An Experimental Study on the Network Performance of SCTP

Aftab Hussain[1], Khalid Mahmood[2], Saif-ul-Islam Khan[3], Syed Muzakkir Ahmed[4]
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh
hussain_aftab@hotmail.com[1], klmahmood@gmail.com[2], saif009972@gmail.com[3],
syed.muzakkir@gmail.com[4]

ABSTRACT

Monitoring network performance parameters like throughput, delay, and overhead help us to judge the quality of transport layer protocols. In this work, we experimentally measure these parameters in order to evaluate the performance of SCTP (Stream Control Transmission Protocol), a protocol widely renowned for its multihoming and multistreaming features. To give credence to our experiment, we simulated the protocol under different topologies and compared the corresponding parameter readings.

***Keywords*— throughput, delay, overhead**

1. INTRODUCTION

SCTP is a reliable, message-oriented transport layer protocol, that was first introduced by Stewart, et al. in RFC 2960[1], which was later obsoleted in RFC 4960[2], the present standard document of the protocol. Services like process-to-process communication, multistreaming and multihoming provided by SCTP make the protocol suitable for real-time applications. To name a few they include, IUA (ISDN over IP), M2UA and M3UA (telephone signaling), H.248 (media gateway control), H.323 (IP telephony), and SIP [3]. In addition to that SCTP retains the congestion, error and flow control mechanisms of TCP (In particular, SCTP follows TCP-SACK for these mechanisms) and thus carries all the advantages of TCP. Another key facility provided by SCTP, which was not present in the TCP variants, is the prevention of SYN-flooding attacks (or Denial of Service Attacks). SCTP's four-way handshake mechanism helps it to achieve this. The HOL (Head-of-line) blocking problem is also averted in SCTP by virtue of its multistreaming facility.

A lot of research work has been carried out in the field of SCTP, especially to study & compare its performances with other protocols, under different types of networks.

Olga Antonova [4] compared SCTP with TCP-MH and DCCP with regard to applications in the mobile network. SCTP was found to be the strongest among the 3 with respect to its multihoming and mobility features, however more investigation is necessary.

Sourabh Ladha and Paul D. Amer in [5], demonstrated how the latency in multiple file transfers can be reduced in FTP with the help of SCTP's multistreaming capability. They also found some key benefits of using FTP over SCTP instead of over TCP: with SCTP, they saw that less packets were transferred in FTP, hence reducing network overload. Also the number of connections a server must maintain is reduced.

In [6], the performance of SCTP was observed in the presence of network redundancy; here in the presence of background traffic, the throughput and end to end delay of SCTP were measured. Network overhead, however, was not measured.

In the following section (Section 2) of the report, we give details of the different features of SCTP and focus on how they are facilitated. In Section 3, we give details of the design of our experiment. In Section 4, we discuss the results we obtained. We conclude our report in Section 5.

2. FEATURES OF SCTP

2.1. Multihoming:-

Multihoming is the ability for a single endpoint to support multiple IP addresses. Therefore, a multi-homed host can be reached by either address. A TCP connection involves one source and one destination IP address. An SCTP *association* (equivalent to a connection in TCP) supports multihoming service. Thus there are multiple paths for multi-homed hosts to communicate with each other. This approach makes the communication more fault-tolerant. An association between multihomed hosts is made by the exchange of INIT chunks, exchanging information about the available IP addresses to each host, and a primary path is chosen for communication. HEARTBEAT chunks are sent over all paths for monitoring purposes. These chunks are acknowledged by HEARTBEATACK chunks. Consequently counters are maintained for each path that depends on departure and arrival of these 2 control chunks respectively. Based on the implementation criteria, paths are marked unreachable according to the value of these counters. Details of the counter management could be found in [2]. Path handovers are made if any path is found to be not responding.

2.2. Multistreaming:-

SCTP allows multistream service in each connection. Multistreaming within an SCTP association separates flows of logically different data into independent streams. Thus, if one of the streams is blocked, the other streams can still deliver their data. This prevents the head-of-line blocking problem.

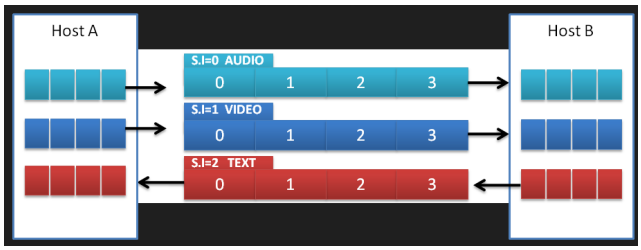

Figure 2.1: SCTP multi-streaming

In Figure 2.1, Hosts A and B have a multistreamed association. Three streams go from A to B, and one stream goes from B to A. Each stream pertains to a certain type of data (audio, video, and text). The number of streams in each direction is negotiated during SCTP's association establishment phase. SCTP uses a set of parameters to distinguish between the stream chunks. Each chunk has the following parameters in its SCTP header,

Transmission Sequence Number (TSN) – used for reliable transmission. The TSN is global over all streams. It uniquely identifies each chunk in the entire association. Reliable delivery is ensured by the help of TSNs. Also, SCTP's *congestion control* mechanism uses these TSNs.

Stream Identifier (SID) – This parameter tells the sequence number of the stream to which a chunk belongs.

Stream Sequence Number (SSN) – indicates the sequence of a chunk within a particular stream. *Ordered delivery* is provided with the help of SSN numbers.

2.3. DoS attack prevention:-

A malicious attacker can flood a server with a huge number of fake SYN packets. Each time the server receives a SYN packet it sets up a state table and allocates resources while waiting for the next packet to arrive. After a while the server may collapse due to exhaustion of resources. This is known as the SYN flooding attack, or the denial of service (DoS) attack [3].

SCTP has a four-way handshake for connection establishment that helps it prevent the DoS attack. First, the client sends an INIT chunk to the server, which consequently replies with an INIT-ACK chunk containing a COOKIE chunk. (Unlike the scenario in TCP where resources are reserved for the client on receiving the request, and then an ACK is sent back.) This chunk contains a Message Authentication Code (MAC), the cookie generation time, and the cookie expiration time. On receiving the cookie chunk, the client sends back a copy of COOKIE in a COOKIE-ECHO chunk. The server calculates the new MAC based on information from this COOKIE-ECHO chunk and compares it with MAC that it sent to the client. If there is a match, resources are allocated and the server acknowledges with a COOKIE-ACK chunk. This is the key step that prevents any possible DoS attack, as the client is verified in this step.

3. EXPERIMENTAL SETUP

In this section we outline the various aspects of our experiment.

3.1. Software used:-

The latest version of Network Simulator available, **(Ns-allinone version 2.34)** [7], was used to conduct the experiments in this work. This version of Network Simulator contains the **SCTP module (release 2.34)** which supports key SCTP features. Details of this module can be found in [8]. The application was installed in **Ubuntu 10.04**, an operating system based on the Debian GNU/Linux platform. For plotting the graphs we used **Gnuplot version 4.4.3**, a portable command-line driven graphing utility for Linux.

3.2. Network topologies used:-

We ran our simulations for SCTP on 3 different network topologies to measure throughput, delay and overhead. For all cases FTP was used in the application layer.

Our first topology, **topology 1** (Fig. 3.1), comprises of 2 nodes, sending messages to each other. Our simulation time for this topology was 30s.

The 2nd topology, **topology 2** (Fig. 3.2), comprises of a slightly more complicated scenario. Here we demonstrate Concurrent Multipath Transfer (CMT). The topology consists of 2 hosts. Each host has 2 interfaces. There exist direct connections between each pair of interfaces (one from each host). Data is transferred using both paths concurrently. Our simulation time for this topology was 10s.

The 3rd topology, **topology 3** (Fig. 3.3), demonstrates multihoming. Two endpoints with 2 interfaces each are connected via a router. The sender has a HEARTBEAT timer for each destination. In the middle of the association (7.5s) a path handover is carried out. Our simulation time for this topology was 12s.

3.3. The parameters measured:-

Throughput. Throughput is a measure of how fast we can send data through a network [3]. In our simulations**,** we calculated the throughput, in kB/s, as follows,

Throughput=total number of bytes received(kB)/total simulation time(s)

Delay. Delay or latency defines how long it takes for an entire message to completely arrive at the destination from the time the first bit is sent out from the source [3]. We calculated the average delays of the data chunks that were sent, in milliseconds, as follows,

Average delay=Total delays for sending all data chunks(ms)/Total number of data chunks sent

Overhead. The overhead for transport layer protocols is the number of additional (control packets) that are being sent in order to transmit a certain number of data packets. We calculate overhead as follows,

Overhead=total no. of control packets sent/total no. of data packets sent * 100 %

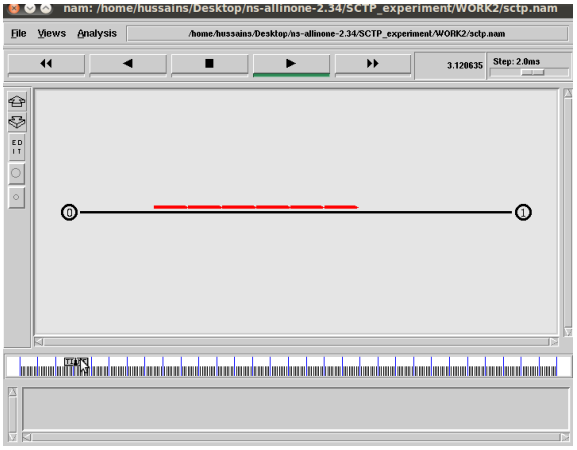

Figure 3.1: Topology 1

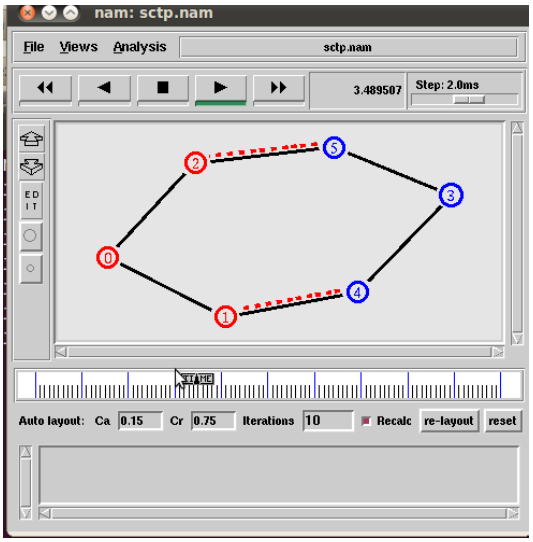

Figure 3.2: Topology 2

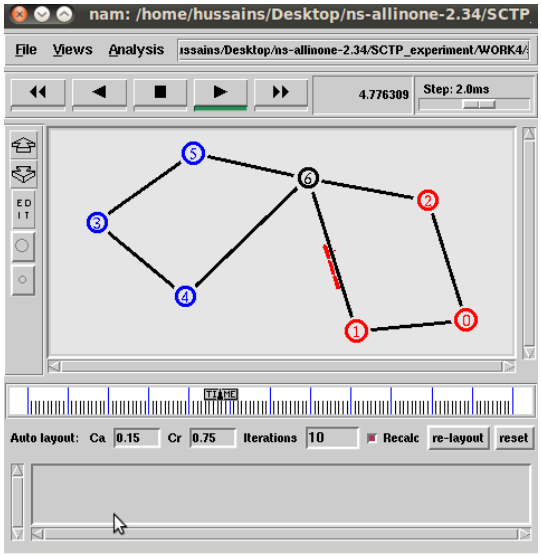

Figure 3.3: Topology 3

3.4. Work methodology:-

We created separate tcl scripts to implement the 3 topologies, codes of which are given in the Appendix. The tcl scripts use features provided in the SCTP module. We used 3 different awk scripts to generate throughput, delay, and overhead respectively for each of the topologies. Codes of the awk scripts have also been given in the Appendix.

After executing the .tcl files in Network Simulator traces are generated. The .awk scripts analyze the traces and generate the required data. The average throughput, average delay, and overhead are shown in the terminal window when each of the awk scripts is executed. Simultaneously, data sheets for each of the metrics are also generated in the form of plain text files. The outputs thus generated are represented in Gnuplot as graphs which give instantaneous readings for each of the metrics.

Fig.3.4 depicts the information flow, at the implementation level, during our experiment.

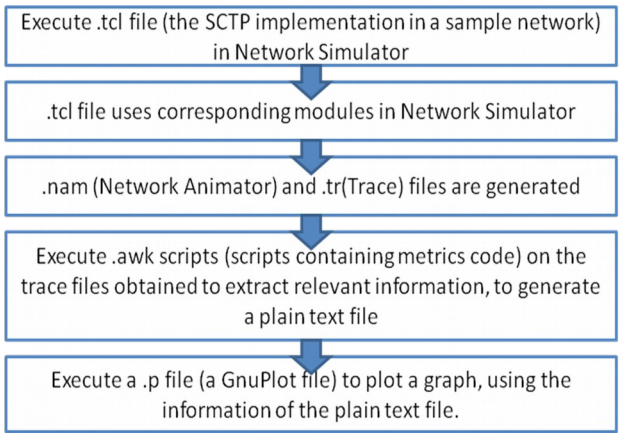

Figure 3.4: Information flow diagram in the experiment

4. RESULTS

4.1. Performance metrics graphs:-

Topology 1:

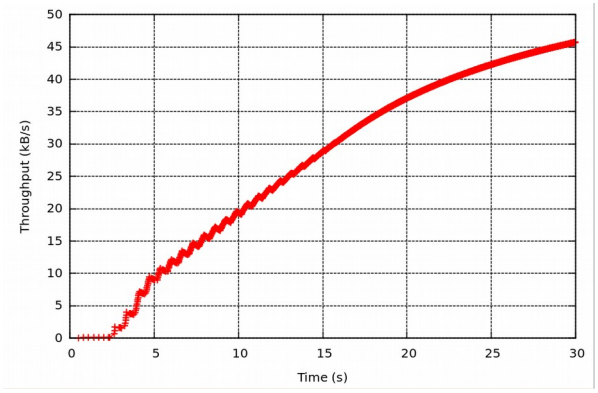

Figure 4.1: Throughput (kB/s) vs. Time(s) for Topology 1

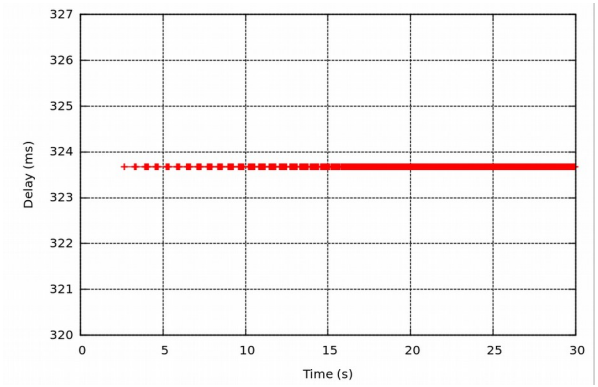

Figure 4.2: Delay (ms) vs. Time (s) for Topology 1

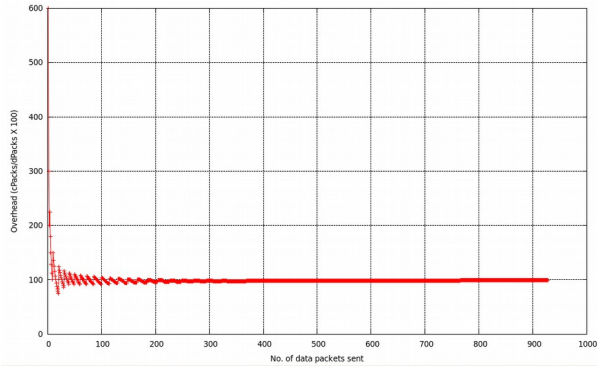

Figure 4.3: Overhead vs. No. of packets sent for Topology1

Table 4.1: Performance metrics for SCTP over Topology 1

Simulation Time (s)	Average Throughput(kB/s)	Average Delay(ms)	Over-head(%)
30	45.71	323.68	99.68

Topology 2:

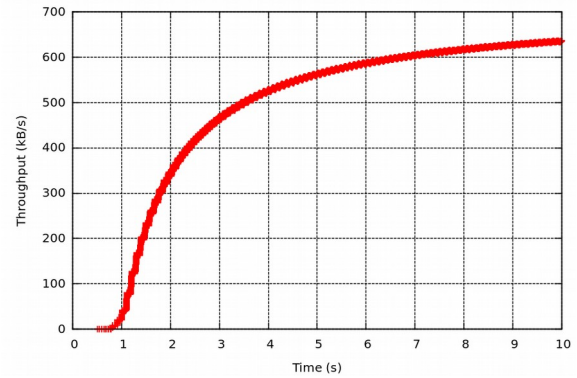

Figure 4.4: Throughput (kB/s) vs. Time(s) for Topology 2

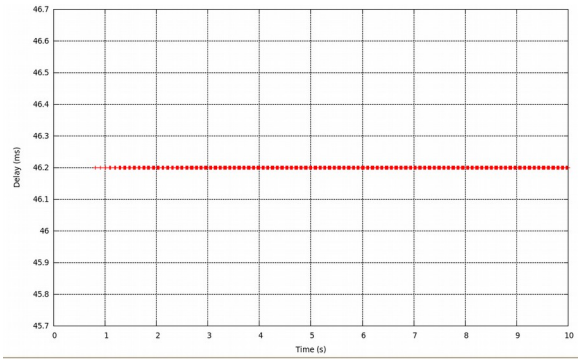

Figure 4.5: Delay (ms) vs. Time (s) for Topology 2

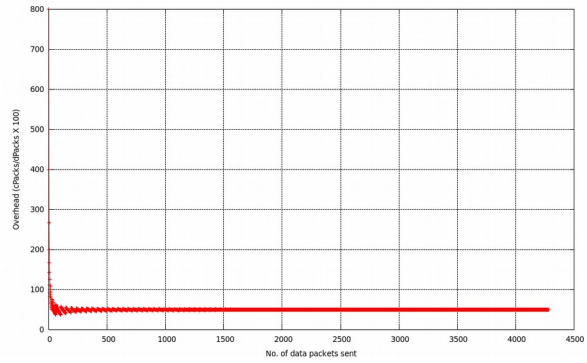

Figure 4.6: Overhead vs. No. of packets sent for Topology 2

Table 4.2: Performance metrics for SCTP over Topology 2

Simulation Time (s)	Average Throughput(kB/s)	Average Delay(ms)	Over-head (%)
30	636.31	46.2	50.07

Topology 3:

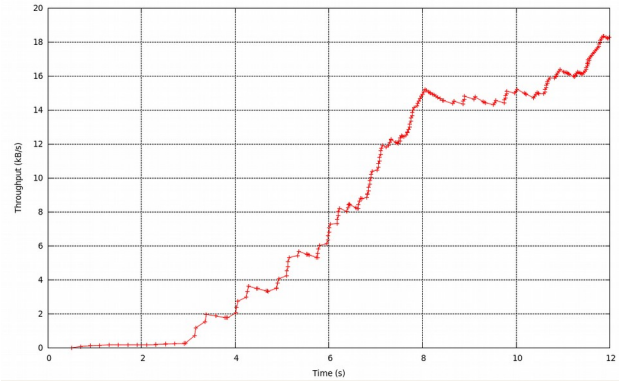

Figure 4.7: Throughput (kB/s) vs. Time(s) for Topology 3

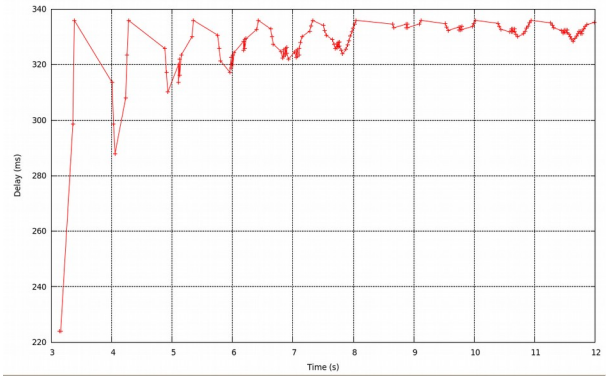

Figure 4.8: Delay (ms) vs. Time (s) for Topology 3

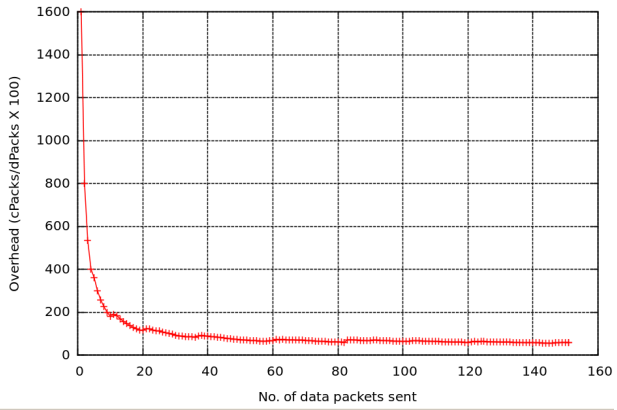

Figure 4.9: Overhead vs. No. of packets sent for Topology 3

Table 4.3: Performance metrics for SCTP over Topology 3

Simulation Time (s)	Average Throughput(kB/s)	Average Delay(ms)	Overhead (%)
12	18.29	335.24	59.60

4.2. Discussion:-

Throughput. In all the topologies, SCTP slowly starts by sending control chunks for establishment of the association. As indicated by the trace files, these include initiation chunks and heart-beat chunks (control chunks for monitoring paths.) Once the association is established, SCTP's throughput increases until it reaches the full capacity. This was reflected in all the throughput graphs (Figs. 4.1, 4.4, 4.7).

Delay. As for the delay metric, once the association is established, it follows the delay for the link set in the corresponding .tcl files. Therefore, no extra time was taken for sending any packet. This was the output observed for all the topologies.

Overhead. During the initial phase of transmission we found the overhead to be very high in all the 3 topologies. This is due to the large number of control packets that are exchanged during association phase of SCTP. After the initial phase (the association establishment phase) the overhead steadies down to a constant value. However, we found the overall overhead of SCTP to be very high, over topology 1 than what it is over the other topologies. This is due to the fact that the delayed acknowledgement mechanism was not used in topology 1, as per the implementation in ***sctp1.tcl*** *(set useDelayedSacks_ $false)*. Hence an acknowledgement for every data packet received was sent, drastically increasing the overhead of the protocol. The overheads in the other 2 topologies were comparatively much lower (50.07% and 59.06%) since the delayed acknowledgement mechanism was used. The reason why topology 2 gave a better overhead reading than topology 3 is because the former implemented concurrent multi-path transfer, and thus more data chunks could be sent simultaneously. This clearly indicates a key advantage which SCTP facilitates with its multihoming feature. Further investigation is necessary for achieving concurrent multi-path transfers with SCTP in real-life implementations as SCTP's current implementation does not support load sharing [4].

In topology 3, a path handover was intentionally implemented at 7.5s of simulation time. No marked variation was observed in any of the network parameters at that time. This clearly indicates that the path handover was smooth and without irregularities.

5. CONCLUSIONS

In this work we studied SCTP's network performance by measuring 3 parameters, throughput, delay, and overhead using Network Simulator 2.34. Simulations were carried out over three widely different topologies. In one of the topologies we observed how the protocol performs with concurrent multi-path transfers. We also observed the parameters in a situation where a path handover took place. With our results we infer that SCTP's delayed SACK mechanism (a mechanism which it carried forward from TCP) helps it to maintain acceptable overheads. We also observed that if concurrent multi-path transfer is implemented (which is possible due to SCTP's multihoming facility), the network overhead could be further improved.

The assumption in our work was the absence of any background traffic. With the inclusion of background traffic it would be possible to analyze the resilience of SCTP in a more real-life network. Then it would be interesting to see how the performance metrics are affected. In addition to that SCTP could be tested in the presence of traffic, which are running under a different protocol, such as TCP. That way, SCTP's friendliness to other protocols could also be assessed.

6. REFERENCES

[1] RFC 2960 - Stream Control Transmission Protocol, *www.ietf.org/rfc/rfc2960.txt*

[2] RFC 4960 – Stream Control Transmission Protocol, *tools.ietf.org/html/rfc4960*

[3] Behrouz A. Forouzan, Data Communications and Networking, 4th edition, Tata McGraw Hill, pp. 736-754

[4] Olga Antonova, Introduction and Comparison of SCTP, TCP-MH, DCCP protocols, University of Helsinki, 2004, *www.tml.tkk.fi/Studies/T-110.551/2004/papers/Antonova.pdf*

[5] Sourabh Ladha, Paul D. Amer, Improving Multiple File Transfers Using SCTP Multistreaming, Conference on Performance, Computing, and Communications, IEEE International, pp.513 – 522, 2004

[6] Rashid Ali, Performance of Network Redundancy in SCTP - Introducing effect of different factors on Multi-homing, Master's Thesis, University West, Department of Economics and IT, Sweden, 2010

[7] Network Simulator Download page, *www.isi.edu/nsnam/ns/ns-build.html*

[8] README file for NS-2 SCTP module release 3.7, *www.cis.udel.edu/~nekiz/sctp.pdf*

APPENDIX

A. The .tcl files for the 3 different topologies:

sctp1.tcl

```
Trace set show_sctphdr_ 1


set ns [new Simulator]

set nf [open sctp.nam w]

$ns namtrace-all $nf


set allchan [open all.tr w]

$ns trace-all $allchan


proc finish {} {

    global ns nf allchan trace_ch


    $ns flush-trace

    close $nf

    close $allchan

    close $trace_ch


    exec nam sctp.nam &


    exit 0

}

set false      0
set true       1


set n0 [$ns node]

set n1 [$ns node]

$ns duplex-link $n0 $n1 .5Mb 300ms DropTail

$ns duplex-link-op $n0 $n1 orient right

$ns queue-limit $n0 $n1 93000

set err [new ErrorModel/List]

$err droplist {15}

$ns lossmodel $err $n0 $n1


set sctp0 [new Agent/SCTP]

$ns attach-agent $n0 $sctp0

$sctp0 set fid_ 0

$sctp0 set debugMask_ 0x00303000  # u can
use -1 to turn on everything

$sctp0 set debugFileIndex_ 0

$sctp0 set mtu_ 1500

$sctp0 set dataChunkSize_ 1448

$sctp0 set numOutStreams_ 1

$sctp0 set initialCwndMultiplier_ 2

$sctp0 set useMaxBurst_ $true


set trace_ch [open trace.sctp w]

$sctp0 set trace_all_oneline_ 0 # do not
trace all variables

$sctp0 trace cwnd_

$sctp0 attach $trace_ch


set sctp1 [new Agent/SCTP]

$ns attach-agent $n1 $sctp1

$sctp1 set debugMask_ -1

$sctp1 set debugFileIndex_ 1

$sctp1 set mtu_ 1500

$sctp1 set initialRwnd_ 65536

$sctp1 set useDelayedSacks_ $false


$ns color 0 Red

$ns color 1 Blue


$ns connect $sctp0 $sctp1

set ftp0 [new Application/FTP]

$ftp0 attach-agent $sctp0
```

```
$ns at 0.5 "$ftp0 start"
$ns at 29.5 "$ftp0 stop"
$ns at 30.0 "finish"

$ns run
```

sctp2.tcl

```
Trace set show_sctphdr_ 1

set ns [new Simulator]
set nf [open sctp.nam w]
$ns namtrace-all $nf

set allchan [open all.tr w]
$ns trace-all $allchan

proc finish {} {
    global ns nf allchan

    $ns flush-trace
    close $nf
    close $allchan

    exec nam sctp.nam &

    exit 0
}

set host0_core [$ns node]
set host0_if0 [$ns node]
set host0_if1 [$ns node]

$host0_core color Red
$host0_if0 color Red
$host0_if1 color Red

$ns multihome-add-interface $host0_core $host0_if0
$ns multihome-add-interface $host0_core $host0_if1

set host1_core [$ns node]
set host1_if0 [$ns node]
set host1_if1 [$ns node]

$host1_core color Blue
$host1_if0 color Blue
$host1_if1 color Blue

$ns multihome-add-interface $host1_core $host1_if0
$ns multihome-add-interface $host1_core $host1_if1

$ns duplex-link $host0_if0 $host1_if0 10Mb 45ms DropTail
[[$ns link $host0_if0 $host1_if0] queue] set limit_ 50
$ns duplex-link $host0_if1 $host1_if1 10Mb 45ms DropTail
[[$ns link $host0_if1 $host1_if1] queue] set limit_ 50

set sctp0 [new Agent/SCTP/CMT]
$ns multihome-attach-agent $host0_core $sctp0
$sctp0 set fid_ 0
$sctp0 set debugMask_ -1
$sctp0 set debugFileIndex_ 0
$sctp0 set mtu_ 1500
$sctp0 set dataChunkSize_ 1468
```

```
$sctp0 set numOutStreams_ 1

$sctp0 set useCmtReordering_ 1   # turn on
Reordering algo.

$sctp0 set useCmtCwnd_ 1         # turn on
CUC algo.

$sctp0 set useCmtDelAck_ 1       # turn on
DAC algo.

$sctp0 set eCmtRtxPolicy_ 4      # rtx.
policy : RTX_CWND



set trace_ch [open trace.sctp w]

$sctp0 set trace_all_ 1          # trace
them all on one line

$sctp0 trace cwnd_

$sctp0 trace rto_

$sctp0 trace errorCount_

$sctp0 attach $trace_ch



set sctp1 [new Agent/SCTP/CMT]

$ns multihome-attach-agent $host1_core
$sctp1

$sctp1 set debugMask_ -1

$sctp1 set debugFileIndex_ 1

$sctp1 set mtu_ 1500

$sctp1 set initialRwnd_ 65536

$sctp1 set useDelayedSacks_ 1

$sctp1 set useCmtDelAck_ 1



$ns color 0 Red

$ns color 1 Blue



$ns connect $sctp0 $sctp1



set ftp0 [new Application/FTP]

$ftp0 attach-agent $sctp0
```

```
# set primary before association starts

$sctp0 set-primary-destination $host1_if0



$ns at 0.5 "$ftp0 start"

$ns at 10.0 "finish"



$ns run
```

sctp3.tcl

```
Trace set show_sctphdr_ 1



set ns [new Simulator]

set nf [open sctp.nam w]

$ns namtrace-all $nf



set allchan [open all.tr w]

$ns trace-all $allchan



proc finish {} {

    global ns nf allchan



    $ns flush-trace

    close $nf

    close $allchan

    exec nam sctp.nam &

    exit 0

}



set host0_core [$ns node]

set host0_if0 [$ns node]

set host0_if1 [$ns node]

$host0_core color Red
```

```
$host0_if0 color Red

$host0_if1 color Red

$ns multihome-add-interface $host0_core
$host0_if0

$ns multihome-add-interface $host0_core
$host0_if1



set host1_core [$ns node]

set host1_if0 [$ns node]

set host1_if1 [$ns node]

$host1_core color Blue

$host1_if0 color Blue

$host1_if1 color Blue

$ns multihome-add-interface $host1_core
$host1_if0

$ns multihome-add-interface $host1_core
$host1_if1



set router [$ns node]



$ns duplex-link $host0_if0 $router .5Mb
200ms DropTail

$ns duplex-link $host0_if1 $router .5Mb
200ms DropTail



$ns duplex-link $host1_if0 $router .5Mb
200ms DropTail

$ns duplex-link $host1_if1 $router .5Mb
200ms DropTail



set sctp0 [new Agent/SCTP]

$ns multihome-attach-agent $host0_core
$sctp0

$sctp0 set fid_ 0

$sctp0 set debugMask_ -1

$sctp0 set debugFileIndex_ 0

$sctp0 set mtu_ 1500

$sctp0 set dataChunkSize_ 1468

$sctp0 set numOutStreams_ 1

$sctp0 set oneHeartbeatTimer_ 0  # each dest
has its own heartbeat timer


set trace_ch [open trace.sctp w]

$sctp0 set trace_all_ 1           # trace
them all on oneline

$sctp0 trace cwnd_

$sctp0 trace rto_

$sctp0 trace errorCount_

$sctp0 attach $trace_ch



set sctp1 [new Agent/SCTP]

$ns multihome-attach-agent $host1_core
$sctp1

$sctp1 set debugMask_ -1

$sctp1 set debugFileIndex_ 1

$sctp1 set mtu_ 1500

$sctp1 set initialRwnd_ 131072

$sctp1 set useDelayedSacks_ 1



$ns color 0 Red

$ns color 1 Blue



$ns connect $sctp0 $sctp1



set ftp0 [new Application/FTP]

$ftp0 attach-agent $sctp0



$sctp0 set-primary-destination $host1_if0



# change primary

$ns at 7.5 "$sctp0 set-primary-destination
$host1_if1"

$ns at 7.5 "$sctp0 print cwnd_"
```

```
$ns at 0.5 "$ftp0 start"

$ns at 12.0 "finish"


$ns run
```

B. The .awk scripts that generate the 3 performance metrics, throughput, delay, and overhead:

throughput.awk

```
BEGIN {

                bytes_counter=0;

}


{

       #Record the time that has elapsed

                     time_elapsed = $2;


       #Record the total number of bytes received

       if ($1=="r") {

                bytes_counter+= $6;

                      }


       #Calculate throughput & create datasheet for plotting graph of throughput vs. time

                if (time_elapsed > 0 ) {

                                  thru = bytes_counter / time_elapsed;

                                  thru /= 1024;

                                  printf("%f %f\n", time_elapsed, thru) > "throughput";

                           }

}


END {


#OUTPUT the Overall Throughput in the Terminal Window

print("Number of bytes received =",bytes_counter);

print("Total simulation time (s) =",time_elapsed);

print("Throughput (kB/s) =",thru);



}
```

delay.awk

```
BEGIN {


             i=0;

             total_delay=0;

             pack_received_count=0;


}


{

             #Here we record the time when a data chunk is sent || The '-' in the first flag of a trace file field, indicates

             #when the corresponding chunk was sent

             if ($1=="-" && $7=="-------D")

                    {

                                  #Keeping a record of the 7th to 15th flags of a field of a packet, denoted in the tracefile, helps us to uniquely identify a packet
```

```
      unique_pack_identifier[i]=$7 $8 $9
$10 $11 $12 $13 $14 $15;


pack_sent_time[i]=$2;
                           i++;
                            }



             #Here we record the time when
the data chunk is received || The '-' in the
first flag of a trace file field, indicates

             #when the corresponding packet
was received

             #consequently, the delay for
that packet is found, and the average delay
is calculated

             if ($1=="r" && $7=="-------D")
                    {


received_pack_identifier=$7 $8 $9 $10 $11
$12 $13 $14 $15;



                      for (j=0;j<i+1;j++)
                           {


if(received_pack_identifier==unique_pack_ide
ntifier[j])
                                          {


      pack_received_count+=1;


      pack_received_time=$2;


      total_delay+=pack_received_time -
pack_sent_time[j];


                                          #The
average delay calculation in milliseconds &
create datasheet for plotting delay



avg_delay=total_delay/pack_received_count*10
00;


      printf("%f %f\n", pack_received_time,
avg_delay) > "delay";


      break;
                                              }
                                 }
                           }


}



END {

#OUTPUT the Overall Delay in the Terminal
Window

print("Number of data chunks received
=",pack_received_count);

print("Total delay time for all the data
chunks (s) =",total_delay);

print("Average_Delay (ms) =",avg_delay);


}
```

overhead.awk

```
BEGIN {


             cntrl_count=0;

             data_count=0;


}


{


      #considering packets that are sent

      if($1=="-")

      {
```

```
                                                    }

              #increment control chunk count
if packet sent is a control chunk

              if ($7=="-------I"||
$7=="-------S"||$7=="-------H"||$7=="-------
B"){

                     cntrl_count+=1;

              }



              #increment data chunk count if
packet sent  is a data chunk

              if ($7=="-------D"){

                     data_count+=1;



                     #Calculate overhead &
create datasheet for plotting graph of
overhead vs. data packets sent



overhead=(cntrl_count/data_count)*100;

                     printf("%f %f\n",
data_count, overhead) > "overhead";

              }


        }

}



END {



#OUTPUT the Overall Overhead in the Terminal
Window

overhead=(cntrl_count/data_count)*100;

print("Number of control packets sent
=",cntrl_count);

print("Number of data packets sent
=",data_count);

print("Overhead =",overhead,"%");

printf("%f %f\n", data_count, overhead) >
"overhead";
```