**Protocol Emulator for TLS**

**(Transport Layer Security)**

**RFC 5246**

**Table Of Contents**

**1**. Abstract: ....................................................................................................................... 3

**2**. Architecture: ................................................................................................................. 3

**3**. TLS Handshaking Protocols: ….................................................................................... 4

**3.1**. The TLS Handshake Protocol Steps: …………..…................................................. 5

**3.2**. MSC of Handshake Process: ................................................................................... 7

**4**. TLS Record Protocol: ….…………………………………………………………….. 9

* Security Parameters: ..…………………………………………………………… 9
* Fragmentation: …………………………………………………………..…….. 10
* Compression Algorithm: ……………………………………………………….. 10
* Encryption Algorithm: …………………………………………………………. 10
* Connection States: ...………………………………………….......................….. 10

**5**. HMAC and PRF: ......................................................................................................... 10

**6.** Principles of REST: .................................................................................................... 12

**6.1** REST Methods: ....................................................................................................... 12

**6.2**. JAX-RS Application, Resources and Sub-Resources: ........................................... 12

**7.** MSC of Implementation: …………………………………………………...….......... 13

**8.** Java Code Details: …...…………………………………………………………….... 14

**8.1**. TLS Client (mobile.computing.ws1819.client): ...………….………………......... 14

* Classes

**8.2**. TLS Server (mobile.computing.ws1819.server): ………………………….…........15

* Classes

**8.3**. Common Classes for both server and client (mobile.computing.ws1819): .………16

**9**. Certificate Generation: ...…...........................................................................................16

**10**. Steps to run the Emulator: ...................………………………………………………17

**11.** Results from terminal: ……………………………………………………………… 17

**12**. Results and captures from WireShark: …..………………………………………...... 18

**13**. References: …………………………………………………………………………..20

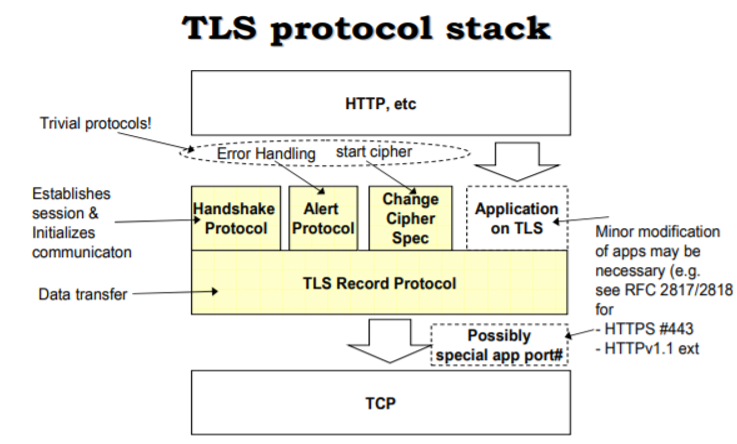
**1. Abstract:**

Transport Layer Security (TLS) is a protocol that provides security for communication over the Internet. TLS encrypts segments of network connections, in order to provide confidentiality when communicating via the Internet. By encrypting the communication, we ensure that no third-party is able to read or tamper with the data that is being exchanged on our connection with the server. An unencrypted communication could and would expose sensitive data such as usernames, passwords, credit card numbers and generally anything that is being sent back and forth during the connection.

TLS is composed of two layers:

* The TLS Record Protocol: Provides connection security and for encapsulation of various higher level protocols.
* The TLS Handshake Protocol: allows authentication and to negotiate encryption algorithms between communicating parties

**2. Architecture:**



**3. TLS Handshaking Protocol:**

Handshake Protocol is responsible for the authentication and key exchange necessary to establish or resume secure sessions. When establishing a secure session, the Handshake Protocol manages the following:

* Cipher suite negotiation
* Authentication of the server and optionally, the client
* Session key information exchange.

**Structure of Handshake Protocol:**

enum {

hello\_request(0), client\_hello(1), server\_hello(2),

certificate (11), server\_key\_exchange (12),

certificate\_request(13), server\_hello\_done(14),

certificate\_verify(15), client\_key\_exchange(16),

finished (20), (255)

} HandshakeType;

struct {

HandshakeType msg\_type; /\* handshake type \*/

uint24 length; /\* bytes in message \*/

select (HandshakeType) {

case hello\_request: HelloRequest;

case client\_hello: ClientHello;

case server\_hello: ServerHello;

case certificate: Certificate;

case server\_key\_exchange: ServerKeyExchange;

case certificate\_request: CertificateRequest;

case server\_hello\_done: ServerHelloDone;

case certificate\_verify: CertificateVerify;

case client\_key\_exchange: ClientKeyExchange;

case finished: Finished;

} body;

} Handshake;

**3.1. The TLS Handshake Protocol involves the following steps:**

1. The client sends a "**Client hello**" message to the server, along with the client's random value and supported cipher suites.
   * + **Contents**: Handshake Type (1 byte), Length (3 bytes), Version (2 bytes)

* Type 01 for Client Hello
* Version: 0303 for TLS

32 bytes Random

Session ID

Cipher suites

Compression algorithm

* + - **Structure of this message :**

struct {

ProtocolVersion client\_version;

Random random;

SessionID session\_id;

CipherSuite cipher\_suites<2..2^16-2>;

CompressionMethod compression\_methods<1..2^8-1>;

} ClientHello;

1. The server responds by sending a "**Server Hello**" message to the client, along with the server's random value.
   * + **Contents:** Handshake Type (1 byte), Length (3 bytes), Version (2 bytes)

* Type 02 for Server Hello
* Version: 0301 for TLS

32 bytes Random (different from client hello)

Session ID

Cipher suite (Selected from client list)

Compression algorithm Selected from client list)

* + - **Structure of this message:**

struct {

ProtocolVersion server\_version;

Random random;

SessionID session\_id;

CipherSuite cipher\_suite;

CompressionMethod compression\_method;

select (extensions\_present) {

case false:

struct {};

case true:

Extension extensions<0..2^16-1>;};

} ServerHello;

1. The server sends its “**Certificate”** (certificate type MUST be X.509v3) along with public key to the client for authentication and may request a certificate from the client

* **Structure of this message:**

struct {

ASN.1Cert certificate\_list<0..2^24-1>;

} Certificate

1. The server MUST send the "**Server** **hello done**" message immediately after Sending certificate to indicate the end of the Server Hello and associated messages. It is an empty message.
2. The client sends “**Client Key Exchange”** message in which client creates a random Pre-Master Secret and encrypts it with the [public key](https://msdn.microsoft.com/en-us/library/ms721603(v=VS.85).aspx) from the server's certificate, sending the encrypted Pre-Master Secret to the server.

* **Structure of this message:**

struct {

select (KeyExchangeAlgorithm) {

case rsa:

EncryptedPreMasterSecret;

case dhe\_dss:

case dhe\_rsa:

case dh\_dss:

case dh\_rsa:

case dh\_anon:

ClientDiffieHellmanPublic;

} exchange\_keys;

} ClientKeyExchange**;**

* **RSA-Encrypted Premaster Secret Message:**

If RSA is being used for key agreement and authentication, the client generates a 48-byte premaster secret, encrypts it using the public key from the server's certificate, and sends the result in an encrypted premaster secret message. This structure is a variant of the ClientKey Exchange message and is not a message in itself.

* + - **Structure of this message:**

struct {

ProtocolVersion client\_version;

opaque random[46];

} PreMasterSecret;

1. The server receives the Pre-Master Secret. The server and client MUST each generate the Master Secret and [session keys](https://msdn.microsoft.com/en-us/library/ms721625(v=VS.85).aspx) based on the Pre-Master Secret.
2. The client then MUST send "**Change Cipher Spec**" notification to server to indicate that the client will start using the new [session keys](https://msdn.microsoft.com/en-us/library/ms721625(v=VS.85).aspx) for [hashing](https://msdn.microsoft.com/en-us/library/ms721586(v=VS.85).aspx) and encrypting messages. The protocol consists of a single message, which is encrypted and compressed under the current connection state. The message consists of a single byte of value 1.
   * **Structure of this message:**

struct {

enum { change\_cipher\_spec(1), (255) } type;

} ChangeCipherSpec;

1. The Client then MUST Send  **‘Client Finished”** message which Authenticate all the previous handshake messages.(Finished = digital signature of all the previous handshake messages as transmitted and received by the peer up to now )
   * **Structure of this message:**

struct {

opaque verify\_data[verify\_data\_length];

} Finished;

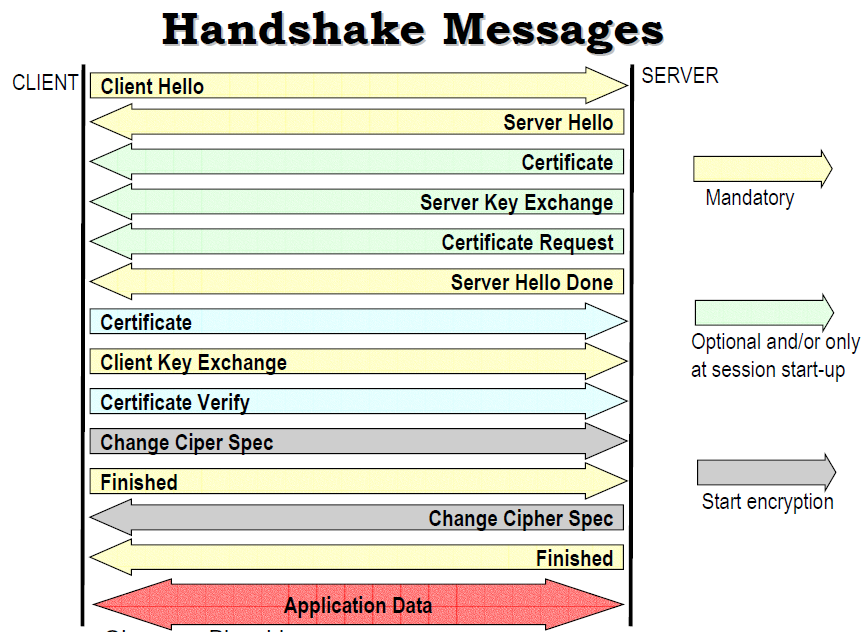
verify\_data

PRF(master\_secret, finished\_label, Hash(handshake\_messages))

[0..verify\_data\_length-1];

1. Server receives "Change cipher spec" and switches its record layer security state to [symmetric encryption](https://msdn.microsoft.com/en-us/library/ms721625(v=VS.85).aspx) using the [session keys](https://msdn.microsoft.com/en-us/library/ms721625(v=VS.85).aspx). Server sends **Change Cipher Spec and "Server finished**" messages to the client.
2. Client and server can now exchange **Application Data** over the secured channel they have established. All messages sent from client to server and from server to client are encrypted using session key.

**3.2. MSC of Handshake Process:**



**Summary of Handshake Process** (from RFC):

* Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.
* Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.
* Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.
* Generate a master secret from the premaster secret and exchanged random values.
* Provide security parameters to the record layer.
* Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

Client Server

ClientHello ----------------------------------------------------------->

ServerHello

Certificate\*

ServerKeyExchange\*

CertificateRequest\*

<------------------------------------------------------------- ServerHelloDone

Certificate\*

ClientKeyExchange

CertificateVerify\*

[ChangeCipherSpec]

Finished --------------------------------------------------------------->

[ChangeCipherSpec]

<---------------------------------------------------------------- Finished

Application Data <-------> Application Data

Fig (RFC page 36)

**4. TLS Record Protocol:**

Record protocol secures application data using the keys created during the [Handshake](https://docs.microsoft.com/en-us/windows/desktop/secauthn/tls-handshake-protocol). The Record Protocol is responsible for securing application data and verifying its [integrity](https://msdn.microsoft.com/en-us/library/ms721588(v=VS.85).aspx) and origin.

It manages the following:

* Dividing outgoing messages into manageable blocks, and reassembling incoming messages.
* Compressing outgoing blocks and decompressing incoming blocks (optional).
* Applying a [Message Authentication Code](https://msdn.microsoft.com/en-us/library/ms721594(v=VS.85).aspx) (MAC) to outgoing messages, and verifying incoming messages using the MAC.
* Encrypting outgoing messages and decrypting incoming messages.

When the Record Protocol is complete, the outgoing encrypted data is passed down to the Transmission Control Protocol (TCP) layer for transport.

* **The Security Parameters:**

The security parameters are determined by the TLS Handshake Protocol when server responds with available and prioritized picks from client and provided as parameters to the TLS record layer in order to initialize a connection state.

* **Structure of this Security parameter:**

struct {

ConnectionEnd entity;

PRFAlgorithm prf\_algorithm;

BulkCipherAlgorithm bulk\_cipher\_algorithm;

CipherType cipher\_type;

uint8 enc\_key\_length;

uint8 block\_length;

uint8 fixed\_iv\_length;

uint8 record\_iv\_length;

MACAlgorithm mac\_algorithm;

uint8 mac\_length;

uint8 mac\_key\_length;

CompressionMethod compression\_algorithm;

opaque master\_secret[48];

opaque client\_random[32];

opaque server\_random[32];

} SecurityParameters

The record layer will use the security parameters to generate the following six items:

* client write MAC key
* server write MAC key
* client write encryption key
* server write encryption key
* client write IV
* server write IV
* **Fragmentation**

Data in blocks of up to 2 ^14 are fragmented bytes in length, whereby different records of the same type in one can be summarized such TLSPlaintext record.

* **Compression algorithm**

Compression, which is basically optional, is performed using the compression methods defined in this session. There is always an active compression method, which can be CompressionMethod.null. Compression must be lossless and never increase the data length by more than 1024 bytes.

* **Encryption algorithm**

Encryption is performed to protect the data from being read or intentionally altered. A counter also detects missing, additional or repeated messages.

* **Connection States** (Operating Environment of TLS Record Protocol)

Comprises of Compression algorithm, an Encryption algorithm and MAC algorithm.

In the TLS connection state, the parameters of the TLS Record Protocols are described. The two participants of a communication are called client and server. In addition to defining the respective algorithms for authentication, ciphering, etc., the current parameters, such as secret keys but also the current value of the initialization vector, which changes at every block, are recorded.

One differentiates current and pending writing and reading states. In pending state, new parameters are negotiated during the handshake and during transmission. Thereafter, the protocol goes into a current state in which the records of the applications are processed.

Algorithms implemented in the project:

* **PRF algorithm:**  An algorithm used to generate keys from the master secret
* **MAC algorithm:** An algorithm to be used for message authentication. This specification includes the size of the value returned by the MAC algorithm.

**5. HMAC and PRF**

A number of operations in the TLS record and handshake layer required a keyed MAC. This is a secure digest of some data protected by a secret. Forging the MAC is infeasible without knowledge of the MAC secret. The construction we use for this operation is known as HMAC.

HMAC can be used with a variety of different hash algorithms.

TLS uses it in the handshake with two different algorithms: MD5 and SHA, denoting these as HMAC\_MD5(secret, data) and HMAC\_SHA(secret, data).

In addition, a construction is required to do expansion of secrets into blocks of data for the purposes of key generation or validation.

This pseudo-random function (PRF) takes as input a secret, a seed, and an identifying label and produces an output of arbitrary length.

In order to make the PRF as secure as possible, it uses two hash algorithms in a way which should guarantee its security if either algorithm remains secure.

First, we define a data expansion function, P\_hash(secret, data) which uses a single hash function to expand a secret and seed into an arbitrary quantity of output:

P\_hash(secret, seed) = HMAC\_hash(secret, A(1) + seed) +

HMAC\_hash(secret, A(2) + seed) +

HMAC\_hash(secret, A(3) + seed) + ...

Where + indicates concatenation.

A() is defined as:

A(0) = seed

A(i) = HMAC\_hash(secret, A(i-1))

TLS's PRF is created by splitting the secret into two halves and using one half to generate data with P\_MD5 and the other half to generate data with P\_SHA, then exclusive-ORring the outputs of these two expansion functions together.

S1 and S2 are the two halves of the secret and each is the same length.

S1 is taken from the first half of the secret, S2 from the second half.

Their length is created by rounding up the length of the overall secret divided by two.

Thus, if the original secret is an odd number of bytes long, the last byte of S1 will be the same as the first byte of S2.

L\_S = length in bytes of secret;   
L\_S1 = L\_S2 = ceil(L\_S / 2);

The secret is partitioned into two halves (with the possibility of one shared byte) as described above, S1 taking the first L\_S1 bytes and S2 the last L\_S2 bytes.

The PRF is then defined as the result of mixing the two pseudorandom streams by exclusive- ORring them together.

PRF(secret, label, seed) = P\_MD5(S1, label + seed) XOR P\_SHA-1(S2, label + seed);

**6. Principles of REST:**

* Resources expose easily understood directory structure URIs.
* Representations transfer JSON or XML to represent data objects and attributes.
* Messages use HTTP methods explicitly (for example, GET, POST, PUT, and DELETE).
* Stateless interactions store no client context on the server between requests. State dependencies limit and restrict scalability. The client holds session state.

**6.1 REST Methods:**

**1. GET**

Retrieve information. GET requests must be safe and idempotent, meaning regardless of how many times it repeats with the same parameters, the results are the same.

**2. PUT**

Store an entity at a URI. PUT can create a new entity or update an existing one. A PUT request is idempotent. Idempotency is the main difference between the expectations of PUT versus a POST request

**3. POST**

Request that the resource at the URI do something with the provided entity. Often POST is used to create a new entity, but it can also be used to update an entity.

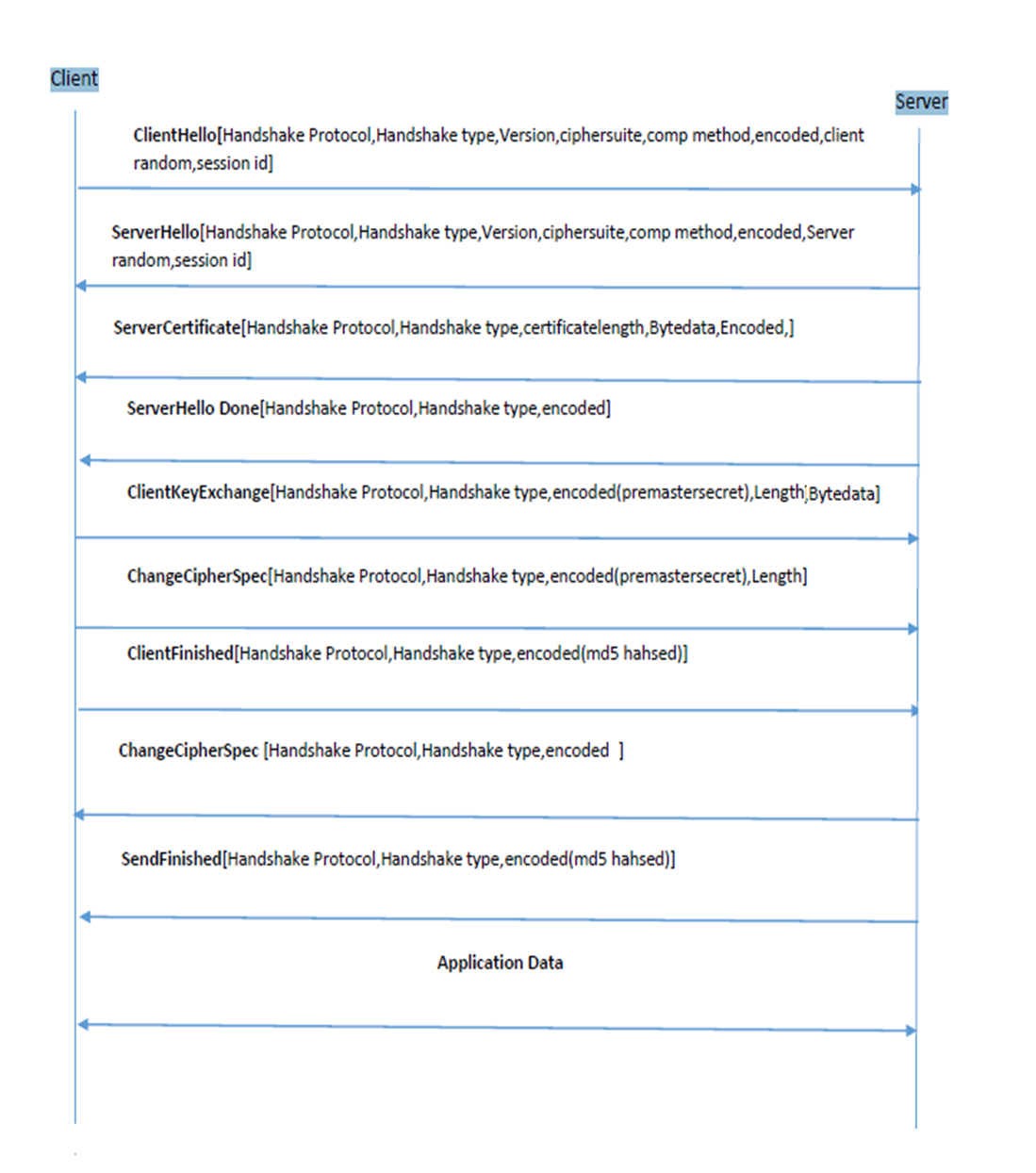
**4. DELETE**

Request that a resource be removed; however, the resource does not have to be removed immediately. It could be an asynchronous or long-running request.

**6.2. JAX-RS Application, Resources and Sub-Resources:**

Root resource classes are POJOs (Plain Old Java Objects) that are annotated with @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] have at least one method annotated with @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] or a resource method designator annotation such as @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ GET.html], @PUT [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PUT.html], @POST [http:// jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/POST.html], @DELETE [http://jax-rs-spec.java.net/ nonav/2.0/apidocs/javax/ws/rs/DELETE.html]. Resource methods are methods of a resource class annotated with a resource method designator.

**7. MSC of Implementation:**

****

**8.** **Java Code Details:**

This project is implemented in **java** inclined with **REST** architecture (GET, PUT, POST, DELETE) and messages are sent as json format.

Main libraries we are using here is **jersey and jackson.**

**8.1.** **TLS Client (mobile.computing.ws1819.client):**

**Classes:**

**1. StartRestClient.java**

This class contains the main method, and a port is created here to process the server’s requests, and to initiate handshake process. This class also creates objects of Class RsaClient.java, RecordLayer.java, and PRF.java.

* **Methods in this class:**
* *doPostRequestClientHello() -* Send Client Hello Message
* *doPostSendClientKeyExchange() -* Send Client Key
* *doPostsendChangeCipherSpec() -* Send Change Cipher Spec Message
* *doPostsendFinished() -* Send Client Finished
* *getRandom() -* Generates random byte array
* *generateMasterSecret() -* Generate a master secret from the given pre Master Secret and store it in Security Parameters
* *generateKeys() -* Generate read and write keys for the Record layer using the Master Secret stored in Security Parameters.
* *updateHashes()-* Private Method for MD5 and SHA hashing

**2. TlsHeadersClient.java**

This class has just variables defined according to RFC which are static.

**3. ClientMessageResouce.java**

This is a root resource class which services the requests from server based on the annotations.

* **Methods in this class:**
* *cipherSpec() -* Changes the cipher spec
* *createMessage() -* Process the server hello.
* *serverCertificate() -* Extract the public key and other information.
* *serverHelloDone()* - Process the server hello done and initiate key exchange.
* *servFinished() -* To process servers read finished message and verify the handshake.
* *appData() -* To process the Application Data message sent by the server

**4. RsaClient.java**

Used to extract the information from certificate and server’s public key and to do RSA encryption.

**5. RecordLayerClient.java**

* This class is used to send and receive the application data which encrypts and decrypts respectively, and also to set key block for pending states.
* **Methods in this class:**
* *sendHandshakeMessage() -* Switch case to send the respective post requests based on handshakes
* *changeClientWriteState()-*Promote the pending write state to be the current state.
* *changeServerWriteState()-*Promote the pending read state to be the current state.
* *readRecord() -* Decrypt message from server.
* *sendMessage() -* Encrypt and send message.
* *setKeyBlock() -* Set key block for pending states.
* *sendHandshakeMessage() -* Send the Handshake message based on type to server.
* *getMAC( ) - C*oncatenate all values to be MACed.
* *long2ByteArray()-*Converting long to Byte Array.

**8.2 TLS Server (mobile.computing.ws1819.server):**

**Classes:**

**1. StartRestServer.java**

This class contains the main method, and a port 8080 is created here to process the client’s requests.

**2. ServerMessageResouce.java**

This is a root resource class which services the requests from client based on the annotations.

* **Methods in this class:**
* *loadCertificateFromFile() -* To load certificate and read as byte and encode.
* *getRandom() -* To generate server random numbers for serverRandom field.
* *generateMasterSecret() -* Generate a master secret from the given pre Master Secret and store it in Security Parameters.
* *generateKeys() -* Generate read and write keys for the Record layer using the Master Secret stored in Security Parameters.
* *updateHashes() -* For verification at the end of finished message.
* *appData() -* Handles the application data.
* *changeCipherSpec() -*Changes the cipher spec.
* *clientKeyExchange() -* Generate master and premaster secret based on client.
* *createMessage() -* Process the client hello and send server hello.
* *readFinished() -* To process clients read finished message and verify the handshake.

**3. TlsHeadersClient.java**

This class has just variables defined according to RFC which are static.

**4. RecordLayerServer.java**

This class is used to send and receive the application data which encrypts and decrypts respectively, and also to set key block for pending states.

* **Methods in this class:**
* *changeClientWriteState() -* Promote the pending write state to be the current state.
* *readHandshakeMessage() –* To receive the handshake messages and pass it without needing to decrypting.
* *changeServerWriteState() -* Promote the pending read state to be the current state.
* *readRecord() -* Decrypt message from client.
* *sendMessage() -* Encrypt and send message.
* *setKeyBlock()* - Set key block for pending states.
* *sendHandshakeMessage()-* Send the Handshake message based on type to client.
* *getMAC( ) - C*oncatenate all values to be MACed.
* *long2ByteArray() -* Converting long to Byte Array.

**5. RsaServer.java**

Uses the private key to decrypt the message which it receives from client.

* **Methods in this class:**
  + - *decrypt()-* To perform decryption using private Key.
    - *getPrivate()* - Decode the pkcs8 encoded private key.

**8.3. Common Classes for both server and client(mobile.computing.ws1819)**

* **ApplicationData.java** - Creating an object to hold the details to be sent in Application Data.
* **ChangeCipherSpec.java** - Creating an object to hold the details to be sent in Change Cipher Spec message.
* **ClientKeyExchange.java** - Creating an object to hold the details to be sent in Client Key Exchange message.
* **FileLoad.java** - To help in loading file and processing.
* **HMAC.java** - Verify both the data integrity and the authentication of a message, PRF.java.
* **SendFinished.java** - Creating an object to hold the details to be sent in Send Finished message.
* **ServerCertificate.java** - Creating an object to hold the details to be sent in Server Certificate message.
* **ServerHelloDone.java** - Creating an object to hold the details to be sent in Server Hello Done message.
* **Tls.java** - Creating an object to hold the details to be sent in hello message.
* **TlsException**.**java** - All the Exceptions of the project are handled by this class.
* **TlsHeaders**.**java** - All headers for client and variables for client are stored here.

**9. Certificate Generation:**

OpenSSL is used to generate the self-signed certificate and encrypt the private key in pkcs8 format.

* **Command to generate X509 certificate:**

*req -x509 -md5 -nodes -days 365 -newkey rsa:512 -keyout privateKey.key -out Certificate.cer*

* **Command to encrypt in pkc8 format:**

*pkcs8 -topk8 -inform PEM -outform DER -in privatekey.key -nocrypt -out privatekey.pkcs8*

**Certificate:** *Certificate.cer*

**Private Key:** *privatekey.pkcs8*

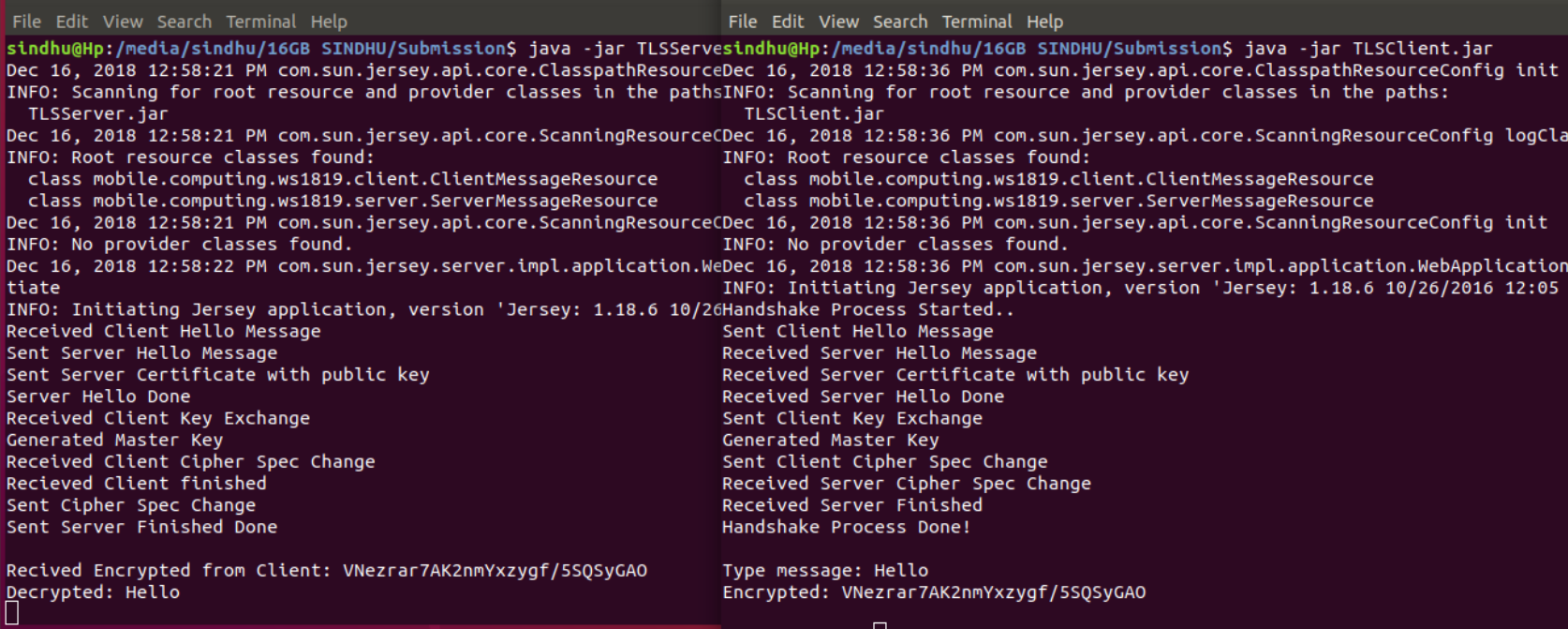
**10. Steps to Run the Emulator:**

The Jar files for Client and Server is generated as follows:

In eclipse Click on File Tab > Export > Java > Runnable Jar File > Next > Under **Launch Configuration** dropdown menu select the StartRestClient Main file of the project for client and StartRestServer Main file of the project for server for which the jar has to be generated and Under **Export Destination** give the path for the Jar file to be stored and Under **Library handling** select ‘Extract required libraries into generated Jar’ > Finish.

1. This requires java version 8.
2. Start wireshark and capture on loopback interface
3. Open terminal in Linux or cmd in windows
4. Change directory to where the **TLSServer**.jar and **TLSClient**.jar files are located.
5. First Server should be run and then Client.
6. Type: **java –jar TLSServer.jar** and hit enter
7. Open one more terminal
8. Type: **java –jar TLSClient.jar** and Hit Enter. When Prompted type messages.

**11. Results from terminal:**

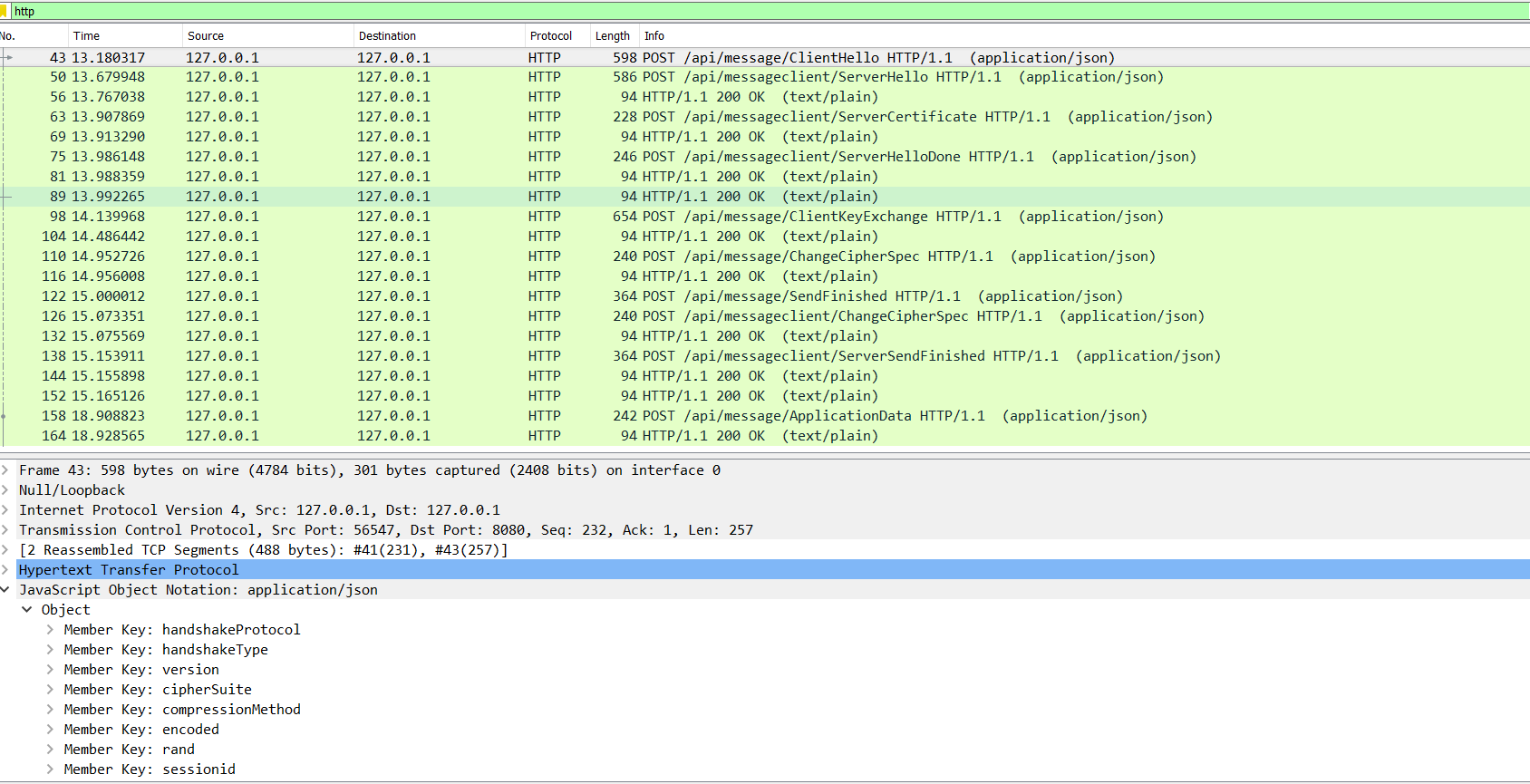


**12.** **Results and captures from Wireshark:**

The project is run in two terminal, one for client and another as server. The wireshark is started on loopback interface to capture the messages between the client and server transaction.

Apply Http filter in wireshark to filter the Http protocol.

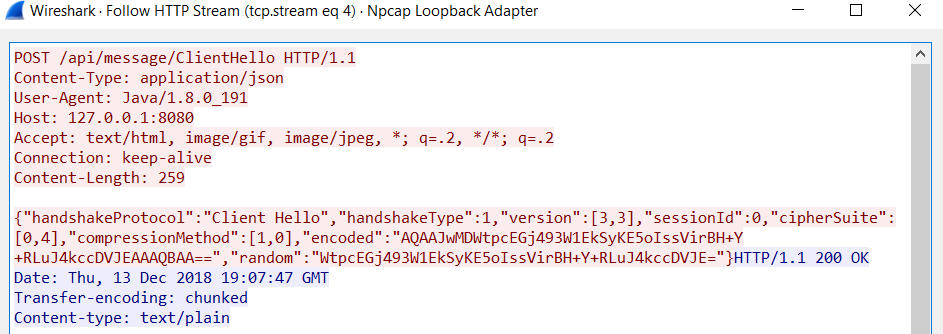
*The below capture shows the handshake process and application data exchange between server and client***.**



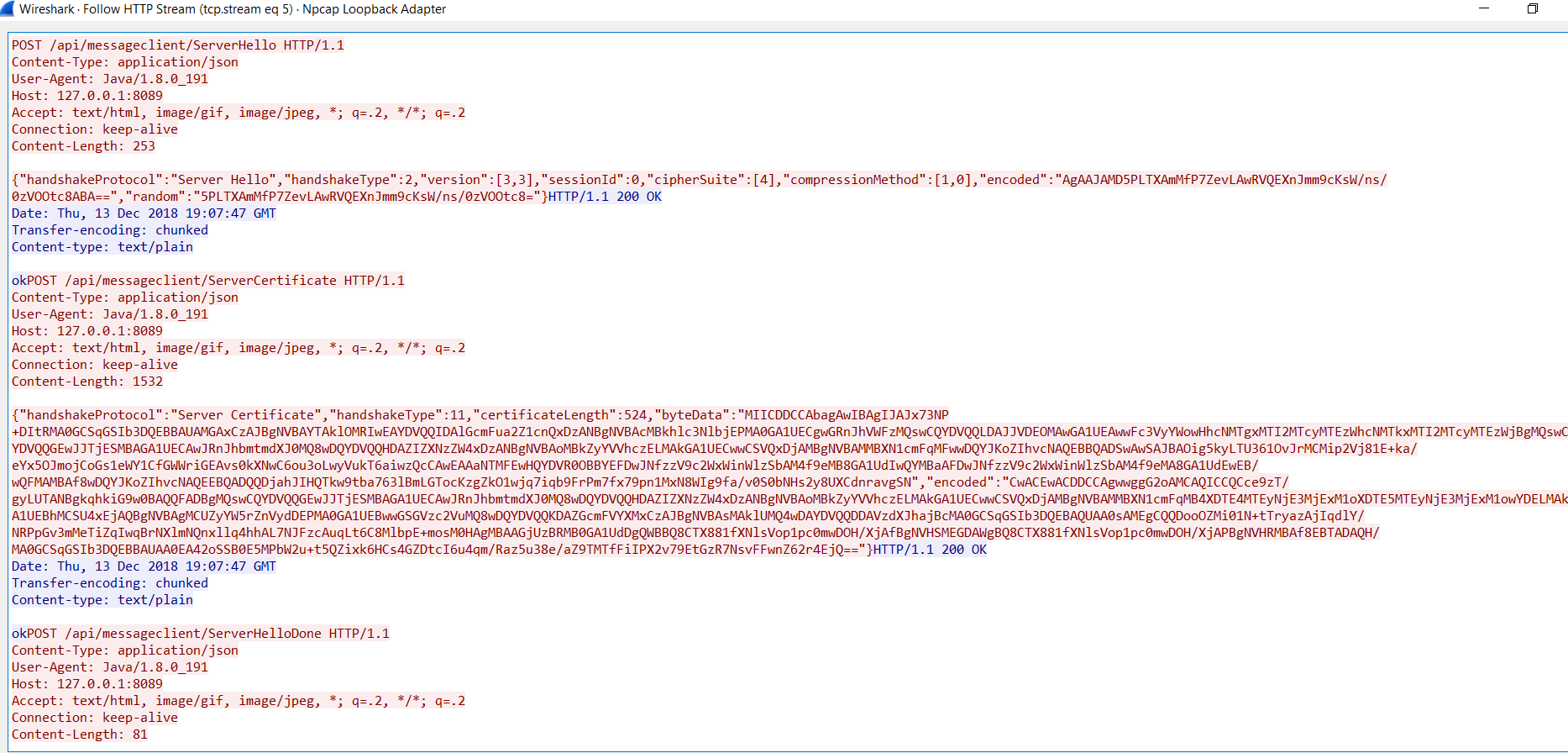
In order to view the json messages:

Right Click on a packet > Follow > HTTP Stream

**The messages are sent in JSON format.**



*Fig: Post message from client to server, Content:ClientHello*



*Fig: Post message from server to client, Content: ServerHello, ServerCertificate,ServerHelloDone*

*From the above captures we can see the request and responses.*

**13. References:**

1. <https://docs.microsoft.com/en-us/windows/desktop/secauthn/tls-handshake-3.protocol>
2. https://tools.ietf.org/html/rfc5246
3. http://kodu.ut.ee/~b04866/tls.pdf
4. https://www.ipa.go.jp/security/rfc/RFC2246-05EN.html