Implementing a PPP-based SSL VPN Client for Clavister Security Gateways

Erik Wiséen Åberg
erik.w.aberg@gmail.com

August 10, 2016
Master’s Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Jerry Eriksson
Examiner: Henrik Björklund

DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN
Abstract

In this thesis, we describe how an SSL VPN client utilizing the UNIX program called pppd was developed. pppd enables the creation of point to point links through the Point-To-Protocol(PPP) which is capable of carrying protocols such as IP, allowing the VPN to connect to the SSL VPN service of Clavisters security gateways. We then perform an analysis of the degradation of network communication throughput caused by this approach. Our findings suggest that the usage of pppd may be the cause of lower than acceptable throughput, making this approach non-viable as the basis for a VPN solution.
Acknowledgments

I would like to thank Clavister for formulating the thesis, for providing the means to execute it and the people working at the company. A special thanks to Mattias Fredriksson at Clavister for his guidance and support, and Jonas Domeij for choosing to provide hints instead of answers to lot of technical questions.

I finally want to thank Jerry Eriksson at Umeå University, who supervised the formal aspects relating to the thesis.
Contents

1 Introduction .............................................. 1
  1.1 Clavister ........................................ 1
  1.2 Purpose ........................................ 1
  1.3 Related Work .................................... 2
  1.4 Outline ........................................ 2

2 Background ........................................... 5
  2.1 Virtual Private Networks ............................ 5
    2.1.1 Introduction ................................ 5
    2.1.2 Definition ................................ 5
    2.1.3 VPN Implementation ......................... 6
  2.2 Secure Communication ............................... 6
  2.3 The Point-To-Point Protocol ......................... 7
    2.3.1 Introduction ................................ 7
    2.3.2 Encapsulation ............................... 7
    2.3.3 Configuration ............................... 8
    2.3.4 Framing ................................ 8
    2.3.5 The PPP Daemon ............................. 9

3 System Description .................................... 11
  3.1 System Architecture ............................... 11
    3.1.1 Utilizing pppd ............................. 11
    3.1.2 System Overview ........................... 12
    3.1.3 Component Description .................... 14
    3.1.4 System API ................................. 15
    3.1.5 HDLC-like Framing Algorithms ............ 16
  3.2 User Interface .................................. 18

4 Results .............................................. 21
  4.1 Performance Evaluation ......................... 21
    4.1.1 Test Equipment ............................ 21
4.1.2 Evaluation Method ............................................... 22
4.1.3 Latency ............................................................ 23
4.1.4 Bandwidth Throughput ......................................... 23
4.1.5 Function Measurement .......................................... 25
4.1.6 Long Term Usage ............................................... 25

4.2 Conclusions .......................................................... 26
4.2.1 Observations On Performance ................................... 26
4.2.2 Discussion .......................................................... 27
4.2.3 Limitations ......................................................... 28
4.2.4 Future work ......................................................... 28
4.2.5 Practical Application ............................................. 28

References ................................................................. 31
Chapter 1

Introduction

A Virtual Private Network (VPN) extends a private network across the public internet so that a remote computer effectively may as well be physically connected to that private network. VPN solutions vary in implementation primarily based on which protocol that is used as the underlying method of transportation, each with its own set of pros and cons. A Secure Socket Layer (SSL) based VPN technology has among its benefits that it often is allowed to pass through firewalls, due to the prevalence of HTTPS servers.

This thesis describes how a prototype VPN client was implemented and how its performance was evaluated. The implementation relies heavily on a program called `pppd` that is provided on OS X and Linux operating systems. `pppd` provide a non-complex method for establishing point to point links between computers through the Point-To-Point Protocol (PPP), allowing exchange of data by sending PPP frames capable of carrying a wide range of different communication protocols. In this way, communication within a Virtual Private Network is achieved by passing IP packets within PPP frames over a secure SSL connection. However, while utilizing `pppd` remove complexity in the developed software, an effort must be made to evaluate the efficiency of using this program as the basis for PPP-communication.

1.1 Clavister

Clavister is a company that offers both software and appliance firewall solutions, with headquarter office in Örnsköldsvik. Clavister stands out among their competitors in that they have built the operating system running their firewalls themselves, rather than extending upon the Linux operating system which is the prevalent approach among their competitors. Clavister sought to investigate the possibility of developing a VPN on the OS X and Linux platforms that is compliant with their currently existing VPN service, and thus allowed a basis for this thesis to take form.

1.2 Purpose

It is today common for companies to maintain various types of internet services such as databases, file storage servers, electronic mail servers and intranet chat, to name a few. In order to protect sensitive information from eavesdroppers, these services are typically only accessible while being physically connected to the internal network infrastructure of
the company. However, it may at times be necessary for employees to access such company resources when working remotely, and in order to provide this service, a common solution is to set up a VPN.

In this thesis we investigate the possibility of implementing a rudimentary VPN client that utilize the Point-to-Point daemon (pppd), a program that is used to establish end-to-end tunnels and is provided in default installations of Unix-like operating systems such as OS X and Linux. The idea is to reduce the complexity of the implementation by piggybacking on pppd. The research question this thesis aim to answer:

Is high network throughput achievable while piggybacking on **pppd**?

Security Gateways developed by Clavister provide a wide variety of services. One service they provide is an SSL VPN service that is based upon the Point-to-Point Protocol (PPP). A key point that is worth noting in this regard is that Clavisters implementation of the Point-to-Point Protocol includes a subtle change to the standard PPP header format. In practice, this means that no publicly available software utilizing PPP can be used without modification to the PPP header. Thus, to investigate the feasibility of utilizing **pppd** and connection to a Clavister security gateway running the SSL VPN service, the goal in terms of software development is to implement a prototype client capable of interfacing with the SSL VPN service.

### 1.3 Related Work

PPP Design and Debugging, published in the year 2000 by James Carlson[1], details use cases and characteristics of PPP. Carlson argues that PPP may be used as the basis for a VPN solution, by encapsulating PPP frames in IP packets addressed to a gateway. The gateway would then decapsulate the PPP frame and deliver the data to the private network. In this example, the PPP frames may themselves carry IP packets, essentially creating an IP over IP tunnel, which is exactly what we want to achieve.

Using PPP in this regard has some similarities with the Point-to-Point Tunneling Protocol (PPTP), which allows PPP to be tunneled through an IP network. PPTP uses an enhanced Generic Routing Encapsulation (GRE) mechanism for carrying PPP packets which include congestion-control facilities[6]. While PPTP provides encapsulation for IP packets over IP networks, similarly to PPP, its specification does not describe encryption features.

There exists various projects on the internet that add encryption to PPP connections in clever ways. One example of this is described by Nurullah Akkaya in his post “Poor man’s VPN using PPP over SSH”[^1], in which he explains how the tunneling support in SSH can be leveraged to maintain a PPP link within an encrypted SSH connection. However, this kind of approach does not allow modifications to the raw PPP packets as required to communicate with the proprietary PPP-implementation used in Clavister VPN service.

### 1.4 Outline

The thesis is organized into four chapters. Chapter 2 introduces the underlying theory that the VPN client is based upon, describing what a Virtual Private Network is, how secure communication is achieved and how the Point-To-Protocol works. Chapter 3 contains a

description of the system architecture and the graphical design. Chapter 4 explains how
the system was evaluated in terms of performance, how the test results were interpreted
and is then concluded with a discussion on the results, limitations of the system, a few
suggestions on future development and finally a set of notes on the practical applications of
the developed software.
Chapter 2

Background

This chapter covers fundamental concepts relating to the implementation of the prototype. We define what a Virtual Private Network is, how secure communication is achieved and how the Point-To-Point Protocol works.

2.1 Virtual Private Networks

This section describes what a Virtual Private Network (VPN) is, its defining characteristics and an overview of how one might be implemented.

2.1.1 Introduction

What is a VPN? According to Gentry[5], a VPN is essentially a system that privatize data from point A to point B, by protecting the data from disclosure during transmission over any given network path. By securing data privacy, information that must be protected from disclosure, modification and unauthorized use, can be safely transmitted over public networks as though they were private.

2.1.2 Definition

The characteristics of a Virtual Private Network is described as follows:

1. Virtual - as data may be exchanged over public networks.
2. Private - as data is protected by encryption.
3. Network - as devices share a communication path.

Additionally, a VPN should provide the following services[5]:

1. Authentication of users, so that only authorized parties may participate in the network.
2. Integrity should be integrated into the encryption method, meaning that data modifications during transit cannot avoid detection.
3. Non-Repudiation ensuring that the origin of transmission cannot be faked.
2.1.3 VPN Implementation

VPNs are created by establishing an end-to-end authenticated tunnel, through which encrypted packets are passed[7]. While there exist many varieties of tunneling techniques, they are typically implemented either at the network layer or the data link layer[11]. A hypothetical implementation of a network layer VPN may be based on the Internet Protocol(IP). Such an implementation would then append an additional IP header onto network layer packets sent from one end of the tunnel, so that the added header has the destination of the other end of the tunnel. Once received on the other end of the tunnel, the additional header is removed, revealing the original packet to be routed toward its destination[7].

In practice, this could be achieved by creating a virtual Network Interface Card(NIC), effectively simulating a different route that a computer may use to communicate over the internet. Communication over this virtual NIC, such as IP packets, may then be intercepted by a VPN application which would encrypt the packet and append an IP header so that it may be routed to the other end of the tunnel over the physical wire. An illustration of this is found in Figure 2.1. Note that the Data Link layer is painted gray to reflect the example implementation described above.

![Figure 2.1: The flow of data within a computer node with or without using VPN.](image)

2.2 Secure Communication

The Secure Sockets Layer(SSL), and its successor, the Transport Layer Security(TLS), are both commonly referred to as SSL. The main goal of the TLS protocol is to provide privacy and data integrity between two communicating parties. TLS is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. The TLS Record Protocol provides a private and reliable connection, while the TLS Handshake Protocol provides
authentication and negotiation of encryption parameters[4].

Quoting the abstract of RFC 5246 for TLS version 1.2:

“The TLS protocol provides communications security over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.”[4]

Recall the definition of a VPN in Section 2.1.2, a VPN should provide authentication, integrity and non-repudiation. TLS fulfill two of these requirements: integrity in prevention of tampering and non-repudiation in prevention of message forgery. SSL/TLS does however not typically provide strong authentication of both parties. For example, usage of SSL/TLS in secure web page viewing provides authentication of the web server, but it does not typically require authentication of the web client. Thus, to fulfill the third requirement for a VPN service, additional steps must be taken in order to authenticate the client. We describe how this is achieved in the VPN client in Section 2.3.5.

2.3 The Point-To-Point Protocol

This section provides an introduction to the Point-To-Point Protocol and an overview of its constituent parts. A description of how it fits into the bigger picture of TCP/IP networking is also provided.

2.3.1 Introduction

The Point-To-Point Protocol (PPP) is used to exchange data between two communication devices over any kind of full-duplex physical media[2], but has historically often been used to connect dial-up modems to an ISP. The purpose of PPP as described in RFC 1661, is to provide a method for transporting multi-protocol datagrams over point-to-point links. To that end, RFC 1661 for PPP define the following functions[9] listed below, further described in the following sections:

1. Encapsulation of datagrams.
3. A family of Network Control Protocols (NCPs).

2.3.2 Encapsulation

The structure of encapsulated datagrams in PPP is illustrated in Figure 2.2. The Protocol field identifies the protocol of the datagram encapsulated in the Information field. The Information field may in turn be padded with an arbitrary number of octets up to the Maximum Receive Unit (MRU) in the Padding field[9].

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Information</th>
<th>Padding</th>
</tr>
</thead>
<tbody>
<tr>
<td>8/16 bits</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

Figure 2.2: PPP encapsulation
However, the PPP encapsulation does not indicate either the beginning or the end of an encapsulated datagram. This means that encapsulated datagrams must be framed in order to allow individual such encapsulations to be identified on the receiving end. We describe how HDLC-like framing is used to achieve the framing of PPP encapsulated packets in Section 2.3.4.

### 2.3.3 Configuration

Detailed descriptions of the Link Control Protocol (LCP) and the Network Control Protocols (NCPs) is not necessary for the purpose of this thesis - we therefore only provide a brief overview of their function in PPP.

Specifically, it suffice to say that LCP is responsible for negotiating and testing the configuration of communication over the point-to-point link. This is achieved by exchanging LCP packets to the effect of either acknowledging or rejecting parameters of the configuration that a peer may request[9]. Once both peers agree upon the link-configuration, authentication of either or both peers may take place, followed by configuration of network-layer protocols such as IP.

PPP is capable of encapsulating a variety of network-layer protocols and may use several over the same communication link. A separate NCP is thus provided for each such network-layer protocol in order to negotiate its properties. The Internet Protocol Control Protocol (IPCP) is an example of a NCP that is used to negotiate the configuration of the Internet Protocol (IP), such as the assignment of IP addresses[3].

### 2.3.4 Framing

As previously mentioned, PPP encapsulated packets are transmitted in HDLC-like frames. This allow the receiving end to identify the beginning and end of PPP frames, and detect erroneous frames through the Frame Check Sequence (FCS) field, which is similar to a cyclic redundancy check (CRC). The frame structure described in “PPP in HDLC-like Framing” RFC 1662[10] is illustrated in Figure 2.3.

![Figure 2.3: HDLC-like frame structure.](image)

The illustration shows that there now are five fields surrounding the PPP encapsulated packet. A valid frame always contain these fields[10]:

- **Flag Sequence** - a single octet of value 0x7e, marking the beginning and end of a frame.

- **Address Field** - a single octet which specifies an address. In PPP, the value is fixed to 0xff - the All-Stations address.

- **Control Field** - a single octet fixed to the value 0x03.
2.3. The Point-To-Point Protocol

- **Frame Check Sequence (FCS)** - consisting of one or two octets. The FCS is similar to a cyclic redundancy check (CRC), an error detection code, calculated over all bits of the Address, Control, Protocol, Information and Padding fields.

**Frame Encoding**

In addition to the header that is added when a PPP packet is framed in the HDLC-like framing format, all transmitted octets are encoded to make sure that a PPP frame can be properly identified. As shown in Figure 2.3, each frame starts and ends with the octet 0x7E. If the *Information* field, which contains the encapsulated packet, also were to contain the octet 0x7E it would open up for the possibility of a premature end-of-frame detection. Thus, the special character 0x7D also known as the *Control Escape Sequence*, is used as a flag to notify the receiver that the immediately following octet Y has been modified by performing the bit-wise XOR operation on the original octet X: \( Y = X \oplus 0x20 \). The original octet is recovered using the same computation again, \( X = Y \oplus 0x20 \). This allows the *Flag Sequence character* (0x7E) to be transmitted as 0x7D 0x5E without being interpreted as the end of a frame. Additionally, values in the inclusive range \([0x00, 0x1F]\), as well as the special characters 0x7D and 0x7E are escaped in this way.

Thus, detecting the beginning of a frame in a stream of octets can be achieved by searching for a sequence of the octets 0x7E 0xFF 0x7D 0x23, where 0x23 translates to the octet 0x03 which is the *Control Field* of the HDLC-like frame.

### 2.3.5 The PPP Daemon

The native PPP Daemon program `pppd` in OS X and Linux provides all features of PPP. An illustrative example of how PPP could be used for TCP/IP-based communications in a typical system is given in Figure 2.4. Note that one would typically use either PPP or Ethernet at the Data Link Layer, not both.

![Figure 2.4: Usage of PPP for TCP/IP communications.](image)

As shown in the illustration, `pppd` also provide authentication of either one or both peers, a feature that will be utilized during the setup of the VPN. At the time of this writing, `pppd` supports three authentication protocols[8]: the Password Authentication Protocol
(PAP), Challenge Handshake Authentication Protocol (CHAP, including MS-CHAP & MS-CHAPv2), and Extensible Authentication Protocol (EAP).

Additionally, once a Point-to-Point link has been established using `pppd`, it automatically creates a virtual network interface that is specified as the default internet access route on the machine. This means that applications requesting internet access will attempt to find a path toward its destination through the PPP interface.
Chapter 3

System Description

This chapter describes the system architecture of the program in terms of how `pppd` was utilized, the tasks that the system is to perform and how incoming and outgoing data flows through the program through its various components. This is followed by a description of how the VPN client may be used by external programs through its application programming interface (API) and an overview of the algorithms developed to perform HDLC-like framing. Lastly, we showcase the graphical user interface for OS X users and the command-line interface which may be used on either OS X or Linux.

3.1 System Architecture

Implementation of the VPN system was divided into two parts: the VPN client written in C and a graphical user interface written in Objective-C which utilize the Cocoa library for OS X. The VPN client is compiled both as a library which graphical applications can use to launch and control the VPN, and as a standalone CLI application. In this section we will begin by describing how `pppd` was utilized in the VPN client, followed by an overview of the structure of the program.

3.1.1 Utilizing `pppd`

Recalling the illustrated use case for `pppd` in Section 2.3.5, it is realized that `pppd` can be utilized in the VPN client by rerouting its input and output through the VPN and redirect it over a TCP/IP connection to a VPN server, as shown in Figure 3.1.
This effectively means that we can pass PPP encapsulated IP-packets as the encrypted payload of TCP/IP-packets transmitted over an SSL connection to the VPN server. Upon reception, the server then decrypt the payload, unpacks the PPP encapsulated packet and reroute it towards its destination.

**Framing**

The output produced by `pppd` is in the HDLC-like frame format described previously, and `pppd` additionally expects its input to be structured in the same fashion. However, since Clavister developed its own PPP implementation in their security gateways knowing that the PPP encapsulated datagrams in turn would be encapsulated into TCP/IP packets, they could omit the HDLC-like framing as an optimization. But since we want to utilize `pppd`, we must support the encoding and decoding of HDLC-like frames entering and leaving `pppd`. While this is not a particularly complex task in itself, when implementing a VPN client one would seek to minimize the latency that is accumulated as packets are encrypted and modified in user-space before being sent. By performing this framing we add additional latency for a task that is seemingly unnecessary, but that we can’t avoid when using `pppd`. The impact on latency when performing this task is shown in Section 4.1 and further discussed in Section 4.2.2.

**3.1.2 System Overview**

Using what we have learned so far, we devise a plan realized as a number of high level tasks required to be carried out by the VPN client:

1. Initiate an SSL/TLS connection to the VPN server.
2. Authenticate the VPN server through SSL/TLS.
3. Request VPN access from the VPN server.
4. Establish a Point-To-Point link through `pppd`.
5. Reroute the input and output of the `pppd`-instance through the VPN client.
6. Decode outgoing HDLC-like frames to PPP-encapsulated packets and send them to the VPN server through the SSL/TLS connection.

7. Perform HDLC-like framing on PPP-encapsulated packets received on the SSL/TLS connection and pass them to the `pppd`-instance.

Once a `pppd`-instance has been initiated, it will commence to transmit LCP-packets to configure the link and perform authentication of the client. Through IPCP, the client is provided with a set of IP-addresses: its local IP, the IP of the gateway as well as the IP of the primary and secondary DNS. When completed, a default route for IP communication is set up that points towards the VPN gateway, through the `pppd`-instance. At that point, the VPN client is tasked with continuously repeating steps six and seven, until termination of the connection or program.

The data-flow of the program is conceptualized in Figure 3.2 and is referenced in the following overview of the program. By inspecting this illustration, we may first note that the application utilize four threads, each with their own specific task that are repeated continuously. You may also note that these threads are represented as circles to symbolize this behavior and that two circles are connected by a uni-directional arrow. These arrows illustrate how data is transferred from one thread to another via two queues: an Input-queue and an Output-queue. More specifically, data packets that are sent to the client from the VPN server arrive at our end of the SSL connection at the SSL Reader, where the data is unpacked and placed in the Input-queue. The PPPD Writer monitors the Input-queue, picks up available data, encapsulates it into a HDLC-like frame and finally forwards it to `pppd`. Similarly, data that is produced by `pppd` arrives at the PPPD Reader where data is extracted from its HDLC-like framing and placed in the Output-queue. The SSL Writer monitors the queue, packs the data into a format accepted by the VPN server and is then sent over the SSL connection.

![Figure 3.2: The data flow of data within the VPN client.](image_url)
3.1.3 Component Description

The VPN client consists of the following main components, followed by a brief description of their responsibilities:

- **config** - Parses a supplied XML-formatted configuration file that contains options such as username, server address and port.

- **hdlc** - The HDLC-module is used to encode PPP-encapsulated packets received from the SSL VPN service into HDLC-like frames, which then are passed to the **pppd**-instance. Similarly, HDLC-frames intercepted from the **pppd**-instance is decoded into PPP-encapsulated packets which can be transmitted over the SSL connection.

- **ppp** - Launches the Point-To-Point-daemon( pppd) program and connects it’s input/output descriptors to a pseudo-tty( pty), and include functions allowing us to read/write data from/to the **pppd**-process.

- **pppqueue** - A thread safe and blocking FIFO-queue implementation used to transport data between threads in a producer-consumer kind of fashion.

- **ssl** - Utilizes the OpenSSL library to provide an SSL connection onto which multiple threads may read and write. It also provides basic SSL-server functionality, which was used in the performance testing described in later sections.

- **vpn** - Initiates the VPN client by parsing command-line options, creating data structures, initiating threads as well as providing the public interface available to external applications.
3.1.4 System API

This section is provided to show which functions one would use to utilize the VPN library in an external program. In essence, the user must import the vpn.h file which contains the function declarations listed below. More detailed comments are included in the source.

```c
#define CONFIG_STR_LENGTH 64
struct VPN_CONFIG {
    char server_hostname[CONFIG_STR_LENGTH]; // example.vpn.clavister.com
    uint16_t server_port; // 443
    char user_name[CONFIG_STR_LENGTH]; // patrik
    char user_password[CONFIG_STR_LENGTH]; // secret
    char server_ssl_fingerprint[CONFIG_STR_LENGTH]; // 96:23:c6:9b...
    struct sockaddr_in server_sockaddr; // (internal)
};
enum VPN_STATE { VPN_DISCONNECTED, VPN_CONNECTING, VPN_CONNECTED }

int vpn_init_with_configuration(int log_level, struct VPN_CONFIG *vpn_config);
// Allocates and initiates structs required for VPN execution and starts the logging system.

void vpn_set_state_callback(void (*state_callback_func)(int));
// Declare that the passed function should be called upon future changes of the VPN state.

int vpn_connect(void);
// Connects to the VPN server and initiates thread execution.

void vpn_terminate(void);
// Disconnects from the VPN server and halts thread execution.

int vpn_get_state(void);
// Fetch the current VPN state.

char* vpn_get_local_ip(void);
// Fetch the local IP-address that we are assigned by the VPN server.

char* vpn_get_remote_ip(void);
// Fetch the remote IP-address of the VPN gateway, as assigned by the VPN server.

char* vpn_get_primary_dns(void);
// Fetch the remote IP-address of the DNS1, as assigned by the VPN server.

char* vpn_get_secondary_dns(void);
// Fetch the remote IP-address of the DNS2, as assigned by the VPN server.

uint64_t vpn_get_bytes_received(void);
// Fetch the current count of bytes received during active VPN use.

uint64_t vpn_get_bytes_sent(void);
// Fetch the current count of bytes transmitted during active VPN use.
```
3.1.5 HDLC-like Framing Algorithms

This section describes the algorithms for encoding and decoding HDLC-like frames.

Encoding

To produce a HDLC-like frame, the following algorithm is performed on an input array containing an PPP-encapsulated packet received on the SSL connection to be passed to the pppd-process:

```plaintext
let bufferOffset = 0
let buffer = # The resulting array containing the encoded frame.
let inputArrayPtr = # An array of octets (the PPP-encapsulated packet).
let fcstab = # An array containing 256 pre-computed values as defined in RFC 1662,
    # used in the calculations of the FCS.

buffer[bufferOffset++] = 0x7E # (Flag Sequence)
buffer[bufferOffset++] = 0xFF # (Address)
buffer[bufferOffset++] = 0x7D # (Character Escape)
buffer[bufferOffset++] = 0x23 # (Control)

let 16bitFCS = 0x3de3 # Pre-computed FCS of above buffer insertions.

for each octet in inputArrayPtr
    let value = octet
    16bitFCS = (16bitFCS >> 8) ^ fcstab[(16bitFCS ^ value) & 0xFF] # FCS calculation
    if value < 0x20 || value == 0x7D || value == 0x7E
        buffer[bufferOffset++] = 0x7D # (Character Escape)
        value = value ^ 0x20 # value XOR’ed by 0x20
    fi
    buffer[bufferOffset++] = value
done

value = (16bitFCS & 0xFF) ^ 0xFF # low-order octet of FCS, XOR’ed by 0xFF
if value < 0x20 || value == 0x7D || value == 0x7E
    buffer[bufferOffset++] = 0x7D # (Character Escape)
    value = value ^ 0x20 # value XOR’ed by 0x20
fi
buffer[bufferOffset++] = value

value = (16bitFCS >> 8) ^ 0xFF # high-order octet of FCS, XOR’ed by 0xFF
if value < 0x20 || value == 0x7D || value == 0x7E
    buffer[bufferOffset++] = 0x7D # (Character Escape)
    value = value ^ 0x20 # value XOR’ed by 0x20
fi
buffer[bufferOffset++] = value
buffer[bufferOffset++] = 0x7E # (Flag Sequence)
```


Decoding

A HDLC-framed PPP-encapsulated packet received from the pppd-process is decoded into a PPP-packet in the somewhat simplified manner shown below:

```plaintext
# Header: (Address)0xFF, (Escape)0x7D, (Control)0x23
let 16bitFCS = 0x3DE3 # pre-computed FCS of a PPP header.
let octet_count = 0
let i = 0 + 3 # offset by length of PPP header

for (; i < hdlc_frame_length; i++)
  let value = hdlc_frame[i]

  if value == 0x7D # (Character Escape)
    escaped = true
    continue
  elif escaped
    value = value ^ 0x20
    escaped = false
  fi

  decode_buffer[octet_count++] = value
  16bitFCS = (16bitFCS >> 8) ^ fcstab[(16bitFCS ^ value) & 0xFF]

done

if 16bitFCS != 0xF0B8 # Good final FCS value, see RFC 1662.
  # Checksum check failed.
fi
```
3.2 User Interface

The design of the graphical user interface is not particularly interesting per se, but included here for completeness. The Figure 3.3 depicts the VPN login screen, including server configuration parsed from a configuration file. An icon in the OS X status bar is shown while the application is running whether it is in the foreground or not, to give the user an indication of the current state of the VPN connection. The possible states indicated by the icons are shown in Figure 3.4.

![GUI: Login view and OS X status bar icon.](image)

Figure 3.3: GUI: Login view and OS X status bar icon.
3.2. User Interface

In Figure 3.4 we show the possible states of the VPN client, as well as the menu that becomes visible when clicking on the status icon. The Connect and Disconnect menu-items are self-explanatory. Choosing the Statistics menu-item opens a view containing some statistics relating to the VPN connection, as shown in Figure 3.5. The Network Information box contains the IP addresses assigned to the client by the VPN server, which is used to configure the virtual network interface on the client machine.

![GUI: Status icon menu and possible states.](image)

![GUI: Statistics view.](image)
The Figure 3.6 depicts a user running the VPN software using the command-line interface in a terminal with the following options:

```
bin/OSX_SSL_VPN_Client
-l 1 (Log verbosity 1, where 3 is the max and would show the contents of packets.)
-x config.csslv (The path to a configuration file.)
-p erik (The password used to authenticate the client, a hidden debug option.)
```

Figure 3.6: Running the VPN from a terminal.
Chapter 4

Results

This is the final chapter of the thesis, in which we will begin with a description how the system was evaluated in terms of performance through a series of tests and how the measurements from these were obtained. We then show the results that we collected while performing these tests, how the results were interpreted and then provide a discussion on the implication of these results. The chapter concludes with a few notes on the limitations of the system and suggestions on how it may be further improved.

4.1 Performance Evaluation

This section describes a set of tests that were devised to evaluate the performance of the application in terms of latency, throughput, CPU-utilization and operation over a long time period. The section begins with a description of the test environment, how the measurements were obtained and a specification of the computer hardware used in the tests. This is followed by the results of the various tests. The interpretation of the results is outlined in Section 4.2.1 and further discussed in Section 4.2.2.

4.1.1 Test Equipment

Measurements described in the following sections were carried out between two computers locally connected through a Clavister E20 Security Gateway, as shown in Figure 4.1.

The hardware of the test computers is specified in Table 4.1, where each computer has been assigned an identifier(ID).
### Test Computers

<table>
<thead>
<tr>
<th>ID</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>MacBook Pro 15&quot; Retina, Mid 2014</td>
<td>iMac 27&quot;, Late 2012</td>
<td>Clavister E20 Firewall</td>
<td>HP EliteBook 8470p</td>
</tr>
<tr>
<td>OS</td>
<td>OS X 10.11.5</td>
<td>OS X 10.11.5</td>
<td>Clavister cOS 11.03.00.44-29188</td>
<td>Windows 10</td>
</tr>
<tr>
<td>CPU</td>
<td>2.2-3.4 Ghz i7-4770HQ</td>
<td>2.9-3.6 Ghz i5-3470S</td>
<td>1.7-2.0 GHz Intel Atom</td>
<td>2.60-3.3 GHz i5-3320M</td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Threads</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Memory</td>
<td>16GB 1600MHz</td>
<td>8GB 1600MHz</td>
<td>956MB 1333MHz</td>
<td>4GB 1600MHz</td>
</tr>
<tr>
<td>NIC</td>
<td>100Base-T (USB Adapter)</td>
<td>1000Base-T</td>
<td>1000Base-T</td>
<td>1000Base-T</td>
</tr>
</tbody>
</table>

Table 4.1: Specification of the hardware found in the computers that were used for performance evaluation and the assignment of an ID to each computer.

We illustrate how the computers were connected while performing the tests in Figure 4.1. Note that when computer W was used in a test scenario, the LAN computer A depicted in the illustration was replaced with computer W.

![Computer network of test computers.](image.png)

Figure 4.1: Computer network of test computers.

### 4.1.2 Evaluation Method

During tests of the VPN client, the Clavister E20 as shown with ID=C in the hardware specification in Table 4.1 provided the VPN service endpoint which the VPN client connected to. With that said, the following tests were performed in four different connection configurations: A test of type *N/A-N/A* means that the test was carried out between A and B without running the VPN client. Similarly, a test of type *vpn-N/A* would mean that computer A was connected via the VPN service, while *vpn-vpn* means that both computers were connected to the VPN service of E20. Tests of type *vpn'-vpn'*, however, indicates that both A and B were running the VPN client, but neither of them were connected to the VPN service of E20. Instead, one of the computers acted as an SSL/TLS server during the initiation of the connection and then used *pppd* to establish a point-to-point communication link between the two computers, in the same way that a VPN connection would be established normally when connecting to the VPN service of E20.
4.1.3 Latency

To get an idea of how much latency is added to network communication while using the VPN, the communication latency between computer A and B of each test configuration was measured using the POSIX ping utility program. The measurements are summarized in Table 4.2.

<table>
<thead>
<tr>
<th>Test Type</th>
<th>n</th>
<th>min</th>
<th>avg</th>
<th>max</th>
<th>stddev</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A-N/A</td>
<td>20</td>
<td>0.704</td>
<td>0.800</td>
<td>0.890</td>
<td>0.047</td>
</tr>
<tr>
<td>vpn-N/A</td>
<td>20</td>
<td>11.09</td>
<td>15.64</td>
<td>20.77</td>
<td>2.861</td>
</tr>
<tr>
<td>vpn-vpn</td>
<td>20</td>
<td>31.23</td>
<td>36.16</td>
<td>40.70</td>
<td>3.056</td>
</tr>
<tr>
<td>vpn'-vpn'</td>
<td>20</td>
<td>1.161</td>
<td>1.463</td>
<td>1.706</td>
<td>0.125</td>
</tr>
</tbody>
</table>

Table 4.2: Statistics from 20 pings between computers A and B, connected through C.

4.1.4 Bandwidth Throughput

The iperf3 software was used to get an estimate of the maximum possible bandwidth throughput rate in the LAN environment shown in Figure 4.1. iperf3 is a client/server application, where the client seeks to transfer as much data as possible during a 10 second interval over a TCP/IP-connection to an iperf3-server. In the context of our throughput tests, this means that one of the computers runs in server mode, while the other acts as a client. Each individual measurement provides a transmission rate for the sender, the iperf3-client, and a reception rate for the receiver, the iperf3-server.

In addition to the previously described test notation, a test of type $N/A(A)-N/A(B)$ means that the computer with $ID = A$ acted as the iperf3-client while the computer with $ID = B$ acted as the iperf3-server. The iperf3-server was initiated with the following command for all tests: `iperf3 -s`. The first set of measurements show the results using a single TCP stream with the client machine initiating the test with: `iperf3 -c 'IP of server'`.

Additionally, the CPU-utilization of each computer was measured for the duration of each test. For computers A and B, CPU-utilization was measured using a script that execute `ps -A -o %cpu | awk 's+=$1 END {print s}'` every second during each test. The result of running the script was a list of values, where each value was the sum of the CPU usage of each running program. The CPU values reported in Tables 4.3-4.5 are the average of each such list of values. The Clavister E20 provides a graph with a curve representing its CPU usage in percent over time. However, as there is no grid in the resulting graph it was difficult to know the exact value of any given point on the curve. The reported CPU-utilization for the E20 were gathered by eyeballing the graph, picking the highest peak in the graph and estimating its value.
Chapter 4. Results

iperf3: 1 TCP Stream

<table>
<thead>
<tr>
<th>Test Type</th>
<th>Sender/Receiver</th>
<th>CPU</th>
<th>Throughput</th>
<th>E20 CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A(A)-N/A(B)</td>
<td>Sender</td>
<td>2.9%</td>
<td>94.2 Mbits/s</td>
<td>0 %</td>
</tr>
<tr>
<td></td>
<td>Receiver</td>
<td>1%</td>
<td>94.2 Mbit/s</td>
<td></td>
</tr>
<tr>
<td>vpn(A)-N/A(B)</td>
<td>Sender</td>
<td>2.6%</td>
<td>0.480 Mbit/s</td>
<td>0 %</td>
</tr>
<tr>
<td></td>
<td>Receiver</td>
<td>1.4%</td>
<td>0.381 Mbit/s</td>
<td></td>
</tr>
<tr>
<td>vpn'(A)-vpn'(B)</td>
<td>Sender</td>
<td>2.3%</td>
<td>0.624 Mbit/s</td>
<td>0 %</td>
</tr>
<tr>
<td></td>
<td>Receiver</td>
<td>1.1%</td>
<td>0.521 Mbit/s</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Single stream test between computers A and B, where A was the sender and B the receiver.

The second set contains measurements from using 120 parallel TCP streams with the client machine initiating the measurement with `iperf3 -c 'IP of server' -P 120`.

iperf3: 120 parallel TCP Streams

<table>
<thead>
<tr>
<th>Test Type</th>
<th>Sender/Receiver</th>
<th>CPU</th>
<th>Throughput</th>
<th>E20 CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A(A)-N/A(B)</td>
<td>Sender</td>
<td>87.2%</td>
<td>97 Mbit/s</td>
<td>0 %</td>
</tr>
<tr>
<td></td>
<td>Receiver</td>
<td>76.8%</td>
<td>93.3 Mbit/s</td>
<td></td>
</tr>
<tr>
<td>vpn(A)-N/A(B)</td>
<td>Sender</td>
<td>38.2%</td>
<td>20.9 Mbit/s</td>
<td>7 %</td>
</tr>
<tr>
<td></td>
<td>Receiver</td>
<td>5%</td>
<td>8.4 Mbit/s</td>
<td></td>
</tr>
<tr>
<td>vpn'(A)-vpn'(B)</td>
<td>Sender</td>
<td>34.7%</td>
<td>18.4 Mbit/s</td>
<td>10 %</td>
</tr>
<tr>
<td></td>
<td>Receiver</td>
<td>14.5%</td>
<td>6.0 Mbit/s</td>
<td></td>
</tr>
<tr>
<td>vpn'(A)-vpn'(B)</td>
<td>Sender</td>
<td>64.8%</td>
<td>29.8 Mbit/s</td>
<td>0 %</td>
</tr>
<tr>
<td></td>
<td>Receiver</td>
<td>37.9%</td>
<td>17.3 Mbit/s</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: 120 stream test between computers A and B, where A was the sender and B the receiver.

The measurements found in Table 4.5 were gathered in the same manner as before, but between computer W and B. Note also that computer W use the Windows 10 operating system, the SSL VPN client developed by Clavister and that the throughput is no longer limited by the 100Base-T USB network adapter used by computer A. Furthermore, the reported CPU utilization values on the sending side where obtained by selecting the highest peak in the CPU graph of the Performance Tab in the Task Manager in Windows.

iperf3: 1 TCP Stream

<table>
<thead>
<tr>
<th>Test Type</th>
<th>Sender/Receiver</th>
<th>CPU</th>
<th>Throughput</th>
<th>E20 CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A(W)-N/A(B)</td>
<td>Sender</td>
<td>9%</td>
<td>941 Mbit/s</td>
<td>18 %</td>
</tr>
<tr>
<td></td>
<td>Receiver</td>
<td>14%</td>
<td>941 Mbit/s</td>
<td></td>
</tr>
<tr>
<td>vpn(W)-N/A(B)</td>
<td>Sender</td>
<td>12%</td>
<td>66.3 Mbit/s</td>
<td>62 %</td>
</tr>
<tr>
<td></td>
<td>Receiver</td>
<td>3.5%</td>
<td>66.1 Mbit/s</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Single stream test between computers W and B, where W was the sender and B the receiver.
4.1. Function Measurement

The Instruments application is a tool provided with Apple XCode that is capable of measuring how much time that is spent in individual functions during execution of an application. Figure 4.2 and Figure 4.3 show the result from a measurement using this tool. The leftmost column in either figure shows execution time in percent relative to the total execution time of the application during the performed test. The four expanded root nodes in the rightmost column in either figure are functions executed by individual threads.

The measurements were observed during an iperf3 test of type vpn'(A)-vpn'(B) with 120 streams. Measurements carried out on the receiving side are shown in Figure 4.2, and measurements on the sending side are shown in Figure 4.3. Furthermore, functions with names starting with ppp are functions which relate to the communication between the VPN client and the pppd program. Functions with names starting with ssl are functions which relate to the SSL-based communication between the local VPN client and the remote VPN client.

<table>
<thead>
<tr>
<th>Total %</th>
<th>Symbol Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>47.5</td>
<td>_v_pthread_body 0x24ae57</td>
</tr>
<tr>
<td>47.5</td>
<td>_v_pthread_body 0x24ae57</td>
</tr>
<tr>
<td>36.7</td>
<td>ppp_writer OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>9.1</td>
<td>hdc_encode OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>1.6</td>
<td>ppp_queue_pop OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>29</td>
<td>_v_pthread_body 0x24ae58</td>
</tr>
<tr>
<td>29</td>
<td>_v_ssl_reader OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>27.8</td>
<td>_v_ssl_read_n OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>0.9</td>
<td>ppp_queue_push OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>12.7</td>
<td>_v_pthread_body 0x24ae59</td>
</tr>
<tr>
<td>12.7</td>
<td>_v_ssl_writer OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>9.1</td>
<td>_v_ssl_writer OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>3.6</td>
<td>ppp_queue_pop OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>10.5</td>
<td>_v_pthread_body 0x24ae56</td>
</tr>
<tr>
<td>10.5</td>
<td>_v_pthread_body 0x24ae56</td>
</tr>
<tr>
<td>6.9</td>
<td>_v_pthread_body 0x24ae56</td>
</tr>
<tr>
<td>3.1</td>
<td>ppp_decode OSX_SSL_VPN_Client</td>
</tr>
</tbody>
</table>

Figure 4.2: Measurement on the receiver of a vpn'(A)-vpn'(B) test with 120 streams.

<table>
<thead>
<tr>
<th>Total %</th>
<th>Symbol Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.1</td>
<td>_v_pthread_body 0x24ae435</td>
</tr>
<tr>
<td>30.1</td>
<td>_v_ssl_reader OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>18.6</td>
<td>ppp_reader OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>10.2</td>
<td>ppp_read OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>28.2</td>
<td>_v_pthread_body 0x24ae437</td>
</tr>
<tr>
<td>28.2</td>
<td>_v_ssl_reader OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>26</td>
<td>ssl_read_n OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>1.8</td>
<td>ppp_queue_pop OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>26.7</td>
<td>_v_pthread_body 0x24ae438</td>
</tr>
<tr>
<td>26.7</td>
<td>_v_ssl_writer OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>24.5</td>
<td>ssl_write OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>1.8</td>
<td>ppp_queue_pop OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>14.2</td>
<td>_v_pthread_body 0x24ae436</td>
</tr>
<tr>
<td>14.2</td>
<td>_v_ssl_writer OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>9.6</td>
<td>ssl_write OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>2.4</td>
<td>ppp_queue_pop OSX_SSL_VPN_Client</td>
</tr>
<tr>
<td>1.2</td>
<td>hdc_encode OSX_SSL_VPN_Client</td>
</tr>
</tbody>
</table>

Figure 4.3: Measurement on the sender of a vpn'(A)-vpn'(B) test with 120 streams.

4.1.6 Long Term Usage

It is important to make sure that a VPN application can run without interruption during long periods of time. Computer B running an active VPN session was therefore set up to continuously perform HTTP GET X every second, where X was an URL to a 1MB image. The computer was then left unmanned and ran without errors during a 72 hour time period. While this does not guarantee that the program is error-proof, it does indeed indicate that the application has some degree of robustness in terms of error accumulation over time. While one may never be completely confident in the robustness of an application, further tests preferably involving end-users running the VPN software on a day-to-day basis is recommended.
4.2 Conclusions

This section concludes the thesis with a couple of observations on the results, a discussion on the observations, a listing of limitations of the program and a few suggestions on how the software may be improved further.

4.2.1 Observations On Performance

Before describing our observations from the performance evaluation, we denote each test type with an abbreviation as defined in Table 4.6, in order to make the text less verbose.

<table>
<thead>
<tr>
<th>Translation</th>
<th>Test Type</th>
<th>Abbrev.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N/A-N/A</td>
<td>NN</td>
</tr>
<tr>
<td></td>
<td>vpn-N/A</td>
<td>VN</td>
</tr>
<tr>
<td></td>
<td>vpn-vpn</td>
<td>VV</td>
</tr>
<tr>
<td></td>
<td>vpn'-vpn'</td>
<td>V'V'</td>
</tr>
</tbody>
</table>

Table 4.6: Defining a simplified notation to be used when referring to the test cases.

Unless otherwise mentioned, the observations listed below are based on the test cases found in Tables 4.2-4.4, between computer A and B, connected through C.

Observations on latency, based on Table 4.2:
1. Average latency of V'V' is roughly 2 times greater than NN.
2. Average latency of VN is roughly 20 times greater than NN.
3. Average latency of VV is roughly 40 times greater than NN.

Conclusions: We observe that connecting the OS X SSL VPN client to the VPN service of E20 introduce an average increase in latency by a factor of 20.

Observations on single-stream throughput, based on Table 4.3:
1. Single stream rates are roughly 0.5 % to the rate of NN.
2. Single stream CPU utilization is low.

Conclusions: All single stream rates were extremely poor and there is no immediately obvious trend found in the data. If the low rates could be attributed solely to the VPN service of E20, we would see much higher rates for V'V'.

Observations on multiple-stream throughput, based on Table 4.4:
1. Test NN shows similar rates and CPU usage for sender and receiver.
2. Reception rate of VN increased by multiple of 20 compared to the single-stream test, CPU usage of sender is roughly 7 times that of the receiver and E20 CPU is at 7%.
3. Reception rate of VV increased by multiple of 25 compared to the single-stream test, CPU usage of sender is slightly less than twice of the receiver and E20 CPU is at 10%.
4.2. Conclusions

4. Reception rate of V’V’ increased by multiple of 34 compared to the single-stream test, CPU usage of sender is slightly more than twice of the receiver.

5. CPU usage of the sender in V’V’ is nearly twice as to that of the sender in VN.

Conclusions: Introducing 120 truly parallel streams, one would expect to see rates increase by a multiple of 120, given support on the hardware level. However, while we can’t expect such dramatic increases in rates, seeing that we don’t have 120 cores on the machine and other hardware limitations, there are some interesting findings in the data.

We may first note that we did not get any increase in throughput for NN using multiple streams, since the USB Ethernet adapter on computer A is limited to 100 Mbit/s. However, NN shows that when the relative difference between the send and receive rates gets small, the relative difference of CPU utilization also gets small. While this is unsurprising on its own, coupled with observation 1 and 5, it would seem to indicate a bottleneck in the VPN service, limiting the rate at which the sender can transmit and further show that the receiver only utilize 5% of the CPU.

4.2.2 Discussion

While the data gathered from tests between computers A and B seem to indicate a bottleneck in the VPN service, tests of type V’V’ make it clear that the VPN client is far from delivering acceptable performance, and the single stream test demonstrated equally poor performance in all test cases between computers A and B. Observing the relative amount of time spent per function on the receiving end of a test using 120 streams as shown in Figure 3.8 of Section 4.1.5, we see that on average, 29% is spent receiving data on the SSL channel, whereas 47.5% is spent encoding received data and passing it to the pppd. In practice, this would mean that if receiving a set of bytes takes 29 ms, it takes an additional 47 ms to pass it to pppd, which in turn must decode the data and pass it to the network stack. This is a strong indicator showing that it’s not feasible to utilize the pppd in this manner, given that one is interested in high network communication rates. Note that this statement requires one to assume that read and writes to the pppd is performed in an optimal manner, which may not be the case.

Recall the measurements found in Table 4.5 from the single stream iperf3 throughput test of type NN and VN, between computers W and B, where W use Clavisters Windows SSL VPN client. Calculating the throughput of VN relative to NN from the measurements obtained from the single stream tests on Windows: \( V_{N}/N_{N} = 66.1/94.1 = 0.07 \), we find that 7.0% of the maximum throughput is utilized while using the SSL VPN. The equivalent calculation for the OS X client: \( V_{N}/N_{N} = 0.380/94.2 = 0.004 = 0.4\% \). So why is the Windows client almost 18 times faster than the OS X client? We may first of all note that one of the primary differences between the Windows client and the OS X client is that Clavisters client does not use pppd. Instead, it is based on Clavisters own implementation of the PPP protocol, which means that they neither have to perform HDLC-like encoding/decoding or writing/reading data to/from an external program. The measurements of relative amount of time spent per function shown in Section 4.1.5, gives you an idea of the performance impact caused by utilizing pppd. With that being said, one must also bare in mind that Clavisters PPP implementation, used in both the Windows SSL VPN client and the VPN service of the E20 gateway, also adds latency to the system. It would therefore be interesting perform further measurements on the Clavister PPP implementation so that a real comparison between to pppd and Clavister PPP can be performed.

While the Windows client indeed achieve higher throughput than the OS X client, 7.0%
of maximum throughput is not ideal either. The purpose of this thesis was to investigate whether it is feasible to utilize pppd in order to avoid porting or re-implementing the PPP protocol, and yet achieve some degree of acceptable throughput as formulated in the research question: “Is high network throughput achievable while piggybacking on pppd?” in Section 1.2. It may be possible to find a more efficient way of interacting with pppd so that the relative throughput between VN and NN of the OS X SSL VPN client approach 7.0% as in the case for the Windows client. But the main cause of the poor throughput performance seems to relate to the encoding and decoding of HDLC-like frames which is required to communicate with pppd. As long as this has to be performed, a pppd-based solution can simply not compete with a client which is not relying on HDLC-like framing. We will therefore conclude that a pppd-based solution is not viable when seeking to achieve high VPN throughput rates.

4.2.3 Limitations

As the pppd-program performs kernel operations, simply starting the program requires root access to the machine and there does not seem to be any way around this fact. Future development into this project may want to investigate Apple’s Authorization procedure, which allows a user to temporarily execute these kinds of programs in a more user friendly manner.

There are considerable speed limitations, as shown in Section 4.1 and discussed in Section 4.2.2.

4.2.4 Future work

In the current implementation, incoming data received from either the SSL connection or the pppd-process is heap-allocated on the fly to fit the size of the incoming data segment. The allocated data is then freed when passed on to the pppd or SSL connection. It would however be better to pre-allocate a larger chunk of memory space at the start of program execution, that can be reused and possibly expanded if required during execution. Circular buffers are often used for data-streams and could be applicable in this situation.

Perform further performance tests of the application and comparisons to the Windows-based implementation. One particular area to investigate could be the fine tuning of the MRU/MTU (Maximum Receive/Transmit Unit in the client, iperf3 and on the server side. Sending larger packets than the MTU leads to packet fragmentation and degrades the VPN performance.

One may also want to explore the details of the Clavister PPP implementation, compare it to pppd and see whether it is somehow possible to disable HDLC-like framing for pppd. If it is found that it indeed is impossible to disable HDLC, one may seek to replace pppd with a different PPP implementation that does not rely on HDLC-like framing.

4.2.5 Practical Application

In preparation of the performance evaluation tests of type V’V’, the program was extended to provide SSL-server capabilities which enables two clients to establish an authenticated and encrypted tunnel between them. This could be useful in cases where one would wish to maintain a secure path of communication between two machines capable of carrying various kinds of communication protocols. For example, Alice and Bob could maintain an always-active secure communication path between them without affecting their regular Internet connections.
4.2. Conclusions

While this VPN implementation use a proprietary PPP-header in order to communicate with Clavister Security Gateways, a trivial modification to the code would render the application capable of connecting to an arbitrary PPP-based SSL VPN service.
References


