ABSTRACT

SEPARATION OF SSL PROTOCOL PHASES ACROSS PROCESS BOUNDARIES

by
Kirthikar Anantharam

Secure Sockets Layer is the de-facto standard used in the industry today for secure communications through web sites. An SSL connection is established by performing a *Handshake*, which is followed by the *Record* phase. While the SSL *Handshake* is computationally intensive and can cause of bottlenecks on an application server, the *Record* phase can cause similar bottlenecks while encrypting large volumes of data.

SSL Accelerators have been used to improve the performance of SSL-based application servers. These devices are expensive, complex to configure and inflexible to customizations. By separating the *SSL Handshake* and the *Record* phases into separate software processes, high availability and throughput can be achieved using open-source software and platforms. The delegation of the *SSL Record* phase to a separate process by transfer of necessary cryptographic information was achieved. Load tests conducted, showed gains with the separation of the *Handshake* and *Record* phases at nominal data sizes and the approach provides flexibility for enhancements to be carried out for performance improvements at higher data sizes.

SEPARATION OF SSL PROTOCOL PHASES ACROSS PROCESS BOUNDARIES

by
Kirthikar Anantharam

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

May 2006

APPROVAL PAGE

SEPARATION OF SSL PROTOCOL PHASES ACROSS PROCESS BOUNDARIES

Kirthikar Anantharam

Dr. Andrew Sohn, Thesis Advisor/Committee Chair — Date
Associate Professor, Department of Computer Science, NJIT

Dr. Alexandros Gerbessiotis, Committee Member — Date
Associate Professor, Department of Computer Science, NJIT

Dr. Teunis J. Ott, Committee Member — Date
Professor, Department of Computer Science, NJIT

BIOGRAPHICAL SKETCH

Author: Kirthikar Anantharam

Degree: Master of Science

Date: May 2006

Undergraduate and Graduate Education:

- Master of Science in Computer Science,
  New Jersey Institute of Technology, NJ, USA, 2006

- Master of Science in Industrial Engineering,
  New Jersey Institute of Technology, NJ, USA, 1999

- Bachelor of Engineering in Mechanical Engineering,
  University of Mysore, India, 1996

Major: Computer Science

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन
मा कर्मफलहेतुर्भुर्मा ते सङ्गोऽस्त्वकर्मणि

Transliteration:

Karmanye Vadhikaraste Ma Phaleshu Kadachana,

Ma Karma Phala Hetuh Bhurmatey Sangostva Akarmani

Translation:

You certainly have the right to perform your duties, but never at any time in their results. You should never be motivated by the results of your actions, never should have any attachment in not performing your duties.

Verse 47 from Chapter 2 of the Srimad Bhagavad-Gita forms one of my core philosophical beliefs. I would like to dedicate this *effort* to my parents and my family members in Bangalore, India. Their love and support during my formative years instilled in me, a strong sense of values and a pure love for learning. I hope to have lived true to my core belief through this research effort.

ACKNOWLEDGMENT

I would like to express my gratitude and appreciation to Dr. Andrew Sohn. He not only inspired to take on this research effort but also supported, mentored and guided me throughout the process. His knowledge and model work ethic helped groom my professional outlook and scientific thinking. In spite of my residency out-of-state, he graciously accepted to advise my Master's thesis primarily during meetings over weekends, without which, this effort would have been impossible. Today, it is hard to measure the growth of my technical capabilities in the field of Computer Science, thanks in large part to Dr. Andrew Sohn. I would also like to thank Mr. Hyung Won Choi who provided me with valuable insight during my research and helped me with all the experimental setup. I request Dr. Alexandros Gerbessisotis and Dr. Teunis Ott to accept my thanks for participating in my committee.

I like to specially thank my wife, Dr. Arti Santhanam and my brother, Karun Anantharam for supporting me in all my endeavors. Without their support, I would lack the motivation to attempt and the perseverance to complete such a research effort.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

SSL	Secure Sockets Layer
VPN	Virtual Private Network
ISO	International Standards Organization
OSI	Open System Interconnection
TCP	Transport Control Protocol
IP	Internet Protocol
HTTP	Hyper Text Transfer Protocol
SMTP	Simple Mail Transfer Protocol
NNTP	Network News Transfer Protocol
MAC	Message Authentication Code
TLS	Transport Layer Security
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
NAT	Network Address Translation

CHAPTER 1

INTRODUCTION

1.1 Virtual Private Network

Securing the network infrastructure of an organization is one of the top priorities for information technology administrators [1]. One way to make a secure connection into an organization is through a Virtual Private Network (VPN) [2, 3] over the public Internet. Users can safely connect into an organization's infrastructure through a VPN service and get access into personal files, email, databases, etc. with the assurance of no eavesdropping on the data being accessed through the public Internet [4].

A VPN uses the Internet as its transport mechanism, while maintaining the security of the data on the VPN [5]. The main benefit of a VPN is the potential for significant cost savings compared to traditional leased lines or dial up networking. However, these savings come with a certain amount of risk, particularly when using the public Internet as the delivery mechanism for VPN data. VPN technology is based on the idea of tunneling, and is commonly used to describe secure remote access tunnels. Network tunneling involves establishing and maintaining a logical network connection (that may contain intermediate hops). Over this connection, packets constructed using a specific VPN protocol format and encapsulated within some other base or carrier protocol, are transmitted between a VPN client and a VPN server, and finally, de-encapsulated on the receiving side. For Internet-based VPNs, packets in one of several VPN protocols are encapsulated within IP packets [6]. VPN protocols also support

authentication and encryption to keep the tunnels secure.

Technologies enabling VPN connectivity include Point-to-Point Tunneling Protocol (PPTP) [7], Layer Two Tunneling Protocol (L2TP) [8], Internet Protocol Security (IPSec) [9] and Secure Socket Layer (SSL)/Transport Layer Service (TLS) [10, 11, 12]. PPTP and L2TP exist at the data link layer (Layer Two) of the OSI model, while IPSec exists at the network layer (Layer Three) of the OSI model.

Typically, IPSec based connectivity solutions are embedded in hardware and/or custom software and that requires the presence of IPSec compliant hardware/software at both end-points of the connection. This requirement for IPSec compliance at both end-points along with its complex setup and management requirements has limited the widespread adoption of IPSec-based VPN in general.

The Layer 2 Transmission Protocol (L2TP) as documented in RFC 2661 [8] is a lower level protocol for remote connectivity. Combining PPTP [7] and Layer 2 Forwarding allows secure communications. All Microsoft Windows-based machines are already equipped with this protocol for establishing secure connections to remote servers. However, this protocol requires both the client and server to use proprietary software to enable this forwarding. As a result, the usage of L2TP has also been limited.

SSL-based VPNs, on the other hand, leverage the existing and widely accepted SSL protocol in standard Web browsers like Internet Explorer or Netscape Navigator. The VPN connection is established and maintained over secure web connections using HTTPS [13, 14, 15]. The base protocol for this connection is done through a higher-level communication protocol (L4/L7) used in applications [16, 17, 18]. In addition, there is no need for proprietary hardware/software compliance at the communication end-points.

1.2 SSL VPN

Architecturally and commercially, SSL-based VPN has significant advantages over the other solutions. Firstly, Web browsers are ubiquitous. There is no need for expensive and time-consuming conversions and training programs. Secondly, the solution is platform independent. Anyone with a Web browser can connect securely into a network with little or no configuration. Thirdly, development and security enhancements automatically evolve as SSL and browser technologies improve (client side). Lastly, the solution is inexpensive, since Web browsers with SSL capabilities are freely available.

While the advantages of an SSL-based VPN solution greatly outweigh the alternatives, this approach is not without its problems, namely performance scalability. At the core of SSL-based VPN or any SSL-based service is its encryption/decryption capability [19]. All data traveling through an SSL tunnel will entail substantial encryption and decryption. During times of high load, a VPN server can potentially end up in a crippled state. Studies have indicated that even a small number of users generating SSL-based traffic can drop server performance to 80% or by 1/5th of its performance capability [20, 21]. Thus, the encryption/decryption overhead of SSL based services poses a significant problem [22].

Dedicated hardware and software technologies have been introduced to accelerate the SSL protocol [23, 24]. Modern L4/L7 switches provide hardware support for SSL acceleration as part of the switch configuration. SSL VPN products from vendors such as Array Networks [25], Aventail [26], Cisco [27], Juniper Networks [28], Nokia [29], NetScaler [30], Netilla Networks [31], NetSilica [32], Nortel Networks [33], Symantec (appliance) [34], Whale Communications [35], etc. come equipped with SSL VPN as one

NetScaler [30], Netilla Networks [31], NetSilica [32], Nortel Networks [33], Symantec (appliance) [34], Whale Communications [35], etc. come equipped with SSL VPN as one of the standard features.

Software approaches to accelerate SSL-based connections have also emerged in an attempt to provide fast and secure VPN connectivity. Software vendors such as Areabe [36], CheckPoint [37], Citrix [38], Menlo Logic [39], OvisGate [40], PortWise [41], Symantec [34], Tarantella [42], V-ONE [43], 3SP (open-source) [44], etc. support SSL VPN features in their product offerings.

1.3 Problem Statement

While the hardware and software solutions described in the previous section increase the performance of SSL computing, two issues remain unanswered, namely performance scalability and closed-box vs. open-source. As these products and approaches are closed-boxes and proprietary in nature, it is very difficult for the vast majority of the open-source community to make any changes for improvement and customize them for specific needs. While scaling with these closed-boxes is cost-prohibitive and complex in most cases, it is simply unrealistic in others.

The primary objective is to examine the feasibility of separating the SSL protocol phases across process boundaries and measure any gains thereof, in an effort to improve the clustering capabilities and increase the performance of SSL-based computing. The outcomes of this investigation hold a direct relevance to the larger problem domain of clustering and scalability in VPN and Web servers using an open-source approach.

SSL is a protocol that ensures the security of the data transmitted over the

Internet, using encryption capabilities that are automatically available in every browser. The SSL protocol consists of two phases, the Handshake phase and the Record phase. Currently, these phases are implemented to execute as part of a single operating system process in user-mode. The goal is to distribute these phases over separate processes executing on multiple servers. A successful separation will not only provide enhanced options for clustering, but also potentially improve performance and scalability. The performance gains demonstrated by such a distributed approach will be especially useful in the area of SSL-based VPN services and HTTPS (HTTP over SSL).

CHAPTER 2

BACKGROUND

Chapter 1 provided an introduction to VPN and the various protocols and approaches used to provide VPN connectivity to clients. It also provided an overview of the SSL-based VPN services and a description of the problem statement as it relates to SSL-based services. This chapter will discuss in brief, the TCP/IP communication protocol and the two phases of the SSL protocol in more detail.

2.1 Protocol Architecture

The International Standards Organization (ISO) proposed a highly modular, layered architecture for the communication protocols over the Internet called the Open Systems Interconnection (OSI) model, published as OSI RM-ISO 7498. Figure 2.1 shows the OSI protocol architecture. Table 2.1 gives a brief description of the services provided by each layer.

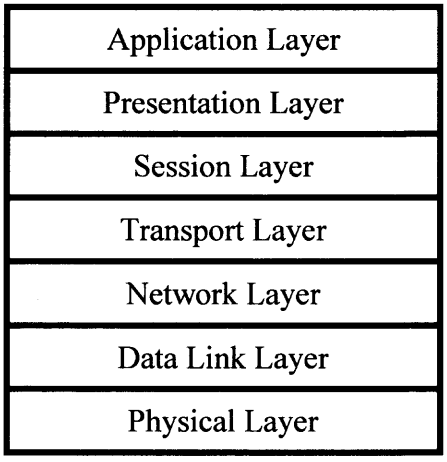


Figure 2.1 OSI layer architecture.

Table 2.1 OSI Layers and Services

Layer No.	Layer Name	Services provided by Layer
1	Physical	Encoding and Signaling, Physical Data Transmission, Hardware Specifications, Topology and Design
2	Data Link	Logical Link Control, Media Access Control, Data Framing, Addressing, Error Detection and Handling, Defining Requirements of Physical Layer
3	Network	Logical Addressing, Routing, Datagram Encapsulation, Fragmentation and Reassembly, Error Handling and Diagnostics
4	Transport	Process-Level Addressing, Multiplexing/De-multiplexing, Connections, Segmentation and Reassembly, Acknowledgments and Retransmissions
5	Session	Session Establishment, Management and Termination
6	Presentation	Data Translation, Compression and Encryption
7	Application	User Application Services

Transmission Control Protocol/Internet Protocol (TCP/IP) is a suite of protocols that enable networks and machines to be interconnected and forms the basic foundation

for the Internet. The TCP/IP suite of protocols is a subset of the OSI model and contains five layers in its architecture. The services provided by the various TCP/IP layers are the same as the ones provided by the OSI model. The layers in the TCP/IP suite are shown in Figure 2.2.

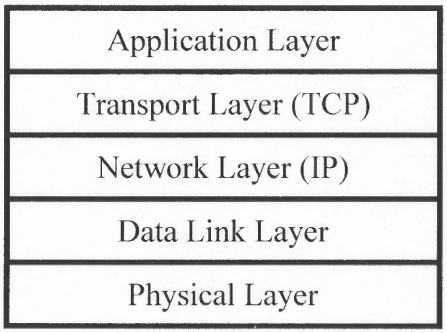


Figure 2.2 TCP/IP layer architecture.

SSL runs beneath application layer protocols such as HTTP, SMTP and NNTP and above the TCP transport layer protocol. Figure 2.3 shows the location of SSL within the TCP/IP protocol suite. In practice, applications that wish to use SSL for communication make use of standard software libraries that provide services constructed around the TCP/IP stack, which is exposed through the system call interface on most operating systems.

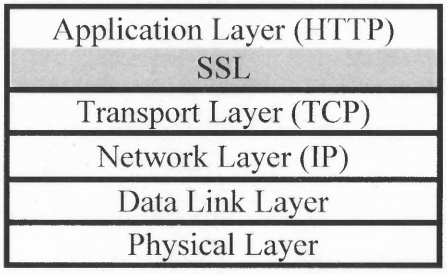


Figure 2.3 Location of SSL in the TCP/IP protocol stack.

2.2 SSL Connection Phases

SSL connections are divided into two phases, the *Handshake* and *Record* phases [43]. The *SSL Handshake* phase authenticates the server (optionally, the client) and establishes the cryptographic keys which are used to protect the transmitted data. The *Handshake* phase must be completed before any application data can be transmitted. Once it is complete, the data to be communicated is broken up into fragments, which are encrypted and then transmitted as a series of packets. This phase is called the *SSL Record* phase.

2.3 SSL Handshake

During the *SSL Handshake*, the server and optionally, the client, are authenticated using digital certificates. An encryption algorithm is selected from a set of predefined algorithms and a symmetric key is chosen for each direction of communication. The following series of messages are exchanged by the server and the client to perform a successful *SSL Handshake* [45]. Figure 2.4 illustrates the same.

1. The client sends the server, the client's SSL version number, cipher settings, randomly generated data, and other information required by the server to communicate with the client using SSL. This step is denoted by the **client-hello** message in Figure 2.4.
2. The server sends the client, the server's SSL version number, cipher settings, randomly generated data, and other information the client needs to communicate with the server over SSL. The server also sends its own certificate and, if the client is requesting a server resource that requires client authentication, requests the client's certificate. This step is denoted by the **server-hello** message in the Figure 2.4.
3. The client uses some of the information sent by the server to authenticate the server. If the server cannot be authenticated, the user is warned of the problem and informed that an encrypted and authenticated connection cannot be established. If the server can be successfully authenticated, the client moves on to step 4.

4. Using all data generated in the handshake so far, the client (with the cooperation of the server, depending on the cipher being used) creates the **pre-master secret** for the session, encrypts it with the server's public key (obtained from the server's certificate, sent in Step 2), and sends the encrypted pre-master secret to the server. This step is denoted by the client-premaster-key message in Figure 2.4.

5. If the server has requested client authentication (an optional step in the handshake), the client also signs another piece of data that is unique to this handshake and known by both the client and server. In this case the client sends both the signed data and the client's own certificate to the server along with the encrypted pre-master secret.

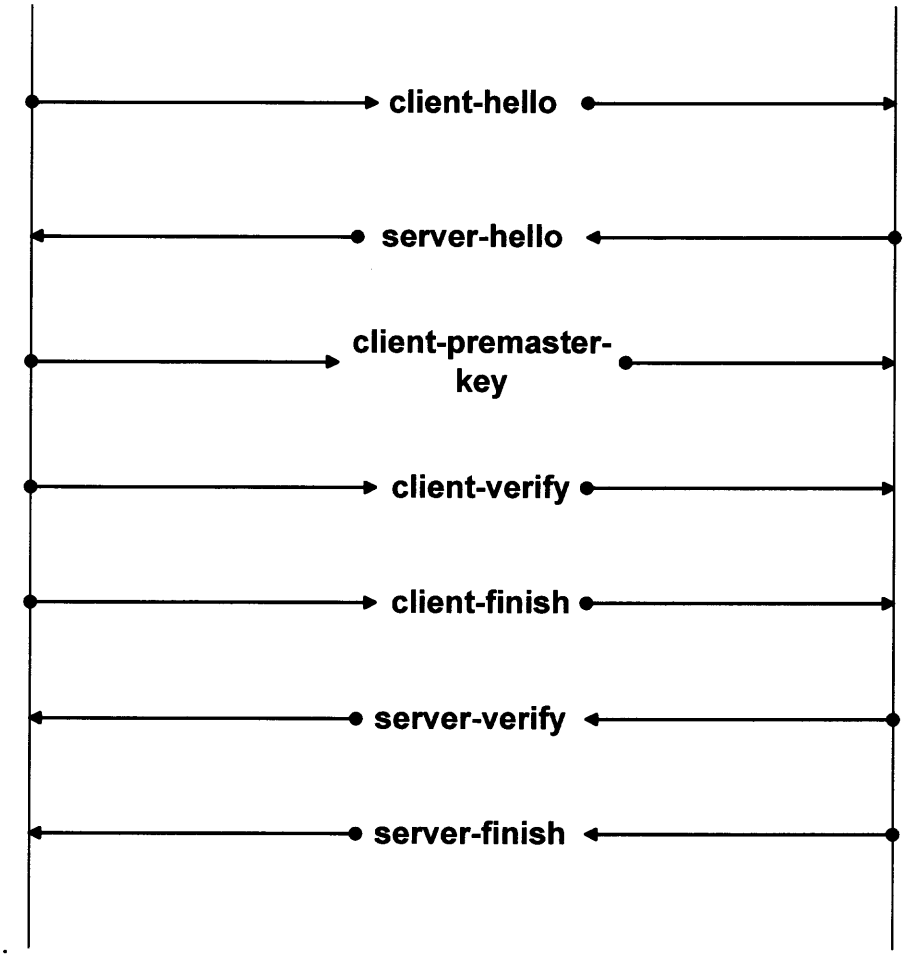


Figure 2.4 SSL Handshake.

6. If the server has requested client authentication, the server attempts to authenticate the client. If the client cannot be authenticated, the session is terminated. If the client can be successfully authenticated, the server uses its private key to decrypt the pre-master secret and performs a series of steps (The steps are also performed by the client, starting from the same pre-master secret) to generate the **master secret**.

7. The client and the server, then, use the master secret to generate the session keys, called the **write-key** and the **read-key**. These are symmetric keys used to encrypt and decrypt information exchanged during the *SSL Record* phase and to verify the integrity of the SSL session, i.e., to detect any changes in the data between the time it was sent and the time it is received over the SSL connection. Different keys are used for each direction of transmission. Feeding the master key, the session identifier, and the challenge data through an algorithm generates the session keys. The two ends of the SSL connection do this independently.

8. The client then sends a message denoted by **client-verify** to the server informing it that future messages from the client will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the client portion of the handshake is finished. This step is denoted by the **client-finish** message in Figure 2.4.

9. The server sends a message denoted by **server-verify** to the client informing it that future messages from the server will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the server portion of the handshake is finished. This step is denoted by the **server-finish** message in Figure 2.4.

10. The *SSL Handshake* phase is now complete, and the SSL session is completely established. The client and the server use the session keys to encrypt and decrypt the data they send to each other and to validate its integrity.

2.4 SSL Record Protocol

The purpose of the *SSL Handshake* phase is to setup the shared data required to enable the sending and receiving of protected data. In SSL, the actual data transfer is accomplished during the second phase using the SSL Record Protocol [46]. The SSL Record Protocol works by breaking up the data stream to be transmitted into a series of fragments, each of which is independently encrypted and transferred. On the receiving end, each record is decrypted and verified.

To ensure the integrity of the message and to guard against an attacker replying to an old message, a Message Authentication Code (MAC) is computed over the data to be transmitted. The MAC is concatenated to the data and the entire block is then encrypted

to form the payload. Finally, a header is attached to the payload to form a record. These records are then transmitted using lower level protocols such as TCP. If necessary, random padding data will be added to an application message to make it the correct length for the encryption algorithm to process.

2.5 SSL Record Format

There are three record types for SSL Version 3: (1) Handshake (2) Alert - warning or fatal error (3) Data - application data

The data in any SSL record has the following characteristics: (1) A variable length and starts with a 5-byte record header; (2) Contains handshake data, alert data or application data; (3) Is encrypted, except for the first few messages in the handshake message flows. The format of an SSL Record is as follows:

Byte 0: SSL Record Type

Bytes 1-2: SSL Version (major/minor)

Bytes 3-4: Length of data in the record (excluding the header itself). The maximum record size supported by SSL 16384 bytes (16K).

CHAPTER 3

PROPOSED APPROACH

The previous chapter explained in detail the SSL protocol phases and the format of the records transmitted between the end points of an SSL connection. This chapter will focus on the computational overhead caused by the encryption and decryption activities carried out by the server during the *SSL Handshake* and *Record* phases, and delve into the strategies available to improve server throughput and performance. The proposed approach consisting of distributing the SSL protocol phases over separate processes is described and the advantages it provides are discussed.

3.1 Cryptography and Clustering

Once a client and a server have completed the *Handshake* phase, they can communicate using standard encryption algorithms such as DES, RC4, etc. These algorithms, called ciphers, use a technique called symmetric-key cryptography. Symmetric-key cryptography uses a common key for both encrypting and decrypting data. Symmetric key cryptography is relatively fast compared to public-key cryptography [47].

Public-key cryptography uses a pair of keys, called private key and public key and is usually based on the RSA algorithm. Typically, 1024-bit RSA key-pairs are used in SSL-based communications. Public-key cryptography is also called asymmetric-key cryptography, since it uses a pair of keys, to perform the encryption and decryption of information. During the *Handshake* phase, public-key cryptography is used to exchange

important pieces of data (pre-master secret and challenge data) used to generate the symmetric keys called session keys. In addition, the *Handshake* phase performs server authentication (optionally, client authentication) and negotiation of ciphers.

The *Handshake* phase is computationally very intensive owing to the public-key cryptography in addition to the load imposed by the authentication and symmetric key generation activities [22]. This is a major cause of bottlenecks in servers supporting SSL-based communications. In many cases, such bottlenecks limit a server to as few as five or ten SSL handshake transactions per second depending on the power of the server. Delegating the computationally intensive cryptographic operations from the server to a separate and specially designed device/entity can distribute the load on the CPU and increase throughput. Thus, cryptographic operations required by SSL create a need for load distribution through clustering.

3.2 SSL Accelerators

A specialized device that is designed to handle the extra computational burden imposed by the *SSL Handshake* phase is called an SSL Accelerator [48, 49]. SSL Accelerators are available in two forms, internal cards and network devices [50]. Internal cards generally handle the SSL encryption and decryption process, leaving the server to cope with activities such as session set-up, key exchange and cipher suite negotiations. External network devices are capable of handling the entire SSL workload, so that the traffic entering and leaving the server is in plain HTTP format or other clear-text. Although SSL Accelerators provide numerous advantages, they pose the following disadvantages:

- Expensive and highly specialized.

- Complex setup and configuration requirements.
- Difficult to perform customizations since all the functionality is built into the hardware.
- Application functionality – Most commercially available SSL Accelerators are designed and built to operate in conjunction with web servers providing HTTPS capabilities.
- Symmetric encryption for large data sizes – SSL Accelerators usually provide assistance with the public-key cryptography portion of a *Handshake*. Once the initial SSL session is established, the Accelerator plays no part in further connections with the same SSL session. Although some are capable of performing symmetric encryption, the overheads of actually sending the information to the hardware to be encrypted or decrypted is often higher (both in terms of latency and system resources) than just performing the operations directly in the server. This problem of increasing loads due to symmetric encryption becomes especially significant when the size of data to be encrypted is large, in the order of hundreds of Kilobytes to Megabytes and higher.
- Large decryption loads on servers – The increased load due to symmetric decryption is also significant when decryption is required at the server for large data volumes In the case of web servers, which typically use SSL, the decryption loads on the server are insignificant compared to the encryption loads for out-going data. SSL VPNs on the other hand, would have to handle large-volumes of data, both incoming and outgoing.
- Logical separation of SSL gateways and application servers providing data for encryption – Separation of secure gateways to an enterprise from the application servers is not possible due to tight coupling between software providing SSL capabilities and application functionality.
- Special requirements such as measurement of SSL encryption performance are not easy to achieve due to proprietary hardware and firmware.

3.3 Separating the Handshake and Record Phases

The proposed approach to achieve performance gains and scalability involves the separation of the *SSL Handshake* and the *Record* phases across process boundaries as described below. A client such as a Web browser wishing to communicate with a server (Web server, VPN server, etc.,) using SSL, establishes an SSL connection (*SSL*

Handshake) followed by exchange of encrypted information through requests and responses (*SSL Record* phase). Typically, the server process connected to the client handles both the SSL protocol phases. The proposed approach involves one server process handling the *SSL Handshake* phase while another handles the *SSL Record* phase.

Henceforth, *SSLClient* will refer to the client and *SSLHandler*, the server process that participates in the SSL Handshake with the client. Initially, *SSLClient* and *SSLHandler* perform an *SSL Handshake*. This is followed by the *SSLHandler* transferring all the cryptographic information required by the *SSL Record* phase to an external process called, *SSLWorker*. Figure 3.1 illustrates this simple two-step process of performing an SSL Handshake.

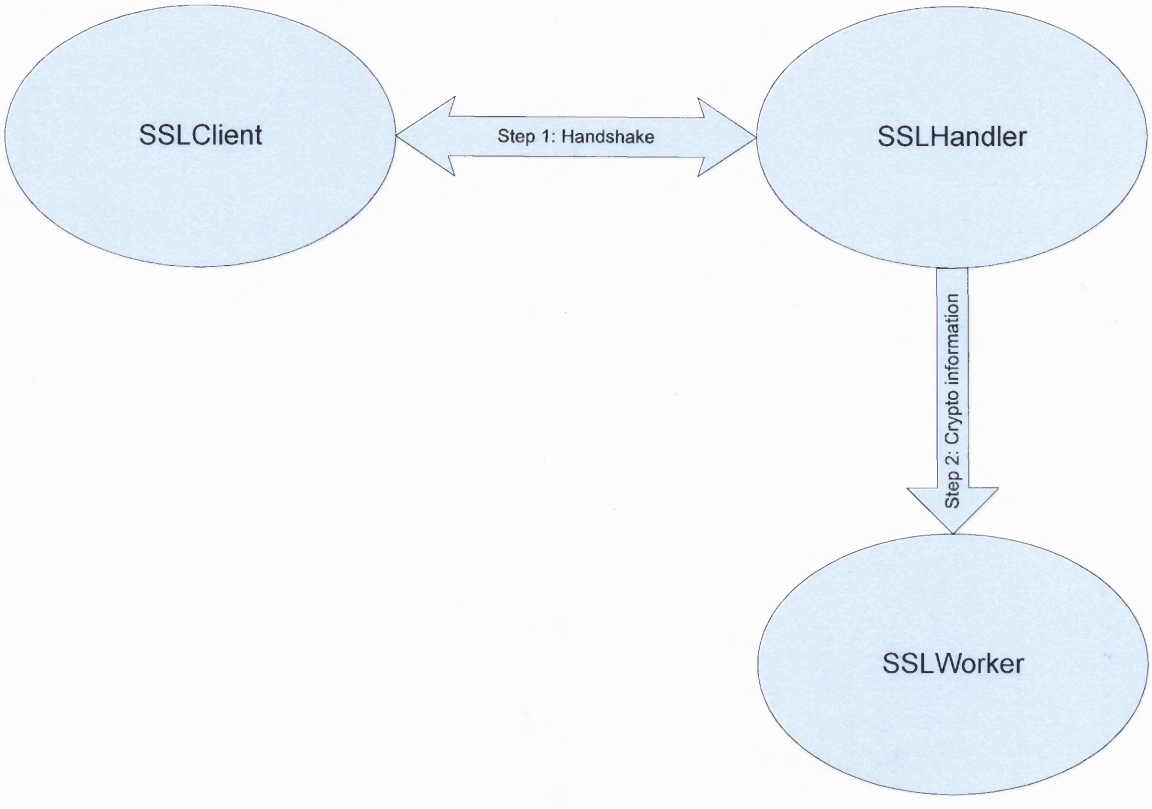


Figure 3.1 SSL Handshake phase with separation.

The 2-step process described creates a pathway for two-way data communication

between the *SSLClient* and the *SSLWorker* during the *SSL Record* phase. Through this pathway, the *SSLWorker* can receive encrypted data from the *SSLClient*, decrypt it, and perform a series of custom actions as requested by the *SSLClient* and send back an encrypted response. Examples of such custom actions include, but are not limited to updating a database, creating media files, etc. The *SSLHandler* simply acts as an intermediary during the *SSL Record* phase, passing encrypted data coming from the *SSLClient* to the *SSLWorker* and vice-versa.

Figure 3.2 depicts *SSL Record* phase communications between the *SSLClient* and the *SSLWorker*. Following Step 2 in Figure 3.1, the *SSLClient* creates a simple HTTP request in Step 3 and transmits it to the *SSLHandler* as shown in Figure 3.2. The *SSLHandler* receives the request and forwards it to the *SSLWorker* in Step 4. The *SSLWorker* decrypts the request using the cryptographic information received from the *SSLHandler* (after the *SSL Handshake*) and creates an encrypted response which is transmitted back to the *SSLHandler* in Step 5. The *SSLHandler* re-transmits the response received from the *SSLWorker* to the *SSLClient* in Step 6.

Steps 3 through 6 describe the crux of the proposed approach to separate the SSL protocol phases across process boundaries. The *SSLHandler* handles all the processing during the *Handshake* phase, and delegates the *Record* phase operations to the *SSLWorker* instance. The separation provides the ability to selectively scale either the servers performing the *SSL Handshake* phase or the ones involved in the *SSL Record* phase or both, as the requirements demand. This is possible because, the *SSLHandler* and *SSLWorker* processes can be executed on separate physical machines. The separation also allows the time-consuming symmetric encryption and decryption of large data (hundreds

of Kilobytes to Megabytes) to be moved and scaled on separate computers, thus enabling the faster SSL Handshakes carried out by dedicated servers.

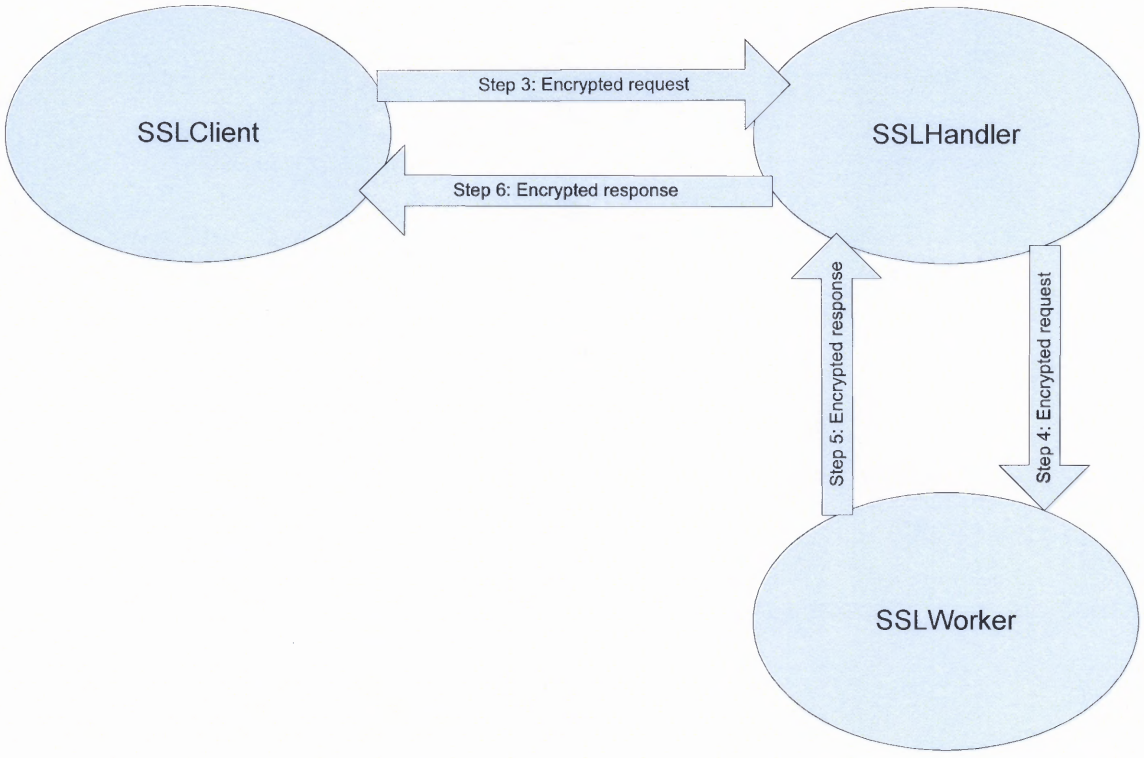


Figure 3.2 SSL Record phase with separation.

3.4 Open-source SSL Software

Open-source SSL software libraries such as OpenSSL provide the ability to perform SSL communications over TCP/IP in a very simple fashion. It is also tailored for open-source platforms such as the Linux operating system that runs on in-expensive x-86 hardware. Put together, they offer a free and highly cost-effective solution in comparison to expensive and proprietary hardware devices such as SSL Accelerators. The caveat is that within OpenSSL, the *SSL Handshake* and *SSL Record* phases are tightly coupled within a single operating system process.

SSL-based communications using OpenSSL as-is, would not be flexible enough to accomplish the separation of the *SSL Handshake* and *Record* phases across the *SSLHandler* and *SSLWorker* processes respectively. No commercial products are available in the market either to achieve such a separation. Custom software developed in conjunction with OpenSSL would offer such a solution. The *SSLHandler* and *SSLWorker* processes mentioned in the previous section constitute such custom software developed around the OpenSSL software library in addition to minor modifications to the library itself. These minor modifications are required to make the open-source library adaptable for such use.

3.5 Advantages of Delegation in Software

A software solution, as described in the previous sections, providing the ability to delegate the *SSL Record p*hase of an SSL connection to an external process provides the following advantages:

- Cost savings owing to the use of a freely available open-source platform such as Linux and the open-source SSL software OpenSSL compared to expensive hardware solutions involving SSL accelerators and load balancers.
- High availability can be easily achieved at low cost by providing fail-over capabilities to servers performing *SSL Handshake* and/or *SSL Record* phases as required. Providing fail-over with SSL Accelerators will increase the already high cost of such hardware.
- Scaling can also be accomplished easily and cost effectively as compared to hardware SSL Accelerators.
- Ability to be designed and developed for easy setup and configuration with varying degrees of customizability.
- Ease of integration - Backend applications such as databases, file servers, multi-media servers, tunneling software, etc., can be easily integrated with processes that

perform *SSL Record* phase communications.

- Special measurements such as *SSL Handshake* and *Record* phase SSL encryption performance can be easily collected by modifying any part of the open-source code (Linux and/or SSL library source) to track any desired metrics.

- Symmetric encryption for large data sizes – The ability to off-load the symmetric encryption to an external process will prove beneficial for large data sizes. Typically, multi-media servers, file servers, etc., which server large amounts of data to clients require such capabilities.

- Large decryption loads on servers – A software solution provides the ability to decrypt large amounts of data on the server in an efficient manner since this activity can be carried out by an external process. Large decryption loads are often experienced by VPN and Secure Shell servers that provide tunneling capabilities to clients.

CHAPTER 4

LOGICAL ORGANIZATION AND COMPONENTS

This chapter discusses the implementation details of the *SSLHandler* and *SSLWorker* components introduced in the previous chapter. Also discussed are two software processes called *Handler Agent* and *Delegate Agent*, used to manage the *SSLHandler* and *SSLWorker* instances, respectively. The implementation includes the use of various open-source libraries and minor modifications to the OpenSSL library. The various data structures used are also described in this chapter.

4.1 Software Platform and Libraries

Red Hat Linux 9 is the software platform used to implement the *SSLClient*, *Handler Agent*, *SSLHandler, Delegate Agent* and *SSLWorker* executables. Code development was carried out using the C programming language and gcc compiler.

OpenSSL version 0.9.7d is used to provide the required SSL functionality [51]. The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and open-source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and the Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library. The project is managed by a worldwide community of volunteers that use the Internet to communicate, plan, and develop the OpenSSL toolkit and its related documentation. OpenSSL is based on the SSLeay library developed by Eric A. Young and Tim J. Hudson.

ASN.1 stands for Abstract Syntax Notation One. ASN.1 allows the description of complex data structures independently of any particular programming language. The ASN.1 compiler can then take these ASN.1 specifications and produce a set of target language (C, C++, Java) files which contain the native type definitions for these abstractly specified structures, and also generate source code (function calls) which can perform the conversions of these structures into/from a series of bytes (serialization/de-serialization) [52]. These function calls provide the capability to transfer data structures with information over the network or to write to external media.

An open-source ancillary library termed libancillary; version 0.9.1 is also used in the implementation to provide an easy interface to UNIX domain sockets [53]. This interface is used to pass file descriptors from one process to another and is used to implement the session resumption capabilities as detailed in a separate section of this chapter.

4.2 Overall Logical Architecture

Figure 4.1 shows the architecture comprising of the various processes and steps involved in the separation of the SSL protocol phases. The *Handler Agent* and the *Delegate Agent* represent daemons listening for incoming connections. The following steps illustrate the process of establishing a new SSL connection before an *SSLClient* instance can send encrypted requests and receive encrypted responses from the server.

1. An *SSLClient* instance initiates a new SSL Connection request with the *Handler Agent*.
2. The *Handler Agent* spawns a child process, called the *SSLHandler*, to perform the *SSL Handshake*.

3. The *SSLHandler* instance connects to the *Delegate Agent* daemon for delegation.
4. In response to the connection request from the *SSLHandler*, the *Delegate Agent* spawns a child process, called *SSLWorker*, to whom all the SSL requests will be delegated to.
5. Then, the *SSLClient* and *SSLHandler* instances perform a full *SSL Handshake*.
6. The *SSLHandler* process collects all the cryptographic state information resulting from the *SSL Handshake* and transports it to the *SSLWorker* instance.

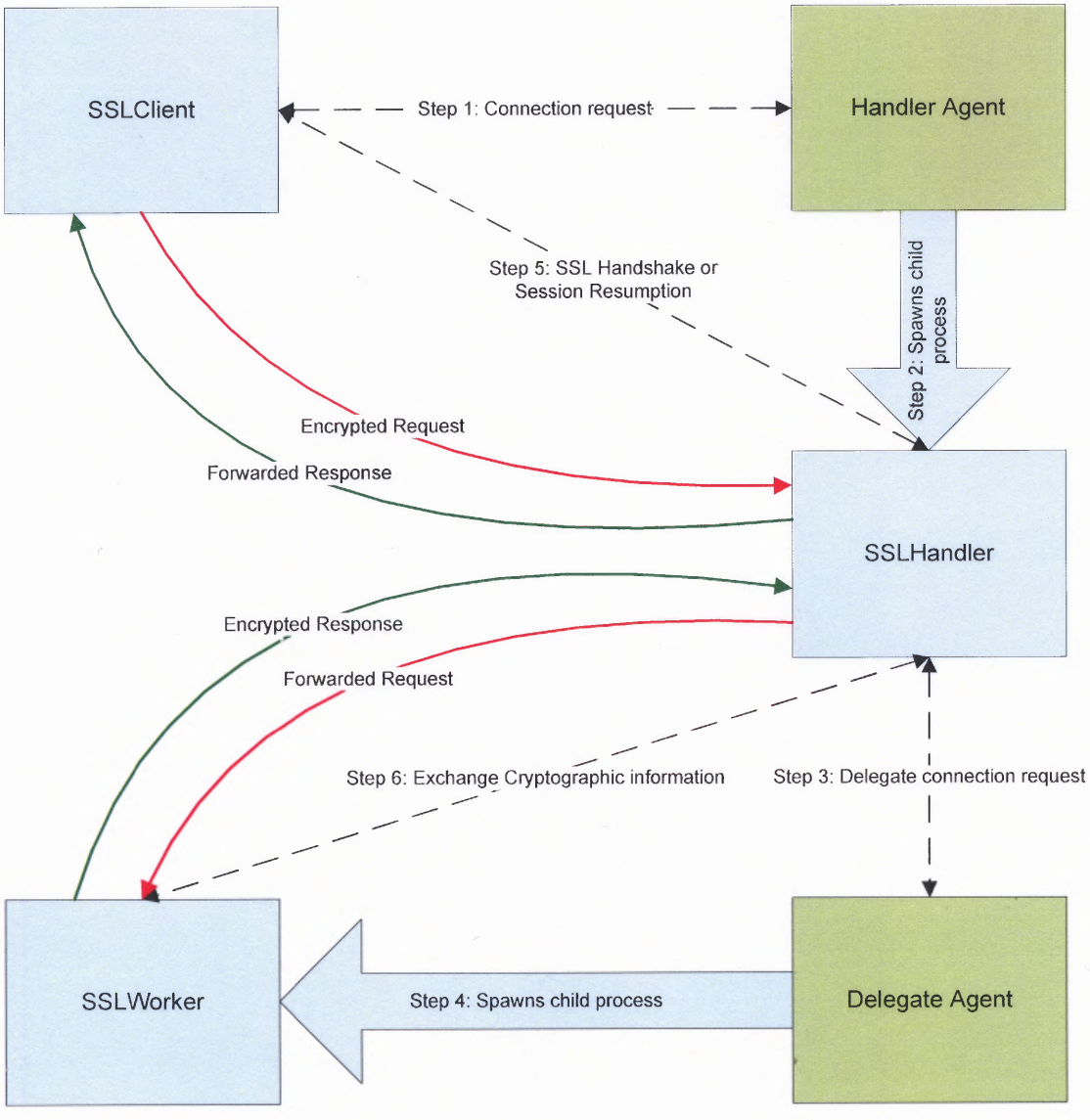


Figure 4.1 Overall logical architecture.

After Step 6, the *SSLHandler* is ready to delegate incoming requests from the *SSLClient* to the *SSLWorker* instance. The *SSLWorker* instance is ready to receive encrypted requests, decrypt it, and create encrypted responses. These responses are sent back to the *SSLHandler* which, in turn forwards them to the *SSLClient*. The red and green arrows shown in Figure 4.1 indicate the flow of requests and responses, respectively.

The process of SSL session resumption does not involve Steps 2 and 4 since there is no need to spawn new *SSLHandler* and *SSLWorker* processes. The *Handler Agent* and the *Delegate Agent* re-use the previously spawned processes by locating them based on the resuming SSL session identifier in the connection request (Step 1). More details on this are provided in a later chapter.

4.3 Transferring the SSL Cryptographic State

While an *SSLHandler* instance performs the handshake with the *SSLClient*, the process of data encryption and decryption in the *SSL Record* phase is performed by a *SSLWorker* instance running on a separate server. So, there is a need to transfer the cryptographic state (keys, ciphers, etc.,) attained by the *SSLHandler* instance after the completion of the handshake to its corresponding *SSLWorker* instance.

The transfer of cryptographic state is achieved by the use of ASN.1. The OpenSSL structure SSL* stores all the cryptographic information required to delegate the SSL Record phase to an external process. This structure in turn, contains OpenSSL structures such as SSLSession* and SSLCipher* along with other information. An ASN.1 specification was developed to represent the minimal information contained in the SSL* and its nested structures, required during the *SSL Record* phase. This specification was

provided as input to the *asn1c* compiler, which produced the C language files with headers.

The *asn1c* is a free, open source compiler of ASN.1 specifications into C source code [54]. It supports a range of ASN.1 syntaxes, including ISO/IEC/ITU ASN.1 1988, '94, '97, '02 and later amendments. The encoding used in this implementation is as per the Basic Encoding Rules (BER) syntax.

4.4 Handler Agent and SSLHandler

The *Handler Agent* spawns a child process, referred to as *SSLHandler*, to serve each new SSL connection request (requests without an SSL session identifier). SSL connection requests with an SSL session identifier are handed over to the previously spawned *SSLHandler* process that served a request with the same session identifier. The flowcharts shown below show the actions taken by the *Handler Agent* and *SSLHandler* processes in serving incoming requests.

Figure 4.2 represents the actions taken by *Handler Agent* in particular. The initialization step involves two main activities: (1) Reading a configuration file called *entryserver.config* (2) Setup of a hash table called the *SessionID* table to keep track of the SSL session identifiers and the *SSLHandler* instances serving these SSL sessions. The *entryserver.config* file contains the server names and the respective port numbers of the *Delegate Agent* instances to connect to in a round-robin fashion. Future sections in this chapter discuss the implementation of SessionID table and the enablement of SSL session resumption in more detail.

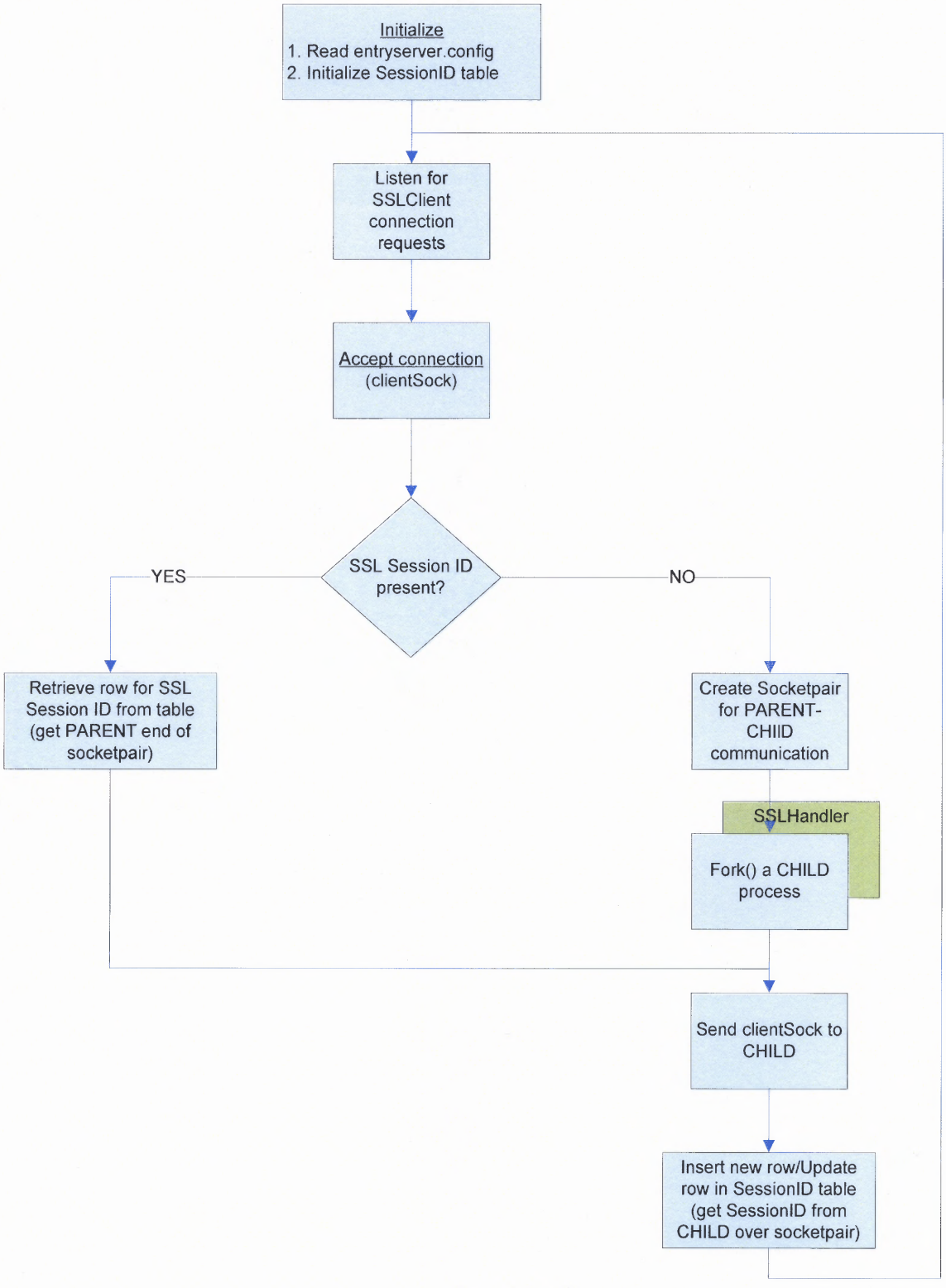


Figure 4.2 Handler Agent.

As the *Handler Agent* listens for incoming requests and accepts a connection, it checks the request to see if an SSL session identifier is present. If it does not find one, the request is treated as one coming from a new *SSLClient*. First, a pair of sockets is created using the *socketpair()* system call. The *Handler Agent* will use one socket from this pair to communicate with its child. This is followed by the spawning of a new child process (*SSLHandler*) using the *fork()* system call. The *SSLHandler* instance can communicate with the *Handler Agent* since it inherits both the socket descriptors from its parent (*Handler Agent*). The *Handler Agent* then sends the socket descriptor obtained by accepting an incoming connection request from an *SSLClient* to the *SSLHandler*. This enables the *SSLHandler* to communicate directly with the *SSLClient* to perform an *SSL Handshake*. Meanwhile, the *Handler Agent* returns to accept any more waiting connection requests.

On the other hand, if a newly accepted connection request contains an SSL session identifier, the *Handler Agent* tries to retrieve the value stored in the *SessionID* hash table stored against a key identical to the SSL session identifier. The value stored against the session identifier key is the socket descriptor (derived from the *socketpair()* call) capable of communicating with the *SSLHandler* instance that had previously served a request with the same session identifier. Upon retrieving this socket, the *Handler Agent* uses it to send the new socket descriptor obtained by accepting the incoming connection request from *SSLClient* to the appropriate *SSLHandler* instance. This enables an *SSLHandler* instance to resume an SSL session without the need to perform a full *Handshake* again.

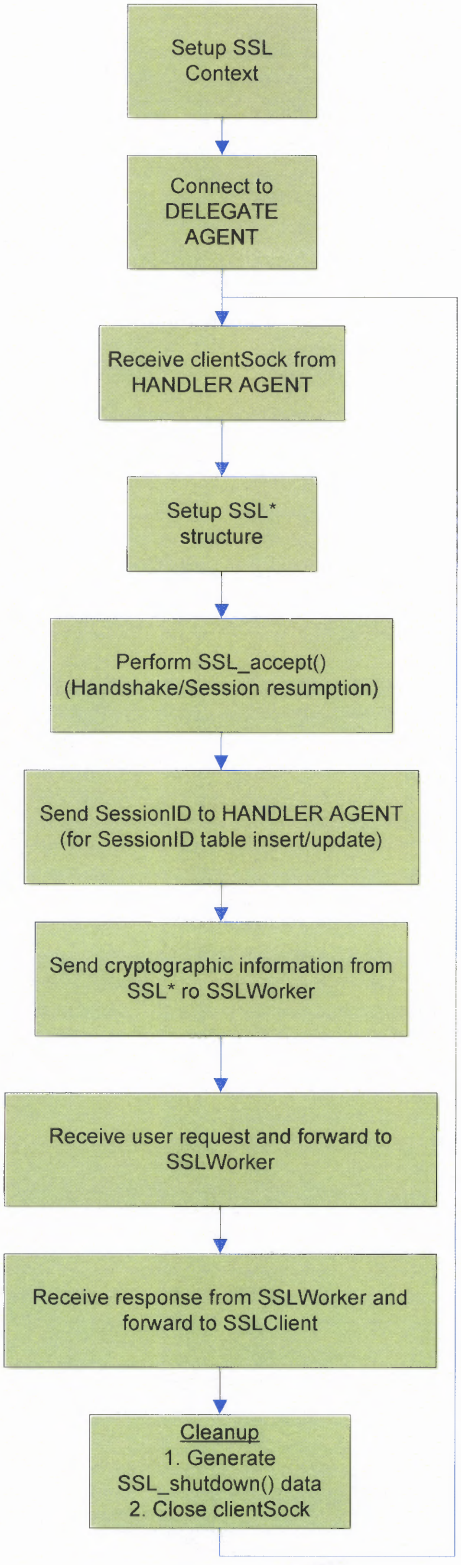


Figure 4.3 SSLHandler.

Figure 4.3 depicts the flow of actions taken by an *SSLHandler* instance. It sets up the OpenSSL SSL_CTX* structure and connects to an instance of *SSLWorker*. All *SSL Record* phase communications from the *SSLClient* will be delegated to this *SSLWorker* instance. This is followed by the receipt of the socket descriptor holding a connection to the *SSLClient*. The OpenSSL library call *SSL_accept()* performs a *Handshake* or a session resumption depending on the presence of an SSL session identifier in the data sent by *SSLClient*. This is followed by the relaying of the session identifier back to *Handler Agent* for inserts/updates to its *SessionID* table. The *SSLHandler* process then uses the ASN.1 routines (generated by the *asn1c* compiler) to communicate all the pertinent cryptographic state information to the *SSLWorker*, which enables it to delegate the *SSL Record* phase.

4.5 Delegate Agent and SSLWorker

The *Delegate Agent* spawns a child process, referred to as *SSLWorker*, to serve the *SSL Record* phase needs of an *SSLHandler* process, which, in turn communicates with an instance of *SSLClient*. The *Delegate Agent* waits for new connection requests and spawns a child process (*SSLWorker*) after accepting a new connection. This is illustrated in Figure 4.4.

An *SSLWorker* sets up the OpenSSL SSL_CTX* structure and waits for incoming data (cryptographic information) from its corresponding *SSLHandler*. It creates the required SSL* and its nested data structures and copies the cryptographic information received into these structures. This enables the *SSLWorker* instance to encrypt/decrypt information. This process is illustrated in Figure 4.5.

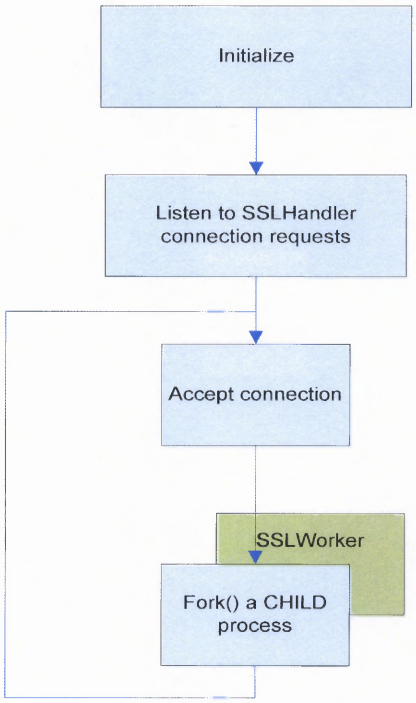


Figure 4.4 Delegate Agent.

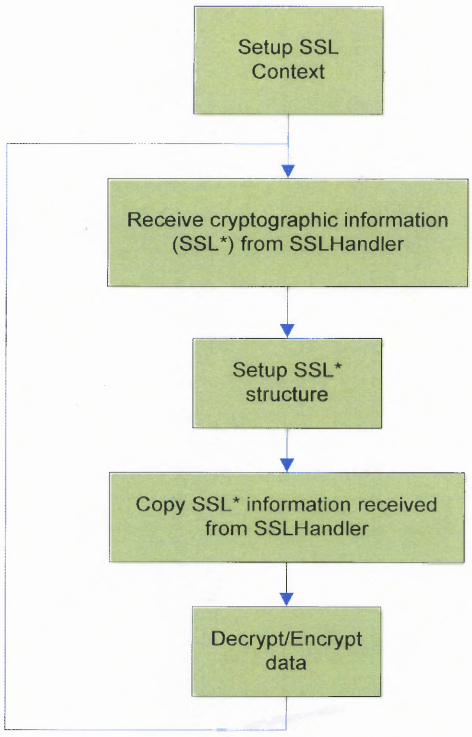


Figure 4.5 SSLWorker.

4.6 SSLClient

The *SSLClient* functions as depicted in Figure 4.6. It initializes and sets up the SSL_CTX* structure. It follows this by creating a TCP/IP based socket connection to the *Handler Agent*. Using this socket, an SSL connection is established (through an *SSL_connect()* call) with an *SSLHandler* process.

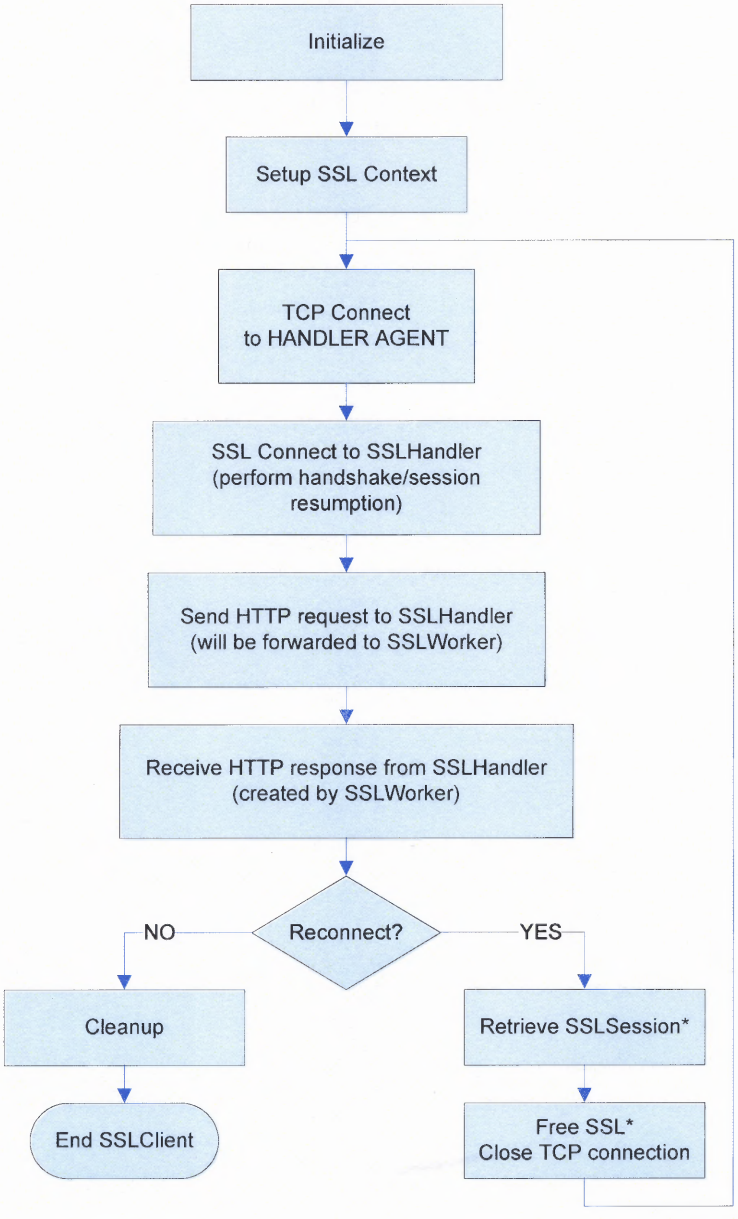


Figure 4.6 SSLClient.

HTTP requests are sent through this SSL connection in an encrypted format and responses are read back and decrypted. After the first request-response cycle, an attempt is made to resume the previously created SSL session by retrieving the SSLSession* from the SSL* structure and re-using it in a new SSL connection. After a pre-determined number of such session resumptions, the *SSLClient* terminates. This sequence of actions simulates a real-world user establishing a new SSL connection with a backend application server, carrying out some transactions and then disconnecting.

4.7 Resuming the SSLSession

The ability to resume an SSL session is very important since SSL resumption is a highly efficient operation that eliminates the need for returning SSL clients to perform a complete *SSL Handshake* [3]. SSL session resumption is implemented by means of a hash table used to keep track of *SSLHandler* processes that have previously performed an *SSL Handshake* and served a request. This hash table is called the *SessionID* table throughout this document. The following C structures show the data stored in each entry of the hash table.

```
struct key {
    char session_id[32];
};
struct value {
    int sd;
};
```

The *SSLHandler* process communicates a newly established SSL session identifier to the *Handler Agent* through a UNIX domain socket. The *Handler Agent* receives this information and creates a new entry in the *SessionID* table with the SSL session identifier as the key (*struct key*) and the corresponding socket descriptor as the value (*struct value*). This socket descriptor represents the *SSLHandler* process in the *Handler Agent* since it provides the ability to communicate with the *SSLHandler* instance, at will. Although session resumptions do not usually result in a change to the SSL session identifier it has been observed that some session identifiers undergo a change upon session resumption. This creates the need for an *SSLHandler* instance to communicate the session identifier back to *Handler Agent* for every request received. This two-way communication channel enables *Handler Agent* to send socket descriptors (representing connections to *SSLClient* instances) to the appropriate *SSLHandler* instances during session resumptions.

New *SSLHandler* instances are created only when a SSL session is to be established for the first time with an *SSLClient*. These instances are not destroyed after the completion of the first request, but are preserved in memory. By preserving these running *SSLHandler* processes, it is possible to resume SSL sessions with minimal overhead by avoiding the creation of new processes with each subsequent request from the same *SSLClient*. The *Handler Agent* peeks into a connection request to see if a SSL session identifier is available. If one is present, it looks up the *SessionID* table to retrieve the UNIX domain socket connecting it to the appropriate *SSLHandler* instance. The socket descriptor representing the new client connection is then transported to the *SSLHandler* via the UNIX domain socket for session resumption to take place.

CHAPTER 5

EXPERIMENTAL SETUP AND IMPLEMENTATION

This chapter discusses in detail the experimental setup used to run the *SSLClient*, *Handler Agent*, *SSLHandler*, *Delegate Agent* and *SSLWorker* processes described in the previous chapter including sections to describe the hardware used, server configurations, metrics collected and the method used to collect them

5.1 Hardware

The hardware used in the experiments along with their specifications is listed below.

- 3 servers with the following specifications: 2.1 GHz, 512 MB RAM, 2 GB swap space. These servers were primarily used as load generators by spawning *SSLClient* instances as background processes through shell script.
- 1 server with the following specifications: 2.1 GHz, 512 MB RAM, 2 GB swap space. This server was used to run the *Handler Agent* and all the *SSLHandler* instances.
- 1 server with the following specifications: 2.1 GHz, 512 MB RAM, 2 GB swap space. This server hosted the *Delegate Agent* and *SSLWorker* processes in the 2-Server configuration. It was also used to run the *datacollector* tool for collection of metrics reported by the *SSLHandler* processes. The *datacollector* tool is described briefly in a separate section of this chapter.

5.2 Server Configurations

Two different server configurations were used to compare the performance of the proposed approach (using separation of SSL protocol phases) against the current approach (without separation). These configurations are referred to as No-delegation and

2-Server configurations. The test results obtained from the No-delegation configuration will be used as a benchmark for comparing the performance gains and losses against the 2-Server configuration.

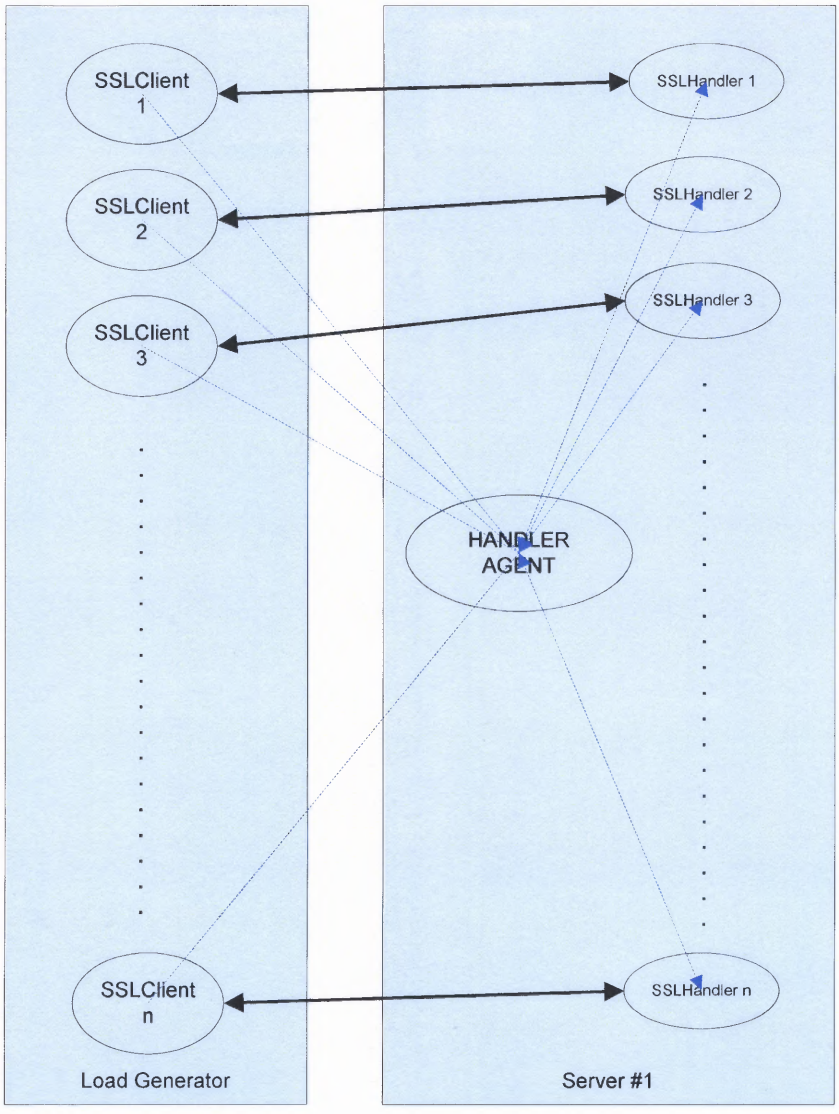


Figure 5.1 No-delegation configuration.

Figure 5.1 shows the No-delegation configuration. In this configuration, the *Handler Agent* and all the *SSLHandler* instances run on the same server. The *SSLHandler* instances perform all the *SSL Record* phase operations without delegating to *SSLWorker*

instances. This configuration mirrors the classical setup used in most applications using SSL. The dark (solid black) lines indicate two-way data flows between pairs of *SSLClient – SSLHandler* processes. A one-to-one correspondence exists between these processes.

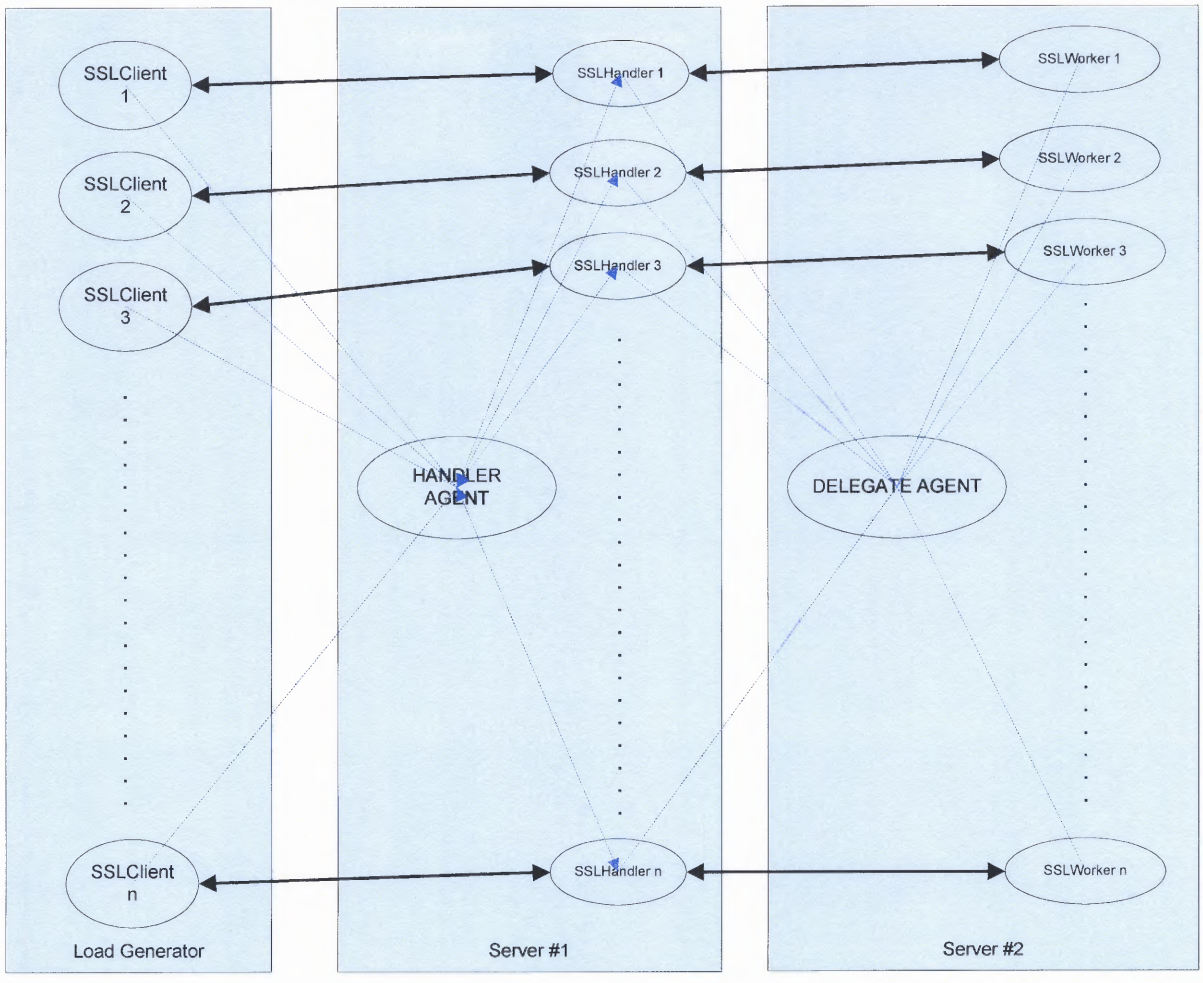


Figure 5.2 2-Server configuration.

Figure 5.2 shows the 2-Server configuration. In this configuration, the *Handler Agent* and its children (*SSLHandler* processes), run on one server while the *Delegate Agent* and its children (*SSLWorker* processes) run on a separate server. The physical server running the *Handler Agent* and its children will be referred to as the *Front-End* server, and the server running the *Delegate Agent* and its children, as the *Back-End*

server. A one-to-one relationship exists between *SSLClient*, *SSLHandler* and *SSLWorker* processes. In this configuration, all *SSL Record* phase processing is off-loaded to the *Back-End* server.

The Load Generator shown in Figure 5.1 and Figure 5.2 comprised of three machines with similar capacities. Such a scheme was adopted to ensure that the concurrency generated by the load was as close to the realistic Internet loads as possible.

5.3 Metrics

Experimental runs were carried out under varying loads generated by the load generating machines. Each run created a load on the server by executing the *SSLClient* instances concurrently. Varying loads were created by executing 100 to 900 *SSLClient* instances in steps of 100 each for each run. Each of these runs was repeated for different data sizes of 2 KB, 6 KB, 10 KB, 20 KB and 200 KB. The following metrics were collected by *SSLHandler* and *SSLWorker* instances during each run for analysis.

- **Handshake time**: This value is the time taken (in milliseconds) to perform a complete *SSL Handshake* by the *SSLHandler* with the *SSLClient*. The relevance of this metric holds for both the No-delegation and 2-Server configurations detailed in the previous sections since the *SSL Handshake* is always performed between an *SSLClient* and an *SSLHandler* processes.
- **Encryption time**: This is the time taken (in milliseconds) by an *SSLWorker* instance to perform symmetric encryptions in response to requests. This value does not include the time taken to send the encrypted responses over the network.
- **Session Initiation time**: Session Initiation represents the first SSL connection negotiated by an *SSLClient* instance with an *SSLHandler* instance and it includes the receipt of a request and the generation of corresponding response. It is relevant for both the No-delegation and 2-Server configurations. The total time taken (in milliseconds) by an *SSLHandler* instance to perform Session Initiation with an *SSLClient* instance is called Session Initiation time.
- **Session Resumption time**: This represents the resumption of an SSL session after its

establishment through the Session Initiation process. The total time taken (in milliseconds) by an *SSLHandler* instance to resume an SSL session is called Session Resumption time. Like Session Initiation, this term holds relevance for both the No-delegation and 2-Server configurations too.

- **Total processing time**: In every run, an *SSLClient* instance generated three requests in succession, the first one being a Session Initiation and the other two, Session Resumptions. Each *SSLHandler* instance tracked the Session Initiation and the two Session Resumption times. The total processing time (in milliseconds) is the difference in the time between the beginning of the Session Initiation and the end of the last Session Resumption.

The above listed metrics were collected by each *SSLHandler* instance and reported to a tool called the datacollector. The datacollector is a simple executable that listens for connections on TCP/IP based sockets and collects the incoming data. This data is then appended to a file for later analysis. One such data file was created by each experimental run.

CHAPTER 6

RESULTS AND ANALYSIS

This chapter lists and discusses the results obtained by carrying out the experiments detailed in the pervious chapter. The analysis performed on these results will not only provide an insight into the benefits gained by the separation of the protocol phases, but also help peer into the future scope and potential gains thereof.

6.1 Handshake Times

Handshake times were measured in the 2-Server configuration (Figure 5.2) on an idle machine and averaged to obtain a value of **64 milliseconds**. Although the SSL Handshake involves the exchange of multiple messages between the client and the server and the latencies depend on the various Internet factors such as routing, congestion, packet loss, etc., the test environment was highly controlled due to the clients and servers residing on the same network. This eliminated the typical latencies introduced by the Internet. Thus, the SSL Handshake time measured is a reasonably good measure of the server's ability to perform the cryptographic operations required. At this rate, the server hosting the *SSLHandler* processes was able to perform a theoretical maximum of **15.62 SSL Handshakes/sec.**

Table 6.1 shows the average Handshake times in milliseconds for the No-delegation and 2-Server configurations at various data sizes. The average Handshake times remain fairly constant at various data sizes, although a slight increase can be seen

for data sizes of 200 KB in both the server configurations. This increase could possibly be due to the increased network latency caused by the large data transfers between the *SSLWorker* - *SSLHandler* and *SSLHandler* - *SSLClient* pairs. Refer to Table A.1 in Appendix A for a detailed listing of Handshake times recorded under various load conditions. It is also worth noting from that the variance of the Handshake times from the mean is very small in Table A.1.

From Table 6.1, it is evident that there is a drop in Handshake times at higher data rates (200 KB) for the 2-Server configuration compared to the No-delegation configuration. This can be attributed to the fact that the *Front-End* server is less loaded and free to handle Handshakes faster due to the off-loading of the *SSL Record* phase encryptions to the *Back-End* server. Refer to Table A.1 for a detailed listing of Handshake times and percentage differences. The negative percentage differences are due to lower average Handshake times in the 2-Server configuration as compared to the No-delegation configuration.

Table 6.1 Average SSL Handshake Times

	Handshake time in milliseconds	
Data Size	**No-delegation**	**2-Server**
2 KB	65	65
6 KB	66	65
10 KB	66	66
20 KB	66	66
200 KB	82	74

6.2 Encryption Times

Encryption times were measured in the 2-Server configuration at various loading conditions and data sizes. Table 6.2 lists the average Encryption time measured at conditions of No-load and loads of 300, 600 and 900 *SSLHandler* instances each, for varying data sizes. The column titled No-load in Table 6.2 indicates measurements taken by running one instance of the *SSLWorker* on an idle machine. These values represent the best performance attainable on the given server. Figure 6.1 displays the same in a graphical format.

Table 6.2 Average Encryption Times

	Encryption time in milliseconds			
Data size	**No-load**	**300 SSLHandlers**	**600 SSLHandlers**	**900 SSLHandlers**
2 KB	<1	0.08	0.09	0.11
6 KB	0.33	0.58	0.51	0.51
10 KB	0.67	1.21	1.35	1.45
20 KB	1.33	3.84	4.62	3.85
200 KB	19.83	68.09	92.54	108.67

From Figure 6.1, it is evident that the Encryption times increase in direct proportion to the data size at No-load conditions. As the load increases, the encryption times increase rapidly with increasing data sizes for each load condition. The increase is more pronounced at higher data sizes (200 KB) and higher loads (900 *SSLHandler* instances). By extrapolation of this trend for data sizes higher than 200 KB, it is clear that a server is significantly burdened at high loads and large data sizes.

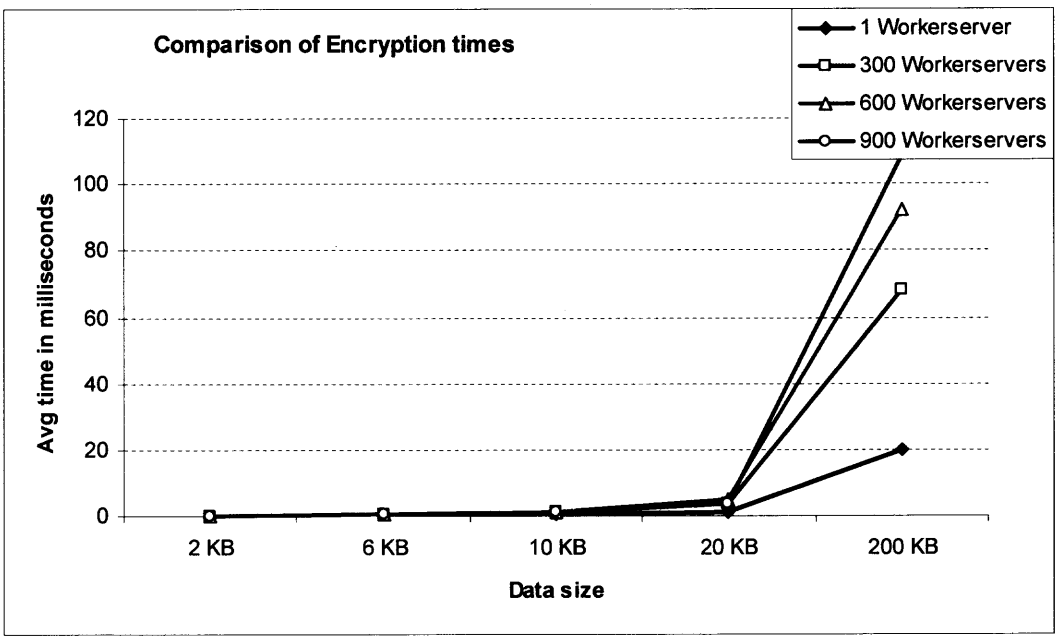


Figure 6.1 Comparison of encryption times.

6.3 Total Processing Times

Total processing times were collected for experimental runs differing in terms of the load and size of the data requested. A Total processing time value represents the time taken to receive the three requests, carry out the required SSL handshake/session resumption activities and perform the data encryption. Since latencies introduced by the Internet are eliminated in the experimental setup, this value is a good measure of the throughput by the server. The values for the No-delegation and 2-Server configurations (Figure 5.1 and Figure 5.2) for each data size and load are listed.

Table 6.4 lists the % difference between the No-delegation and 2-Server configurations for the various data sizes and loads. The positive values in green cells indicate improved throughput in the 2-Server configuration over the No-delegation

configuration, while the negative values indicate degradation in throughput experienced in the 2-Server configuration. The red cells indicate especially large amounts of degradation (10% or more).

From Table 6.4, a relatively large number of green cells are seen in columns for smaller data sizes (2 KB, 6 KB and 10 KB) than at larger data sizes (20 KB and 200 KB). This indicates that at small data sizes, the 2-Server configuration provides improved throughput over a wide range of load conditions, in spite of the additional overhead caused by TCP/IP communication between with the *SSLHandler* and *SSLWorker* instances. As the data size increases, the performance deteriorates as is evident by the clustering of red cells in the 20 KB and 200 KB categories. This can be attributed to the overhead caused by the introduction of additional network communication between the *SSLHandler* and *SSLWorker* instances running on physically separate servers.

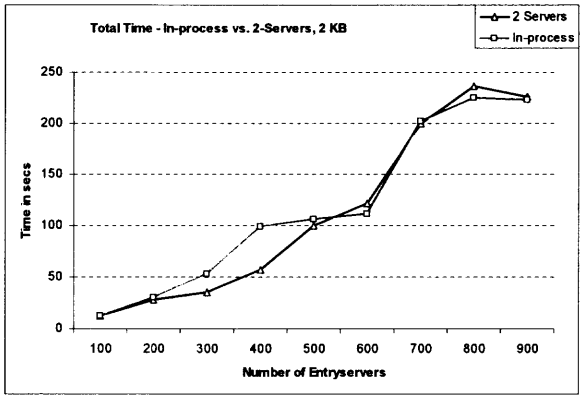
Table 6.3 Total Processing Times*

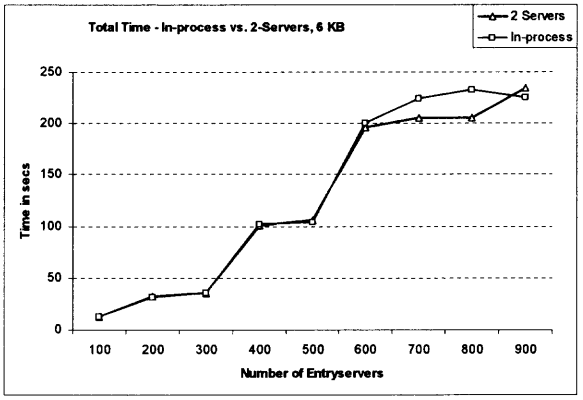
	2K		6K		10K		20K		200K	
SSLHandlers	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**
100	12992	12146	12277	12213	12085	12240	12686	14371	16473	18534
200	30148	28461	30751	32266	30554	27764	28165	28689	31653	49986
300	52658	35164	35457	35781	35313	37077	35403	38736	69073	72399
400	98917	57550	101687	100541	54760	57173	58594	100296	103286	97812
500	106246	100252	103248	105474	99687	99015	114885	104829	111141	126694
600	111905	121365	200113	195896	110869	197278	197072	198844	199972	233922
700	201871	199791	224090	205139	207789	203041	209263	203768	219664	221333
800	224866	236615	232573	205737	223980	221710	223008	216553	224173	231996
900	223362	225980	225341	234247	240802	245075	225134	246726	293698	288752

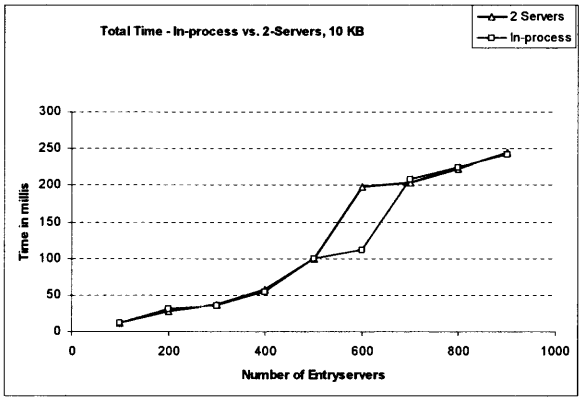
*All values in Table 6.3 are in milliseconds.

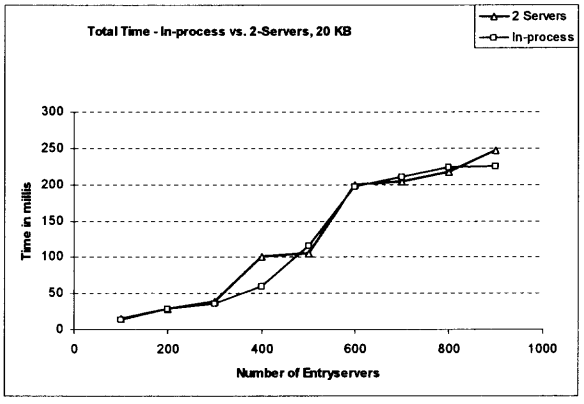
Table 6.4 % Difference in Total Processing Times

SSLHandlers	2K % Diff	6K % Diff	10K % Diff	20K % Diff	200K % Diff
100	7%	1%	-1%	-13%	-13%
200	6%	-5%	9%	-2%	-58%
300	33%	-1%	-5%	-9%	-5%
400	42%	1%	-4%	-71%	5%
500	6%	-2%	1%	9%	-14%
600	-8%	2%	-78%	-1%	-17%
700	1%	8%	2%	3%	-1%
800	-5%	12%	1%	3%	-3%
900	-1%	-4%	-2%	-10%	2%









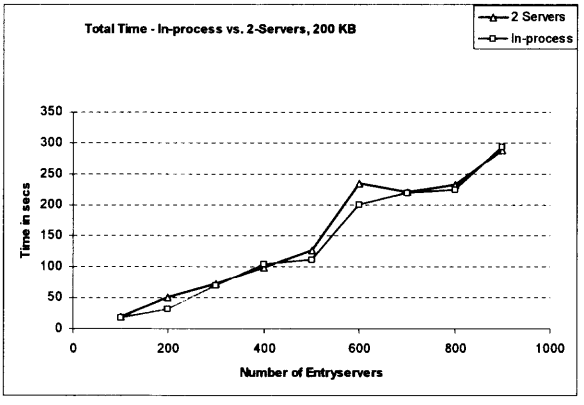


Figure 6.2 Comparison of total processing times.

Notice that the degradation in the 2-Server configuration's performance is ≤5% in 15 out of 25 cells in Table 6.4 (see red cells with negative percentages). This, in spite of increased communication overhead (almost doubled), indicates that the 2-Server

configuration provides potential for significant improvements in server throughput via a mechanism, where-in, a *SSLWorker* instance completely by-passes its corresponding *SSLHandler* instance while responding to a request. This fact is evident in Figure 6.2 where the lines closely follow each other or separate in favor of the 2-Server configurations except in a few cases at large data sizes.

6.4 Session Initiation and Resumption Times

Session Initiation is the process of serving the first request from an *SSLClient* by the *SSLHandler*, where a full *SSL Handshake* is performed. Session Resumption is the process of resuming an already existing SSL session, which is usually the preferred method for purposes of efficiency. In this case, a complete *SSL Handshake* is not performed.

Table 6.5 % Increase in Session Initiation and Resumption Times

Session Initiation		Session Resumption	
Data Size	**% Increase**	**Data Size**	**% Increase**
2 KB	8.409	2 KB	7.556
6 KB	8.391	6 KB	7.565
10 KB	8.375	10 KB	8.056
20 KB	8.324	20 KB	14.141
200 KB	51.278	200 KB	215.436

The average values of Session Initiation and Session Resumption times for each load condition was computed for a given data size. A listing of these average Session Initiation and Session Resumption times are as shown in Appendix B and Appendix C respectively. A mean of these averages over different load conditions was computed for each data size. The resulting percentage increases of these averages for the 2-Server

configuration over the No-delegation configuration are listed in Table 6.5. Figure 6.2 displays the log (base 2) of these percentage increases plotted against the data size.

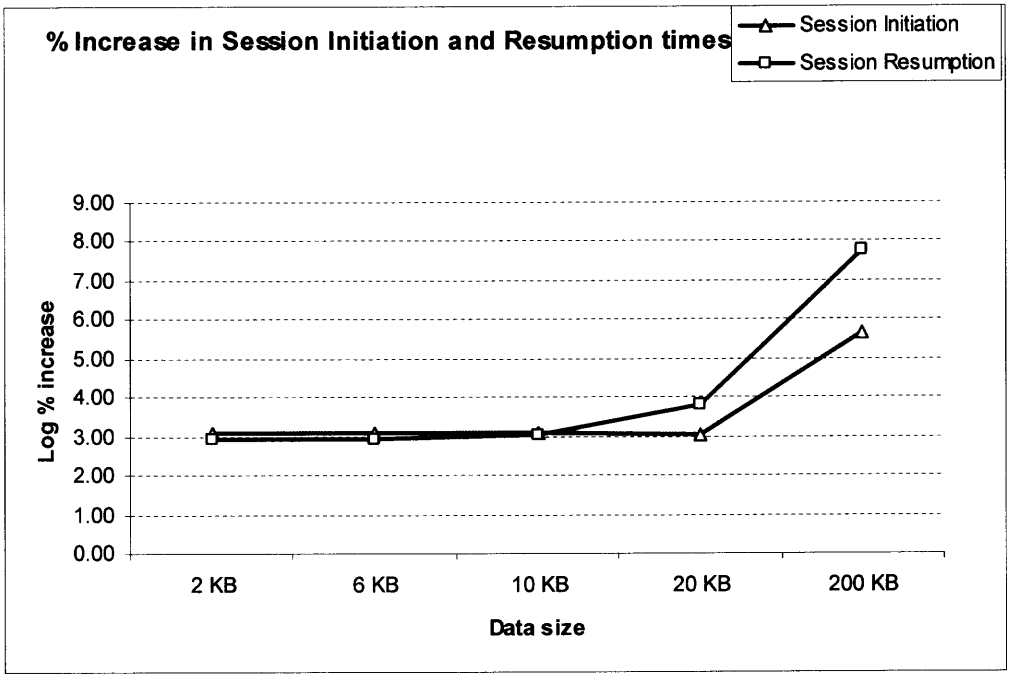


Figure 6.3 Log % increase versus data size.

The pattern of increase in average Session Initiation times (Figure 6.3) closely mirrors the average Handshake times (Table 6.1). The % increase stays mostly constant for data sizes 2 KB, 6 KB, 10 KB and 20 KB, and increases sharply for 200 KB. This implies the addition of an overhead that is a proportionally constant to the No-delegation Session Initiation times at every data size. A similar observation can be made regarding the Session Resumption times and the average Handshake times too. The % increase remains roughly constant for data sizes 2 KB, 6 KB and 10 KB, with a small increase for 20 KB followed by a sharp increase for 200 KB. Network communication overhead caused by the *SSLHandler* acting as an intermediary between the *SSLWorker* and the *SSLClient* during the Record phase provides a plausible explanation to such observations.

This network communication overhead is not introduced in the No-delegation configuration since the *SSLHandler* instances perform all of the *SSL Record* phase encryption and return the data directly to the client.

6.5 Experimental Limitations

During the experimental runs, it was observed that the *Front-End server* reached the limit of its processing capabilities when the number of *SSLHandler* processes reached slightly beyond 1000. The 1-minute *loadavg* valued reached highs of 50-60. Such values for the *loadavg* indicate that the server is stretched well beyond its capabilities. In such cases, it was also noticed that many *SSLHandler* instances failed to complete their processing. Ideally, a cluster of *Front-End* servers is desirable to distribute the load under such conditions. A limitation in available hardware prevented such clustering and conducting experimental runs exceeding 900 *SSLHandler* instances.

The number of machines used to generate the load imposes a limitation on the experiments too. In the experiments conducted, three machines were used to generate loads with a peak load of 300 *SSLClient* instances from each machine contributing to 900 *SSLClient* instances in total. This number is not nearly close enough to realistic Internet loads in the tens to thousands to millions of clients.

CHAPTER 7

CONCLUSIONS

The separation of the SSL protocol phases across process boundaries, which can be distributed over multiple machines was investigated and the results, discussed in the previous chapter. A set of conclusions can be drawn from these results that direct our attention to the potential benefits of using such a distributed approach and entail the future scope of work that can be carried out in this regard.

Although the *SSL Handshake* consumes computational resources at a much higher order of magnitude as compared to the *SSL Record* phase at low data sizes and loads, the *SSL Record* phase consumption grows to comparable orders and beyond, under high loads and for large data sizes. Given the fact that the Handshake times measured in these experiments are a good measure of the actual cryptographic loads involved due to the isolation of the testing environment from the Internet, it is evident that clustering capabilities and scalability provided by the separation of the SSL protocol phases will provide benefits as the *SSL Record* phase begins to consume a larger proportion of resources in comparison to the *SSL Handshake* during encryption and decryption of large amounts of data. An indication of improvement in Handshake times at higher data rates (200 KB) due to the offloading of *SSL Record* phase computations points toward a payoff that can be gained at high data rates and at high loads.

At smaller data sizes (2 KB, 6 KB and 10 KB), the approach using SSL protocol separation begins to show some improvement in server throughput in terms of the total

number of requests processed per unit time. However, at larger data sizes of 20 KB, 200 KB and possibly higher, the benefits of separation begin to erode owing to the higher communication overheads in the current implementation. This overhead is caused by the data packets containing the encrypted response traversing the network protocol stack of the *SSLHandler* up to the application layer (TCP) before being re-transmitted to the client. This can be avoided by *direct routing* of responses to the client instead of a passing through an intermediary. Direct routing can be achieved using NAT or such other means.

There is a slight degradation in the Session Initiation and Session Resumption times with SSL protocol separation. Again, this is caused by the additional overhead introduced by the communication between the *SSLHandler* and the *SSLWorker* instances. While the percentage increase in average initiation and resumption times is mostly constant for smaller data sizes (up to 10 KB), it increases rapidly as the data size grows larger (20 KB and higher). This increase directly contributes to the latency in response times experienced by the clients requesting large amounts of data such as for large images, video files, etc. Direct routing of responses from the *SSLWorker* to the SSL clients would address the reduction of this overhead.

The load tests carried out in the experiments detailed in previous chapters are not an exact representation of true load experienced by a server in the Internet, but only an approximation. Concurrent requests were created by background clients running on multiple machines residing within the same network as the servers. While such an approach eliminated the latencies introduced by the Internet, large scale loads involving tens of thousands to millions of clients is hard to create with limited hardware resources.

In view of such limitations, the results obtained from tests conducted provide only a reasonable peek into the trends that can be expected under more realistic loads. These trends can then be used to design more elaborate and accurate experiments to improve server throughput.

It was observed that the Front-End server eventually posed a bottleneck. Scaling and load-balancing multiple Front-End servers would solve the problem. The ideal solution would contain m Front-End servers performing SSL Handshakes and delegating the SSL Record phase to n Back-End servers. The m Front-End servers also require seamless migration of SSL session information so that Session Initiation and Session Resumptions can be carried out by different Front-End servers. Direct routing of responses from Back-End servers to SSL clients would eliminate the communication overhead experienced in our experiments.

Separation of the protocol phases is not intended for improving server throughput alone, but also to enable the clustering of SSL based applications such as Web servers, VPN servers, etc. for high availability. Hardware SSL Accelerators increase the cost of providing high availability due to increased costs incurred in procuring and maintaining backups for fail-over. Clustering by phase separation provides an economic alternative to gain high availability since SSL sessions can be seamlessly migrated between processes from one machine to another in case of failure. Scaling is also achieved at low cost by adding additional computing power as required. Such open-source software based solutions are also conducive to customizations as opposed to expensive and proprietary hardware.

The key contribution of this thesis is the separation of the SSL Handshake and

Record phases across processes executing on different machines. It was aimed at clustering the SSL services offered by VPN and Web servers by offloading the encryption and decryption services. It was demonstrated through load tests that improved throughput can be achieved by such a separation, although at higher data rates, the communication overhead erased the gains. Future scope of work was identified in the form of eliminating the communication overhead through direct routing to improve performance. Direct routing can be achieved by means of NAT or socket migration. This concept is illustrated in Figure 7.1.

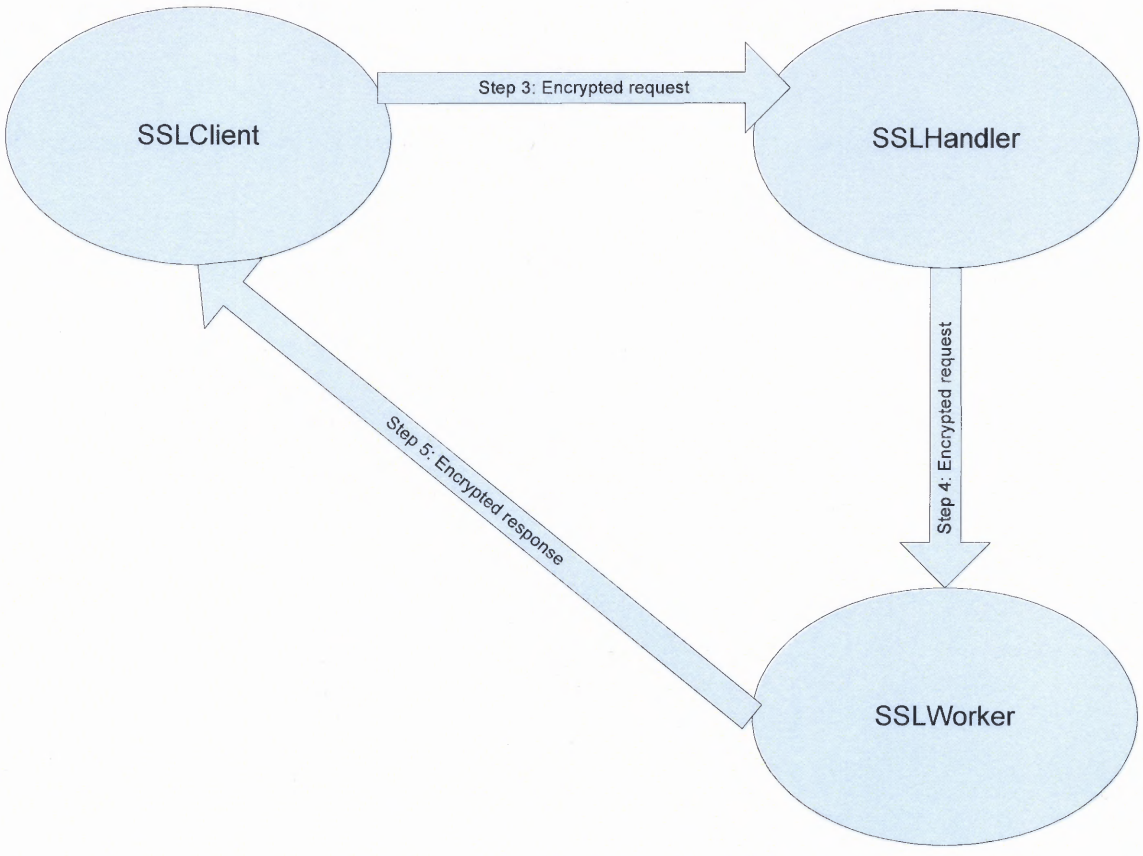


Figure 7.1 Direct routing of SSL responses.

CHAPTER 8

FUTURE SCOPE

Future work can be conducted in this area to address the shortcomings of the separation and further enhance the scalability of servers providing SSL-based services. The key shortfall in the approach developed arises from the routing of encrypted request from *SSLClient* through the *SSLHandler* process en-route to the *SSLWorker* process and the same of the encrypted response from the *SSLWorker* to the *SSLClient*. The latencies introduced are mainly due to the fact that the data does not traverse just up to Layer 3 of the protocol stack in the intermediary, but all the way up to the application layer and back out again onto the network. Direct routing of data between the *SSLClient* and *SSLWorker* achieved through NAT can eliminate this overhead during the *SSL Record* phase. The proposed direct routing is illustrated in Figure 8.1.

Currently, the *SSLHandler* instances are responsible for both the *SSL Handshake* and session resumption activities. Since SSL session resumption is relatively easy and non-intensive, this activity can be carried out by the *SSLWorker* directly instead of the *SSLHandler*. This will require the *Handler Agent* to hand-off SSL session resumption requests to the appropriate *SSLWorker* instance as shown in Step 7 and 8 of Figure 8.1. Such a hand-off will provide two benefits:

- Eliminate the overhead of transmitting the SSL cryptographic information from the *SSLHandler* to the *SSLWorker*
- Remove the need for preserving *SSLHandler* instances on the Front-End server, thereby reducing the number of running processes.

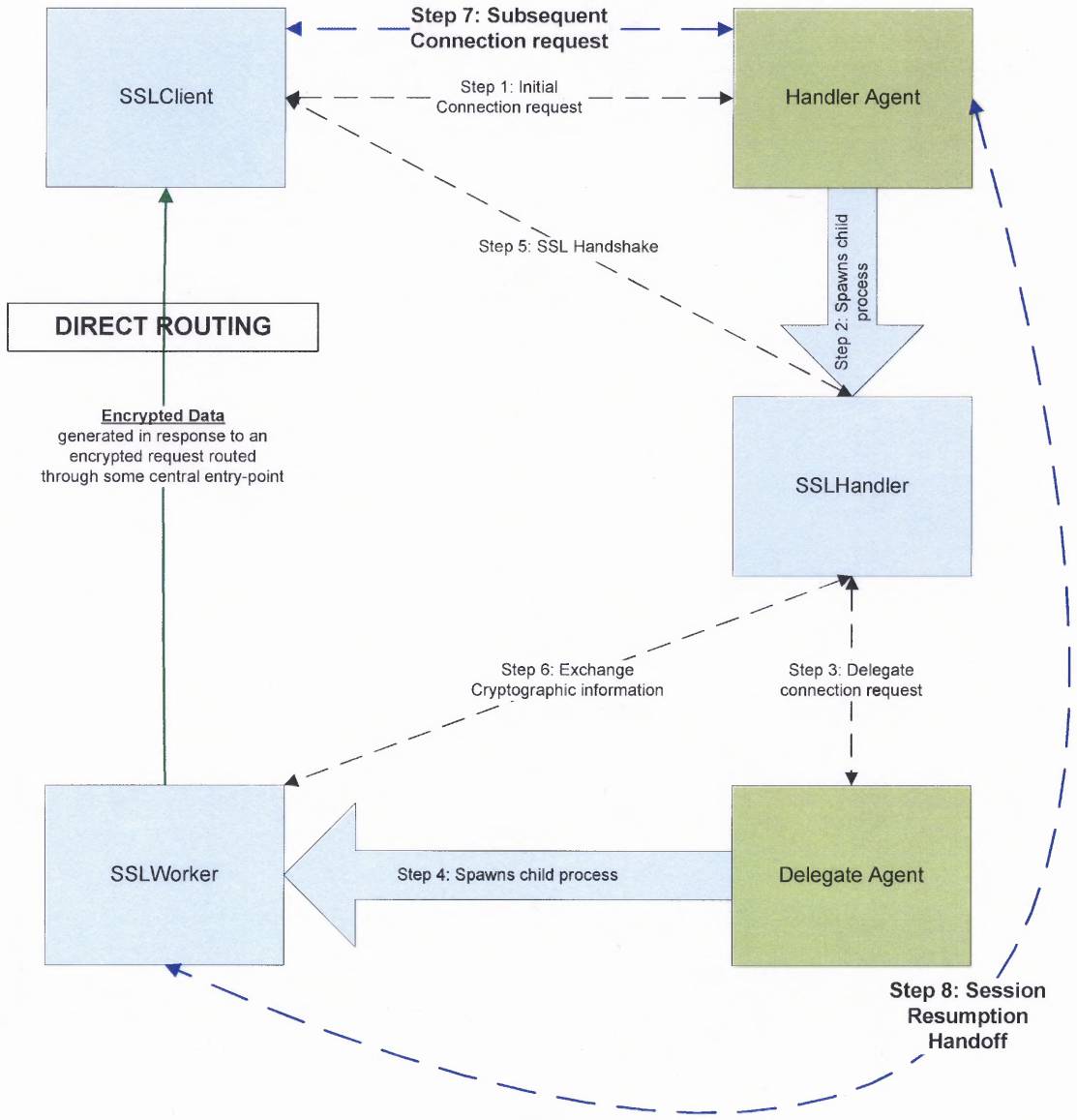


Figure 8.1 Proposed enhancements.

Clustering of multiple *Back-End* servers providing *SSL Record* phase services has already been made possible with the current implementation. A further extension of clustering multiple *Front-End* servers will provide comprehensive scalability to the solution. While an Initial Connection Request (Step1 in Figure 8.1) can be satisfied by any *Front-End* server, a Subsequent Connection Request (Step 8 in Figure 8.1) containing an SSL session identifier can arrive at a *Front-End* server other than the one

that processed the Initial Connection Request. At this time, the *Handler Agent* will require the *Back-End* server and *SSLWorker* port number information to perform the hand-off. Thus, to achieve *Front-End* clustering, the *Handler Agent* instances running on separate *Front-End* servers will require the ability to share information mapping SSL session identifiers to *SSLWorker* instances running on different *Back-End* servers. Such sharing can be achieved by means of a shared cache or a simple communication protocol between the *Handler Agents*.

APPENDIX A

HANDSHAKE TIMES

Handshake times for experimental runs under different loads and data sizes are listed in Table A.1.

Table A.1 Handshake Times*

	2K		6K		10K		20K		200K	
Clients	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**
100	0.065	0.065	0.065	0.065	0.066	0.065	0.068	0.065	0.076	0.073
200	0.065	0.066	0.065	0.065	0.065	0.065	0.065	0.065	0.087	0.073
300	0.065	0.065	0.066	0.066	0.065	0.066	0.066	0.066	0.087	0.075
400	0.065	0.065	0.065	0.066	0.065	0.065	0.066	0.066	0.083	0.075
500	0.065	0.066	0.066	0.066	0.066	0.066	0.066	0.066	0.078	0.076
600	0.065	0.065	0.065	0.065	0.066	0.066	0.066	0.066	0.081	0.074
700	0.066	0.065	0.065	0.066	0.065	0.065	0.066	0.066	0.082	0.074
800	0.065	0.065	0.066	0.065	0.065	0.066	0.066	0.066	0.082	0.075
900	0.066	0.065	0.065	0.065	0.066	0.066	0.066	0.066	0.081	0.074
Average	0.065	0.065	0.066	0.065	0.066	0.066	0.066	0.066	0.082	0.074
% Difference	**0.01%**		**-0.02%**		**0.01%**		**-0.03%**		**-0.80%**	

*All values in Table 6.3 are in seconds.

APPENDIX B

SESSION INITIATION TIMES

Session Initiation times for experimental runs under different loads and data sizes are listed in Table B.1.

Table B.1 Session Initiation Times*

	2K		6K		10K		20K		200K	
Clients	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**
100	1.074	1.157	1.074	1.159	1.074	1.161	1.094	1.159	1.134	1.277
200	1.075	1.162	1.075	1.157	1.075	1.163	1.083	1.168	1.227	1.670
300	1.074	1.159	1.077	1.159	1.076	1.156	1.080	1.168	1.217	2.114
400	1.074	1.159	1.074	1.162	1.077	1.158	1.084	1.171	1.194	1.634
500	1.074	1.159	1.075	1.161	1.079	1.163	1.083	1.168	1.151	1.693
600	1.074	1.159	1.075	1.158	1.077	1.162	1.082	1.168	1.210	1.736
700	1.075	1.158	1.075	1.161	1.076	1.159	1.083	1.169	1.193	1.692
800	1.075	1.157	1.075	1.159	1.077	1.162	1.084	1.170	1.200	1.705
900	1.075	1.158	1.076	1.157	1.078	1.160	1.085	1.166	1.206	1.826
Average	1.075	1.159	1.075	1.159	1.077	1.160	1.084	1.168	1.193	1.705
% Difference	**8.41%**		**8.39%**		**8.38%**		**8.32%**		**51.28%**	

*All values in Table 6.3 are in seconds.

APPENDIX C

SESSION RESUMPTION TIMES

Session Resumption times for experimental runs under different loads and data sizes are listed in Table C.1.

Table C.1 Session Resumption Times*

	2K		6K		10K		20K		200K	
Clients	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**	**No-delegation**	**2-Server**
100	1.011	1.085	1.018	1.091	1.027	1.103	1.076	1.218	1.798	2.987
200	1.009	1.086	1.012	1.087	1.025	1.099	1.033	1.109	1.659	3.820
300	1.011	1.088	1.016	1.090	1.023	1.096	1.064	1.200	1.890	4.414
400	1.010	1.086	1.016	1.095	1.019	1.095	1.045	1.234	1.852	3.882
500	1.009	1.090	1.012	1.093	1.021	1.110	1.045	1.226	1.818	4.308
600	1.011	1.085	1.016	1.091	1.022	1.102	1.049	1.150	1.790	3.557
700	1.011	1.084	1.016	1.093	1.018	1.101	1.044	1.195	1.871	4.138
800	1.012	1.086	1.014	1.090	1.016	1.103	1.044	1.240	1.725	3.910
900	1.011	1.085	1.017	1.087	1.017	1.102	1.047	1.149	1.674	4.450
Average	1.010	1.086	1.015	1.091	1.021	1.101	1.050	1.191	1.786	3.941
% Difference	**7.56%**		**7.56%**		**8.06%**		**14.14%**		**215.44%**	

*All values in Table 6.3 are in seconds.

REFERENCES

1. G. Spafford, *Web Security, Privacy & Commerce*, O'Reilly and Associates, 2001.

2. E. Herscovitz, Secure virtual private networks: the future of data communications. *International Journal of Network Management*, vol. 9, issue 4, July-August 1999, pp. 213-220.

3. D. Kosiur, *Building and managing virtual private networks*, Wiley, September 1998.

4. M. Sarrel, "Improving Performance and Availability of SSL VPN Solutions", August 2003, http://www.findarticles.com/p/articles/mi_zdpcm/is_200308/ai_ziff45224.

5. Matthew D. Wilson, "VPN HOWTO", December 1999, http://www.tldp.org/HOWTO/VPN-HOWTO/index.html.

6. Introduction to VPN-VPN Tunnelling, February 2006, http://compnetworking.about.com/od/vpn/l/aa010701d.htm.

7. K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, G. Zorn, RFC 2637: Point-to-Point Tunneling Protocol (PPTP), July 1999, http://www.ietf.org/rfc/rfc2637.txt.

8. W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, B. Palter, RFC 2661: Layer Two Tunneling Protocol (L2TP), August 1999, http://www.ietf.org/rfc/rfc2661.txt.

9. S. Kent, R. Atkinson, RFC 2401: Security architecture for the Internet Protocol (IPSec), November 1998, http://www.ietf.org/rfc/rfc2401.txt.

10. T. Dierks, and C. Allen, RFC 2246: The TLS Protocol Version 1.0, January 1999, http://www.ietf.org/rfc/rfc2246.txt.

11. A. Freier, P. Karlton, and P. Kocher, The SSL Protocol Version 3.0, November 1996, Netscape, http://home.netscape.com/eng/ssl3/draft302.txt

12. R. Khare, S. Lawrence, RFC 2817: Upgrading to TLS Within HTTP/1.1, May 2000, http://www.ietf.org/rfc/rfc2817.txt.

13. D. Bhatt, S. Schulze, G. Hancke, L. Horvath, "Secure Internet access to gateway using secure socket layer", in *Proceedings of IEEE International Symposium on Virtual Environments*, Human-Computer Interfaces and Measurement Systems, July 2003, pp.157-162.

14. C. Crall, M. Danseglio, and D. Mowers, SSL/TLS in Windows Server 2003, Microsoft Corporation, July 2003, http://www.microsoft.com/technet/prodtechnol/windowsserver2003/technologies/security/sslws03.mspx.

15. E. Rescorla, RFC 2818: HTTP over TLS, May 2000, http://www.ietf.org/rfc/rfc2818.txt.

16. G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradham, D. Saha, "Design, implementation and performance of a content-based switch", in *Proceedings of the Conference on Computer Communications (IEEE INFOCOM'00)*, March 2000, vol. 3, pp. 1117-1126.

17. E. Casalicchio, M. Colajanni, "A client-aware dispatching algorithm for Web clusters providing multiple services", in *Proc. of 10th International World Wide Web Conference, Hong Kong*, May 2001, pp. 535-544.

18. V. Pai, M. Aron, G. Banga , M. Svendsen, P. Druschel, W. Zwaenepoel, E. Nahum, "Locality-aware request distribution in cluster-based network servers", in *Proceedings of the Eighth ACM ASPLOS, San Jose, California,* October 02-07, 1998, pp. 205-216.

19. K. Kant, R. Iyer, P. Mohapatra, "Architectural impact of secure socket layer on Internet servers", in *Proceedings of IEEE International Conference on Computer Design*, Sept. 2000, pp. 7-14.

20. Ed. R. Hinden, RFC 3768: Virtual Router Redundancy Protocol (VRRP), April 2004, http://www.ietf.org/rfc/rfc3768.txt

21. M. Colajanni, P.S. Yu, "A performance study of robust load sharing strategies for distributed heterogeneous Web server systems", *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, issue 2, pp. 398-414, March/April 2002.

22. E. Rescorla, A. Cain, B. Korver, "SSLACC: A clustered SSL Accelerator", in *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA*, August 2002.

23. D. Boneh, H. Shacham, "Improving SSL handshake performance via batching", in *Proceedings of the RSA Conference, San Francisco, CA*, April 2001, pp. 28-43

24. C. Coarfa, P. Druschel, and D. Wallach, "Performance analysis of TLS web servers", in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, February 2002, pp. 183-194.

25. ArrayNetworks, Inc., Array Networks SPX Series, February 2006, http://www.arraynetworks.net/products/EnterpriseSSLVPN.asp.

26. Aventail Corporation, Aventail SSL VPN appliances (Aventail EX-2500, Aventail EX-1500, Aventail EX-750), February 2006, http://www.aventail.com/products_services/appliances/default.asp.

27. Cisco Systems, Inc., Cisco VPN 3000 Series Concentrators, February 2006, http://www.cisco.com/en/US/products/hw/vpndevc/ps2284/index.html.

28. Juniper Networks, Inc., Juniper Networks NetScreen-SA 5000 Series, February 2006, http://www.juniper.net/products/ssl.

29. Nokia, Nokia SSL VPN, February 2006, http://www.nokia.com/nokia/0,8764,43098,00.html

30. NetScaler, Inc., NetScaler SSL VPN feature option for the NetScaler 9000 Series, February 2006, http://www.netscaler.ca/9000_series/options/ssl_vpn.htm.

31. Netilla Networks, The AEP Netilla Security Platform, March 2006, http://www.aepnetworks.com/products/ssl_vpn/nsp/overview.htm.

32. NetSilica, Inc., NetSilica Application Security Gateway, March 2006, http://www.netsilica.com/nshome/index.htm.

33. Nortel Networks, Nortel SSL VPN, July 2005 http://www.nortel.com/products/01/alteon/sslvpn/.

34. Symantec Corporation, Symantec Clientless VPN Gateway 4400 Series, February 2006, http://enterprisesecurity.symantec.com/products/products.cfm?productid=342.

35. Whale Communications Ltd., Intelligent Application Gateway, April 2006 http://www.whalecommunications.com/site/Whale/Corporate/Whale.asp?pi=30.

36. Areabe, inc., SWANStor SSL VPN Remote Access Solution, March 2006, http://www.areabe.com/eng/products/index.html

37. Check Point Software Technologies, Inc., SSL Network Extender, March 2006, http://checkpoint.com/products/ssl_network_ext/index.html.

38. Citrix Systems, Inc., Citrix Access Gateway, March 2006, http://www.citrix.com/English/ps2/products/product.asp?contentID=15005.

39. Menlo Logic, AccessPoint SSL VPN, February 2006, http://www.menlologic.com/ssl_vpn_products.html.

40. OvisGate Software, SSL VPN Windows Solution, February 2006, http://www.ovisgate.com/product.html.

41. PortWise, PortWise mVPN, April 2006, http://www.portwise.com/portImg/Download_press/ProductSheet_PortWise_mVPN.pdf.

42. Tarantella Inc., Secure Global Desktop Enterprise Edition, April 2006, http://www.tarantella.com/support/documentation/sgd/ee/3.42/help/en-us/base/gettingstarted/tarantella_intro.html .

43. V-ONE Corporation, V-ONE clientless solution, January 2005, http://www.v-one.com/ssltestdrive.html.

44. 3SP Ltd., SSL-Explorer, April 2006, http://3sp.com/showSslExplorer.do.

45. Jason Everard, Introduction to SSL , http://docs.sun.com/source/816-6156-10/contents.htm.

46. E. Rescorla, *SSL and TLS: Designing and Building Secure Systems*, Addison-Wesley, New York, NY 2000.

47. Mark J. Cox, Geoff Thorpe, "Apache e-Commerce Solutions", in *ApacheCon 2000,* Florida 2000, http://www.geoffthorpe.net/apcon2000/apachecon2000.pdf.

48. Logan G. Harbaugh, SSL Accelerators – Spped up SSL transactions with hardware, May 2002, http://www.newarchitectmag.com/documents/s=2455/new1017959283467/.

49. Dan Kegel, SSL Acceleration, May 2001, http://www.kegel.com/ssl/hw.html.

50. SC Magazine, SSL accelerators, March 2006, http://www.scmagazine.com/uk/grouptest/details/27f879d4-8d4c-4dd4-9871-6ce94f50efbe/ssl+accelerators/.

51. OpenSSL, January 2005, http://www.openssl.org/.

52. The ASN.1 Consortium, January 2006, http://www.asn1.org/

53. Nicolas George, Ancillary library, June 2005, http://www.eleves.ens.fr/home/george/info/prg/libancillary.html.

54. Open Source ASN.1 Compiler, January 2005, http://lionet.info/asn1c/