gekommen ist.

7 Zusammenfassung

Die umfangreichen Messungen und deren Auswertung und Interpretation erlauben die Feststellung, dass es häufig (ca. 0,5%) zu Paketverlust beim Verbindungsaufbau und damit verbundenen Wartezeiten kommt. Inwieweit der User diesen Verlust feststellt und darauf mit einem Klick auf den „Refresh Button“ reagiert lässt sich nicht eindeutig sagen. Allerdings kann man davon ausgehen, dass zumindest bei einem Teil der erneut gesendeten Pakete der User eingegriffen hat, da diese zu einem Zeitpunkt geschickt wurden, der nicht in der Nähe eines automatischen Intervalls liegt.

Da der Paketverlust nicht selten auftritt scheint eine Optimierung des Verhaltens von TCP in Verbindung mit http Anfragen sinnvoll. Da dies den Rahmen dieser Arbeit sprengen würde, wäre es sinnvoll dies in einer Nachfolgearbeit zu versuchen. Die dafür notwendigen Messergebnisse stehen dank dieser Arbeit in ausrechendem Maße zur Verfügung.

Danksagung

An dieser Stelle möchte ich mich bei den Personen und Instituten bedanken, die mir bei der Beschaffung der Messwerte, sowie bei der Analyse der Messwerte geholfen haben. Dies ist zum einen mein Betreuer Michael Welzl.

Besonderer Dank gilt dem ZID der Universität Innsbruck, allen voran Walter Müller, welcher die Messungen an den beiden Webmail Servern und dem Internet Proxy Server koordinierte bzw. durchführte.

Dank auch an die Firma Alupress, die mir erlaubte, Messungen an ihrem Internet Proxy Server durchzuführen.

Literatur und Quellen

1. „Transmission Control Protokoll“ Artikel deutsche Wikipedia. Auf http://de.wikipedia.org/wiki/Transmission_Control_Protocol, zuletzt besucht am 22.01.2008
2. “The LBNL/ICSI Enterprise Tracing Project” auf http://www.icir.org/enterprise-tracing/index.html, zuletzt besucht am 22.01.2008
3. Microsoft Knowledge Base Artikel zum Ändern des Startwerts der RTT http://support.microsoft.com/kb/q223450/, zuletzt besucht am 22.01.2008
4. Aleksandar Kuzmanovic, Department of Computer Science, Northwestern University, “The Power of Explicit Congestion Notification”, 23.08.2005
5. V. Paxson and M. Allman. Computing TCP’s retransmission timer, RFC 2988, 11.2000
6. Braden, R., "Requirements for Internet Hosts -- Communication Layers", STD 3, RFC 1122, 10.1989
7. Prof. Plate Jürgen, Grundlagen Computernetzwerke auf http://www.netzmafia.de/skripten/netze/netz8.html, zuletzt aktualisiert am 27.08.2007, zuletzt besucht am 22.01.2008
8. Linux Kernel 2.6, speziell „tcp.h“
9. Alexander Keller und Maik Burandt, Spezielle Techniken der Rechnerkommunikation – Projektarbeit „Transmission Control Protocol“, http://www.ak-solution.de/downloads/uni/TCP.pdf , 13.06.2007
10. Junker Holger, Fakultät für Informatik Universität Karlsruhe, „Proseminar Rechnerkommunikation und Telefon“ auf http://goethe.ira.uka.de/seminare/rkt/tcp+udp/, zuletzt geändert 07.02.2001, zuletzt besucht am 22.01.2008
11. "Transmission Control Protocol," J. Postel, RFC-793, 09.1981

Anhang A

Anweisungen für das ZID

1 Webmail Server Testmessung

Die erste Testmessung wurde vom ZID anhand dieser Anweisungen durchgeführt. Das ZID hat hierbei selbstständig ein Script geschrieben, das diese Anweisungen der Reihe nach ausführt.

1.) Tcpdump auf dem Webserver mit folgenden Optionen starten:
tcpdump –i „interface" –w tcpdump.dump '(tcp[13] == 2 and dst host "ip_webserver" and dst port 80) or (tcp[13] == 18 and src host "ip_webserver" and src port 80) or (tcp[13] == 16 and dst host "ip_webserver" and dst port 80)'

Beispiel: *tcpdump –i eth0 –w tcpdump.dump '(tcp[13] == 2 and dst host 138.232.1.217 and dst port 80) or (tcp[13] == 18 and src host 138.232.1.217 and src port 80) or (tcp[13] == 16 and dst host 138.232.1.217 and dst port 80)'*

Das Programm sollte etwa 4 bis 5 Tage laufen.
Das Dumpfile bitte ein paar Tage aufbewahren, damit man bei Bedarf das Programm zum Erstellen der Statistiken nochmals darüber laufen lassen kann.

2.) Das Dumpfile „tcpdump.dump"nehmen und in den selben Ordner wie das Program „tcp_syn_stat.out" kopieren

3.) "./tcp_syn_stat.out > stats.csv" laufen lassen

4.) „stats.csv" Datei an benjamin.kaser@student.uibk.ac.at mailen

2 Webmail Server Langzeitmessung

Da die Testmessung erfolgreich war, wurde das Auswerteprogramm erweitert (nennt sich nun TSYN_Stat.c). Der Aufzeichnungs- und Auswerteprozess wurde weitgehend mit einem Script automatisiert.
Das ZID wurde gebeten das Script auf Ihren Webmail Servern über zwei Wochen laufen zu lassen und am Ende die Ergebnisse zur Verfügung zu stellen. Die Scripts sind in Kap. 3.1 aufgeführt.

3 Internet Proxy Server Messung

1. Ip Adresse im Job für Solaris pflegen und laufen lassen (erstellt einen Ordner raw_traces). Aufruf: ./job_solaris

2. Auf Linux Maschine beliebigen Ordner erstellen (z.b "main") und dort den linux job "job_linux", sowie das Program TSYN_Stat.out reinkopieren

3. In den eben erstellten Ordner "main" den "raw_traces" Ordner reinkopieren

4. linux job im Ordner "main" laufen lassen -> Parameterübergabe Anzahl wie viel snoop traces (letztes File zb. snoop49.sol -> Aufruf: ./job_linux 49)

5. es wird wieder ein Ordner "stats" erstellt. Diesen bitte mir zukommen lassen.

Anhang B

Programmcode TSYN_Stat.c

```
#define APP_NAME  "TSYN_stat"
#define APP_DESC  "Erstellt Statistiken zum Senden von SYN und SYN/ACK Paketen an
und von Webservern (Port 80)"
#define APP_COPYRIGHT   "Copyright (c) 2007 Kaser Benjamin"

#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* default snap length (maximum bytes per packet to capture) */
#define SNAP_LEN 1518

/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN  6

/* Ethernet header */
struct h_ethernet {
        u_char  ether_dhost[ETHER_ADDR_LEN];    /* destination host address */
        u_char  ether_shost[ETHER_ADDR_LEN];    /* source host address */
        u_short ether_type;                     /* IP? ARP? RARP? etc */
};

/* IP header */
struct h_ip {
        u_char  ip_vhl;                 /* version << 4 | header length >> 2 */
        u_char  ip_tos;                 /* type of service */
        u_short ip_len;                 /* total length */
        u_short ip_id;                  /* identification */
        u_short ip_off;                 /* fragment offset field */
        #define IP_RF 0x8000            /* reserved fragment flag */
        #define IP_DF 0x4000            /* dont fragment flag */
        #define IP_MF 0x2000            /* more fragments flag */
        #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
        u_char  ip_ttl;                 /* time to live */
        u_char  ip_p;                   /* protocol */
        u_short ip_sum;                 /* checksum */
```

```
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
};
#define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip)                (((ip)->ip_vhl) >> 4)

/* TCP header */
struct h_tcp {
        u_short th_sport;               /* source port */
        u_short th_dport;               /* destination port */
        u_int th_seq;                  /* sequence number */
        u_int th_ack;                  /* acknowledgement number */
        u_char  th_offx2;               /* data offset, rsvd */
      #define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
        u_char  th_flags;
        #define TH_FIN  0x01
        #define TH_SYN  0x02
        #define TH_RST  0x04
        #define TH_PUSH 0x08
        #define TH_ACK  0x10
        #define TH_URG  0x20
        #define TH_ECE  0x40
        #define TH_CWR  0x80
        #define TH_FLAGS        (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
        u_short th_win;                 /* window */
        u_short th_sum;                 /* checksum */
        u_short th_urp;                 /* urgent pointer */
};

struct t_matrix {
      u_short host;
      u_short port;
      u_int seq_nr;
      u_int ack_nr;
      double syn_time;
      double synack_time;
      int sent_syns;
      int sent_synacks;
};

#define MAX_ENTRIES 5000
#define MAX_HOSTS 5000
#define MAX_TIMEOUT 250
#define MAX_RETRIES 10

char * hosttable[MAX_HOSTS];        // array with ip adresses
struct t_matrix matrix[MAX_ENTRIES];
int double_syn[MAX_TIMEOUT];
int double_synack[MAX_TIMEOUT];
int syn_successful[MAX_RETRIES];
int syn_unsuccessful[MAX_RETRIES];
int synack_successful[MAX_RETRIES];
int synack_unsuccessful[MAX_RETRIES];

// function for covering ip-addresses from user
int host_lookup_table(char *host)
```

```
{
	int index = 1;

	while (hosttable[index] != NULL)
	{
		if (strcmp(host,hosttable[index]) == 0) //if ip is known return the
index
			return index;
		else
			index++;
	}
	hosttable[index] =(char *) malloc (sizeof(char)*16); // reservation of memory
for the ip-address
	strcpy(hosttable[index],host); //save the ip-address
	return index;
}

// prints a single line for a packet
void printline(int shost, int dhost, u_short sport, u_short dport, int count, u_int
seqnr, u_int acknr, double timestamp, char * flag, u_short window,u_short ttl, int
double_syn, int double_synack)
{
	printf("%d;",count);
	printf("%d;",shost);
	printf("%d;",sport);
	printf("%d;",dhost);
	printf("%d;",dport);
	printf("%s;",flag);
	printf("%u;",seqnr);
	printf("%u;",acknr);
	printf("%u;",window);
	printf("%u;",ttl);
	printf("%f;",timestamp);
	printf("%d;",double_syn);
	printf("%d;\n",double_synack);

}

void got_double_syn(double time) {

	double upper=0.1;
	double lower=0.0;
	int i;

	for (i=0 ; i < MAX_TIMEOUT ; i++)
	{
		if ((lower <=time) && (time < upper))
		{
			double_syn[i]++;
			break;
		}
		else
		{
			upper=upper+0.1;
			lower=lower+0.1;
```

```
            }
      }
}

void got_double_synack(double time) {

      double upper=0.1;
      double lower=0.0;
      int i;

      for (i=0 ; i < MAX_TIMEOUT ; i++)
      {
            if ((lower <=time) && (time < upper))
            {
                  double_synack[i]++;
                  break;
            }
            else
            {
                  upper=upper+0.1;
                  lower=lower+0.1;
            }
      }
}

// resetting the arrays
void init()
{
      int i = 0;
      for (i = 0; i < MAX_TIMEOUT; i++)
      {
            double_syn[i] = 0;
            double_synack[i] = 0;
      }

      for (i = 0; i < MAX_RETRIES; i++)
      {
            syn_unsuccessful[i] = 0;
            syn_successful[i] = 0;
            synack_unsuccessful[i] = 0;
            synack_successful[i] = 0;
      }

      for (i = 0; i < MAX_ENTRIES; i++)
      {
            matrix[i].host = 0;
      }
}

// function if captured a syn packet
void got_syn(int count, int shost,int dhost,u_short sport,u_short dport,double
timestamp,u_int seqnr,u_int acknr,u_short window,u_short ttl)
{
      int i = 0;
      int insert = 1;
```

```
	int double_syn = 0;

	for (i = 0 ; i < MAX_ENTRIES ; i++) {
		if ((matrix[i].host == shost) && (matrix[i].port == sport))// &&
(matrix[i].sent_synacks > 0))
		{
			got_double_syn(timestamp-matrix[i].syn_time);
			matrix[i].syn_time = timestamp;
			matrix[i].sent_syns = matrix[i].sent_syns + 1;
			insert = 0;
			double_syn = 1;
			break;
		}
	}

	if (insert == 1)
	{
		for (i = 0 ; i < MAX_ENTRIES ; i++) {
			if (matrix[i].host == 0)
			{
				matrix[i].host = shost;
				matrix[i].port = sport;
				matrix[i].seq_nr = seqnr;
				matrix[i].ack_nr = acknr;
				matrix[i].syn_time = timestamp;
				matrix[i].synack_time = 0;
				matrix[i].sent_syns = 0;
				matrix[i].sent_synacks = 0;
				break;
			}
		}
	}
	printline(shost,dhost,sport,dport,count,seqnr,acknr,timestamp,"SYN",window,tt
l,double_syn,0);
}

void got_synack(int count, int shost,int dhost,u_short sport,u_short dport,double
timestamp,u_int seqnr,u_int acknr,u_short window,u_short ttl)
{
	int i = 0;
	int double_synack = 0;
	for (i = 0 ; i < MAX_ENTRIES ; i++) {
		if ((matrix[i].host == dhost) && (matrix[i].port == dport) &&
(matrix[i].seq_nr == (acknr-1)))
		{
			matrix[i].ack_nr = seqnr;
			matrix[i].sent_synacks = matrix[i].sent_synacks + 1;
			if (matrix[i].sent_synacks > 1)
			{
				got_double_synack(timestamp-matrix[i].synack_time);
				double_synack = 1;
			}
			matrix[i].synack_time = timestamp;
			break;
		}
```

```
	}
	printline(shost,dhost,sport,dport,count,seqnr,acknr,timestamp,"SYNACK",window
,ttl,0,double_synack);
}

void got_ack(int count, int shost,int dhost,u_short sport,u_short dport,double
timestamp,u_int seqnr,u_int acknr,u_short window,u_short ttl)
{
	int i = 0;

	for (i = 0 ; i < MAX_ENTRIES ; i++) {
		if
((matrix[i].host==shost)&&(matrix[i].port==sport)&&(matrix[i].seq_nr==(seqnr-
1))&&(matrix[i].ack_nr==(acknr-1)))
		{

	printline(shost,dhost,sport,dport,count,seqnr,acknr,timestamp,"ACK",window,tt
l,0,0);

				if (matrix[i].sent_syns < 9)
					syn_successful[matrix[i].sent_syns]++;
				else
					syn_successful[9]++;

				if (matrix[i].sent_synacks < 9)
					synack_successful[matrix[i].sent_synacks]++;
				else
					synack_successful[9]++;

				matrix[i].host=0;
				matrix[i].port = 0;
				matrix[i].seq_nr = 0;
				matrix[i].ack_nr = 0;
				matrix[i].syn_time = 0;
				matrix[i].synack_time = 0;
				matrix[i].sent_syns = 0;
				matrix[i].sent_synacks = 0;
				break;
		}
	}
}

void calculate_syn_unsuccessful() {
	int i=0;
	for (i = 0 ; i < MAX_ENTRIES ; i++) {
		if (matrix[i].host != 0)
		{
				if (matrix[i].sent_syns < 9)
					syn_unsuccessful[matrix[i].sent_syns]++;
				else
					syn_unsuccessful[9]++;
		}
	}
}
```

```
void calculate_synack_unsuccessful() {
      int i=0;
      for (i = 0 ; i < MAX_ENTRIES ; i++) {
            if (matrix[i].host != 0)
            {
                  if (matrix[i].sent_synacks < 9)
                        synack_unsuccessful[matrix[i].sent_synacks]++;
                  else
                        synack_unsuccessful[9]++;
            }
      }
}

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char
*packet)
{

      static int count = 1;                   /* packet counter */

      /* declare pointers to packet headers */
      const struct h_ethernet *ethernet;  /* The ethernet header [1] */
      const struct h_ip *ip;              /* The IP header */
      const struct h_tcp *tcp;            /* The TCP header */

      int size_ip;
      int size_tcp;
      int shost=0;
      int dhost=0;

      /* define ethernet header */
      ethernet = (struct h_ethernet*)(packet);

      /* define/compute ip header offset */
      ip = (struct h_ip*)(packet + SIZE_ETHERNET);
      size_ip = IP_HL(ip)*4;

      /* define/compute tcp header offset */
      tcp = (struct h_tcp*)(packet + SIZE_ETHERNET + size_ip);
      size_tcp = TH_OFF(tcp)*4;

      /* if it is a regular packet */
      if ((size_tcp >= 20) && (size_ip >= 20))
      {
            shost = host_lookup_table(inet_ntoa(ip->ip_src)); // cover the source-ip
            dhost = host_lookup_table(inet_ntoa(ip->ip_dst)); // cover the
destination-ip
            switch(tcp->th_flags) // different functions for SYN, SYN/ACK and ACK
packets
            {
            case 2:     got_syn(count,shost,dhost,ntohs(tcp->th_sport),ntohs(tcp-
>th_dport),header->ts.tv_sec+(header->ts.tv_usec/1000000.0), ntohl(tcp->th_seq),
ntohl(tcp->th_ack),tcp->th_win, ip->ip_ttl);
                  break;
```

```
            case 18:got_synack(count,shost,dhost,ntohs(tcp->th_sport),ntohs(tcp-
>th_dport),header->ts.tv_sec+(header->ts.tv_usec/1000000.0), ntohl(tcp->th_seq),
ntohl(tcp->th_ack),tcp->th_win, ip->ip_ttl);
                  break;
            case 16:got_ack(count,shost,dhost,ntohs(tcp->th_sport),ntohs(tcp-
>th_dport),header->ts.tv_sec+(header->ts.tv_usec/1000000.0), ntohl(tcp->th_seq),
ntohl(tcp->th_ack),tcp->th_win, ip->ip_ttl);
                  break;
            default:break;
            }
            count++;
      }
return;
}

void print_statistics(char * double_syn_filename, char * double_synack_filename,
char * syn_unsuccessful_filename, char * syn_successful_filename, char *
synack_unsuccessful_filename, char * synack_successful_filename)
{
      int i=0;
      FILE * pfile;

      pfile = fopen(double_syn_filename,"a");
      for (i = 0; i < MAX_TIMEOUT; i++)
            fprintf(pfile,"%d;",double_syn[i]);
      fprintf(pfile,"\n");
      fclose(pfile);

      pfile = fopen(double_synack_filename,"a");
      for (i = 0; i < MAX_TIMEOUT; i++)
            fprintf(pfile,"%d;",double_synack[i]);
      fprintf(pfile,"\n");
      fclose(pfile);

      pfile = fopen(syn_successful_filename,"a");
      for (i = 0; i < MAX_RETRIES; i++){
            fprintf(pfile,"%d;",syn_successful[i]);
      }
      fprintf(pfile,"\n");
      fclose(pfile);

      pfile = fopen(syn_unsuccessful_filename,"a");
      for (i = 0; i < MAX_RETRIES; i++)
            fprintf(pfile,"%d;",syn_unsuccessful[i]);
      fprintf(pfile,"\n");
      fclose(pfile);

      pfile = fopen(synack_successful_filename,"a");
      for (i = 0; i < MAX_RETRIES; i++){
            fprintf(pfile,"%d;",synack_successful[i]);
      }
      fprintf(pfile,"\n");
      fclose(pfile);

      pfile = fopen(synack_unsuccessful_filename,"a");
```

```
	for (i = 0; i < MAX_RETRIES; i++)
		fprintf(pfile,"%d;",synack_unsuccessful[i]);
	fprintf(pfile,"\n");
	fclose(pfile);

}

int main(int argc, char **argv)
{

	char errbuf[PCAP_ERRBUF_SIZE];		/* error buffer */
	pcap_t *handle;			/* packet capture handle */
	int num_packets = 1000000;		/* number of packets to capture */
	char * dumpfilename;
	char * double_syn_filename;
	char * double_synack_filename;
	char * syn_unsuccessful_filename;
	char * syn_successful_filename;
	char * synack_unsuccessful_filename;
	char * synack_successful_filename;

	struct pcap_pkthdr header;

	init();

	/* define dump file*/
	dumpfilename = (char *) malloc (sizeof(char)*32);
	sprintf(dumpfilename,"./traces/tcpdump%s.pcap",argv[1]);
	printf("%s",dumpfilename);

	/* open dump file */
	handle = pcap_open_offline(dumpfilename,errbuf);
	if (handle == NULL) {
		fprintf(stderr, "Couldn't open file: %s\n", errbuf);
		exit(EXIT_FAILURE);
	}

	printf("Nr;Source Host;Source Port;Dest Host;Dest
Port;Flag;SeqNr;AckNr;Windows;TTL;Timestamp;Double_syn;Double_synack\n");

	/* callback function */
	pcap_loop(handle, num_packets, got_packet, NULL);


	// calculate syn_ and synack_unsuccessful connections
	calculate_syn_unsuccessful();
	calculate_synack_unsuccessful();

	// print statistics
	double_syn_filename = (char *) malloc (sizeof(char)*32);
	sprintf(double_syn_filename,"./stats/syn_stats.csv");
	double_synack_filename = (char *) malloc (sizeof(char)*32);
	sprintf(double_synack_filename,"./stats/synack_stats.csv");
	syn_successful_filename = (char *) malloc (sizeof(char)*32);
	sprintf(syn_successful_filename,"./stats/syn_successful.csv");
```

```
      syn_unsuccessful_filename = (char *) malloc (sizeof(char)*32);
      sprintf(syn_unsuccessful_filename,"./stats/syn_unsuccessful.csv");
      synack_successful_filename = (char *) malloc (sizeof(char)*32);
      sprintf(synack_successful_filename,"./stats/synack_successful.csv");
      synack_unsuccessful_filename = (char *) malloc (sizeof(char)*32);
      sprintf(synack_unsuccessful_filename,"./stats/synack_unsuccessful.csv");

      print_statistics(double_syn_filename,double_synack_filename,syn_unsuccessful_
filename,syn_successful_filename,synack_unsuccessful_filename,synack_successful_fil
ename);

      /* cleanup */
      pcap_close(handle);

return 0;
}
```