University of Amsterdam

MSc System and Network Engineering
Research Project One

Investigating the Potential for SCTP to be used as a VPN Transport Protocol

by

Joseph Darnell Hill

February 7, 2016

Abstract

SCTP is a message oriented, connection based protocol with flexible ordering options. This research aims to determine when this protocol would be useful as a transport protocol for VPN traffic. Specifically, the throughput of SCTP, UDP and TCP while encapsulating IP packets is compared. SCTP is shown to clearly have the potential to out perform TCP. Also, due to its connection oriented nature, it may be preferable to UDP in some circumstances. However implementation issues on various platforms limit the practicality of its use.

Contents

1 Introduction

A Virtual Private Network (VPN) allows two endpoints to communicate securely over an untrusted connection. Conceptually a VPN can be viewed as a tunnel through an untrusted network, through which various traffic can pass. This is typically implemented by encapsulating and encrypting traffic as it enters the tunnel, then decapsulating and decrypting the traffic as it exits. An advantage of this method is that the tunnel can support various types of traffic. Also, the source and destination of the traffic need not know that the tunnel exits. Some of the challenges of implementing a VPN are how best to encapsulate this wide variety of data and how to transport it over the network. While the application of encryption is also an important topic, it is not the focus of this research. Various techniques have been used to implement tunnels. The open source VPN solution, OpenVPN, can use either TCP or UDP to transport traffic[4]. With IPSec the Encapsulating Security Payload (ESP) protocol is used[7]. Cisco developed its own protocol for encapsulation, Generic Routing Encapsulation (GRE)[2], which is also used by the Point-to-Point Tunneling Protocol (PPTP)[5]. This research will investigate the possibility of using the Stream Control Transmission Protocol (SCTP) and compare it to TCP and UDP. As a transport protocol SCTP is similar to TCP in that it is a connection oriented protocol that provides reliable delivery[9]. This is important as a connection-less protocol like UDP, may have issues with stateful firewalls not being able to track the connection[10]. Unlike TCP, SCTP is message based[9], meaning that the transport protocol will maintain message boundaries. When encapsulating IP traffic, this will result in only entire packets being delivered to next higher layer. With a stream based protocol such as TCP, the application must implement a method of identifying the boundaries of packets. Where TCP provides ordered delivery and UDP does not, SCTP provides options for more flexible ordering. This is important in mitigating the Head-of-Line blocking problem that can occur with ordered delivery[11]. The issue occurs when a message is lost and all the messages ordered after it must not be delivered until the lost message is recovered. SCTP allows each message to be assigned to a stream, with each stream being independently ordered[14]. This could be used to prevent unrelated traffic passing through a tunnel from blocking each other. SCTP also allows for the selective disabling of the ordering requirement for individual messages[9]. For these reasons the possibility of using SCTP to transport VPN traffic is being investigated.

2 Research Question

This research will determine under what circumstances, if any, is SCTP a suitable choice of a transport protocol for VPN traffic. This research will specifically focus on how SCTP performance compares to TCP and UDP, how the SCTP ordering options affect performance and the practicality of implementing a tunnel using SCTP.

3 Related Work

Other research has considered the use of SCTP as a transport protocol for VPN traffic. However, that research has focused on the use of the multi-homing feature of SCTP to create a multi-homed VPN[17]. This research does not involve multi-homing and instead focuses on the selective ordering option of SCTP and how it affects performance. There has also been research done into the secure transporting of SCTP traffic[12]. This research differs in that it is concerned with the performance of transporting other IP protocols over SCTP not the encapsulation of SCTP traffic. The ARPA 2 Project has several open projects involving the use of SCTP as a transport protocol[1]. However these projects involve the mapping of specific higher layer protocols into SCTP. Whereas, this research is about the encapsulation of arbitrary IP traffic and the simultaneous encapsulation of multiple connections.

4 Method

A tunneling application was written that was capable of setting up a tunnel between two endpoints using TCP, UDP or SCTP. The application was also capable of enabling or disabling ordering on the SCTP tunnel. FreeBSD 10.2 was chosen as the operating system for each endpoint due to its long standing support for SCTP[13]. Encapsulation was performed by taking IP packets received on a tunnel adapter and encapsulating them as the payload of the selected protocol. The application was written to be as simple as possible, using the default options for each protocol with the exception of SCTP ordering. It also was designed to minimize the differences in how each protocol was handled. However, due to TCP being a stream based protocol, the application had to implement a means of identifying message boundaries. This was done by prepending two bytes to each message containing the total size of the message. The application was tested in a lab environment to ensure that it operated correctly before being used to gather data.

In order to test SCTP performance in a variety of situations, six virtual machines were setup to act as tunnel endpoints. Each virtual machine was setup in a disparate location and paired with another. Resulting in three tunnels each with substantially different lengths, both in the physical and network sense (Table 1).

Locations	**Distance**	**Hops**	**Round Trip Time**
Amsterdam to Frankfurt	360 km	7	7.36 ms
London to New York	5500 km	8	71.5 ms
San Francisco to Singapore	13,600 km	10	196 ms

Table 1: Endpoint Pairs

Four tunnels were setup between each pair to encapsulate traffic. TCP and UDP, were used on two of the tunnels. The other two tunnels used SCTP, one with ordering enabled the other without. The iperf utility was used to send traffic through the tunnels, one tunnel at a time, and measure the throughput. Given that TCP is used for the vast majority of network traffic[8][3], it was used for the traffic going through the tunnel. Ten simultaneous streams were used

for thirty seconds at a time. Each tunnel was tested five times in each direction. Throughput measurements were taken on the receiving end.

5 Experiments

Table 2 shows the measurements taken for for each trial of each protocol at every pair. The averages for each tunnel are also shown. The first five trials show traffic moving in the same direction as the endpoints are listed. Trials six through ten are in the reverse direction. Throughput measurements were always taken at the receiving end.

Tunnel	Amsterdam to Frankfurt				London to New York				San Francisco to Singapore			
Protocol	TCP	UDP	SCTP (ordered)	SCTP (unordered)	TCP	UDP	SCTP (ordered)	SCTP (unordered)	TCP	UDP	SCTP (ordered)	SCTP (unordered)
Trial 1	60.9	116	148	147	6.1	25.4	30.5	28.8	2.03	11	3.57	4.75
Trial 2	60.5	132	134	136	6.07	34.1	45.7	32.2	1.95	11.4	3.93	10.4
Trial 3	61.2	124	131	134	6.11	29.1	26.7	49.5	2.24	11.1	6.72	10.6
Trial 4	60.9	123	128	126	6.27	30.3	41.9	35.3	2.27	11.4	6.49	4.82
Trial 5	60.8	123	130	129	6.11	19.1	42	28.3	2.26	11.5	8.37	10.8
Trial 6	60.7	140	143	150	6.19	45.7	31	39.3	0.669	6.31	0.596	0.773
Trial 7	61.8	126	138	136	6.15	38.2	21.2	14.4	0.688	5.86	0.704	0.648
Trial 8	61.2	139	130	132	6.2	54.5	19.2	21.9	1.56	7.73	1.03	1.26
Trial 9	55.8	136	125	132	6.17	57.4	18.1	12.8	1.23	9.8	1.3	1.42
Trial 10	56.1	132	132	124	6.26	48.4	13.2	14.9	1.56	14.5	1.19	1.41
Average	**59.99**	**129.1**	**133.9**	**134.6**	**6.163**	**38.22**	**28.95**	**27.74**	**1.6457**	**10.06**	**3.39**	**4.6881**

Table 2: Measured Performance Data (Mb/s)

Figure 1 shows the average throughput of each protocol between each pair of endpoints. As expected each protocol performed better the shorter the tunnel. It can be seen that UDP is the best performer or performs close to the best performer within any pair of endpoints. TCP is consistently the worse performer between any pair. The performance between SCTP ordered and unordered is close on all pairs.

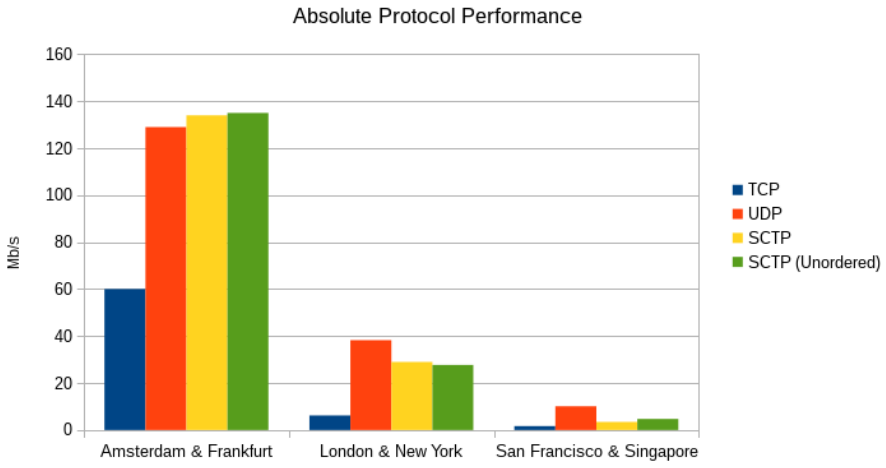

Figure 1: Average throughput measured on each tunnel.

To better visualize the relative performance within each pair of endpoints, Figure 2 shows the performance of each protocol relative to how UDP performed between the same endpoints.

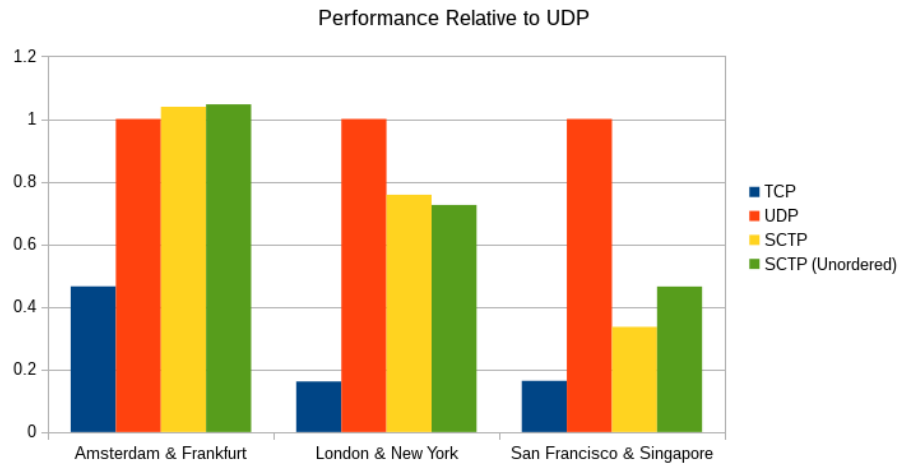

Figure 2: Throughput relative to UDP on the same pair.

6 Analysis

Two areas seem to be significant in evaluating the current potential of SCTP to be used for the encapsulation of data. The performance data gathered is clearly important when evaluating the protocol. However, a number of issues where encountered while implementing the SCTP tunnels which are also relevant.

6.1 Performance

Before comparing SCTP to TCP and UDP it is interesting to consider ordered versus unordered delivery within SCTP. Figure 2 shows that unordered delivery does not seem to have any significant advantage except between the pair of endpoints with the greatest delay. This would suggest that performing selective ordering on a per message basis may not yield a noticeable advantage given the increased over head inspecting each packet would cause. In fact the overhead of doing such classification in real time would more likely cause a degradation in performance. When comparing the three protocols, UDP is clearly the best performer in two out of three cases, outperforming the next best protocol by 32% in one case and 114% in the other. Even in the one case where it was not the best performer it was only outperformed by 4%. TCP is clearly the worse performer in all cases, with every other protocol having at least twice the throughput. This could be due to the TCP over TCP problem[16], where the reliability mechanism of the transport protocol interferes with mechanism on the traffic being encapsulated. These tests used TCP traffic passing through

the tunnels as it is by far the most commonly used protocol [8, 3]. In an environment with an unusual traffic pattern, such that TCP is not the majority, these results could differ. The relative performance of SCTP seems to be related to the delay between the two endpoints. Its performance relative to UDP is best between the shortest link and the worst over the longest link. This holds true weather or not ordered deliver was disabled.

6.2 Implementing

A number of issued were encountered while attempting to implement SCTP tunnels. Initial testing was done using Ubuntu Linux 14.04 endpoints. SCTP showed a substantially lower performance compared to TCP and UDP. This occurred even when not encapsulating other traffic. Similar results were achieved using multiple Linux distributions. After further investigation it was discovered that the Linux implementation of SCTP has severe performance issues in some circumstances. This was documented in research performed by Asim Iqbal while at CERN[6]. As the goal of this research was to determine the true potential of SCTP, a different implementation was used to conduct further testing. FreeBSD which uses the reference implementation of SCTP[13], was chosen for testing. However the FreeBSD implementation was not without its own issues. Throughout the conducted research an SCTP connection was unable to be established when using IPv4. Also, driver options had to be applied to the network interface in order to establish a IPv6 SCTP connection. This was of course dependent on the driver in use, but with the commonly used virtio network device driver, IPv6 checksum offloading and segmentation offloading needed to be disabled. Tests showed that this did not hamper the performance of the other protocols.

7 Conclusions

While UDP clearly has the best consistent performance across all the endpoint pairs that does not mean that a case cannot be made for the use of SCTP. There are currently cases where TCP is chosen over UDP. In the book Mastering OpenVPN it is suggested to use UDP if it works and if it does not then to use TCP[4]. Since UDP is a connection-less protocol, tunnels using UDP can have issues transversing stateful firewalls. While OpenVPN provides options to mitigate these issues[10], there are occasions where a connection oriented protocol must be used. This is where SCTP might be the best option. As it is connection oriented and outperforms TCP, at least when carry TCP data, it may be the best choice when UDP does not work. However, given the issues encountered during this research it should be expected that there will be some situations where SCTP will just not work as well. For instance if a stateful firewall is not SCTP aware, a SCTP tunnel may fair no better then a UDP tunnel. Also care needs to be taken what platform is used for the tunnel endpoints as not all implementations of SCTP perform equally.

8 Future Work

Given the implementation issues discovered during this research, one area of future work could be a comparison of SCTP support and performance in various

platforms. This could include network devices such as firewalls as well computer operating systems. This research focused on a one to one tunnel, but a VPN can also be implemented in a one to many fashion. In that scenario, one end point would be a server or network device that many clients would connect to. The clients could often be using a mobile device so it may also be worth while to survey support in Android and iOS. Also, there are several RFC's proposing extensions to the SCTP protocol, such as the partial reliability extension[15]. Some of these extension may provide a performance benefit when encapsulating data. This could be researched along with the current state of support for the extension.

9 Appendix

Source code of tunnel application, written in C for FreeBSD.

```
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include <fcntl.h>
#include <poll.h>
#include <arpa/inet.h>
#include <netinet/sctp.h>

#define  MAX_MSG_SIZE 16384
#define  MAX_DATA_SIZE 16386

// Display Error Message and Exit
void die(char *msg) {
  perror(msg);
  exit(-1);
}

// Opens the Tunnel Device Specified by Command Line
int get_tun(char *dev_name) {
  char dev_path[16] = "/dev/";
  int tun;

  strncat(dev_path, dev_name, 8);

  printf("Opening Tunnel... ");
  if ((tun = open(dev_path,O_RDWR)) == -1)
    die("Tunnel Open Error");
  printf("Done (%s).\n", fdevname(tun));

  return tun;
}

// Opens a network socket of the specified protocol
// U: UDP
// T: TCP
// S: SCTP (ordered)
// X: SCTP (unordered)
int get_socket(char proto) {
  int sckt;
  struct sctp_sndrcvinfo sctp_defaults;
```

```
  bzero(&sctp_defaults, sizeof(sctp_defaults));

  switch(proto) {
    case 'U' :
      sckt = socket(AF_INET6, SOCK_DGRAM, 0);
      break;
    case 'T' :
      sckt = socket(AF_INET6, SOCK_STREAM, 0);
      break;
    case 'X' :
      sctp_defaults.sinfo_flags = SCTP_UNORDERED;
    case 'S' :
      sckt = socket(AF_INET6, SOCK_STREAM, IPPROTO_SCTP);
      setsockopt(sckt, IPPROTO_SCTP, SCTP_DEFAULT_SEND_PARAM,
                 &sctp_defaults, sizeof(sctp_defaults));
      break;
    case 'Q' :
      sckt = socket(AF_INET6, SOCK_SEQPACKET, IPPROTO_SCTP);
      break;
    default :
      die("Invalid Socket Type");
  }

  if (sckt < 1) die("Socket Error");

  return sckt;
}

// Binds Network Socket to Local Address and Port
void bind_socket(int sckt, const char *local_ip,
                 const char *local_port) {
  struct sockaddr_in6 local_address;

  bzero(&local_address,sizeof(local_address));
  local_address.sin6_family = AF_INET6;
  local_address.sin6_port   = htons(atoi(local_port));
  if (inet_pton(AF_INET6, local_ip,
                local_address.sin6_addr.s6_addr) != 1)
    if (inet_pton(AF_INET, local_ip,
                  local_address.sin6_addr.s6_addr) != 1)
      die("Invalid Address (Local)");

  if (bind(sckt, (struct sockaddr *) &local_address,
           sizeof(local_address)) == -1)
    die("Bind Error");
}

// Sets the network Socket to Listen For connections
void listen_socket(int sckt) {
  if (listen(sckt,1) < 0)
```

```
    die("Socket Listen Error");
}

// Accepts an incoming connection request
// spawns new socket
int accept_connection(int sckt) {
  int new_sckt;

  new_sckt = accept(sckt,NULL,NULL);
  if (new_sckt < 0)
    die("Socket Accept Error");
  close(sckt);

  return new_sckt;
}

// Sets up connection with remote IP Address and port
void connect_socket(int sckt, const char *remote_ip,
                    const char *remote_port) {
  struct sockaddr_in6 remote_address;

  bzero(&remote_address,sizeof(remote_address));
  remote_address.sin6_family = AF_INET6;
  remote_address.sin6_port   = htons(atoi(remote_port));
  if (inet_pton(AF_INET6, remote_ip,
                remote_address.sin6_addr.s6_addr) != 1)
    if (inet_pton(AF_INET, remote_ip,
                  remote_address.sin6_addr.s6_addr) != 1)
      die("Invalid Address (Remote)");

  if (connect(sckt, (struct sockaddr *) &remote_address,
              sizeof(remote_address)) == -1)
    die("Unable To Connect");
}

// Transfers message oriented data
// between socket and tunnel device
void transfer_data(int sckt, int tun) {
  struct pollfd pfds[2];
  unsigned char buffer[MAX_MSG_SIZE];
  int msg_len;

  pfds[0].fd = sckt;
  pfds[0].events = POLLIN;
  pfds[1].fd = tun;
  pfds[1].events = POLLIN;

  while (1) {
    poll(pfds, 2, -1);
```

```
    if (pfds[0].revents == POLLIN) {
      msg_len = recv(sckt, buffer, MAX_MSG_SIZE, 0);
      if (msg_len < 1)
        die("Socket Recv Error");
      if (write(tun, buffer, msg_len) < msg_len )
        die("Tunnel Write Error");
    }
    if (pfds[1].revents == POLLIN) {
      msg_len = read(tun, buffer, MAX_MSG_SIZE);
      if (msg_len < 1)
        die("Tunnel Read Error");
      if (send(sckt, buffer, msg_len, 0) < msg_len)
        die("Socket Write Error");
    }
  }
}

// Transfers stream oriented data
// between tunnel device and socket
void transfer_stream(int sckt, int tun) {
  struct pollfd pfds[2];

  unsigned char sckt2tun[MAX_DATA_SIZE*2];
  unsigned char *data_ptr = sckt2tun;
  int data_len = 0;
  int recv_status, recv_len = 0;

  unsigned char tun2sckt[MAX_DATA_SIZE*2];
  unsigned char *msg_ptr = tun2sckt+2;
  int msg_len;

  pfds[0].fd = sckt;
  pfds[0].events = POLLIN;
  pfds[1].fd = tun;
  pfds[1].events = POLLIN;

  while (1) {
    poll(pfds, 2, -1);

    if (pfds[0].revents == POLLIN) {
      // START RECEIVE STREAM

      // Receive Beginning of Next Packet
      if (recv_len == 0 && data_len == 0) {
        if ((recv_len = recv(sckt, sckt2tun,
                             MAX_DATA_SIZE, 0)) < 1)
          die("Socket Recv Error");
        if (recv_len < 2)
          recv_len += recv(sckt, data_ptr+recv_len, 1, 0);
        data_len = data_ptr[0] + (data_ptr[1] << 8);
```

```
      } else if (data_len > recv_len) {
        if ((recv_status = recv(sckt, data_ptr+recv_len,
                                data_len-recv_len, 0)) < 1)
          die("Socket Recv Error");
        recv_len += recv_status;
      }

      // Process leading packets
      while (recv_len > data_len) {
        if (write(tun, data_ptr+2, data_len-2) < 1)
          die("Tunnel Write Error");
        data_ptr += data_len;
        recv_len -= data_len;
        if (recv_len < 2)
          recv_len += recv(sckt, data_ptr+recv_len, 1, 0);
        data_len = data_ptr[0] + (data_ptr[1] << 8);
      }

      // Process Final Packet
      if (recv_len == data_len) {
        if (write(tun, data_ptr+2, data_len-2) < 1)
          die("Tunnel Write Error");
        recv_len = 0;
        data_len = 0;
        data_ptr = sckt2tun;
      }
      // END RECEIVE STREAM
    }
    if (pfds[1].revents == POLLIN) {
      if ((msg_len = read(tun, msg_ptr, MAX_MSG_SIZE)) < 1)
        die("Tunnel Read Error");
      msg_len += 2;
      tun2sckt[0] = msg_len & 0xff;
      tun2sckt[1] = msg_len >> 8;
      if (send(sckt, tun2sckt, msg_len, 0) < msg_len)
        die("Socket Write Error");
    }
  }
}

int main(int argc, char *argv[]) {
  int sckt, tun;
  char *local_ip, *local_port, *remote_ip;
  char *remote_port, *dev, role, proto;

  if (argc != 8 )
    die("Wrong Number of Arguments");

  local_ip    =  argv[1];
  local_port  =  argv[2];
```

```
  remote_ip   =  argv[3];
  remote_port =  argv[4];
  dev         =  argv[5]; // Tunnel Device Name
  role        = *argv[6]; // Act as Client or Server
  proto       = *argv[7]; // Protocol to use for tunnel

  setvbuf(stdout, NULL, _IONBF, 0);

  tun = get_tun(dev);
  sckt = get_socket(proto);

  if ( proto == 'U') role = 'U';

  switch(role) {
    case 'U' :
      bind_socket(sckt,local_ip,local_port);
      connect_socket(sckt,remote_ip,remote_port);
      break;
    case 'S' :
      bind_socket(sckt,local_ip,local_port);
      listen_socket(sckt);
      sckt = accept_connection(sckt);
      break;
    case 'C' :
      connect_socket(sckt,remote_ip,remote_port);
      break;
    default :
      die("Unknown Role");
  }

  printf("Connection Established (%c)\n", proto);

  switch(proto) {
    case 'U' :
    case 'S' :
    case 'X' :
      transfer_data(sckt,tun);
      break;
    case 'T' :
      transfer_stream(sckt,tun);
      break;
    default :
      die("Invalid Protocol Type");
  }

  close(sckt); close(tun);

  return 0;
}
```

References

[1] ARPA2 Research. Web, 2015. http://research.arpa2.org.

[2] Cisco Systems, Inc., http://docwiki.cisco.com/wiki/Internetworking_Terms:_Generic_Routing_Encapsulation_(GRE). *Internetworking Terms: Generic Routing Encapsulation (GRE)*, 2011.

[3] K. Claffy, Greg Miller, and Kevin Thompson. The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone. Technical report, ISOC, https://www.isoc.org/inet98/proceedings/6g/6g_3.htm, 1998.

[4] Eric F Crist and Jan Just Keijser. *Mastering OpenVPN*. Packt Publishing Ltd., August 2015.

[5] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, and G. Zorn. Point-to-Point Tunneling Protocol (PPTP). RFC 2637, RFC Editor, July 1999.

[6] Asim Iqbal. SCTP - DataTAG. Technical report, CERN, http://datatag.web.cern.ch/datatag/WP3/sctp/, 2003.

[7] Stephen Kent and Randall Atkinson. IP Encapsulating Security Payload (ESP). RFC 2406, RFC Editor, November 1998.

[8] David Murray and Terry Koziniec. The State of Enterprise Network Traffic in 2012. Technical report, Murdoch University, 2012.

[9] L. Ong and J. Yoakum. An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286, RFC Editor, May 2002.

[10] OpenVPN Community Wiki and Tracker, https://community.openvpn.net/openvpn/wiki/Openvpn23ManPage. *OpenVPN 2.3 Manual*, 2016.

[11] Michael Scharf and Sebastian Kiesel. Quantifying Head-of-Line Blocking in TCP and SCTP. Technical report, IETF, July 2013.

[12] Robin Seggelmann. SCTP: Strategies to Secure End-To-End Communication. Technical report, University of Duisburg-Essen, 2012.

[13] Ken Smith. FreeBSD 7.0-RELEASE Announcement. Web, 2008. https://www.freebsd.org/releases/7.0R/announce.html.

[14] R. Stewart. Stream Control Transmission Protocol. RFC 4960, RFC Editor, September 2007.

[15] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. RFC 3758, RFC Editor, May 2004.

[16] Olaf Titz. Why TCP Over TCP Is A Bad Idea. Web, 2001. http://sites.inka.de/bigred/devel/tcp-tcp.html.

[17] Bruce Zamaere, Markus Hidell, and Peter Sjdin. CVPN: A multi-homed VPN Solution for Remote Patient Monitoring. Technical report, KTH Royal Institute of Technology, 2012.