Virtual Private Network Communication over a session layer socket protocol (SOCKS)

by

C Richard Soler Avellén

LIU-IDA/LITH-EX-G—11/005—SE

2011-02-17
Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© C Richard Soler Avellén
Virtual Private Network Communication over a session layer socket protocol (SOCKS)

by

C Richard Soler Avellén

LIU-IDA/LITH-EX-G—11/005—SE

2011-02-17
SYSTeam is an IT solutions supplier company that wants to develop a product which permits users to communicate with each other, over the Internet, in a secure way. The solution creates communication between two subnets which are connected through a Virtual Private Network (VPN) Gateway. The security of the communication is implemented at the application layer by using the Secure Socket Layer (SSL) protocol which carries, encrypted within it, a session layer technology called Sock-ets (SOCKS). The communication prototype is developed in a Linux platform with the Integrated Development Environment (IDE) Eclipse and Java programming language.

There are many software-hardware similar based products in the market, but these solutions usually demand high budgets. This thesis shows the development of a communication prototype of a new, and low cost, alternative product. This report also describes how the Java SOCKS methods are increased with further functionality in order to reach the designed communication infrastructure. The entire implementation is tested by using a network analyzer software called Wireshark and a log function which writes out messages in order for us to know which part of the code is running.

Nyckelord
VPN, SOCKS, JAVA, Socket, SSL
Abstract

SYSTeam is an IT solutions supplier company that wants to develop a product which permits users to communicate with each other, over the Internet, in a secure way. The solution creates communication between two subnets which are connected through a Virtual Private Network (VPN) Gateway. The security of the communication is implemented at the application layer by using the Secure Socket Layer (SSL) protocol which carries, encrypted within it, a session layer technology called Sock-et-s (SOCKS). The communication prototype is developed in a Linux platform with the Integrated Development Environment (IDE) Eclipse and Java programming language.

There are many similar software-hardware based products in the market, but these solutions usually demand high budgets. This thesis shows the development of a communication prototype of a new, and low cost, alternative product. This report also describes how the Java SOCKS methods are increased with further functionality in order to reach the designed communication infrastructure. The entire implementation is tested by using a network analyzer software called Wireshark and a log function which writes out messages in order for us to know which part of the code is running.
Acknowledgement

I would like to give thanks to SYSTeam AB for giving me the possibility of doing this work. Thank you Stefan Rosell for contacting me in the very first place and thank you Andreas R Johansson for all the help and support with the project, I did really appreciate all your explanations and ideas.

Special thanks to Simin Nadjm-Tehrani because she had the patience with me and my thesis, and also because she always had the time to guide me into the world of well done technical documentation.

Finally, thank you Jordi Cucurull Juan for all your comments and explanations. With your help, I have completed my thesis and I'm very grateful for it. Gracias Jordi.
## Contents

1 INTRODUCTION .......................................................................................................................... 13
  1.1 Background ............................................................................................................................... 13
  1.2 Prerequisite .............................................................................................................................. 13
  1.3 Purpose ..................................................................................................................................... 14
  1.4 Limitations ............................................................................................................................... 15
  1.5 Abbreviations .......................................................................................................................... 15

2 BACKGROUND ............................................................................................................................... 16
  2.1 Networking ............................................................................................................................... 16
    2.1.1 The TCP/IP protocol suite ............................................................................................... 16
      2.1.1.1 Network layer ............................................................................................................ 17
      2.1.1.2 Transport layer .......................................................................................................... 17
    2.1.2 Proxy .................................................................................................................................. 18
    2.1.3 Tunneling ........................................................................................................................... 18
  2.2 Security ..................................................................................................................................... 18
    2.2.1 Cryptography ..................................................................................................................... 19
      2.2.1.1 Symmetric key cryptography ...................................................................................... 19
      2.2.1.2 Public key cryptography ............................................................................................ 20
    2.2.2 Network security protocols ............................................................................................... 20
      2.2.2.1 IPsec .......................................................................................................................... 20
      2.2.2.2 SSL protocol .............................................................................................................. 21
      2.2.2.3 SOCKS protocol ......................................................................................................... 21
    2.2.3 Firewalls ............................................................................................................................ 22
    2.2.4 VPN .................................................................................................................................... 22
  2.3 Software .................................................................................................................................... 22
    2.3.1 Java ..................................................................................................................................... 22
    2.3.2 Java SOCKS server ............................................................................................................ 23

3 PROBLEM AND SOLUTION ANALYSIS ................................................................................. 24
  3.1 The environment ....................................................................................................................... 24
  3.2 Problems and solution ............................................................................................................... 26
  3.3 Tools ......................................................................................................................................... 28

4 ARCHITECTURE, COMPONENTS AND THEIR RELATIONS .................................................. 29
  4.1 The big picture .......................................................................................................................... 29
4.2 Components and relations ........................................................................................................31
  4.2.1 The broker module ...........................................................................................................31
  4.2.2 The gateway module .........................................................................................................31
  4.2.3 The user module ...............................................................................................................32

5 TEST AND IMPLEMENTATION .................................................................................................33
  5.1 Class overview .....................................................................................................................33
  5.2 The source code and its improvement ....................................................................................35
    5.2.1 The SOCKS server .........................................................................................................35
    5.2.2 The SOCKS client ...........................................................................................................36
  5.3 The broker ............................................................................................................................37
    5.3.1 The register .....................................................................................................................37
    5.3.2 The SOCKS server at the external interface ......................................................................38
      5.3.2.1 The SOCKS client at the external interface ...............................................................39
    5.3.3 The SOCKS server at the internal interface ....................................................................40
  5.4 The gateway ..........................................................................................................................40
    5.4.1 The SOCKS client at the gateway ....................................................................................40
      5.4.1.1 The SOCKS server at gateway ..................................................................................41
  5.5 The user module ...................................................................................................................42
    5.5.1 The SOCKS client at the user ..........................................................................................42
  5.6 The test ...................................................................................................................................43
    5.6.1 Proving a correct authentication process .........................................................................43
    5.6.2 Testing the prototype’s communication process .............................................................45
    5.6.3 The first communication step .........................................................................................46
    5.6.4 The second communication step .....................................................................................46
    5.6.5 The third communication step and final connection .......................................................47

6 THE DISCUSSION ....................................................................................................................50

7 BIBLIOGRAPHY ......................................................................................................................51

APPENDIX A .............................................................................................................................53
Figures

FIGURE 1: THE ENVIRONMENT .................................................................................. 14

FIGURE 2: SOCKS CONNECTION (EXAMINING THE GENERIC SOCKS PROXY PROTOCOL, JULY 2005) ................................................................. 21

FIGURE 3: THE MODULES ......................................................................................... 24

FIGURE 4: THE PROCESS ............................................................................................ 25

FIGURE 5: THE ARCHITECTURE ............................................................................... 30

FIGURE 6: THE CLASS DIAGRAM ............................................................................. 34

FIGURE 7: THE USER INTERFACE ........................................................................... 44

FIGURE 8: THE FIRST SOCKS DATAGRAM ................................................................. 44

FIGURE 9: THE SECOND SOCKS DATAGRAM ............................................................. 44

FIGURE 10: THE THIRD SOCKS DATAGRAM ............................................................. 45

FIGURE 11: THE FOURTH SOCKS DATAGRAM ......................................................... 45

FIGURE 12: THE “CONNECT” REQUEST ................................................................. 47

FIGURE 13: THE SUCCEED TARGET CONNECTION REQUEST .............................. 47

FIGURE 14: THE USER’S “HELLO GATEWAY” MESSAGE ...................................... 48

FIGURE 15: ANONYMITY ......................................................................................... 48

FIGURE 16: TRANSPARENT CONNECTION ............................................................. 49

FIGURE 17: SSL PROTOCOL STEPS ...................................................................... 53

FIGURE 18: THE CLASSES INVOLVED IN THE SSL IMPLEMENTATION PROCESS .............................................................................................................. 55
1 Introduction

1.1 Background

Internet is growing, no doubt about it. Communication between computers is also growing in form of email, instant message, remote control and so on. Companies use also this kind of services and delicate data is exchanged. Today, companies don't have just one Local Area Network (LAN), they have different LANs at different locations in the world and they need to communicate with each other in a secure way. For this, the existence of a secure channel is a priority factor when growing the computer networks of the companies all over the world. These secure channels between LANs creates a kind of global private network which physically doesn't exist but acts like one, that is a Virtual Private Network (VPN).

1.2 Prerequisite

The company SYSteam wants to develop a new VPN product. To make this happen, the product must be developed in several steps like designing socket communication, implementing security aspects, implementing transparency for users and so on. There are many VPN solutions in the market but the company wants a more personalized and cheaper version in which applications, like instant messaging, can run over it.

The company wants to develop in the very first place a prototype that represents the socket communication between computers in order to create final VPN product. This socket communication prototype has not only to succeed but also to achieve certain security aspects. The intention is that this version has to be applicable in a certain scenario where restrictions or adversities are present. The scenario, presented by the company, consists in two private networks located physically at different parts of the world and these two have not been connected before. One of these LANs is very restricted for incoming calls and the company wants that the prototype achieves secure socket communication without affecting the configuration of these restrictions.

The company recommends that this communication uses an authentication protocol and technology called Sock-et-s (SOCKS) [1]. In order to implement encrypted communication through Internet, the company wants to implement the Secure Socket Layer (SSL) in combination with Hypertext Transfer Protocol (HTTP). This combination is called Hypertext Transfer Protocol Secure (HTTPS) and SOCKS communication will run encrypted within it.
In the final VPN product, an internal node at LAN A (see Figure 1) will have access to a database stored at the A's bounding point. This data base has information about the very restricted LAN B and gives the possibility to the internal node of A to know which node is reachable at B and therefore be able to connect to it.

1.3 Purpose

This thesis describes the creation of a socket communication prototype which is composed by clients and servers, these ones authenticate each other using SOCKS protocol. This will create an open secure channel which crosses, not only the Internet but all connections involved in the scenario. The prototype will be capable of sending, in a secure way, socket messages through this channel without affecting the previous security restrictions, proving that a basic socket architecture of the desired final VPN product is reached. This channel will be used by many possible applications in the future because SOCKS is a generic protocol, meaning that several application protocols can run over it.

A great part of the thesis is concentrated in the implementation of SOCKS communication, security aspects like authentication and the anonymity of internal nodes are implemented at the prototype. Other security aspects like encryption and integrity of data are achieved by implementing the SSL protocol. The whole prototype is written in Java for commercial and interoperability reasons.

The theoretical background needed to understand the project is reviewed in chapter 2. Chapter 3 shows the design of the prototype, this chapter studies the commented restrictions and propose a concrete solution. The architecture, components and relations between these components are explained in chapter 4 and the implementation of the prototype and the results in chapter 5. Chapter 6 is devoted to discussions and, finally, the SSL protocol process implementation is presented in the appendix A.
1.4 Limitations

The entire practical implementation is done with the Linux machine loop back interface so there are no public IP addresses. The final test in a virtual environment at the company couldn’t be done because of time limitations, therefore, functionality at the network layer is limited.

The implementation of the SSL protocol was not done because of time limitations but the process to implement it in the future is reviewed in this thesis.

The code developed is not entirely shown for one reason. The entire project is not composed only by self made modules but also by many open source codes that have been edited so, in order to safe space, only the most relevant parts of the code will be reviewed.

1.5 Abbreviations

AES Advanced Encryption Standard
API Application Programming Interface
CA Certification Authorities
GNU = GNU is Not Unix (recursive)
FTP File Transfer Protocol
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure
IP Internet Protocol
JSSE Java Secure Socket Extension
LAN Local Area Network
RFC Request for comments
RSA Rivest Shamir Adleman (inventors of the RSA algorithm)
SHA Secure Hash Algorithm
SMTP Simple Mail Transfer Protocol
SOCKS Sock-et-s internal development name
SSH Secure Shell
SSL Secure Socket Layer
TCP Transmission Control Protocol
TELNET Teletype Network
VPN Virtual Private Network
2 Background

This chapter describes the background of this thesis. It goes through network basic concepts, network security and a short review of the programming language used in this thesis.

2.1 Networking

2.1.1 The TCP/IP protocol suite

The TCP/IP protocol suite is the set of communication protocols that implement the protocol stack on which the Internet and most commercial networks run. It is named from two of the protocols involved in the suite: the Transmission Control Protocol (TCP) and the Internet Protocol (IP).

The TCP/IP Protocol suite can be viewed as a set of layers. Each layer refers to a group of protocols that solves a set of problems involving the transmission of data, and provides a well defined service to the upper layer protocols based on using services from some lower layers. Upper layers are logically closer to the user, relying on lower layer protocols to translate data into forms that can eventually be physically transmitted. The TCP/IP reference model consists of four layers. From lowest to highest, these are the link layer, the network layer, the transport layer, and the application layer.

The link layer is the lowest layer in the TCP/IP suite and refers to the group of protocols that operate at the physical and logical network components used to interconnect machines. The network layer is the second layer from the bottom in the TCP/IP suite and refers to the group of protocols that are responsible of transferring data packets from a source to a destination by using one or more networks. The IP protocol works at the network layer. The transport layer is the third layer from the bottom and this one groups protocols that are responsible for maintaining reliable end to end communication across the network and control the reliability of a given link through flow control, segmentation/desegmentation and error control. The TCP protocol works at the transport layer. The application layer is the top layer of the TCP/IP suite and refers to the group of protocols that falls into the realm of (remote located) process to process communication.

The TCP/IP model is often compared with the seven layer Open Systems Interconnection (OSI) reference model. This one was designed after the TCP/IP model meaning that the OSI model is nearly the same but with a important difference, that is, three additional layers. These layers are: the presentation layer, the session layer and the physical layer. The presentation and the session layers reside between the application and the transport layer, and the physical layer reside at the bottom, right under the link layer. The session layer controls the connections (sessions) between computers. It establishes, manages and terminates the connections between the local and remote application.
How two machines communicate with each other using the TCP/IP protocol suite can be described with the following example. First, application programs working at the application layer, send messages or streams of data to the transport layer protocol. This protocol receive the data from the application, divide it into smaller pieces called packets, add a destination address, and then pass the packets along to the next protocol layer, the network layer. The network layer encloses the packet in to a datagram, puts in the datagram header and trailer, decides where to send the datagram, and passes the datagram on to the network interface layer composed by the link and physical layer. The datagram is send it from the network interface layer to the another machine and that this one is take it from the respective network interface layer. Then, the protocol at the network layer strips off the datagram and sends the packet up to the transport layer. The protocol in the transport layer strips off the packet and sends the data up to the application layer [2].

2.1.1.1 Network layer

The network layer sends datagrams from one network to another. The IP protocol defines the fields in the IP datagram and how routers act on these fields. All Internet components that have a network layer must run the IP protocol. Although the network layer contains both the IP protocol and other routing protocols, it is often simply referred to as the IP layer [3, page 51].

2.1.1.2 Transport layer

The transport layer provides the service of transporting application-layer messages between the client and server sides of an application. In the Internet there are two transport protocols, TCP and the User Datagram Protocol (UDP). The TCP protocol provides a connection oriented service to its applications. This service includes guaranteed delivery of application-layer messages to the destination and flow control. TCP also divides long messages into shorter segments and provides a congestion control mechanism [3, page 50-51]. With UDP applications can send messages without requiring a priori communications. UDP uses a simple transmission model without implicating handshaking dialogues for guaranteeing reliability.

The choice of one of these transport protocols depends on which application protocol is used. This thesis refers only to the TCP protocol.

2.1.1.3 Application layer

The application layer supports network applications. The application layer includes many protocols, including HTTP for supporting the Web, SMTP to support electronic mail, and FTP for file transfer. Application-layer protocols are implemented in software in the end systems [3, page 50].
2.1.2 Proxy

In computers networks, a proxy server is a computer system or an application program that acts as an intermediary for requests from clients seeking resources from other servers. A client connects to the proxy server, requesting some service, such as a file, web page, or other resources, available from a different server. The proxy server evaluates the request according to its filtering rules. For example, it may filter traffic by IP address or protocol. If the request is validated by the filter, the proxy provides the resource by connecting to the relevant server and requesting the service on behalf of the client. Proxies are used, among other things, to keep machines behind it anonymous and to provide users Internet usage reporting.

2.1.3 Tunneling

Tunneling can be defined as a way to transfer data between two similar networks over an intermediate network. Tunneling encloses one type of data packet into the packet of another protocol, the tunnelling protocol. Before the encapsulation takes place, the packets are usually encrypted so that the data is unreadable to anyone monitoring the network. These encapsulated packets travel through the Internet until they reach their intended destination. Once there, the packets are separated and returned to their original format [4, page 217].

2.2 Security

There are two kinds of security commented in this section, those are network security and information security.

The area of network security consists of the security measures made in a computer network infrastructure. These security measures can be interpreted as policies adopted by the network administrator to protect the network and the network accessible resources from unauthorized access. Network security starts from authenticating the user, commonly with a user name and a password. Once authenticated, a device enforces access policies such as what services are allowed to be accessed by the network users. The network administrator may also monitor the network and traffic for suspicious user behaviour.

Information security refers to protecting information and information systems from unauthorized access. The goal of information security is to protect the confidentiality, integrity and availability of information. Confidentiality ensures that information is accessible only to those authorized to have access. Confidentiality can be accomplished by requiring strong authentication for any access to data and by ensuring encryption of the data so it cannot be accessed during transmission. Integrity ensures that information is not altered by unauthorized persons in a way that is not detectable by authorized users and this is accomplished by implementing message integrity. Availability ensures that information has to be accessible when and where we need it.
**Authentication** is the process of verifying a user’s identity. Human communication solves the problem of identity by visual recognition but when entities exchange messages over a medium, where they cannot “see” the other party, authentication is not so simple. Authentication must be done on the basis of messages and data exchanged as part of an authentication protocol. Typically, an authentication protocol runs before the two communicating parties run another protocol [3, page 566].

Only the sender and the receiver should understand the contents of the message. Because intruders may intercept the message, this requires that the message be encrypted so that an intercepted message cannot be decrypted by an interceptor [3, page 566].

Even if the sender and receiver are able to authenticate each other, they also want to be sure that the content of the messages is not altered during transmission. Message integrity can be implemented by message digest algorithms [3, page 566]. A message digest algorithm takes a message and produces a “fingerprint” of the data known as a message digest. This message digest protects the original message, so if the message m is maliciously changed to m' then f(m) will not match f(m')[3, page 590].

**2.2.1 Cryptography**

Cryptographic techniques guarantee the privacy, authenticity and integrity of data so that an intruder can gain no information from the intercepted data. In cryptographic terminology, the original message is referred to as plain text and the encrypted result of that message is the cipher text. The process of retrieving the plain text from the cipher text is called decryption. Encryption and decryption generally require the use of a key (a string of numbers). The method of encryption takes as parameters the cipher text and the key and produces a cipher text. The method of decryption takes the cipher text and the key and this produces the original plain text. The modern field of cryptography can be divided in two several areas of study: Symmetric key cryptography and the asymmetric or public key cryptography.

**2.2.1.1 Symmetric key cryptography**

In symmetric key cryptography the same key is used in order to encrypt a message as to decrypt the message. The advantages of using symmetric key cryptography is that is the encryption process can be simple and that it doesn't demand a lot of resources. The drawback with symmetric key algorithms comes from finding a way to exchange this key over a public network.
2.2.1.2 Public key cryptography

Public key cryptography is based on using a public/private key pair, these ones are generated together. Data encrypted by one of these keys can be decrypted by the other, and vice versa. The private key is kept private and used only by the owner, while the public key can be given out freely without compromising any data.

Public key encryption can be used in order to reach authentication. In a digital world, one often wants to indicate the owner or creator of a document. Digital signatures are methods used to verify that a message came from a given party and that the contents of the message that has been sent has not been altered with in any way during its transport. How this is exactly reached is described by John Mairs 2000[4, page 321]. Signing data via complete encryption-decryption can be overkill, there is the alternative of signing a message digest algorithm of the message to consume less resources.

A certificate is used when two machines exchange the public key and is a digitally signed statement vouching for the identity and public key of an entity (person, company, etc.). Certificates can either be self-signed or issued by a Certification Authority (CA). Certification Authorities are entities that are trusted to issue valid certificates for other entities. The certificate generally includes items like: The name of the certificate, the name and attributes of the person that the public key belongs to, the actual public key of the person represented by the certificate authority and a digital signature of the certificate authority at the end of the certificate itself [4, page 334]. The Public key infrastructure is an infrastructure that enables users to securely and privately exchange data through the use of the key pair that is obtained and shared through a trusted authority, this solves the problem of how two people can exchange a public key over a public medium by using all elements commented above.

2.2.2 Network security protocols

This section describes the basic protocols that secure Internet applications and traffic. These protocols use the security techniques described in the last section.

2.2.2.1 IPsec

IPsec is short for P Security and is a suite of protocols designed to provide security at the Network layer. IPsec provides two choices of security service: Authentication Header (AH), which essentially allows authentication, and Encapsulating Security Payload (ESP), which supports both authentication and encryption of data as well. The specific information associated with each of these services is inserted into the packet in a header that follows the IP packet header. A big advantage of IPsec is that security arrangements can be handled without requiring changes to the intermediate computers.
2.2.2.2 SSL protocol

Secure Sockets Layer (SSL) is a protocol that provides data encryption and authentication between a client and a server over the transport layer. This protocol uses public key cryptography to exchange keys, and then uses symmetric encryption in order to exchange data. The protocol starts with a handshake process that negotiates an encryption algorithm and keys, and authenticates the server to the client. There is also the option of authenticate the client to the server. Once the handshake is complete and the transmission begins, all data is encrypted using the negotiated session keys a priori. SSL is widely used in Internet commerce [3, page 609].

2.2.2.3 SOCKS protocol

SOCKS protocol is a session layer protocol, i.e. it lies between the transport and the application layer. In the last years it became normal to provide control at the application layer, this is done by the firewall. The problem is that the firewall must have a proxy for each application: HTTP, the File Transfer Protocol (FTP), the Simple Mail Transfer Protocol (SMTP), and so on. The SOCKS protocol is a generic proxy, accepting connections for any application protocols in port 1080.

The SOCKS process works in such way that a client makes a request that includes the application server address and the user's identification to the SOCKS in order to communicate with the application server. Once this is done, a circuit is set up between the client and the server, and the information travels transparently between them [4, page 194].

![Figure 2: SOCKS connection (Examining the generic SOCKS proxy protocol, July 2005)](image-url)
The client SOCKS proxy starts by opening a TCP connection to a server SOCKS proxy; the connection uses port 1080 by default. The client sends a negotiation packet suggesting a few authentication methods. If the server accepts, it uses a server negotiation packet to tell the client its preferred authentication method. The server then proceeds to authenticate the client. The client sends a possible connect request to the server stating which service it requires, with target IP and port. The server reply to a connect request contains the bind port and bind address, that is the address at which the server has connected the target server. The bind address is typically not identical to the SOCKS server address, to which the client sent the original request. After the successful connect command, the client and the target server can communicate transparently through the SOCKS server; this one simply forwards any data [5].

### 2.2.3 Firewalls

A firewall is essentially a software or hardware device that examines and filters information that comes out or comes in protected networks. It can protect you from outside intrusion over the Internet while permitting employees to access Internet services. Firewalls examine every packet, check its IP headers and its higher level protocols headers, in order to figure out whether it is a TCP or UDP. The firewall also uses, in normal cases, a list of rules in order to decide whether or not to let the packet through.

### 2.2.4 VPN

A Virtual Private Network is a virtual network that is private by “tunneling” private data through Internet. Tunneling allows two sites to connect in a secure way by using the infrastructure of the Internet transparently as what appears to be their own private network [4, page 208-209]. Organizations which decide to use VPNs as their means of secure communication should choose between the more commonly used IPsec and SSL secure protocols to encrypt and authenticate the data.

### 2.3 Software

This section describes the programming language used in this thesis and also describes the open source which has been found in order to avoid starting from scratch.

#### 2.3.1 Java

Java is general-purpose, concurrent, class-based, and object-oriented programming language and is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere".
Java Platform, Standard Edition or Java SE is a widely used platform for programming in the Java language. The Java Platform acts as a buffer between a running Java program and the underlying hardware and operating system. Java programs are compiled to run on a Java Virtual Machine, with the assumption that the class files of the Java API will be available at runtime. The virtual machine runs the program; the API gives the program access to the underlying computer resources. No matter where a Java program goes, it need only interact with the Java Platform. It doesn't need to worry about the underlying hardware and operating system. As a result, it can run on any computer that hosts a Java Platform. In practical terms, Java SE consists of a virtual machine which must be used to run Java programs, together with a set of libraries (or "packages") needed to allow the use of file systems, networks, graphical interfaces, and so on, from within those programs. Java SE was known as Java 2 Platform, Standard Edition or J2SE from version 1.2 until version 1.5.

The entire application programming interface of Java SE can be found at [6] and is the one used in this thesis.

### 2.3.2 Java SOCKS server

It is a SOCKS server written entirely in Java, which supports SOCKS protocol. This software is distributed under GNU Lesser General Public License, meaning that both binary and source code are freely available and can be modified and distributed [7]. This code is very basic and has a limited functionality but this one provides a good skeleton to the code developed in this thesis.
3 Problem and solution analysis

This chapter describes the environment and the entities involved in the communication of such environment. Then, a description of the expected functionality with its complications and solutions.

3.1 The environment

The company’s proposed scenario consists of two private networks located physically at different parts of the world. These networks, A and B, are composed of a number of internal nodes and bounding points. The machines involved in the scenario are: the internal node at A, the broker (or bounding point at A) and the gateway (or bounding point at B). The rest of the nodes at LAN B became spectators because the prototype developed had as priority the communication access to LAN B.

These three machines will be equipped with the software modules developed in this thesis. In the absence of neither a real nor a virtual environment, these modules are developed and tested within a Linux machine. The broker module is the bounding point at A, meaning that this module is able to keep with many different dialogs, at the same time, pointing to different directions. The broker module also has a register, which stores a connection to the gateway module. The gateway module will run in a machine equipped with a firewall that doesn't accept calls at port 1080, that is the SOCKS port. The broker module, in the other hand, has the SOCKS port open at both sides of the broker.

Figure 3: The modules
As pointed out above, the firewall at the gateway doesn’t accept incoming SOCKS calls. This will force the change of the standard SOCKS connection process (chapter 2, section 2.2.2.3) where the user starts the whole process by connecting first to the SOCKS proxy. In this prototype, it is the final target point (the gateway) which starts the SOCKS connection process and thereafter leaves a session open, this in order to give the broker the possibility of passing through the firewall at B.

![Diagram](image)

*Figure 4: The process*

The internal node module at A will have the possibility to access the broker’s register and then connect, in a secure way, to the gateway through the broker module. Several security aspects are taken into account: the authentication of the users that use the developed modules, the anonymity (at transport level) of the internal node at A, and the confidentiality and the integrity of the data exchanged between bounding points.

In order to authenticate all users involved in this communication, the SOCKS protocol is implemented. To guarantee the confidentiality of the data exchanged, a secure channel must also be implemented.
3.2 Problems and solution

The SOCKS protocol is used to proportionate a general proxy in which users are authenticated before connecting to the final destination. Some issues have been identified before the implementation and some solutions are proposed:

• Dealing with the firewall

As pointed out in section 3.1, the port 1080 at B's firewall is closed and that's why it is the gateway which connects first to the broker. But the gateway must also leave the session open in order to pass the restricted area. To implement SOCKS protocol, the Java “socks” and “socks.server” packages are used [7]. The functions of these packages represent the client and the server side functionality of the SOCKS protocol and they are used inside the code of the developed modules.

A SOCKS client must be situated at the gateway. This client will authenticate itself to the broker and will leave open the socket used for this first communication. The body of the function at the client side must be changed in order to be able to leave this socket open. This gives access to LAN B because it is the gateway which has started the connection and because it leaves a certain port open, giving the broker permission of passing the restricted area. The client at the gateway doesn't start the communication by the petition of someone at A. This client can start the communication right at the beginning (when the code at the gateway's module starts to run) or at certain time. This is something excluded from this thesis and belongs to the company's further implementations.

• Phase separation

As described in RFC 1928 [1], there is an authentication process between the client and the server side, then, the client sends a remote connection request to the server. As pointed out, it is the gateway which starts the connection process, therefore the gateway must leave a session open without making a final request. The broker acts as a proxy for the user at A but not for the gateway at B and that's why there is a need of separating the authentication process from the final request.

Once again, the solution is to edit the functions in the “socks” and “socks.server” packages. In this manner, the authentication process can be separated from the final client request. This can be done by finding the part of the code where the authentication process occurs and, if the function has been called from one of the bounding points, force the function to exit before the connection request. This doesn't change the SOCKS protocol specification or change the predefined packets of the SOCKS protocol, it only separates the entire process in two phases.
• Storing information

The broker must know information about the gateway. This information equals to the socket used for communication and its values, i.e., ending points IP addresses and ports. This, in order to offer this information to the internal node at A which asks the broker about whose socket is out there. The server side, situated at the broker, must leave this socket open and then store a reference to this socket in to the register.

One way of doing this is by editing the body of the corresponding function at server side. The SOCKS server at the broker module will pass a new parameter into the function, this parameter is the register object. The new body definition will leave the socket (used for communication between bounding points) open and will store it in to the register. Thereafter, the socket with its entire information is offered to the internal node at A.

• The channel

The prototype has to provide encryption and the integrity of the data exchanged between the broker and the gateway.

This is reached by implementing the SSL protocol between end points of the channel. The carrier protocol which contains the SOCKS communication is the HTTP and this one is encapsulated by SSL. This channel proportionates the confidence of being invisible to anyone in the Internet and, therefore, protects the SOCKS socket communication.

• Functionality

How can the developed prototype demonstrate a correct functionality?

The user at A will send a “HELLO GATEWAY” message which will travel through the broker to the gateway. If this prototype delivers the message (passed through all identified peers) to the gateway, provide anonymity (at transport level) to the internal node, and provide encryption and the integrity of the data, we can then prove the achievement of the prototype wished functionality. In order to demonstrate the expected results, a network analyzer software, called Wireshark (version 1.0.7.)(8], is used.

Further complications may exist but these are not decisive for the desired functionality. The problems commented above have been the most discussed at the company, where this project has been implemented, because these aspects form the socket strategy.
3.3 Tools

The operating system used in this thesis is the Kubuntu-9.04 Linux Version. As pointed out before, the programming language in use is Java, and the Java development kit version 6 is a development environment for building applications using Java [9]. To compile, administrate and run the code, Eclipse Platform v 3.5.0 [10] is in use.

The desired code has to be implemented with help of SOCKS package and its libraries (SOCKS & SOCKS.server), the rest of Java TM 2 Platform Standard Edition and the totality of the API Specification is also in use.

The code found, in order to avoid to start from scratch, in [7] is very simple. This code has a SOCKS server and a SOCKS client, they only implement a very basic connection and don't include a proper authentication method. The authentication of SOCKS server service is based in IP addresses and host names, instead of supplying a proper user/password authentication method.
4 Architecture, components and their relations

This chapter describes the design of the communication prototype. It shows the modules, the components that will compose the system and how they communicate with each other.

4.1 The big picture

The project is composed of three modules: the broker, the gateway and the internal node at LAN A. These three modules will implement the SOCKS servers and clients that communicate and authenticate to each other.

The broker module manages the connections and acts as a proxy between the user and the gateway. It has two SOCKS servers (see the Figure 5), one which points to the Internet and one inside the broker's LAN which handles outgoing connections to the Internet. The broker module has also a register object. This object is used to store an existing socket connection between the broker and a target destination (the gateway in this case). The content of the register will be available to the identified internal node who wants to communicate with the gateway.

The gateway module has a SOCKS client and has the task of starting the connection to the broker, therefore, the gateway will open a door through the gateway's firewall and will accept everything sent by the broker.

The internal node module has only a SOCKS client, this one is in charge of identifying itself to the SOCKS server located at the broker.
Figure 5: The architecture
4.2 Components and relations

4.2.1 The broker module

The broker is possibly the most important module of this design. The broker handles three dialogs, stores and delivers the gateway's information and finally forwards any data sent from the user to the gateway.

The register is used to store an existing socket connection between the broker and the gateway. This register will be managed by both sides of the broker, the internal and the external side. The external side will store the socket in to the register while the internal side will fetch the socket from the register.

The sides of the broker are composed by two SOCKS servers, one server at each side. The first one has the job of identifying the gateway (step 1 in Figure 5). The server has this dialog at port 1080 while the gateway opens a random port number. When this authentication process is finished, the server leaves the socket open, stores a reference to this socket in to the register and passes the socket, as a parameter, to the SOCKS client situated at the external side of the broker (step 2 in Figure 5). The other SOCKS server, situated at the internal side, is in charge of identifying the internal node and will offer her/him the socket in order to give her the possibility of sending a message to the gateway through the broker, without implicating her identity (step 4 in Figure 5).

The SOCKS client, situated at the external side, is responsible of identifying the broker to the gateway. This client takes the socket parameter and uses it to identify itself to the gateway by using the same socket.

4.2.2 The gateway module

The gateway module is responsible of identifying itself to the broker. Thereafter, the gateway has also the task of identifying the broker. The gateway module has a SOCKS client, which starts (as commented) the connection process, and a SOCKS server.

The gateway uses a similar structure seen in the broker, it has two components that share the same socket. The gateway has a SOCKS client which starts a connection (step 1) and passes the socket to a SOCKS server, the one which authenticates the broker.
4.2.3 The user module

The SOCKS client at this module will start the authentication dialog with the broker (step 3). When the client is identified, the broker will offer the client a reference with the open socket used by the broker and the gateway. The final connection is made at the client side by including an IP address and a port, data previously offered by the broker to the client.

The user module is the only one which doesn't have a SOCKS server. It is only implemented with a SOCKS client and this one will send the named “hello” message after successful authentication.

Finally, the client sends the “hello message” which travels through the broker to the gateway without implicating the internal node's identity (step 4).
5 Test and implementation

This chapter goes through the basic open source code found for SOCKS (in order to avoid starting from zero) and the final implementation. This implementation will follow an iterative developing process. First, the implementation of a standard communication paradigm in which a SOCKS server authenticates a SOCKS client. Then, the creation of the modules and the different stages of communication between these modules. Finally, the execution of a test in order to prove the correct implementation.

5.1 Class overview

The class structure in this thesis is composed initially by three modules (see Figure 6): The broker, the gateway and the user. These modules are implemented by SOCKS servers and clients. An additional instance of an object, the register, is also implemented at the broker.

The original SOCKS code [7] was very simple. It did not have the expected functionality for this thesis and that's why it has been reused and modified. The SOCKS servers and clients developed will be implemented within the classes described at Figure 6.

In order to reduce the content, only the most relevant methods and variables are included in this picture. The classes of the diagram will be referred throughout the chapter.
Figure 6: The class diagram
5.2 The source code and its improvement

This section will show parts of the original code, found at [7], and will explain the integration of the authentication functionality to the initial source code. The SOCKS package provides a number of classes and methods that can be used, for this purpose these classes are, for example, `usermethodauthentication` class, `Proxyserver` class, `SOCKSsocket` class and many more.

5.2.1 The SOCKS server

The part in the SOCKS server code which was responsible of the original and basic authentication process looked like this:

```java
ProxyServer server = new ProxyServer (auth);
server.setLog(log);
server.start(port, 5,localIP);
```

This piece of code uses admissible IP addresses which are permitted to connect to the `ProxyServer` instance without demanding user name and password. The `ProxyServer` class represents the SOCKS proxy object and its functionality, it has methods to: start the session, stop it, use UDP instead of TCP, managing timeouts and so on. The `auth` variable is an instance of a very simple identification class, but it doesn't implement the wished user and password authentication type. The function `server.start()` runs finally the server.

The new server's functionality is achieved by: (1) creating a new class called `SOCKS`, which represents a user, (2) declare an instance of this class and (3) add it as a parameter to the class `UserPasswordAuthenticator`, a class which implements the user and password process and is included later on to the `ProxyServer` constructor.

(1) `String user, password;
SOCKS (String user, String password) {
this.user = user;
this.password = password;
..`

(2) `SOCKS us = new SOCKS ("[user]","[password]");`

(3) `UserPasswordAuthenticator autho = new UserPasswordAuthenticator (us);
ProxyServer server = new ProxyServer (autho);
server.start(port, 5,localIP);`
This code creates a new user at server side. This is done by giving the user name and password string parameters to the new SOCKS server code, these parameters are checked when a SOCKS client identifies itself to this code. Thereafter, a constructor of a UserPasswordAuthenticator class is created by including the us parameter, which represents the user added to this server. The UserPasswordAuthenticator class implements the server side of the user password authentication scheme as defined in RFC 1928.

When the client starts the connection it sends the desired authentication method to the server. The server responds with an affirmative datagram saying that supports the method. The client receives the datagram and sends the corresponding user and password fields to the server. Then, the SOCKS server verifies these fields, if they are correct the server continues with the dialog otherwise drops the connection by saying “authentication fails”. The server will continue with the remote target connection if the authentication is succeed, giving his own IP address and hiding the original client IP address.

5.2.2 The SOCKS client

The code for the client wasn't made from scratch either. The following code is used as start point of the client implementation and works together with the original SOCKS server. It's important to see that, as in the server side, an instance of a proxy object and it's corresponding initialization has to be done. The socket subclass SOCKSsocket class is very similar to a normal socket, but this one allows connections through the SOCKS proxy.

    Proxy Proxy;
    Proxy.setDefaultProxy (proxyHost, proxyPort);
    ..
    SS = new SOCKSSocket (host, port);

This code tries to connect to a given target IP address and port using a proxy instance. This code doesn't implement a proper authentication method and is very limited, like the corresponding and original SOCKS server code.

The new code implemented at the client side has to communicate perfectly with the SOCKS server implemented at section 5.2.1. All steps described in RFC 1928, like authentication process negotiation, sending user data and the final connect request have to match exactly as described in the RFC.

This new client code has been implemented with a new proxy instance. This proxy is a subclass of the original proxy class and implements robust authentication mechanisms and extends support to UDP.
/* initiate proxy with proxy address + proxy port */

Proxy = new SOCKS5Proxy (defaultProxyHost, defaultProxyPort):

/* creation of SOCKS dialog which does run a 
* SetAuthenticationMethod */

SOCKSDialog dialog = new SOCKSDialog (frame, proxy);
Proxy temp = dialog.getProxy (proxy);
Ss = new SOCKSSocket (temp, Host, Port);

The proxy object is included in a new object, the SOCKSdialog. This instance is a subclass of the Dialog class and belongs to the package java.awt which contains all the classes for creating user interfaces and for printing graphics and images. The Dialog class is a toplevel window with a title and a border that is typically used to take some form of input from the user. An instance of the SOCKSDialog class is created and initialized with a frame and the proxy parameters, this creates a pop up window which takes as input the user name and password. Later on, the dialog.getProxy() method initiates the dialog and displays the SOCKS window. Finally, the SOCSKsocket constructor takes as parameters: the proxy, the target address and port. The target parameters correspond to the desired final location which the proxy connects to. When this client runs, it asks about a name and a password, this data is inserted and sent through the socket and it must equals to that data included at the server.

5.3 The broker

This section describes the broker module which is a class which contains three objects: the register and two servers. Recall that the broker module manages the connections and acts as a proxy between the user and the gateway.

5.3.1 The register

The register is the data object which will store the socket in use between the broker and the gateway. The Socket class includes, among others, the methods socket.getaddress() and socket.getport(), meaning that the broker can call these methods from the socket stored in the register and give these return values to the authenticated user in order to inform her or him who's socket is out there.
The register has only two public methods implemented. These methods are: the `add_socket()` method, which takes a socket as an argument, and the `read_socket()` method which returns the socket.

An instance of this register will be given to the SOCKS servers at the broker and later on moved between functions as a parameter. This in order to be able to send the register inside the methods of the SOCKS package, where the creation and management of the socket resides.

### 5.3.2 The SOCKS server at the external interface

The broker has a SOCKS server which points out to the Internet. This server is represented by the `SOCKSserver_ex_b` class and it's in charge of listening and validating SOCKS clients.

The register is sent through the `ProxyServer` class `start()` function, this in order to manage the register inside the function definition. This `ProxyServer` class belongs to SOCKS package and must be edited in order to: use the register right after successful authentication and store the socket used for communication inside the register.

The following code, inside the `start()` function definition, shows how a session is authenticated by the function `startSession()`, that is, a `ProxyServer` class private method. This private method has been edited in order to control if a register has been included as a parameter and stores the socket if the register is not null. Then, the code returns to the last method invocation with the socket stored in the register.

```java
Try {
  ...
  Auth = auth.startSession (socket);
  .
  .
  If (register != null)
  {
    register.add_address (socket);
  }
  return;
}
```
5.3.2.1 The SOCKS client at the external interface

The broker also identifies itself to the gateway, this is done by implementing a SOCKS client at the external side of the broker.

An instance of this client, which is represented by the `SOCKSclient_mini` class, is created when the `start()` function, at the `SOCKSserver_ex_b`, returns with the register. The server at the external interface authenticates the client at the gateway, stores the socket in the register and passes this socket to the SOCKS client.

```java
SOCKSclient_mini miniclient = new SOCKSclient_temp (server.get_ex_socket());
```

This constructor takes the socket as a parameter, in such manner the created SOCKS client can communicate, through the same socket, with the server located at the gateway. The following code shows how the client creates and initializes a proxy with the socket values. The last row indicates that a dialog will start between the client and the server through the socket used a priori. The `SOCKSSocket` constructor takes three parameters: the configured proxy, the target node IP address and port. The last two parameters are null in this case because this client doesn't make a final target request.

```java
Proxy = new SOCKS5Proxy (socket.address, socket.port);
SOCKS5Proxy temp = (SOCKS5Proxy) dialog.getProxy (Proxy);
Ss = new SOCKSSocket (temp, "", 0);
```

The class `SOCKSSocket` has been edited in order to separate the final request from the authentication process. The following code has been added in to the body of the `SOCKSSocket` constructor, this code checks if the remote parameters, the target IP address and port, are included. If they aren't included, this code returns right after the authentication process, otherwise it continues with the remote connection. If these parameters are included means that this constructor is called from some other place, that is from the user module, because it is the only one which can make a final connection request in this thesis.
If ((host && port) == null)
{
    proxy.startSession();
    return;
}
Else
{
    do a regular connection
}

5.3.3 The SOCKS server at the internal interface

This SOCKS server is represented by the class `SOCKSserver_in_b` and has the function of authenticate the SOCKS client located at the user module. In this case, the `start()` function takes an additional parameter, that is, the socket chosen by the authenticated user.

```
server.start(proxyport,5,localIP, null, this.b_register.read_socket());
```

The socket parameter is sent to the `ProxyServer` class `start()` method definition. Inside the function body, this socket is used right after success authentication and, instead of returning (like a priori cases), the code continues with the remote client request and uses the socket to make that request. The part of the code, inside the `start()` function definition, which handles the connection between the broker and the target node has been edited in order to use the socket stored in the register, instead of creating a new socket instance like in the original method definition.

5.4 The gateway

The following section describes the gateway module which is composed by a SOCKS client and a SOCKS server. The gateway has the task of starting the connection to the broker, therefore, the gateway will open a route through the gateway's firewall and will accept everything sent by the broker.

5.4.1 The SOCKS client at the gateway

This client is represented by the `SOCKSclient_g` class, this one runs first and starts the connection by calling the broker (the SOCKS server located at the external interface of the broker). The parameters are the broker's IP address and port 1080.
Proxy = new SOCKS5Proxy ([broker address], 1080);

SOCKS5Proxy temp = (SOCKS5Proxy) dialog.getProxy (Proxy);

SS = new SOCKSSocket (temp,"", 0);

5.4.1.1 The SOCKS server at gateway

This server is represented by the SOCKSserver_mini class, an instance of this server is created within the SOCKSclient_g class commented above. The constructor takes as parameters: the authorized user, her password and the proxy socket used before. The following code correspond to a continuation of the code commented above.

SS = new SOCKSSocket (temp,"", 0).getProxySocket ();
SOCKSserver_mini miniserver = new SOCKSserver_mini (user, password, SS);
miniserver.run ();

Public void run ()
{
    ..
    ..
    server.start (proxypotr, 5, localIP, null, SS);
}

The server is initiated by invoking the start() method. This method is called passing the parameters: the local proxy host, the local port and the socket passed by the SOCKSclient_g. Recall that this function behaves differently from site to site, meaning that the definition of this function at the SOCKS package has been edited in order to accept different parameters and that the functionality of this function depends of the combination of these parameters.

When this SOCKS server runs, it listens to the respective SOCKS client located at the external interface of the broker. After identifying the client it waits for the client to talk.
5.5 The user module

This section describes the user module. This module connects to the broker in order to get the proxy service, thereafter it sends the “hello message” which travels trough the broker to the gateway.

5.5.1 The SOCKS client at the user

This part of the architecture has not embedded objects inside others, this part equals to a regular negotiation and authentication between the user client and the broker.

In previous implementations of SOCKS clients, the SOCKSSocket constructor didn't have more than the proxy parameter, this is because the previous modules were not implemented to make a connection request to a target node. In this case, the user needs not only the proxy parameter but also the remote parameters.

The following code shows the difference between constructors:

```java
SS = new SOCKSSocket (temp,"127.0.1.1", 53163);
```

In contrast with a priori calls.

```java
SS = new SOCKSSocket (temp,"", 0);
```

Notice that the parameters included this time correspond to the remote values of the socket stored in the register, those are the IP address and open port of the gateway. These parameters were expected to be sent by the broker right after a succeed authentication but this functionality has been offered in order to reach minimal functionality. This info is collected by running an output stream at the broker and included them manually by the user.
5.6 The test

This section describes the functional test and the results of the first attempt to a SOCKS client and server authentication process and the four steps of communication, between SOCKS clients and servers, described in chapter 4.

5.6.1 Proving a correct authentication process

The first approach is to achieve a correct authentication process in which a SOCKS client and a SOCKS server follow the steps described in RFC 1928. This must be tested first in order to continue later on with the implementation of all steps of communication. In this thesis the authentication method is user/password.

To be sure of the results, a network protocol analyzer was in use; this software is the Wireshark version 1.0.7. and it shows the content of the TCP/IP datagrams.

The IP addresses used in this test are local host addresses. As commented in previous chapters, there is a limitation when using the IP layer with the loop back interface meaning that only the transport and session layers are reviewed during the tests.

The client starts by opening a TCP connection to the server (port 1080). Then, the client sends a negotiation packet suggesting an authentication method, that is, the number which identifies the method which in the current case is 02. If the server accepts the request, it uses a Server negotiation packet to tell the client that the method is supported or not. If the method is supported, the client sends a packet with the user name and password. The server receives this packet, checks if the arguments are valid and sends (if succeed) to the client a packet with a succeed message. The SOCKS server is then ready for a connect request from the client. The SOCKS server responds, like mentioned, if the negotiation and authentication are valid, otherwise drops the connection.

The test is made it by running the SOCKS server and the socks client edited so far. Right after the activation of both components, a proxy window appears demanding the user Id and password. There is also the possibility of choosing the version of SOCKS and changing the authentication protocol among other things.
When the OK button is activated, the following datagrams are exchanged between the SOCKS client and the SOCKS server. To differentiate layers, the transport layer is marked in blue while the session layer, where SOCKS runs, is marked in white.

**Figure 8: The first SOCKS datagram**

**Figure 9: The second SOCKS datagram**
This process behaves exactly as the process described in RFC 1928. Therefore, there is a correct functional authentication process between SOCKS components.

5.6.2 Testing the prototype's communication process

After correct authentication process functionality between a client and a server, it is time to prove that this process can be implemented to the states of the prototype's communication. The whole communication process is divided in 4 steps of communication. Each of these steps are reviewed in the following sections where the results of the communication, via Wireshark, are compared with the results expected for this thesis.

As commented in the previous section, only the transport and session layer are reviewed by the Wireshark but on the other hand, the code has been implemented with a log function that writes out messages in order to have an orientation of which module, with its corresponding local host IP address, is running. The IP addresses used in this test are: 127.0.1.1 (the gateway), 127.0.0.1 (the external server) and 127.0.2.1 (the internal server). The task of this test is to prove that communication works as expected by sending a “HELLO gateway” message through the prepared socket connection.
5.6.3 The first communication step

The first communication step is between the SOCKS client at the gateway and the SOCKS server at the external interface of the broker. After the activation of the broker and the gateway modules, we can observe that:

ExBroker server started, listening on: ServerSocket[addr=/127.0.0.1,port=0,localport=1080]
ExBroker, inbound connection received: Socket[addr=/127.0.0.1,port=53163,localport=1080]
[Client->ExBroker]: client authenticated.
[Client->ExBroker]: added client socket to register.
[Client->ExBroker]: thread stopped.

The broker starts to listen to the port 1080 and waits. When the client connects to the broker, a socket is created with two end points: the port 1080 at the broker and the port 53163 at the gateway. Later on, the authentication process succeeds and therefore the current socket is stored in the register.

5.6.4 The second communication step

Right after the first communication step and after storing the socket, a second authentication process occurs but this time differs from the first one. The SOCKS client located at the external interface of the broker connects to the gateway at port 53163 instead of 1080, this is because the client starts this connection through the same socket used in the first dialog. The following output shows what happens:

Started Gateway server, listening on: ServerSocket[addr=/127.0.1.1,port=0,localport=53163]
Gateway, inbound connection received: Socket[addr=/127.0.1.1,port=54580,localport=53163]
[Client->Gateway]: client authenticated.
[Client->Gateway]: thread stopped.
Gateway: waiting for incoming data.

This dialog occurs through the same socket but the bounding port used by the client differs from before. The first dialog had the ports 1080 and 53163 as ending ports. This time, the ports are: 54580 and 53163. This is because a server socket creates first a “welcome socket”, this one listens and waits for a call to the port 1080. When someone calls, it creates a new socket with the same remote port but with different local port, in order to dispatch other calls to the port 1080 (see page 145 of [3]).
5.6.5 The third communication step and final connection

The a priori steps achieves the availability of an existing open socket which is ready for it's use. The user, on the other hand, identifies it self to the internal interface of the broker and thereafter gets an output from the broker. This output correspond to Gateway’s IP address and port which are "127.0.1.1" and "53163", this is needed by the user in order to make a target connection to the gateway using the proxy service.

The figure 12 shows how the user sends through a socket, between the user and the broker, the information needed by the proxy service in order to make the target request.

```
InBroker, inbound connection received: Socket[addr=/127.0.2.1,port=58199,localport=1080]
[Client->InBroker]: client authenticated.
Request version: 5 Command: CONNECT
IP:/127.0.1.1 Port:53163
```

**Figure 12: The “connect” request**

The internal server evaluates the request and if the evaluation is affirmative, the broker communicates the affirmative answer to the user by sending a datagram which contains the bind port and bind address (look at Figure 13). This data correspond to the IP address and port at which the proxy has connected the target server, that is, the IP address and port of the external interface of the broker.

```
Transmission Control Protocol, Src Port: 58199 (58199), Dst Port: socks (1080)
Socks Protocol
Version: 5
Command: Connect (1)
Reserved: 0x0 (should = 0x00)
Address Type: 1 (IPV4)
Remote Address: 127.0.1.1 (127.0.1.1)
Port: 53163
```

**Figure 13: The succeed target connection request**
The datagram in figure 14, send it by the user client to the broker, contains the “HELLO gateway” message. This datagram also contains the target IP address and port of the gateway.

The final datagram (Figure 15) is sent it by the external server to the gateway. This datagram is sent through the already running socket which has been stored and managed by the broker and the gateway and contains the message “hello gateway” which has been written a priori by the user. This means that the IP address and port of the user are invisible for everybody out in the Internet meaning that the user keeps her anonymity.
This test show us (see Figure 16) that we achieve a VPN socket communication structure which: has a successful utilization of SOCKS authentication protocol, is applicable in a certain scenario where existing restrictions reside and doesn't implicate the user's identity.
6 The Discussion

The priority in this thesis was the implementation of the SOCKS communication prototype in a certain time, that’s why there are some limitations. These limitations correspond to functional aspects which weren’t implicated directly in the SOCKS authentication process and, therefore, they can be seen as unfinished parts in the final product. However, they did not affect reaching the goal of the thesis which was the authentication functionality. The next paragraphs go through those limitations, they are commented and covered by giving a possible solution.

The first limitation is located at the register, managed by the broker. In this thesis, the SOCKS server at the external side of the broker uses first the register by loading the socket used for communication. Thereafter, the internal SOCKS server reads the socket. This is reached by simply having two functions, the store function and the read function, which is contemplated as a temporal solution. The register functionality should be extended in order to cover future requirements, e.g. dealing with more target LANs. This register should be implemented with a vector which stores many other possible sockets representing different target nodes. The vector should have certain functionality like: a search function which searches through the vector in order to find the desired socket and, also, the implementation of methods which guarantee the atomicity of execution for all those who want to use the vector, providing consistency of data.

Another remaining part is integration with the application layer. Currently, the proper delivery of the gateway’s data to the user is not implemented. The broker doesn’t send the content of the register to the user, instead, the broker prints out the content of the register. In such manner, the gateway’s IP and port are included manually in to the user module source code when creating the socket which communicates with the internal server of the broker. This was a fast solution but not the proper one. Instead, the broker should have sent, through the socket, the content of the register to the user. This is something that demanded further study and should be considered as future works.

Encapsulating the entire VPN solution with the SOCKS endpoints was not implemented due to time constraints, but the appendix A shows which classes are involved in the SSL protocol process and in which order these classes have to be implemented.

This prototype has been tested with loop back IP addresses and therefore correct functionality has only been proved at the transport level. In order to prove correct functionality at the IP layer, the source code should be placed in a machine which runs a virtual environment.

The goal of this thesis was to design, and thereafter to implement, a VPN communication prototype which runs over an existing infrastructure without affecting the existing security restrictions. Despite the limitations explained, the goal has been reached.
7 Bibliography


   <http://java.sun.com/j2se/1.5.0/docs/api/> Accessed: 20090501

[7] Java Socks Server project

[8] Wireshark software
   <http://www.wireshark.org/> Accessed: 20090520
[9] Open Java Development Kit
<http://openjdk.java.net/> Accessed: 20090501

[10] Eclipse programming platform

Accessed: 20090915
Appendix A

4 The SSL implementation

This chapter describes the SSL protocol and how to apply it to the project. The documentation used in this chapter is the Java Secure Socket Extension (JSSE) Reference Guide [11] for Java TM Platform Standard Edition 6.

4.1 The SSL

The complete SSL protocol description can be found in detail at the RFC 4346 (SSL 3.0). An overview of the steps involved between the client and the server (see the Figure 17) is also reviewed at the reference guide commented above.

Figure 17: SSL protocol steps
The SSL protocol combines the use of asymmetric cryptography with symmetric cryptography. In the figure above, the authentication of the server demands a third party, that is, a certification authority (CA). The CA provides the certificate of the server and this is sent to the client in order to authenticate the server. The same could be demanded by the server (to demand a certificate from the client) but this is something that is excluded from this project. The certificate, not only gives the client the conviction of the true identity of the server but also the server's public key. The public key is created together with a private key (the one kept by the server). This pair of keys is needed by the RSA public key encryption algorithm to cipher the information (with the public key) and decipher it (with the private key). Thereafter, the client generates information which is used to create a key for symmetric encryption and sends (encrypted with the server public key) this information to the server. The server decrypts (with its private key) this key information in order to use it later for symmetric encryption.

In this thesis, only the authentication of the server is taken in account, meaning that not all the steps above are required to implement the SSL protocol. The following steps are the needed ones for the application and those steps which are involved in the authentication of the client are excluded from the process.

1. **Client hello** - The client sends the server information including the highest version of SSL it supports and a list of the cipher suites it supports. The cipher suite information includes cryptographic algorithms and key sizes.

2. **Server hello** - The server chooses the highest version of SSL and the best cipher suite that both the client and server support and sends this information to the client.

3. **Server certificate** - The server sends the client a certificate or a certificate chain. A certificate chain typically begins with the server's public key certificate and ends with the certificate authority's root certificate. This message is optional, but is used whenever server authentication is required.

4. **Server hello done (step 6 at Figure 17)** - The server tells the client that it is finished with its initial negotiation messages.

5. **Client key exchange (step 8 at Figure 17)** - The client generates information used to create a key to use for symmetric encryption. For RSA, the client then encrypts this key information with the server's public key and sends it to the server.

6. **Change cipher spec (step 10 at Figure 17)** - The client sends a message telling the server to change to encrypted mode.

7. **Finished (step 11 at Figure 17)** - The client tells the server that it is ready for secure data communication to begin.

8. **Change cipher spec (step 12 at Figure 17)** - The server sends a message telling the client to change to encrypted mode.

9. **Finished (step 13 at Figure 17)** - The server tells the client that it is ready for secure data communication to begin. This is the end of the SSL handshake.
10. **Encrypted data (step 14 at Figure 17)** - The client and the server communicate using the symmetric encryption algorithm and the cryptographic hash function negotiated in Messages 1 and 2, and using the secret key that the client sent to the server in Message 5 (step 8 at Figure 17).

11. **Close Messages (step 15 at Figure 17)** - At the end of the connection, each side will send a `close_notify` message to inform the peer that the connection is closed.

The company wants the implementation of the SSL protocol by using a certain combination of algorithms. For this purpose it is suggested an algorithm for symmetric encryption, the AES-256, and a hash algorithm for the integrity of messages, the SHA-256. These security mechanisms are supported by SSL and can be implemented by using APIs like `java.security` and `javax.net.ssl`. The implementation of the SSL protocol can be done in many ways, in a high level (by calling default methods), in a very low level (in which the user can even develop her own cryptography algorithms), or by using the approach that gives a granulated control of the implementation without going deeper in to the cryptography methods.

**4.2 How to implement it**

This section goes through the process which integrates the SSL protocol into the project and gives the possibility of choosing a combination of certain cryptography methods, like the AES-256 and SHA-256. The cipher suite chosen is the “TLS_RSA_WITH_AES_256_CBC_SHA”.

---

![Diagram](Image.png)

*Figure 18: The classes involved in the SSL implementation process*
To communicate securely, both sides of the connection must be SSL-enabled. This is done by creating and initializing a number of classes described in the Figure 18, where the endpoint class of the connection is the “SSLSocket”. The JSSE enables secure Internet communications and provides a framework for a Java version of the SSL protocols, it includes functionality for data encryption, server authentication, message integrity, and optional client authentication.

4.2.1 The process

The “SSLSocket” class demands the creation and initialization of several objects. The purpose is to create an instance of the SSLSocket at both sides of the communication and initialise them with keys and certificates in the repository. The following steps are done by taking in account that only the server authentication is demanded.

1. **The key material.** First, the key material (keys and certificates) must be generated. In order to do so, a key and certificate management tool provide by Java is used. The keytool comes with a manual that can be found in the Linux machine by typing in the bash “man keytool”. In this extended manual, you can find examples about generating keys and requesting signed certificates to Certification Authorities.

   The creation of a file at the client machine is required. This file will store the certificate, and the corresponding public key, sent by the server to the client when server authentication is demanded (step 3 at Figure 17). The server machine also requires a file which stores the server’s private key and the certificate (with its corresponding public key). This file is used when the server uses its own private key in order to decrypt the secret key (for symmetric encryption) sent by the client (step 5 at Figure 17).

   The “KeyStore” class (this one isn’t at the Figure 18) is responsible of loading the key material created above. This is done by creating a “KeyStore” object, initialize it by giving the type of the key store, which in this case is the JKS (Java Key Store), and later on calling the “KeyStore.load” function with the files created above as parameters.

   The “KeyStore” instance is then used in order to initialize the objects “KeyManagerFactory” and “TrustManagerFactory”, these ones are used to create the corresponding “KeyManager” and “TrustManager” objects. The “KeyManager” manages the pair of keys (in the case of the server) while “TrustManager” manages the certificates (at the client side).

2. **The “SSLContext” class.** The SSLContext class acts as a factory for the SSL socket factories and associates key material and certificates (created above) with the SSL Sockets used for this communication. An instance of the “SSLContext” class must be created by calling the “SSLContext.getInstance” method, this function takes as an argument the type of protocol, the SSL for example. This context can be initialized by calling the instance's “init” method which takes three arguments: an array of “KeyManager” objects, an array of “TrustManager” objects, and a “SecureRandom”
random. From the server point of view, only the “KeyManager” object has to be included while from the client point of view, only the “TrustManager” object will be demanded.

After the initialization of the “SSLContext” object, it is time to create an “SSLServerSocketFactory” or “SSLSocketFactory”. These classes are created by calling methods of the “SSLContext” class in which the return value is either a “SSLServerSocketFactory” or “SSLSocketFactory”. Finally, a create socket function is used in order to create the respective “SSLServerSocket” and “SSLSocket” instances.

3. Implementing cryptography. The cryptography mechanism (the cipher suite) must be provided to the “SSLServerSocket/SSLSocket” object. These two classes include a method which adds supported cipher suites. The “setEnableCipherSuites(String)” method takes the desired combination of cryptography algorithms and implements these mechanisms with the keys already produced. In this thesis, the selected cipher suite is the “TLS_RSA_WITH_AES_256_CBC_SHA” combination. If both peers support a list with many cipher suites, then the most advanced suite (that resides at both parts) is selected.

Finally, the “SSLSession” can be initialized by calling the “SSLSocket.getSession” method. Having an initialized instance of a “SSLSession” and an initialized instance of the “SSLSocket” proportionates control over the “SSLSession” and the output and input streams of the “SSLSocket”.