David LE, Mehdi BOULHARTS, 3A RTS
Supervisor: Linda SENIGAGLIESI

TP5 – Java for networks

2 Practical Exercises

2.1 Exercise 1: Setting Up SSL/TLS Environment

2.1.1 Task 1.1: Generate Self-Signed Certificate

We wanted to create a server identity (private key + certificate) that can be used to authenticate the SSL/TLS server and encrypt the communication.

Using the keytool -genkeypair command, we generated a 2048-bit RSA key pair and a self-signed certificate for CN=localhost, OU=Development, O=MyCompany, L=City, ST=State, C=FR. The key and certificate were stored in a server.jks file, which is actually a PKCS#12 keystore protected by a password. We then ran keytool -list -v -keystore server.jks to inspect the entry (alias myserver), the owner/issuer fields and the validity dates.

We obtained a PKCS#12 keystore (server.jks) containing one PrivateKeyEntry with an RSA 2048-bit key and a self-signed certificate signed with SHA256withRSA. The certificate is valid from 3 December 2025 to 3 December 2026 and will be used by our SSL/TLS server during the TLS handshake.

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % keytool -genkeypair -alias myserver -keyalg RSA -keysize 2048 -validity 365 -keystore server.jks
Entrez le mot de passe du fichier de clés :
Ressaisissez le nouveau mot de passe :
Quels sont vos nom et prénom ?
  [Unknown]:  localhost
Quel est le nom de votre unité organisationnelle ?
  [Unknown]:  Development
Quel est le nom de votre entreprise ?
  [Unknown]:  MyCompany
Quel est le nom de votre ville de résidence ?
  [Unknown]:  City
Quel est le nom de votre état ou province ?
  [Unknown]:  State
Quel est le code pays à deux lettres pour cette unité ?
  [Unknown]:  FR
Est-ce CN=localhost, OU=Development, O=MyCompany, L=City, ST=State, C=FR ?
  [non]:  oui

Génération d'une paire de clés RSA de 2 048 bits et d'un certificat auto-signé (SHA256withRSA) d'une validité de 365 jours
        pour : CN=localhost, OU=Development, O=MyCompany, L=City, ST=State, C=FR
```

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % keytool -list -v -keystore server.jks
Entrez le mot de passe du fichier de clés :
Type de fichier de clés : PKCS12
Fournisseur de fichier de clés : SUN

Votre fichier de clés d'accès contient 1 entrée

Nom d'alias : myserver
Date de création : 3 déc. 2025
Type d'entrée : PrivateKeyEntry
Longueur de chaîne du certificat : 1
Certificat[1]:
Propriétaire : CN=localhost, OU=Development, O=MyCompany, L=City, ST=State, C=FR
Emetteur : CN=localhost, OU=Development, O=MyCompany, L=City, ST=State, C=FR
Numéro de série : c32633230a004a99
Valide du Wed Dec 03 14:38:25 CET 2025 au Thu Dec 03 14:38:25 CET 2026
Empreintes du certificat :
         SHA 1: 1D:1E:E9:6A:A8:72:5F:2F:20:56:0D:22:1C:40:57:95:83:66:9C:E3
         SHA 256: 82:DC:F9:B5:C5:49:5A:D2:F8:D9:CC:B8:FD:CA:95:C9:1F:7B:FB:E3:89:44:51:A3:8C:CE:AE:61:19:C3:37:39
Nom de l'algorithme de signature : SHA256withRSA
Algorithme de clé publique du sujet : Clé RSA 2048 bits
Version : 3

Extensions :

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: E2 09 A3 B2 52 5D 5F 4E   2D 75 EC 50 76 31 DF 71  ....R]_N-u.Pv1.q
0010: 71 EC 23 2E                                        q.#.
]
]



*******************************************
*******************************************
```

2.1.2 Task 1.2: Understand Certificate Contents

1. What is the certificate's owner name ?
   => The owner is: CN=localhost, OU=Development, O=MyCompany, L=City, ST=State, C=FR.
2. Who issued this certificate ?
   => It is a self-signed certificate, so the issuer is the same as the owner: CN=localhost, OU=Development, O=MyCompany, L=City, ST=State, C=FR.
3. What encryption algorithm was used ?
   => The key pair uses the RSA algorithm with a 2048-bit key.
4. When does the certificate expire ?
   => The certificate is valid for 365 days from the creation date.
   The exact expiration date is shown in the Valid from ... until ... field.

2.2 Exercise 2: SSL Server Implementation

2.2.4 Testing Your Implementation

Here, the objective was to check that the SSL server correctly presents its certificate and can establish an encrypted TLS session with an external client (OpenSSL), independently from our Java client.

We started SSLTCPServer on port 8443, then used openssl s_client -connect localhost:8443 from the terminal. On the client screenshot, we can see the server certificate details and the negotiated TLS version/cipher. On the server screenshot, we see the new connection, the successful TLS handshake and the messages received/echoed.

OpenSSL successfully established a TLSv1.3 session with the server and displayed our self-signed certificate (CN=localhost, RSA 2048 bits, SHA256withRSA). The verification warning “self-signed certificate” is expected in this lab context. The echo protocol worked correctly: all lines sent from the OpenSSL client were received, tagged and echoed by the server, and the /quit command closed the TLS connection cleanly.

server:

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out SSLTCPServer
[16:37:31] SSL server socket created on port 8443
[16:37:31] SSLTCPServer(port=8443) starting...
[16:37:37] New SSL connection from 0:0:0:0:0:0:0:1 (client #1)
[16:37:37] Handling SSL client #1 from 0:0:0:0:0:0:0:1
[16:37:37] TLS handshake successful with client #1 (0:0:0:0:0:0:0:1)
[#1 0:0:0:0:0:0:0:1] hello
[#1 0:0:0:0:0:0:0:1] I am client 1 (openssl)
[16:38:39] SSL client #1 (0:0:0:0:0:0:0:1) disconnected
```

client:

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % openssl s_client -connect localhost:8443
CONNECTED(00000005)
Can't use SSL_get_servername
depth=0 C = FR, ST = State, L = City, O = MyCompany, OU = Development, CN = localhost
verify error:num=18:self-signed certificate
verify return:1
depth=0 C = FR, ST = State, L = City, O = MyCompany, OU = Development, CN = localhost
verify return:1
---
Certificate chain
 0 s:C = FR, ST = State, L = City, O = MyCompany, OU = Development, CN = localhost
   i:C = FR, ST = State, L = City, O = MyCompany, OU = Development, CN = localhost
   a:PKEY: rsaEncryption, 2048 (bit); sigalg: RSA-SHA256
   v:NotBefore: Dec  3 13:38:25 2025 GMT; NotAfter: Dec  3 13:38:25 2026 GMT
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDeDCCAmCgAwIBAgIJAMMmMyMKAEqZMA0GCSqGSIb3DQEBCwUAMGoxCzAJBgNV
BAYTAkZSMQ4wDAYDVQQIEwVTdGF0ZTENMAsGA1UEBxMEQ2l0eTESMBAGA1UEChMJ
TXlDb21wYW55MRQwEgYDVQQLEwtEZXZlbG9wbWVudDESMBAGA1UEAxMJbG9jYWxo
b3N0MB4XDTI1MTIwMzEzMzgyNVoXDTI2MTIwMzEzMzgyNVowajELMAkGA1UEBhMC
RlIxDjAMBgNVBAgTBVN0YXRlMQ0wCwYDVQQHEwRDaXR5MRIwEAYDVQQKEwlNeUNv
bXBhbnkxFDASBgNVBAsTC0RldmVsb3BtZW50MRIwEAYDVQQDEwlsb2NhbGhvc3Qw
ggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCOzap7OTFzoJzJ/P46dfqz
UwgWOYMEE4JanqYybECV1ivCPXlDd/GKlcltEYknYTs54ollK568LW2TfNeRepd5
YGSKYsDVKNRiWJYp4f5OraWAmyzeot1h6iq//yPO+KFWmIGGiTZAUMXStCsBuiZu
BCE9ZiVapg7KSgTjfrrdErqazv77yLZZBD/xCcp7/Z+SkS/0+bGNAxfRBnLgrfvb
9KM4NnWfPQmB5brtpD8spLQNVWCm1Cdp8yug0NS///UqqNEP599tuMPmw0pLwseN
eehLs0VbOa15Z4/dCu2LX90AM19B/VoNC3RKunm0B4aODXMaJeMQAeco+V/Yb+jZ
AgMBAAGjITAfMB0GA1UdDgQWBBTiCaOyU11fTi117FB2Md9xcewjLjANBgkqhkiG
9w0BAQsFAAOCAQEAL1NvZotO5yHEu0Ok2bJAQRD2Yc9Cnu0YHdx6vnZCDU6p/SWX
nUWtj5LBlpB9cltdR19MWNlTAFJCxBsr44KF7Bm8GPkAPkHkhdpnNM1VJ6UG2Q1v
MOSYQJIuOflebovP494wSUHSOVbeQ12Ajq4P7UZfXzkb2wns6LL2lLPqw2nial7W
1iXpLrHuWdgrZhX+Dr2JZqEqGLA3xu4R6yPJylZAaNQlfj+UxY/EZuzFoHuHd+Q8
beDtNHvt8WSOAhzebb/sNQ3+S17yxTMm+n2tu5iOPU4J7haxcBDyBu70q+X73irF
HqPA4bTjw808IWea3uWQlRTr+nKZgftH2Di+YQ==
-----END CERTIFICATE-----
subject=C = FR, ST = State, L = City, O = MyCompany, OU = Development, CN = localhost
issuer=C = FR, ST = State, L = City, O = MyCompany, OU = Development, CN = localhost
---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA-PSS
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 1538 bytes and written 377 bytes
Verification error: self-signed certificate
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 18 (self-signed certificate)
---
---
Post-Handshake New Session Ticket arrived:
SSL-Session:
    Protocol  : TLSv1.3
    Cipher    : TLS_AES_256_GCM_SHA384
    Session-ID: 5D14B17A9431667C0CF8B216315EB9B45C83968B2B2693EAD679E5734699E135
    Session-ID-ctx:
    Resumption PSK: B8FB15B22153DB1329A2110B865E0C689F05DB1E2792DD6E03BC85B70867A49DD7780F07F50856196B3AB66ED7CE3F5D
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 86400 (seconds)
    TLS session ticket:
    0000 - 7b 48 1c 97 60 1d c5 97-5f 46 3d b5 b6 56 bb 8e   {H..`..._F=..V..
    0010 - 1c 6e 07 4c 6e e8 fb 94-a5 b0 ec 0b d8 c8 21 90   .n.Ln.........!.
    0020 - 50 b4 cf 69 82 9d 2a 4c-2e e4 ba 88 d4 82 d5 7a   P..i..*L.......z
    0030 - 40 04 c0 9c 00 b2 b0 e5-01 0f 83 26 a0 bd d6 ad   @..........&....
    0040 - d4 92 6a 14 60 19 a0 c3-e8 d4 9f 4c 53 f8 9e 12   ..j.`......LS...
    0050 - 88 26 03 1e 25 c2 32 e1-20 f6 42 0b a4 d3 cb 12   .&..%.2. .B.....
    0060 - 4e e1 85 13 4e a0 ec cf-78 c3 73 c0 8f 6e 9b 31   N...N...x.s..n.1

    Start Time: 1764776257
    Timeout   : 7200 (sec)
    Verify return code: 18 (self-signed certificate)
    Extended master secret: no
    Max Early Data: 0
---
read R BLOCK
Welcome (over TLS)! You are secure client #1
hello
[#1 0:0:0:0:0:0:0:1] hello
I am client 1 (openssl)
[#1 0:0:0:0:0:0:0:1] I am client 1 (openssl)
/quit
Goodbye secure client #1
closed
```

2.3 Exercise 3: SSL Client Implementation

2.3.3 Implementation Tasks

The aim of this part was to evaluate how our SSL client behaves with different trust settings when connecting to the local SSLTCPServer that uses a self-signed certificate.

We started SSLTCPServer on port 8443 and tested two modes of SSLClient:

- trustAll = true: java -cp out SSLClient localhost 8443 true
  The client printed a warning about the “trust-all” context, connected to the server and we exchanged a few messages ("hi, I am the SSL client (client 2)", "thanks", "bye").
- trustAll = false: java -cp out SSLClient localhost 8443 false
  This time, the TLS handshake failed. On the server side we saw a log for client #3 with the message “TLS handshake failed … Received fatal alert: certificate_unknown”.

With trustAll = true, the client successfully completed the TLS handshake with the self-signed certificate and all messages were correctly echoed over TLS. With trustAll = false, the connection was rejected because the self-signed certificate is not trusted by the default CA store. This clearly shows the difference between the testing mode (accept any certificate) and the normal validation mode (reject untrusted self-signed certificates).

server:

```
[16:47:30] New SSL connection from 127.0.0.1 (client #2)
[16:47:30] Handling SSL client #2 from 127.0.0.1
[16:47:30] TLS handshake successful with client #2 (127.0.0.1)
[#2 127.0.0.1] hi, I am the SSL client (client 2)
[#2 127.0.0.1] thanks
[#2 127.0.0.1] bye
[16:59:14] SSL client #2 (127.0.0.1) disconnected
```

```
[17:03:25] New SSL connection from 127.0.0.1 (client #3)
[17:03:25] Handling SSL client #3 from 127.0.0.1
[17:03:25] TLS handshake failed with client #3 (127.0.0.1): Received fatal alert: certificate_unknown
[17:03:25] SSL client #3 (127.0.0.1) disconnected
```

client:

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out SSLClient localhost 8443 false
SSL client error: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target
javax.net.ssl.SSLHandshakeException: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target
        at java.base/sun.security.ssl.Alert.createSSLException(Alert.java:131)
        at java.base/sun.security.ssl.TransportContext.fatal(TransportContext.java:383)
        at java.base/sun.security.ssl.TransportContext.fatal(TransportContext.java:326)
        at java.base/sun.security.ssl.TransportContext.fatal(TransportContext.java:321)
        at java.base/sun.security.ssl.CertificateMessage$T13CertificateConsumer.checkServerCerts(CertificateMessage.java:1294)
        at java.base/sun.security.ssl.CertificateMessage$T13CertificateConsumer.onConsumeCertificate(CertificateMessage.java:1169)
        at java.base/sun.security.ssl.CertificateMessage$T13CertificateConsumer.consume(CertificateMessage.java:1112)
        at java.base/sun.security.ssl.SSLHandshake.consume(SSLHandshake.java:396)
        at java.base/sun.security.ssl.HandshakeContext.dispatch(HandshakeContext.java:481)
        at java.base/sun.security.ssl.HandshakeContext.dispatch(HandshakeContext.java:459)
        at java.base/sun.security.ssl.TransportContext.dispatch(TransportContext.java:206)
        at java.base/sun.security.ssl.SSLTransport.decode(SSLTransport.java:172)
        at java.base/sun.security.ssl.SSLSocketImpl.decode(SSLSocketImpl.java:1510)
        at java.base/sun.security.ssl.SSLSocketImpl.readHandshakeRecord(SSLSocketImpl.java:1425)
        at java.base/sun.security.ssl.SSLSocketImpl.startHandshake(SSLSocketImpl.java:455)
        at java.base/sun.security.ssl.SSLSocketImpl.startHandshake(SSLSocketImpl.java:426)
        at SSLClient.connect(SSLClient.java:79)
        at SSLClient.main(SSLClient.java:139)
Caused by: sun.security.validator.ValidatorException: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target
        at java.base/sun.security.validator.PKIXValidator.doBuild(PKIXValidator.java:439)
        at java.base/sun.security.validator.PKIXValidator.engineValidate(PKIXValidator.java:306)
        at java.base/sun.security.validator.Validator.validate(Validator.java:264)
        at java.base/sun.security.ssl.X509TrustManagerImpl.checkTrusted(X509TrustManagerImpl.java:231)
        at java.base/sun.security.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:132)
        at java.base/sun.security.ssl.CertificateMessage$T13CertificateConsumer.checkServerCerts(CertificateMessage.java:1278)
        ... 13 more
Caused by: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target
        at java.base/sun.security.provider.certpath.SunCertPathBuilder.build(SunCertPathBuilder.java:148)
        at java.base/sun.security.provider.certpath.SunCertPathBuilder.engineBuild(SunCertPathBuilder.java:129)
        at java.base/java.security.cert.CertPathBuilder.build(CertPathBuilder.java:297)
        at java.base/sun.security.validator.PKIXValidator.doBuild(PKIXValidator.java:434)
        ... 18 more
Disconnected from SSL server.
```

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out SSLClient localhost 8443 true
[WARNING] Trust-all SSL context enabled (TESTING ONLY!)
Connected to SSL server localhost:8443
Type messages to send over TLS. Use /quit to exit.
> hi, I am the SSL client (client 2)
Welcome (over TLS)! You are secure client #2
> thanks
[#2 127.0.0.1] hi, I am the SSL client (client 2)
> bye
[#2 127.0.0.1] thanks
> quit
User requested to quit.
Disconnected from SSL server.
```

3 Custom Protocol Design

3.2 Exercise 5: Protocol Message Design

We wanted to implement a portable serialization format for our ChatMessage objects, suitable for transmission over a TLS stream, and to reconstruct messages safely on the receiver side.

We implemented a toBytes() method in ChatMessage that converts the object into a length-prefixed JSON message: first a 4-byte big-endian integer indicating the JSON payload size, then the UTF-8 encoded JSON string. The reverse operation is implemented in fromBytes(byte[]), which reads the length, validates it against the actual buffer size, decodes the JSON string in UTF-8 and parses the fields (type, version, sender, recipient, room, content, timestamp) to build a new ChatMessage instance.

Using a small test main() method, we verified a complete serialization/deserialization round-trip: the decoded message matches the original one. This confirms that our protocol messages can be safely transformed to and from byte arrays, with proper framing and UTF-8 handling, and that basic length validation is in place to detect malformed messages.

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out ChatMessage
Original: ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T08:36:54.251587Z, sender='alice', recipient='null', room='general', content='Hello JSON over TLS'}
Decoded : ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T08:36:54Z, sender='alice', recipient='null', room='general', content='Hello JSON over TLS'}
```

3.3 Exercise 6: Protocol Server Implementation

In this part, the goal was to turn our SSL echo server into a secure multi-client chat server using the ChatMessage protocol over TLS, with user login and room-based broadcasting.

We implemented SecureChatServer on port 8444, loading server.jks and accepting TLS connections via SSLServerSocket. The server reads length-prefixed ChatMessage objects, keeps a map of active users and chat rooms, and expects a LOGIN_REQUEST followed by JOIN_ROOM_REQUEST and TEXT_MESSAGE. Using ProtocolTestClient, two users (mehdi and david) connected to localhost:8444, joined the "general" room and each sent a text message (“Hello from mehdi”, “Hello from david”).

The server logs show two successful TLS handshakes, User logged in: mehdi/david and User X joined room general. Client 1 (mehdi), already connected, correctly received the server message announcing that david joined the room, as well as the TEXT_MESSAGE sent by client 2 (“Hello from david”). Each client also saw its own message, which confirms that the secure chat server correctly manages TLS sessions and broadcasts messages to all clients in the same room.

server:

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out SecureChatServer
[11:03:20] SecureChatServer listening on port 8444
[11:03:25] New connection from 127.0.0.1
[11:03:25] TLS handshake successful with 127.0.0.1
[11:03:25] User logged in: mehdi
[11:03:25] User mehdi joined room general
[11:03:46] New connection from 127.0.0.1
[11:03:47] TLS handshake successful with 127.0.0.1
[11:03:47] User logged in: david
[11:03:47] User david joined room general
```

client 1 (mehdi):

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out ProtocolTestClient localhost 8444 mehdi general
Connected to 127.0.0.1:8444
Sent LOGIN_REQUEST as mehdi
Received: ChatMessage{type=LOGIN_RESPONSE, version='1.0', timestamp=2025-12-05T10:03:25Z, sender='server', recipient='mehdi', room='null', content='Welcome mehdi!'}
Sent JOIN_ROOM_REQUEST for room 'general'
Received: ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T10:03:25Z, sender='server', recipient='null', room='general', content='mehdi joined the room.'}
Sent TEXT_MESSAGE to room 'general'
Received: ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T10:03:25Z, sender='mehdi', recipient='null', room='general', content='Hello from mehdi'}
Received: ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T10:03:47Z, sender='server', recipient='null', room='general', content='david joined the room.'}
Received: ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T10:03:47Z, sender='david', recipient='null', room='general', content='Hello from david'}
```

client 2 (david):

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out ProtocolTestClient localhost 8444 david general
Connected to 127.0.0.1:8444
Sent LOGIN_REQUEST as david
Received: ChatMessage{type=LOGIN_RESPONSE, version='1.0', timestamp=2025-12-05T10:03:47Z, sender='server', recipient='david', room='null', content='Welcome david!'}
Sent JOIN_ROOM_REQUEST for room 'general'
Received: ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T10:03:47Z, sender='server', recipient='null', room='general', content='david joined the room.'}
Sent TEXT_MESSAGE to room 'general'
Received: ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T10:03:47Z, sender='david', recipient='null', room='general', content='Hello from david'}
```

4 Testing and Validation

4.1 Exercise 7: Security Testing

4.1.1 SSL/TLS Configuration Testing

For this part, we did not introduce new tests but reused the TLS experiments from section 2. Using openssl s_client on port 8443, we confirmed that the server negotiates TLSv1.3 with our self-signed certificate, and that certificate validation correctly reports error code 18 (“self-signed certificate”). With the Java SSL client in “production” mode (certificate validation enabled), the handshake fails as expected with the untrusted certificate, while it succeeds in “testing” mode. Finally, we roughly measured the handshake time with time openssl s_client, which shows that the TLS setup overhead on localhost is very small. (See previous screenshots in section 2 for detailed outputs.)

server:

```
[14:02:13] New SSL connection from 0:0:0:0:0:0:0:1 (client #2)
[14:02:13] Handling SSL client #2 from 0:0:0:0:0:0:0:1
[14:02:13] TLS handshake successful with client #2 (0:0:0:0:0:0:0:1)
[14:02:13] SSL client #2 (0:0:0:0:0:0:0:1) disconnected
```

client:

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % time openssl s_client -connect localhost:8443 </dev/null
CONNECTED(00000005)
Can't use SSL_get_servername
depth=0 C = FR, ST = State, L = City, O = MyCompany, OU = Development, CN = localhost
verify error:num=18:self-signed certificate
verify return:1
depth=0 C = FR, ST = State, L = City, O = MyCompany, OU = Development, CN = localhost
verify return:1
```

```
SSL handshake has read 1538 bytes and written 377 bytes
Verification error: self-signed certificate
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 18 (self-signed certificate)
---
DONE
openssl s_client -connect localhost:8443 < /dev/null  0,02s user 0,01s system 21% cpu 0,128 total
```

4.1.2 Protocol Security Testing

server:

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out SecureChatServer
[10:57:35] SecureChatServer listening on port 8444
[10:57:40] New connection from 127.0.0.1
[10:57:41] TLS handshake successful with 127.0.0.1
[10:57:41] User logged in: mehdi
[10:57:41] User mehdi joined room general
[10:57:52] New connection from 127.0.0.1
[10:57:52] TLS handshake successful with 127.0.0.1
[10:57:52] Login failed, closing connection from 127.0.0.1
```

client mehdi:

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out ProtocolTestClient localhost 8444 mehdi general interactive
Connected to 127.0.0.1:8444
Mode = interactive
Sent LOGIN_REQUEST as mehdi
Received: ChatMessage{type=LOGIN_RESPONSE, version='1.0', timestamp=2025-12-08T09:57:41Z, sender='server', recipient='mehdi', room='null', content='Welcome mehdi!'}
Sent JOIN_ROOM_REQUEST for room 'general'
Received: ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-08T09:57:41Z, sender='server', recipient='null', room='general', content='mehdi joined the room.'}
Interactive mode.
  - plain text       -> message à la room 'general'
  - /msg user text   -> message privé à 'user'
  - /quit            -> quitter proprement
> /msg bob are you there ?
[SENT] ChatMessage{type=PRIVATE_MESSAGE, version='1.0', timestamp=2025-12-08T09:59:20.836713Z, sender='mehdi', recipient='bob', room='null', content='are you there ?'}
>
[INCOMING] ChatMessage{type=ERROR_RESPONSE, version='1.0', timestamp=2025-12-08T09:59:20Z, sender='server', recipient='mehdi', room='null', content='User not found: bob'}
```

fake client:

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out ProtocolTestClient localhost 8444 mehdi general interactive
Connected to 127.0.0.1:8444
Mode = interactive
Sent LOGIN_REQUEST as mehdi
Received: ChatMessage{type=ERROR_RESPONSE, version='1.0', timestamp=2025-12-08T09:57:52Z, sender='server', recipient='null', room='null', content='Username already in use: mehdi'}
Sent JOIN_ROOM_REQUEST for room 'general'
Server closed connection (EOF).
Received: null
Interactive mode.
  - plain text       -> message à la room 'general'
  - /msg user text   -> message privé à 'user'
  - /quit            -> quitter proprement
> Server closed connection (EOF).

[INCOMING] Server closed connection.
```

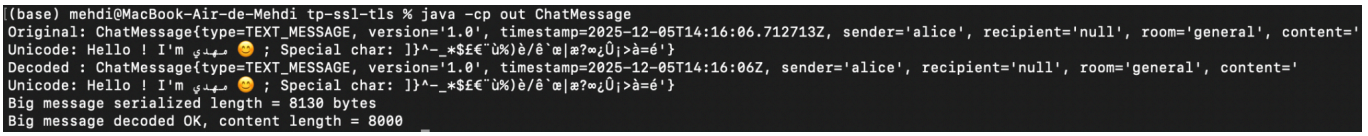
In this section, we validated the robustness and secure error handling of our TLS-protected chat protocol. The confidentiality and integrity of all protocol messages (LOGIN_REQUEST, JOIN_ROOM_REQUEST, TEXT_MESSAGE, etc.) were already demonstrated in Section 3.3, where the multi-client chat scenario runs entirely inside a TLS session, and in 4.1.1 with openssl s_client. All application traffic in 4.1.2 therefore benefits from TLS authenticated encryption; only encrypted records appear on the wire.

We then performed two negative tests whose traces are shown in the 4.1.2 screenshots. First, we tried to open a second client with the same username mehdi while a valid mehdi session was already connected. The server rejected this login by sending an ERROR_RESPONSE (content="Username already in use: mehdi") to the fake client and logging “Login failed, closing connection from 127.0.0.1” before closing the TLS session. Second, from the authenticated mehdi client we sent /msg bob are you there ?, which generates a PRIVATE_MESSAGE targeting a non-existing user. The server replied with another ERROR_RESPONSE (content="User not found: bob") but did not crash and did not expose any stack trace or internal details to the client. Together with the length checks and exception handling already implemented in SecureChatServer, these tests show that invalid or inconsistent protocol operations are cleanly rejected, while all details remain confined to the server logs.

4.2 Exercise 8: Protocol Compliance Testing

4.2.1 Message Format Testing

terminal:



For message format testing, we reused the ChatMessage main method introduced in section 3.2. This test serializes a ChatMessage instance to a length-prefixed JSON/UTF-8 byte array using toBytes(), then reconstructs it with fromBytes(byte[]) and prints both the original and decoded objects. We used a content string containing accents, Arabic characters, an emoji and various special symbols to verify correct UTF-8 handling, and we also generated a large text payload (8000 characters) to check that big messages are serialized and deserialized without errors. In all cases, the decoded message matches the original one (same type, version 1.0, sender, room and content), which confirms that our custom format supports round-trip conversion, Unicode data and basic length validation.

4.2.2 Functional Testing

In this part, we functionally tested the secure chat protocol with several concurrent TLS clients. Using ProtocolTestClient in interactive mode, we started three users (mehdi, other, david) and connected them to SecureChatServer on port 8444. Each client sent a LOGIN_REQUEST followed by a JOIN_ROOM_REQUEST for room "general". The server replied with LOGIN_RESPONSE messages (“Welcome X!”) and broadcast notifications (“X joined the room”) to the room. The server log shows three successful TLS handshakes, logins and room joins, and when each client used /quit the corresponding sessions were closed (“Session closed for user X”), which validates the complete authentication flow and room joining/leaving behaviour.

We then tested private versus global messaging. From client 1 (mehdi), we sent /msg david Hi, this is my secret…, which is encoded as a PRIVATE_MESSAGE with recipient="david". The server delivered this message only to client 3 (david), who replied with another PRIVATE_MESSAGE (“OK, this conv is private”). Client 2 (other) did not see any of these private messages, but all three clients did receive the subsequent global TEXT_MESSAGE broadcasts (“hello everyone”, “hello guys”) to room "general". This confirms that the server correctly routes PRIVATE_MESSAGE point-to-point while TEXT_MESSAGE is broadcast to all users in the same room, fulfilling the functional test objectives.

server:

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out SecureChatServer
[17:04:57] SecureChatServer listening on port 8444
[17:05:00] New connection from 127.0.0.1
[17:05:00] TLS handshake successful with 127.0.0.1
[17:05:00] User logged in: mehdi
[17:05:00] User mehdi joined room general
[17:05:03] New connection from 127.0.0.1
[17:05:03] TLS handshake successful with 127.0.0.1
[17:05:03] User logged in: other
[17:05:03] User other joined room general
[17:05:04] New connection from 127.0.0.1
[17:05:04] TLS handshake successful with 127.0.0.1
[17:05:04] User logged in: david
[17:05:04] User david joined room general
[17:07:00] Client mehdi disconnected.
[17:07:00] Session closed for user mehdi
[17:07:05] Client other disconnected.
[17:07:05] Session closed for user other
[17:07:10] Client david disconnected.
[17:07:10] Session closed for user david
```

client 1 (mehdi):

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out ProtocolTestClient localhost 8444 mehdi general interactive
Connected to 127.0.0.1:8444
Mode = interactive
Sent LOGIN_REQUEST as mehdi
Received: ChatMessage{type=LOGIN_RESPONSE, version='1.0', timestamp=2025-12-05T16:05:00Z, sender='server', recipient='mehdi', room='null', content='Welcome mehdi!'}
Sent JOIN_ROOM_REQUEST for room 'general'
Received: ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:00Z, sender='server', recipient='null', room='general', content='mehdi joined the room.'}
Interactive mode.
  - plain text       -> message à la room 'general'
  - /msg user text   -> message privé à 'user'
  - /quit            -> quitter proprement
>
[INCOMING] ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:03Z, sender='server', recipient='null', room='general', content='other joined the room.'}
>
[INCOMING] ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:04Z, sender='server', recipient='null', room='general', content='david joined the room.'}
> /msg david Hi, this is my secret...
[SENT] ChatMessage{type=PRIVATE_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:29.204474Z, sender='mehdi', recipient='david', room='null', content='Hi, this is my secret...'}
> hello everyone
[SENT] ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:57.638173Z, sender='mehdi', recipient='null', room='general', content='hello everyone'}
>
[INCOMING] ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:57Z, sender='mehdi', recipient='null', room='general', content='hello everyone'}
>
[INCOMING] ChatMessage{type=PRIVATE_MESSAGE, version='1.0', timestamp=2025-12-05T16:06:27Z, sender='david', recipient='mehdi', room='null', content='OK, this conv is private'}
>
[INCOMING] ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:06:36Z, sender='david', recipient='null', room='general', content='hello guys'}
> /quit
User requested to quit.
Client closing socket.

[INCOMING] Receiver stopped: Socket closed
```

client 2 (other):

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out ProtocolTestClient localhost 8444 other general interactive
Connected to 127.0.0.1:8444
Mode = interactive
Sent LOGIN_REQUEST as other
Received: ChatMessage{type=LOGIN_RESPONSE, version='1.0', timestamp=2025-12-05T16:05:03Z, sender='server', recipient='other', room='null', content='Welcome other!'}
Sent JOIN_ROOM_REQUEST for room 'general'
Received: ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:03Z, sender='server', recipient='null', room='general', content='other joined the room.'}
Interactive mode.
  - plain text       -> message à la room 'general'
  - /msg user text   -> message privé à 'user'
  - /quit            -> quitter proprement
>
[INCOMING] ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:04Z, sender='server', recipient='null', room='general', content='david joined the room.'}
>
[INCOMING] ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:57Z, sender='mehdi', recipient='null', room='general', content='hello everyone'}
>
[INCOMING] ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:06:36Z, sender='david', recipient='null', room='general', content='hello guys'}
> /quit
User requested to quit.
Client closing socket.
Server closed connection (EOF).

[INCOMING] Server closed connection.
```

client 3 (david):

```
(base) mehdi@MacBook-Air-de-Mehdi tp-ssl-tls % java -cp out ProtocolTestClient localhost 8444 david general interactive
Connected to 127.0.0.1:8444
Mode = interactive
Sent LOGIN_REQUEST as david
Received: ChatMessage{type=LOGIN_RESPONSE, version='1.0', timestamp=2025-12-05T16:05:04Z, sender='server', recipient='david', room='null', content='Welcome david!'}
Sent JOIN_ROOM_REQUEST for room 'general'
Received: ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:04Z, sender='server', recipient='null', room='general', content='david joined the room.'}
Interactive mode.
  - plain text       -> message à la room 'general'
  - /msg user text   -> message privé à 'user'
  - /quit            -> quitter proprement
>
[INCOMING] ChatMessage{type=PRIVATE_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:29Z, sender='mehdi', recipient='david', room='null', content='Hi, this is my secret...'}
>
[INCOMING] ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:05:57Z, sender='mehdi', recipient='null', room='general', content='hello everyone'}
> /msg mehdi OK, this conv is private
[SENT] ChatMessage{type=PRIVATE_MESSAGE, version='1.0', timestamp=2025-12-05T16:06:27.113806Z, sender='david', recipient='mehdi', room='null', content='OK, this conv is private'}
> hello guys
[SENT] ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:06:36.140561Z, sender='david', recipient='null', room='general', content='hello guys'}
>
[INCOMING] ChatMessage{type=TEXT_MESSAGE, version='1.0', timestamp=2025-12-05T16:06:36Z, sender='david', recipient='null', room='general', content='hello guys'}
> /quit
User requested to quit.
Client closing socket.
Server closed connection (EOF).

[INCOMING] Server closed connection.
```

5 Assignment: Secure Chat System

5.1 Part 1: Secure Server Implementation (50%)

In this assignment, we implemented a secure chat server on port 8444 using Java SSL/TLS and a self-signed certificate stored in server.jks. The server loads the keystore through an SSLContext, performs a TLS handshake with each client, and then handles all application messages with our custom ChatMessage protocol. SecureChatServer manages multiple concurrent users using a thread per connection and shared thread-safe maps for active sessions and chat rooms. The server supports login, room joining, broadcast messages, and private messages, and it logs all important events (TLS handshake, login, room join, disconnection, errors). Error conditions such as unknown users, duplicate usernames or invalid room names are reported to clients as ERROR_RESPONSE messages, while detailed information stays only in the server logs.

5.2 Part 2: Protocol Client Implementation (30%)

In this assignment, we implemented a secure chat server on port 8444 using Java SSL/TLS and a self-signed certificate stored in server.jks. The server loads the keystore through an SSLContext, performs a TLS handshake with each client, and then handles all application messages with our custom ChatMessage protocol. SecureChatServer manages multiple concurrent users using a thread per connection and shared thread-safe maps for active sessions and chat rooms. The server supports login, room joining, broadcast messages, and private messages, and it logs all important events (TLS handshake, login, room join, disconnection, errors). Error conditions such as unknown users, duplicate usernames or invalid room names are reported to clients as ERROR_RESPONSE messages, while detailed information stays only in the server logs.

5.3 Part 3: Testing Documentation (20%)

Security Testing:
We first verified the TLS configuration using openssl s_client on port 8443/8444. The command shows the server certificate (CN=localhost), the validity period, and the negotiated cipher suite (e.g. TLS_AES_256_GCM_SHA384), confirming that all application data is encrypted on the wire. The same command also reports a verification error because the certificate is self-signed, which is expected in this lab setup and is documented as a security limitation. All subsequent chat experiments (sections 3.3, 4.1.1 and 4.1.2) run over this TLS channel, so chat messages are protected by TLS authenticated encryption and cannot be read or modified by a passive network observer.

Protocol Testing:
The ChatMessage class was tested in isolation (section 4.2.1) with several scenarios: (1) a full serialization/deserialization round-trip of a normal text message, (2) a stress test with a long message close to the allowed size limit, and (3) a payload containing Unicode characters and special symbols. In all cases, fromBytes() reconstructed the original message correctly, which validates the length-prefix framing, UTF-8 handling and basic length checks. At the functional level (section 4.2.2), we tested a complete chat scenario with three concurrent clients (mehdi, other, david): each user successfully logged in, joined the generalroom, saw the room-join notifications, exchanged room messages, and used the /msg command to send private messages that were only delivered to the intended recipient. Error cases such as sending a private message to a non-existent user or reusing an already-taken username produced the expected ERROR_RESPONSEon the client and a clear log on the server.

System Limitations and Improvements:
This secure chat system is suitable for a small number of concurrent users, but it has not been stress-tested for high load, so the maximum number of supported clients is not precisely measured. Performance bottlenecks (thread-per-connection model, simple JSON parsing) were not optimized, as the focus of the lab was correctness and security rather than throughput. The client currently uses a “trust-all” SSL context for the self-signed certificate, which simplifies the lab but would be insecure in production; a proper deployment should instead import the server certificate into a dedicated truststore and enable strict certificate validation. Future improvements could include authenticated user accounts with passwords, more advanced room management commands (/rooms, /who), and performance tuning to scale to many more simultaneous users.

6 Troubleshooting Guide

During this lab we encountered several of the typical issues described in the troubleshooting guide. On the TLS side, the main difficulty was the keystore format on macOS: keytool created a PKCS#12 keystore by default, while the TP examples used the .jks extension. We confirmed that Java still loads it correctly through KeyStore.getDefaultType()and validated the certificate with keytool -list and openssl s_client, understanding that the persistent “self-signed certificate” warning is normal in this context. On the application side, most problems came from protocol mistakes: trying to log in twice with the same username, sending messages before logging in, or mis-routing private messages. In each case we relied on server logs, ERROR_RESPONSE messages and Wireshark / terminal traces to identify the cause and adjust the code (for example, introducing a distinct PRIVATE_MESSAGE type and checking activeSessions before sending). These debugging steps helped us better understand how TLS errors, protocol errors and concurrency issues appear in practice and how to handle them cleanly.

Conclusion

In this TP, the goal was to design, implement and test a small secure chat system over TLS. We first set up a TLS server with a self-signed certificate and verified the handshake and cipher suite using openssl s_client, ensuring that all application data would be protected in transit. We then designed a simple JSON-based ChatMessage protocol with length-prefix framing, implemented multi-client handling on the server (SecureChatServer, ChatRoom, ClientSession), and built a protocol-aware client (ProtocolTestClient) capable of login, joining rooms, broadcasting messages and sending private messages.

Through the different test campaigns (echo server, two-client and three-client scenarios, negative tests and serialization tests), we observed that the system behaves as expected: TLS sessions are successfully established, users authenticate with unique usernames, room messages are broadcast only within the correct room, private messages are delivered only to the intended recipient, and invalid operations produce controlled ERROR_RESPONSE messages without crashing the server or exposing internal details. Even though the system is not optimized or production-grade (self-signed certificate, trust-all client, limited scalability), this TP allowed us to go from a basic TCP echo server to a fully working secure chat application, and to understand concretely how TLS, Java keystores, custom wire protocols and multi-client server design fit together in a real networked application.