Project report on Communication through TCP/IP using TLS over SSL

BY

Anupam Aggarwal

f2016033@pilani.bits-pilani.ac.in

Shubham Tiwari

f2016935@pilani.bits-pilani.ac.in

AT

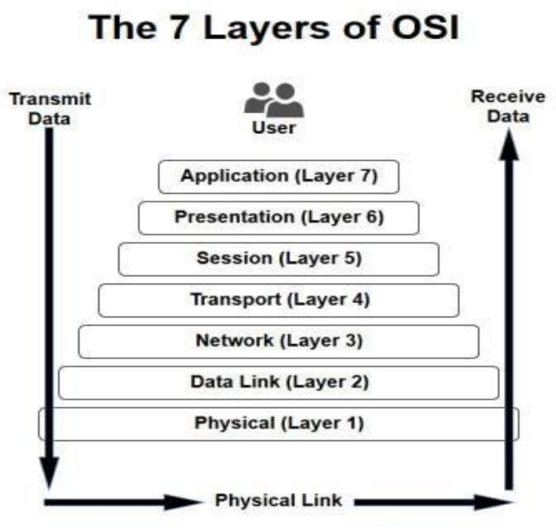
MapmyIndia

CONTENTS

1. Understanding Networking

A computer network is a set of computing devices (Computers, IoT devices, phones etc...) that are connected to each other using a media link (For example - ?) and can communicate with each other through digital data and/or share resources. Each such computing device is called a node in the network. The standard reference model used in networking is the **Open Systems Interconnection** model (OSI Model).

The OSI Model consists of the following 7 layers:



(via https://www.webopedia.com/imagesvr_ce/8023/7-layers-of-osi-icon.jpg)

While OSI model is a generic, protocol independent network model, the actual implementation which the internet today uses is the **Transmission Control Protocol/Internet Protocol (TCP/IP).** The TCP/IP does not have a separate presentation, session and a physical layer. Therefore it only has the following four layers.

4 - Application Layer: This is where our application runs. The application calls the transport layer for sending the data.

3 - Transport Layer: The transport layer ensures that the bytes that the application wants to send are delivered correctly, ones that are missing and re-transmitted and are delivered in order.

2 - Network Layer: The network layer divides the data that the application wants to send into smaller segments which contain some bytes of the data called **packets**. The network layer also adds the destination IP address to each packet.

1 - Link Layer: The packet is transmitted through a physical link such as LAN cable or WiFi. Here the unit of communication is an Ethernet frame which consists of the destination and source MAC address.

These layers are abstracted from each other, that is, each layer hides the details of implementation from the layer above and below them. These layers of network are collectively called the network stack. The part of the network layer where the operating system comes into play is the transport layer and the network layer. The application uses the **socket API** to call the network stack running inside the OS. The TCP/IP layer of the OS converts the data into packets, attaches the appropriate destination IP address and sends the packets to the Link Layer. The packets are then transported link by link until they are received by the destination Link Layer. On reaching the target destination, data in the packet is decoded through the layers until the application layer in the target destination receives it.

2. Socket Programming in Java

A network **socket** is an abstract handle which is used by the application layer to communicate with the transport layer, that is, send/receive stream of bytes. Sockets can therefore be considered as two endpoints of the channel through which the communication is taking place. In a client-server model, the socket is used for client/server interaction. In the client-server model, one of the computers/processes acts as the server while other computers/processes act as clients which connect to the server.

In a client-server model, the computer acting as the server first binds a socket to an address which the clients can use to connect to the server. The socket then waits (or listens) for clients to initiate a connection request through a socket. Once a request is made, the server accepts the client socket and data can be exchanged through the client-server sockets.

The package java.net provides the classes used in networking.

The InetAddress class

public class InetAddress extends Object implements Serializable

InetAddress class is the Internet Protocol (IP) class. The class abstracts out the IP address and the domain name. It provides member methods to resolve the IP address from the domain name and vice-versa (reverse name resolution). InetAddress stores the IP address in the network byte order (Big-endian format). InetAddress does not have a visible constructor, therefore one of the factory methods have to be used to get an instance of the InetAddress class.

Commonly used factory methods for getting an instance of InetAddress.

static InetAddress getLocalHost() throws UknownHostException	Resolves the IP address of the localhost.
static InetAddress getByName(String hostname) throws UknownHostException	Resolves the IP address of the specified host name.
static InetAddress[] getAllByName(String hostname) throws UknownHostException	Resolves the IP address of all the hosts having the specified hostname.
static InetAddress getByAddress(String addr) throws UnknownHostException	Resolves the name of the host using the IP address specified.

The Socket and the ServerSocket classes

The Socket and the ServerSocket classes are used for initializing the socket on both client and server side.The socket class has many constructors which can be used to initialize the class.

Socket (String hostname, int port) throws UknownHostException, IOException.	Creates a socket to the specified hostname and the port. The hostname's IP address is resolved automatically and is abstracted out.
Socket (InetAddress ipAddr, int port) throws IOException	Creates a socket using an instance of the InetAddress class (which abstracts out the name and IP address of the host) and the specified port.

ServerSocket class is used to create sockets on the server side. The accept() method of the ServerSocket is used to call the initialized server socket to listen for a connection. Once a connection is made, the accept() method returns an instance to the Socket class.

The most commonly used constructor for initializing ServerSocket is *ServerSocket (int port)* which creates a server socket bound to a specific port.

3. Java multithreading

A **thread** is an independent path of execution of a program. When a program has two or more such paths executing concurrently, we call it multithreading. In a multiple-client- single-server, while receiving data from one of the clients, the I/O blocks the execution of the program preventing the receiving of data from other clients. To solve this problem, we used the concept of multithreading. Multithreading creates a separate thread (or an execution path) for every client. Multithreading is implemented even in the physically single core using **context switching**. Context switching is used for multi-tasking by dividing the CPU time available to the threads. The number of threads that can actually run concurrently is the number of physical cores that the CPU has. If the number of threads is more than that, then the CPU time is sliced amongst the threads. (That is some threads execute concurrently while other threads are waiting for their turn.) For example, if there two physical cores and four threads executing, then two of them will be running concurrently while the other two wait for their turn to get CPU time available for their execution. This ensures that the CPU time is used optimally. Also it should be noted that all the threads share the same memory address space. *(https://stackoverflow.com/questions/1713554/threads-processes-vs-multithreading-multi-core-multiprocessor-how-they-are)*

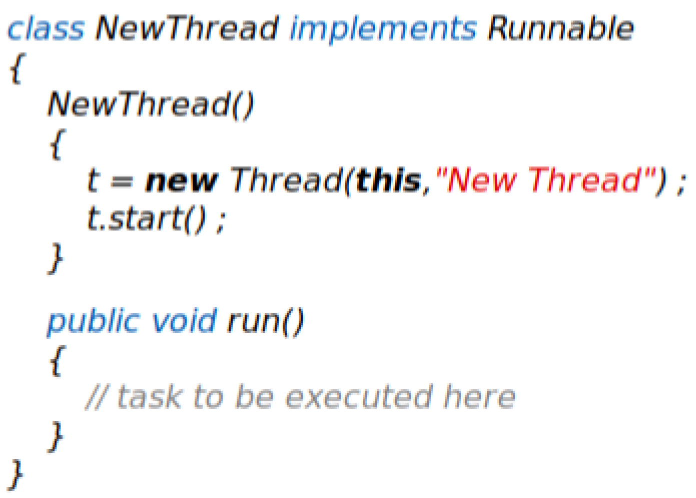
The package which is used to implement multithreading is *java.lang*.

The Main Thread

When java program starts, the *main* thread is created. There a java program always has at least one thread which is executing. All the other threads are created from the main thread.

Methods of creating a thread:

1. ***implement* Runnable interface.**



```
class NewThread implements Runnable
{
   NewThread()
   {
      t = new Thread(this,"New Thread") ;
      t.start() ;
   }

   public void run()
   {
      // task to be executed here
   }
}
```

2. ***extend* thread class.**

```
class NewThreads extends Thread
{
   NewThreads()
   {
      super("ChildThread") ;
      start() ;
   }

   public void run()
   {
      // task to be executed here
   }
}
```

Synchronization

When more than one thread try to access the same resources asynchronously, then it might lead to unwanted results. For example, when one thread is writing to a file, and at the same time another thread tries to write to the same file, then the contents written by the first thread might get erased/mixed-up with the contents written by the second thread. To avoid such a condition the keyword *synchronized* is used with the block of code that has to be used synchronously (that is, only by one thread at a time). Synchronization therefore blocks until the thread currently using the synchronized block has exited the block**.**

Example usage:

Synchronized monitor object:

```
synchronized (obj) {
// only one thread can access this block of code at a time using the object obj
}
```

Synchronized function:

```
public void synchronized write (String msg) {
// write to a file using I/O. Only one thread can call this method at a time
}
```

Deadlock

Deadlock is a special condition that occurs in multi-threaded programs in java. Consider two threads Th1 and Th2 and two synchronized objects obj1 and obj2. Suppose Th1 has blocked obj1 and Th2 has blocked obj2. Now, suppose Th1 is waiting for Th2 to release lock on obj2 and Th2 is waiting for Th1 to release lock on obj1. But since Th1 has blocked obj1 and Th2 has blocked obj2, both of them keep waiting. This condition is known as **deadlock**.

4. I/O IN JAVA

Use of networking is to access information from a remote source and for that communication between a client and server is required. This can be achieved using Input-Output operations offered by java. It also offers to transmit data in various encodings such as UTF-8, Unicode, 7-bit latin etc, according to our need.

The socket is the endpoint to which connections on both ends are made. The socket provides the input and output Stream at both ends for communication.

```
Socket s = new Socket("www.google.com",80);
InputStream in = s.getInputSream();
OutputStream out = s.getOuptputStream();
```

These streams of both clients are connected to each other. InputStream of Client Socket is connected to OutputStream of Server Socket and vice versa. So when data is sent by Client socket over its outputStream, it is received by InputStream of ServerSocket.

Then, the result is sent back by Server through its OutputStream and received by the client through its InputStream.

These DataStreams can be buffered to get a large amount of data to be read or write once enabling faster data transmission.

```
BufferedReader br = new BufferedReader(new InputStreamReader(s.getInputstream()));
BufferedWriter bw = new BufferedWriter(new OuputStreamWriter(s.getOutputStream()));
```

5. Cryptographic fundamentals

Consider two parties Alice and Bob who want to communicate with each other. Consider another party Eve who has the control of the communication media through which Alice and Bob want to communicate. Cryptography is the method of encryption of data from plain text to an unintelligible form called cipher text and decryption from cipher text to plaintext to ensure that Eve cannot read the information or alter it in the process of communication.

Private key cryptography

Private key cryptography uses those algorithms which require only one key (known as the secret key) for both encryption and decryption. For the same reason, encryption and decryption take less time and the scheme is therefore used for encryption of bulk data. Since only one key is required, key management should be secure enough to avoid any leak/theft of key while distribution of keys. Examples of Private key algorithms are AES and 3DES.

Public key cryptography

Public key cryptography uses algorithms which require two different keys for encryption and decryption – the public key and the private key. The public key is known to all (hence the name “public”) while the private key is known only to the party which receives the encrypted information. Public key cryptography makes the use of **Trapdoor One-Way functions**. Trapdoor one-way functions are those which are easy to compute in one direction, but hard to compute in reverse (that is, inverse) without some special information called **trapdoor**. There is no issue of key management in Pkc, since the private key does not have to be distributed. Pkc algorithms are slow and are used for encryption of small data (for example exchange of cipher suite during handshaking process in SSLSockets) and exchange of private keys. Example of public key cryptography are RSA, D-H, and RC4.

Digital Signatures

Digital signature is a string generated using the contents of a digital document which ensures that the data in the document was not tampered and verifies the identity of the sender.

Cryptographic hash functions

A hash is a unique string of fixed length which can be generated for message of any length. No key is required for generating a hash. Examples of cryptographic hashing algorithms are **SHA256** and **MD5**.

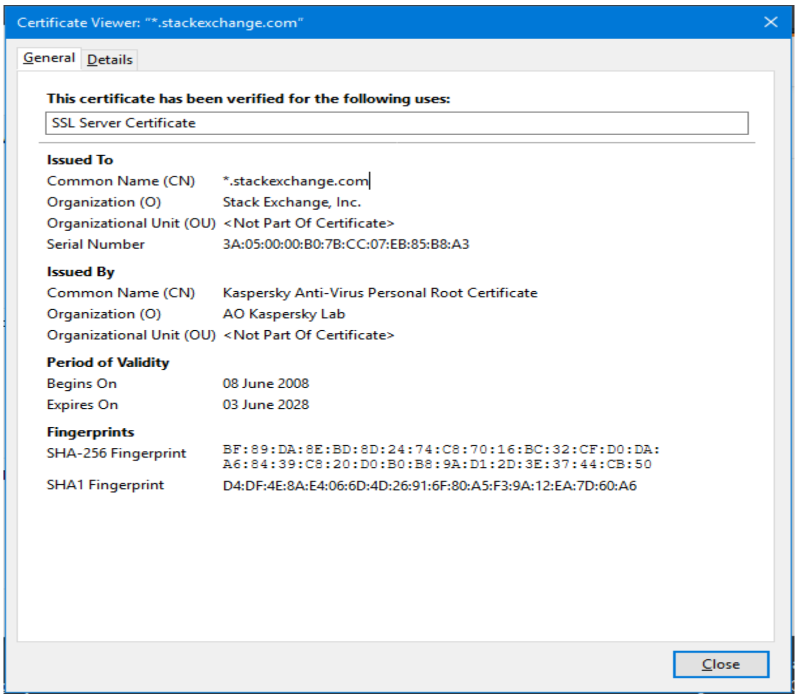
Message authentication code (MAC) and HMAC

MAC is a string of bits that is generated using the message and a secret key. The secret key is shared between both the communicating parties. The generated string is sent along with the message. The receiver of the message can compute the MAC of the message using the secret key and compare them. If they match, then the data was not tampered in the process. When hash functions are used to generate the MAC, it is known as HMAC. Some common HMAC algorithms are **HMAC-MD5** and **HMAC-SHA256**.

Public key certificate

Consider two parties Alice and Bob communicating with each other. Eve has the control of the communication medium. Eve can create his own public and private keys and send the public key to Bob, claiming that he is Alice. Bob will use the public key to encrypt data and send it, thinking he is communicating with Alice. To avoid such a situation, **authentication** is needed, which is done using Public key certificates. PKC is a digital document which issued by a **Certificate Authority** - a certifying authority which is trusted by both the sides. The contents of the PKC, which consists of senders public key along with some other data is digitally signed by the CA, to ensure its legitimacy. The contents of the PKC are:

- Issuer
- Period of validity
- Subject
- Subjects public key
- Digital signature – created using Certified Authority's (CA) private key and ensures the validity of the certificate.



This certificate has been verified for the following uses:
SSL Server Certificate

Issued To

Common Name (CN)	*.stackexchange.com
Organization (O)	Stack Exchange, Inc.
Organizational Unit (OU)	<Not Part Of Certificate>
Serial Number	3A:05:00:00:B0:7B:CC:07:EB:85:B8:A3

Issued By

Common Name (CN)	Kaspersky Anti-Virus Personal Root Certificate
Organization (O)	AO Kaspersky Lab
Organizational Unit (OU)	<Not Part Of Certificate>

Period of Validity

Begins On	08 June 2008
Expires On	03 June 2028

Fingerprints

SHA-256 Fingerprint	BF:89:DA:8E:BD:8D:24:74:C8:70:16:BC:32:CF:D0:DA: A6:84:39:C8:20:D0:B0:B8:9A:D1:2D:3E:37:44:CB:50
SHA1 Fingerprint	D4:DF:4E:8A:E4:06:6D:4D:26:91:6F:80:A5:F3:9A:12:EA:7D:60:A6

Example of a PKC

6. Understanding SSL/TLS Protocol

The Secure Sockets Layer sits in between the Application layer and the transport layer. It provides authentication (guarantee of identity of both client and server, privacy (encryption/decryption) and data integrity (data is not tampered in the process) to the applications. Due to various security flaws, SSL was replaced by TLS (Transport Security Layer) which is considered to be an advanced version of SSL (SSL 3.1). **Java Secure Sockets Extension (JSSE)** is a set of packages in java that provide the framework for the implementation of SSL/TLS in java.

SSL uses public key cryptography for authentication and secret key cryptography with digital signatures for data privacy and integrity.

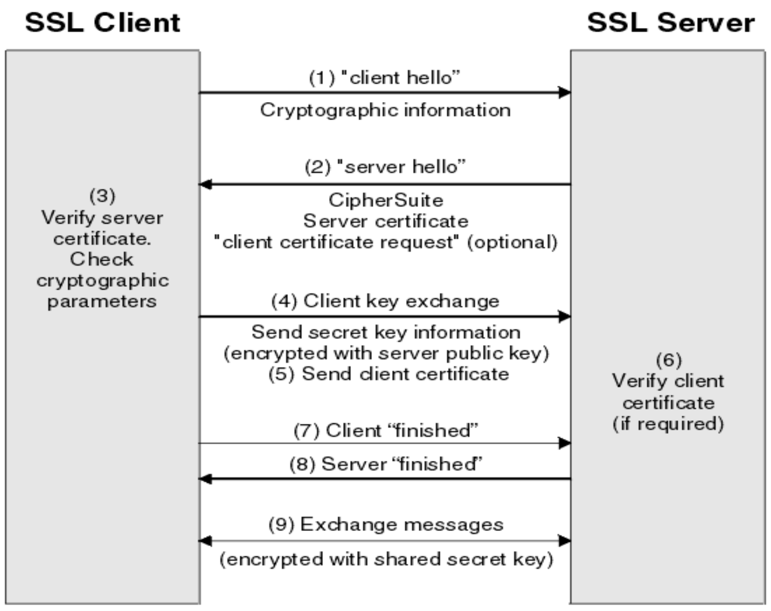
Cipher suite

A set of algorithms used for secure communication by SSL. It consists of

- Key exchange algorithm (Public key algorithm for example RSA, DH)
- Private key algorithm (for encryption of bulk messages).
- Hashing algorithm and MAC algorithm (to ensure data integrity).

SSL Handshake

During the SSL handshake, the client and the server exchange the secret keys (for symmetric key cryptography) for bulk encryption and decryption of messages and verify each other's identity.


SSL Client
SSL Server
(1) "client hello"
Cryptographic information
(2) "server hello"
CipherSuite
Server certificate
"client certificate request" (optional)
(3) Verify server certificate. Check cryptographic parameters
(4) Client key exchange
Send secret key information (encrypted with server public key)
(5) Send client certificate
(6) Verify client certificate (if required)
(7) Client "finished"
(8) Server "finished"
(9) Exchange messages
(encrypted with shared secret key)

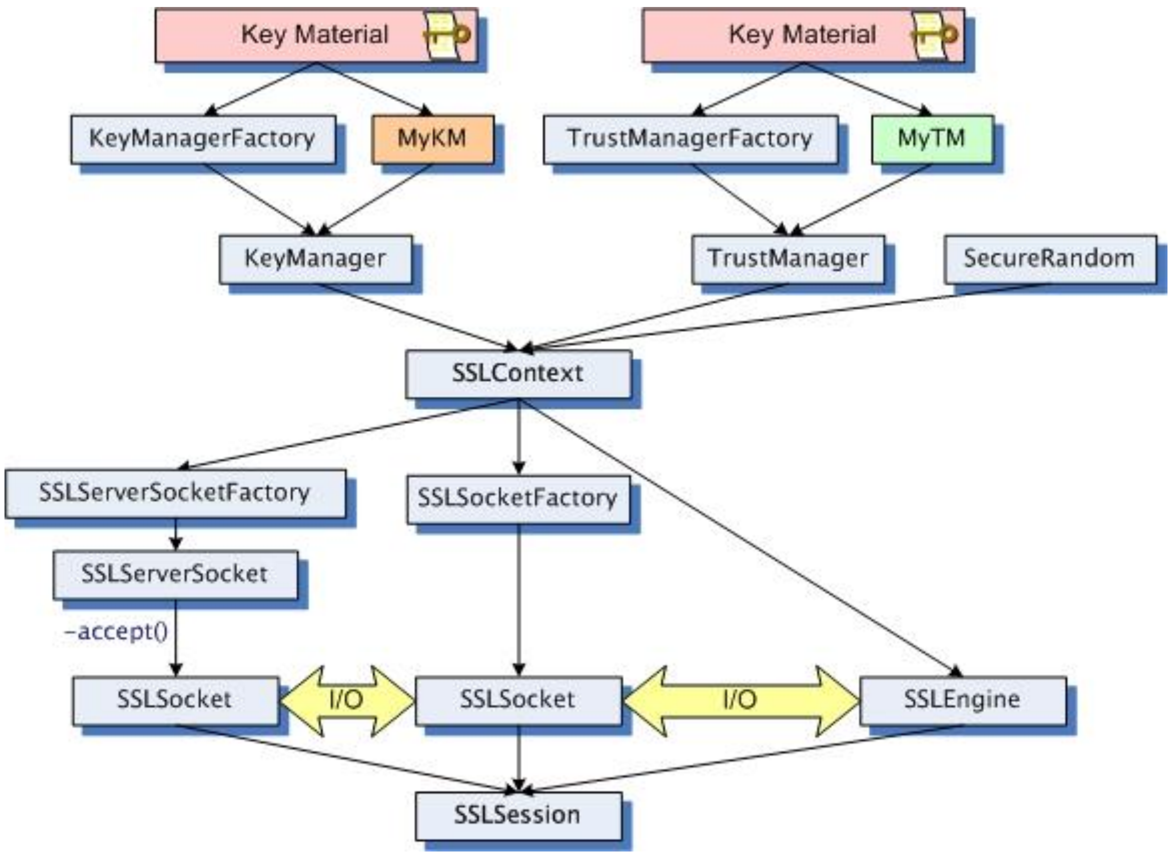
via https://www.ibm.com/support/knowledgecenter/SSFKSJ_7.5.0/com.ibm.mq.sec.doc/q009930a.gif

Summary of the steps involved during the handshake process:

1. Exchanging the available versions of protocols that the client and server are using and selecting the best one they are both compatible with. Client presents the server with the list of all the cipher suites that it supports
2. Server selects the best cipher suite that both the client and the server have in common and presents its public key certificate (which contains the public key of the server). Server also optionally requests the client for its public key certificate for client verification.
3. Client verifies the public key certificate, and generates a private key (for symmetric encryption of bulk messages). Client uses the servers public key (obtained for the servers public key certificate) to encrypt the generated secret (used for symmetric encryption) key and sends it to the server along with its own public key certificate (if requested by server).

4. Server verifies the clients certificate (if required) and sends a “finish” message to the client. The client also sends the “finish” message to indicate that it is ready to exchange the messages.

An overview of the classes used in implementing SSL secure client/server:



via https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/classes1.jpg

Packages which contain the classes are *javax.net* and *javax.net.ssl*

KeyStore/TrustStore

A KeyStore in java is a file that stores the client’s/server’s private key and the public key certificate (issued by CA or self-signed). A TrustStore in java is also a file that contains the root or the intermediate CA certificates. This allows the client/server to determine whether the certificate presented by the peer is signed by one of the CA whose certificate is stored in the TrustStore. If not, the certificate presented by the peer is not trusted. A KeyStore can also be used as a TrustStore.

KeyManager/TrustManager

On the server side, KeyManager is used by the server to present its public key certificate (which contains the server’s public key) for authenticating itself to the client. For authenticating the server, the TrustManager on the client side determines whether the server’s public key certificate is signed by one of the trusted CA’s (whose root certificates are present in the TrustStore). If client authentication is also required then KeyManager and TrustManager are present on both client and server side.

Obtaining SSLSocket

The private, public key pair and the public key certificate can be generated using a key tool (JDK provides its own key tool for this purpose) and stored in a file of appropriate format which acts as the Key Material. The Key Material is used to initialize the KeyStore and the TrustStore using the method *KeyStore.getInstance (String type),* which returns a KeyStore object of the specified type. The KeyStore file is loaded using the method *KeyStore.load (InputStream stream, char[] password).*

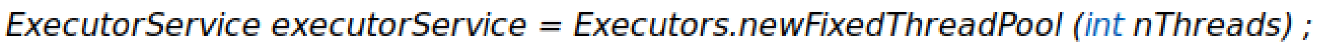
The TrustManagerFactory and the KeyManagerFactory are created using the *TrustManagerFactory.getInstance (String algorithm) and KeyManagerFactory.getInstance (String algorithm)*. The algorithm provided by the package is “Sunx509” and are initialized by *TrustManagerFactory.init (KeyStore ks)* and *KeyManagerFactory.init (KeyStore ks, char[] password)*. SSLContext object is created by *SSLContext.getInstance (String protocol)* and is initialized using *SSLContext.init (KeyManager[] km, TrustManager[] tm, SecureRandom random)*. For using TLS protocol, protocol = “TLS”. The SSLSocketFactory and SSLServerSocketFactory objects are created using the *SSLContext.getSocketFactory()* and *SSLContext.getServerSocketFactory()* which can in turn be used to create sockets using the *SSLSocketFactory.createSocket (int port)* for client *and SSLServerSocketFactory.createServerSocket (InetAddress ipAddr, int port)* for server. On the server side *SSLServerSocket.accept()* is used to listen to incoming connection requests which on receiving a request returns an object of type Socket. To get object of type SSLSocket, Socket has to be explicitly type casted to SSLSocket.

7. Executor services and thread pool

The Executor Services is a thread creation and management framework which is used when the multithreaded application requires creating a large number of threads. In our case, a new thread is created for every client that connects to the server. Since the server’s capacity might be limited, the number of threads that can be executed concurrently should be limited. For this purpose we used a thread pool of fixed size.

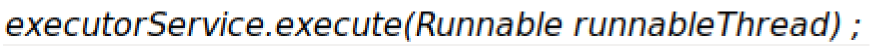
The executor service maintains a queue of waiting threads internally. The queue is non-empty when the number of threads to be executed exceeds the maximum pool size. As soon as one of the threads has finished executing, next thread in the waiting queue is allowed to execute.

Initializing a thread pool of fixed size



```
ExecutorService executorService = Executors.newFixedThreadPool (int nThreads) ;
```

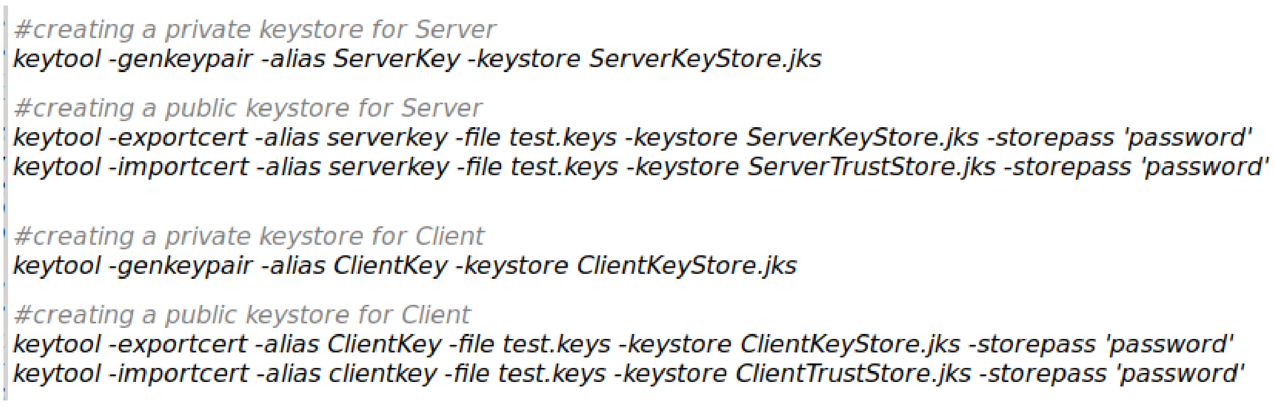
A thread is submitted to the ExecutorService for execution in the following way.



```
executorService.execute(Runnable runnableThread) ;
```

8. Implementation and working

Firstly, we made 4 KeyStores- two each for client and server, one acting as a trust store and other as KeyManager. This was necessary as client and server both need to be authenticated. The certificates in those Keystores were created using Keytool offered by Java and were self-signed.



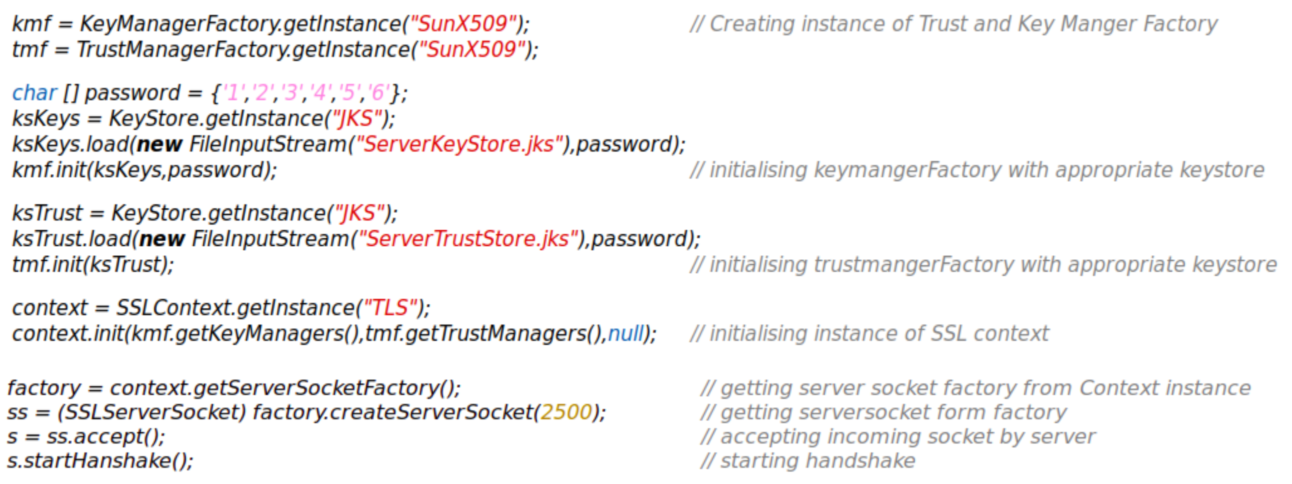
```
#creating a private keystore for Server
keytool -genkeypair -alias ServerKey -keystore ServerKeyStore.jks

#creating a public keystore for Server
keytool -exportcert -alias serverkey -file test.keys -keystore ServerKeyStore.jks -storepass 'password'
keytool -importcert -alias serverkey -file test.keys -keystore ServerTrustStore.jks -storepass 'password'


#creating a private keystore for Client
keytool -genkeypair -alias ClientKey -keystore ClientKeyStore.jks

#creating a public keystore for Client
keytool -exportcert -alias ClientKey -file test.keys -keystore ClientKeyStore.jks -storepass 'password'
keytool -importcert -alias clientkey -file test.keys -keystore ClientTrustStore.jks -storepass 'password'
```

Then we have to make sure that certificate of server is in client trust store and vice versa.

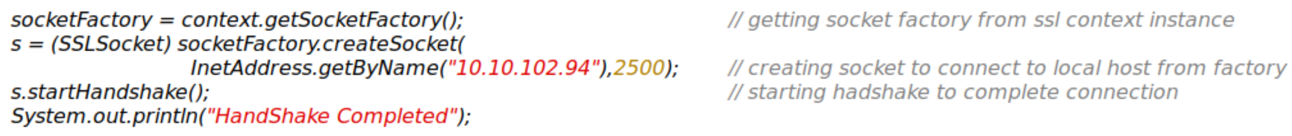
Two classes were made - one for client and server and following tasks were achieved:

1. **Secure Connection:**

   First step was to establish a secure connection with proper handshaking between them. For this SSLContext object was created and initialised with a KeyManagerFactory, TrustManagerFactory and a SecureRandomNumber. This is where KeyStores come in picture these files were read by appropriate Factory Managers to get initialised.
   After this we get the Socket for Client and ServerSocket for Server. When client gets activated it tries to connect with Server thus beginning the handshaking process after which connection is established.



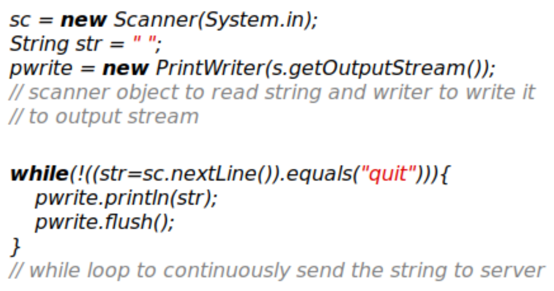
```
kmf = KeyManagerFactory.getInstance("SunX509");                 // Creating instance of Trust and Key Manger Factory
tmf = TrustManagerFactory.getInstance("SunX509");

char [] password = {'1','2','3','4','5','6'};
ksKeys = KeyStore.getInstance("JKS");
ksKeys.load(new FileInputStream("ServerKeyStore.jks"),password);
kmf.init(ksKeys,password);                                       // initialising keymangerFactory with appropriate keystore

ksTrust = KeyStore.getInstance("JKS");
ksTrust.load(new FileInputStream("ServerTrustStore.jks"),password);
tmf.init(ksTrust);                                               // initialising trustmangerFactory with appropriate keystore

context = SSLContext.getInstance("TLS");
context.init(kmf.getKeyManagers(),tmf.getTrustManagers(),null);  // initialising instance of SSL context

factory = context.getServerSocketFactory();                      // getting server socket factory from Context instance
ss = (SSLServerSocket) factory.createServerSocket(2500);         // getting serversocket form factory
s = ss.accept();                                                 // accepting incoming socket by server
s.startHanshake();                                               // starting handshake
```



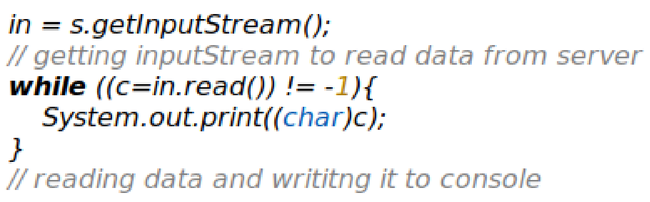
```
socketFactory = context.getSocketFactory();                      // getting socket factory from ssl context instance
s = (SSLSocket) socketFactory.createSocket(
                InetAddress.getByName("10.10.102.94"),2500);     // creating socket to connect to local host from factory
s.startHandshake();                                              // starting hadshake to complete connection
System.out.println("HandShake Completed");
```

   Then begins the actual communication between Client and Server.

   For this InputStream and OutputStream of sockets at both ends are used. InputStream of Client is connected to OutputStream of Server and vice-versa. Data is sent from OutputStream of Client and received by InputStream of server. Server processes upon the data and gives back the response through its OuptutStream which is then received by InputStream of Client as output to its request.



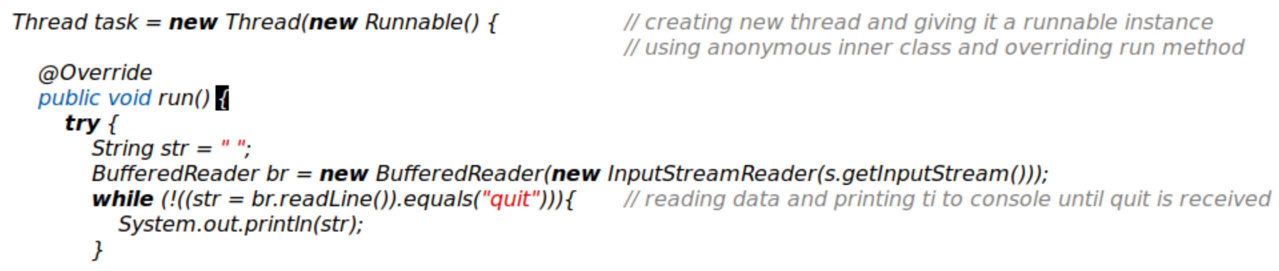
```
sc = new Scanner(System.in);
String str = " ";
pwrite = new PrintWriter(s.getOutputStream());
// scanner object to read string and writer to write it
// to output stream

while(!((str=sc.nextLine()).equals("quit"))){
    pwrite.println(str);
    pwrite.flush();
}
// while loop to continuously send the string to server
```



```
in = s.getInputStream();
// getting inputStream to read data from server
while ((c=in.read()) != -1){
    System.out.print((char)c);
}
// reading data and writitng it to console
```

2. **Multi-Client Server Communication:**

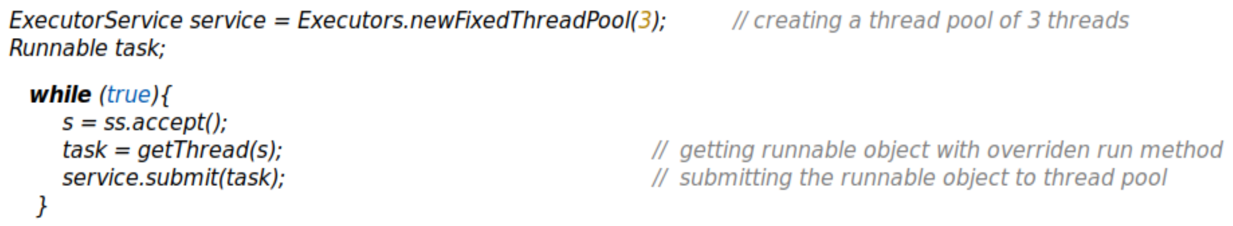
   Using multi-threading we can make as many as clients to connect to server. Each new client when tries to connect to server a new thread is created and its run method is called for further communication.



```
Thread task = new Thread(new Runnable() {              // creating new thread and giving it a runnable instance
                                                        // using anonymous inner class and overriding run method
   @Override
   public void run() {
      try {
         String str = " ";
         BufferedReader br = new BufferedReader(new InputStreamReader(s.getInputStream()));
         while (!((str = br.readLine()).equals("quit"))){      // reading data and printing ti to console until quit is received
            System.out.println(str);
         }
```

3. **Executor Service and Thread Pool:**

   Although multi-threading can provide the functionality to a server to connect to many clients but each new thread created makes an extra overhead on the flow of server so its better to use a thread pool. Basically, a thread pool contains fixed number of services to which a thread can be assigned. If that becomes full thread object has to wait and can continue to its run method once gets space.



```
ExecutorService service = Executors.newFixedThreadPool(3);        // creating a thread pool of 3 threads
Runnable task;

  while (true){
      s = ss.accept();
      task = getThread(s);                              //  getting runnable object with overriden run method
      service.submit(task);                             //  submitting the runnable object to thread pool
  }
```