UNIVERSITY OF PISA

MASTER'S DEGREE IN COMPUTER ENGINEERING

Foundations of Cybersecurity

Bulletin-Board-System

Professor:

Gianluca Dini

Student:

Giovanni Ligato

ACADEMIC YEAR 2023/2024

Index

1. Introduction

The system under review is a Bulletin Board System (*BBS*), designed as a distributed service where users can interact by reading and posting messages. The BBS provides users with the following core functionalities:

- *List*: Displays the latest messages available on the BBS.
- *Get*: Downloads a specific message from the BBS based on the provided message identifier.
- *Add*: Allows users to add a new message to the BBS.

Before using these operations, users must successfully log into the system. A secure channel ensures that all operations are executed safely. Additionally, new users must register before they can log in and access the system. Once logged out, users must log in again to perform further actions.

The system is designed with a strong focus on security, requiring user authentication for all operations and safeguarding communication through encryption. The BBS strives to deliver a reliable and user-friendly platform for securely sharing and organizing information.

2. Design

This section provides detailed information about all entities involved in the Bulletin Board System (BBS) (i.e. **Systemt Specification** sub-section), along with the underlying assumptions and requirements (i.e. **Requirements** sub-section).

2.1. System Specification

2.1.1. Server

The BBS server is a centralized entity responsible for *securely* handling client requests in a multi-threaded environment. It is hosted at a fixed (IP, port) pair. The server possesses a public-private key pair, with the public key ($pubK_S$) being known to all clients. This public key is a crucial part of the system's design, simplifying the protocol for establishing a secure channel between the client and server. Through this secure channel, both parties can share a secret key used to encrypt and decrypt messages exchanged between them.

2.1.2. User

Users interact with the BBS through *clients*. Each user is uniquely identified by a nickname, set during registration, along with a password. Additionally, users provide an email address, which the system uses as a secure authentication channel during registration.

2.1.3. Message

A message in the BBS is represented as a tuple consisting of an identifier, title, author, and body. The identifier is unique to each message, while the author field records the nickname of the user who created the message.

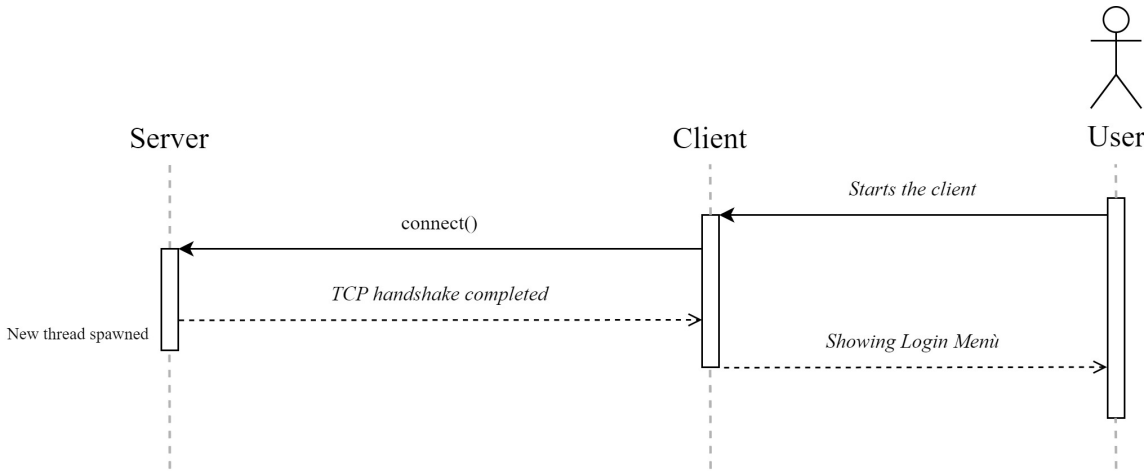


Figure 1: Initial unsecured TCP connection between the client and server.

2.2. Requirements

The system must meet the following key security requirements:

1. **Confidentiality**: Protect sensitive information from unauthorized access.
2. **Integrity**: Ensure data remains unaltered during transmission.
3. **Replay Protection**: Prevent attackers from reusing previous messages to gain unauthorized access.
4. **Non-Malleability**: Ensure that any intercepted and modified encrypted messages cannot affect the system in a valid way.
5. **Perfect Forward Secrecy (PFS)**: Guarantee that even if a user's long-term secret key is compromised, past session data remains secure.

Additionally, passwords must never be stored or transmitted in the clear. The system is implemented in C++ with the *OpenSSL* library to minimize code vulnerabilities.

2.3. Secure Channel

When a client *starts*, a *TCP connection* is established between the client and the BBS server, as shown in Figure 1. However, this type of connection, on its own, is insecure and susceptible to several vulnerabilities:

- **Eavesdropping**: Unauthorized parties can intercept the exchanged messages, compromising confidentiality.
- **Message Tampering**: The integrity of the communication is at risk, as messages can be altered.
- **Replay Attacks**: Previously transmitted messages can be reused to bypass security measures.
- **Message Modification**: Even encrypted messages can be manipulated, compromising non-malleability.

To mitigate these risks, the system establishes a *secure channel* before any client-initiated operations

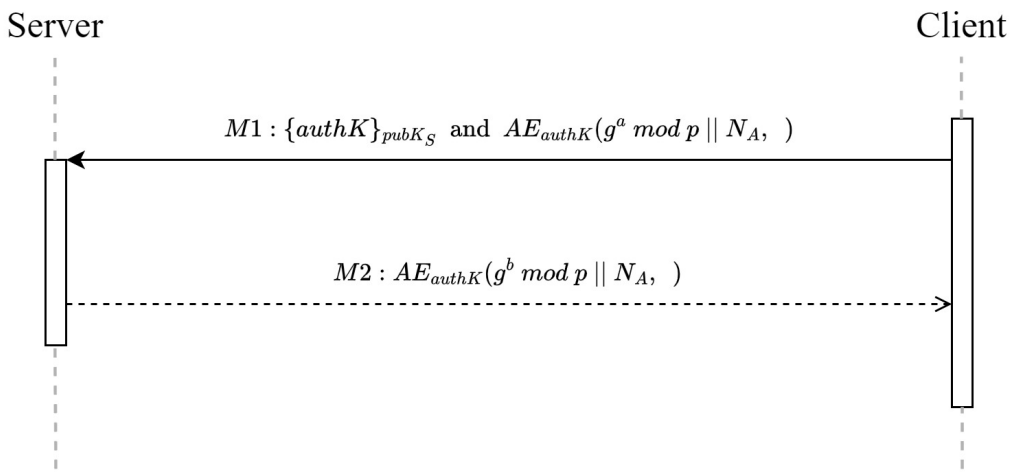


Figure 2: Secure channel establishment between client and server using a custom protocol.

using a custom security protocol. This protocol ensures the generation of a shared secret key between the client and the server, which is subsequently used as a symmetric key for encrypting and decrypting further communication. Figure 2 illustrates the protocol in action.

In the first message (**M1**), the client sends two components to the server: 1. The first component is a client-generated authentication key ($authK$), encrypted using the server's public key to ensure that only the server can decrypt it. 2. The second component contains the client's public key for the Diffie-Hellman key exchange and a nonce, which helps link **M1** to **M2** and prevents replay attacks.

Before delving into the details of the second component, it's important to understand the concept of *Authenticated Encryption (AE)*. AE serves two purposes: it provides both encryption and authentication for the exchanged data. Specifically, *Additional Authenticated Data (AAD)* is authenticated but not encrypted, while the plaintext is both encrypted and authenticated. The tag produced by the AE process ensures the integrity of both the ciphertext and the AAD, and it is used for verification. The AE process can be summarized as follows:

$$AE_K(AAD, Plaintext) = (Ciphertext, Tag)$$

Where:

- AE_K: Authenticated encryption using key K.
- AAD: Additional authenticated data that is not encrypted.
- *Plaintext*: The data that is encrypted.
- *Ciphertext*: The encrypted form of the plaintext.
- *Tag*: The tag used to verify the authenticity of both the ciphertext and the AAD.

In the protocol under consideration, only the AAD (Additional Authenticated Data) is specified, as no plaintext is present. Specifically, the AAD of M1 consists of the client's Diffie-Hellman public key and a client-generated nonce, neither of which require confidentiality. This is because the focus

is on tampering protection, and recovering the private key a from the public key $g^a \mod p$ would require solving the discrete logarithm problem, which is computationally infeasible. These values are authenticated using *authK*, a key shared between the client and server. The nonce serves to link M1 to M2 and prevents replay attacks.

The Diffie-Hellman parameters (g and p) are publicly known and standardized, meaning both the client and server use the same values. Upon receiving M1, the server prepares M2, which also contains only AAD for the same reasons. In M2, the server includes its own Diffie-Hellman public key and the client-provided nonce. The server does not generate a new nonce because its public key will naturally differ, ensuring that the same nonce (N_A) corresponds to a unique shared secret. This nonce is only used to protect the client against replay attacks from a man-in-the-middle.

When the client receives M2, it checks N_A and confirms the authenticity of the server's public key, as only the server has access to *authK*. The integrity of M2 is verified by computing the tag, ensuring that no tampering has occurred.

After exchanging M1 and M2, both the client and server can compute the shared secret, $g^{ab} \mod p$. Before using this shared secret as a symmetric key, it is hashed to enhance security. This hashed value becomes the key for symmetric encryption and decryption.

Evaluating the protocol against security requirements:

1. **Confidentiality** is ensured in subsequent packet exchanges through symmetric encryption using the shared symmetric key, known only to the client and server.
2. **Integrity** is guaranteed by the AE process, which uses a tag to verify the data.
3. **Replay Protection** is achieved in this phase through nonces, and in future phases by additional safeguards.
4. **Non-Malleability** is enforced by the AE process, which prevents tampering with the ciphertext or AAD.
5. **Perfect Forward Secrecy (PFS)** is guaranteed because the shared secret is unique to each session and cannot be recovered even if the server's long-term private key is compromised. The use of ephemeral Diffie-Hellman key exchange ensures this, as the server's public key is only used for authentication, not for encrypting the shared secret.

---

3. Design

3.1. System Specification

Here are all the details related to all the entities involved in the system. Hypotheses and requirements are also specified.

3.1.1. Server

The BBS server is a centralized entity that handles in a multi-threaded, and secure way all the requests coming from the clients. The server is attested at a well-known (ip, port) couple. Furthermore, the BBS server is equipped with a private-public key pair of which the public component pubK bbs is known to users. So all the clients know the public key of the server, this is one of the main

assumptions of the system, that allows to simplify the protocol used for establishing a secure channel between the client and the server, where both share a secret key that will be used for encrypting and decrypting the messages exchanged between them.

3.1.2. User

THe users are those that will utilize the clients to interact with the server. Each user is identified by means of a nickname that is established at registration time together with a password. The password is never stored or transmitted in the clear. The user is also associated with an email address that simulates a secure channel, sent by the server in the registration phase. For simulating the secure channel in order to authenticate the user with his email address, the server will write the challenge inside the Storage of the clients in the emails folder, in a file named as the email address of the user. The client will read the challenge from the file, as if it was sent by email. An example will clarify this concept. If the mail of the user is `user123@gmail.com`, the server will write the challenge inside the file `Client\Storage\Emails\user123@gmail.com.txt`. The client will read the challenge from the file and paste it to the terminal enabling the registration phase to correctly conclude.

3.1.3. Message

A message is a tuple composed of the following fields: identifier, title, author, and body. The identifier field uniquely identifies the message within the BBS. The author field specifies the nickname of the user who added the message to the BBS.

3.2. Requirements

Between the requirements of the system, the following it is possible to find requirements related to:

- Confidentiality: ensuring that sensitive information is protected from unauthorized access.
- Integrity: guaranteeing that data remains unchanged and uncorrupted during transmission.
- No-replay: preventing attackers from reusing messages to gain unauthorized access.
- Non-malleability: This requirement ensures that even if an attacker intercepts and modifies an encrypted message, the resulting message will have no valid effect on the system.
- Perfect Forward Secrecy (PFS): This property ensures that even if the long-term secret key of a user is compromised, the past sessions are still secure.

Additional requirments are related to the fact that passwords should never be stored or transmitted in the clear. The system should reduce code vulnerabilities as much as possible, using the C or C++ programming language and the *OpenSSL* library.

3.3. Secure Channel

When the clients are started, a simple TCP connection between the particular client and the only server of The BBS is established. It is clear that using such a connection is not secure, becuase an attacker could exploit this channel to: - eavesdrop the messages exchanged between the client and the server, breaking the confidentiality requirement. - modify the messages exchanged between the client and the server, breaking the integrity requirement. - replay the messages exchanged between the client and the server, breaking the no-replay requirement. - modify the messages exchanged between the client and the server, breaking the non-malleability requirement. So to fulfill to those

requiremnts it is necessary to establish a secure channel by means of a proper defined security protocol. The protocol that is implemented before the transmission of each packet to the server is the one described in the figure x. Here going into the details there is a first message, M1 that the client sends to the server. This message is composed of two different packets, as the first one is just the encryption by public key of the server of a client generated authentication key (i.e. `authK`) used just for authentication purposes. Being encrypted by the public key of the server, the only one that can read it is the server and obviously the client that generated it. Using that key, now the client makes use of AE, that is an Authenticated Encryption method that follows these details:

When specifying both the Additional Authenticated Data (AAD) and the plaintext in the context of authenticated encryption, you can use a notation that includes both components. The notation should clearly differentiate between the AAD and the plaintext (or message) being encrypted. Here's how you might represent this:

3.3.1. General Notation

For authenticated encryption with AAD, the notation might be:

[AE_{K}(AAD, Plaintext) = (Ciphertext, Tag)]

where: - **(AE_{K})** denotes the authenticated encryption operation with key (K). - **(AAD)** represents the additional authenticated data. - **(Plaintext)** represents the data being encrypted. - **(Ciphertext)** is the result of encrypting the plaintext. - **(Tag)** is the authentication tag used to verify both the ciphertext and the AAD.

In the figure there are only AAD, as the Plaintext is not sent to the server. Here there's no need to encrypt gˆa mod p and N_A because we only want that the integrity of them is preserved and this is the case as the AE method computes the tag associated to those quantities, that the server will check at the destination and if the tag is correct, the server will be sure that the values of gˆa mod p and N_A are the same as the ones sent by the client.

Now the server after receiving M1, and decrypting authJ with its private key, it can compute the tag of the second packet of M1 and check if it is the same as the one sent by the client. If it is the same, the server will be sure that the client is the one that sent the message and that the message has not been modified by an attacker. Now the server will prepare M2, where it will put his counterpart of the public key of the Diffie Hellman key exchange, i.e. gˆb mod p. It will append also the N_A of the client to the AAD and authenticate and encrypt the packet with the same AE method. The client will receive M2, and after decrypting it with the session key, it will check the tag and if it is correct, it will be sure that the server is the one that sent the message and that the message has not been modified by an attacker. The client will also check that the N_A is the same as the one sent by the client in M1. Otherwise a replay attack would have been happened.

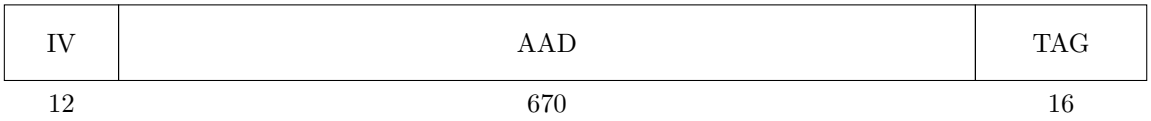
Note that sending the same M1 will result in general in a different M2 and hence in a different share key, aas gˆb mod p is chosen by the server and hence is like a nonce, because the same session key can't be forced by the attacker to be used in another session, because the attacker can't know the value of gˆb mod p before the server sends it to the client. And in general it will be different. Replay attacks from the server are neither possible as before said because the N_A is different in each session and the server can't know the value of N_A before the client sends it to the server.

After having received correctly gˆb mod p the client and gˆa mod p the server, they can compute

the shared key that will be used for encrypting and decrypting the messages exchanged between them. Before that of course the shared secred is hashed as it is a good practice to do that, before using the shared secret as a simmetric key for the encryption and decryption of the messages. The hash function used is SHA256. The autentication encryption used instead is the AES 128 GCM, that is a good choice for the encryption of the messages exchanged between the client and the server, as it is a secure and efficient method for the encryption and decryption of the messages and furthermore there is no expansion of the ciphertext with regard to the plaintext The details of the packets exchanged in the secure channel are the following:

First packet:

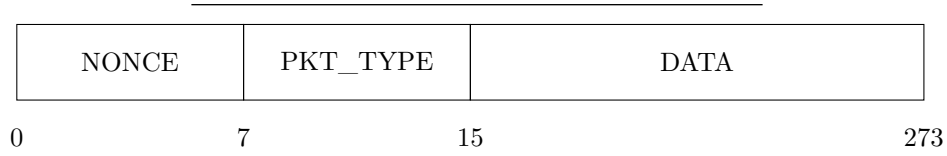
Second packet of m1 and also m2



In Aad there are both the public key of Diffie Hellman (654 bytes) and concatenated to it the nonce of the client (16 bytes).

Registration Phase A user securely connects to the BBS server and specifies an email address, a nickname and a password; the server sends a challenge to the email address specified by the user and waits for receiving the challenge back; if the user correctly returns the challenge to the server, the registration phase concludes successfully. Otherwise, it is aborted. Login phase A registered user securely connects to the BBS server and logins by means of his/her nickname and password. The server lets the user log in if the submitted user's nickname and password are correctly verified. Upon successful login, a secure session is established and maintained until the user logs out.

Requirements Never store or transmit passwords in the clear. Fulfill confidentiality, integrity, no-replay, and non-malleability in communications. Guarantee perfect forward secrecy (PFS). Reduce code vulnerabilities as much as possible. Use C or C++ programming language and OpenSSL library but OpenSSL API TLS cannot be used. Deliverables System specification and design. A running early prototype.



This is the general packets format exchanged during all the subsequent phases after that the secure connection is established.

IV	AAD_SIZE	AAD	CIPH_SIZE	CIPH	TAG
12	4	AAD_SIZE	4	CIPH_SIZE	16

Possible values for type: #define T_REGISTRATION 1 #define T_LOGIN 2 #define T_OK 3 #define T_KO 4 #define T_LIST 5 #define T_GET 6 #define T_ADD 7 #define T_LOGOUT 8

Add is different because the replay could do bad things...

Ask the Server the possibility to add a message In this way it will send back a its nonce to be used in next packets

We close the connection between client and server everytime that happens something of unexpected, as a protection mechanism.

Dire add si fa perché può causare effetti indesiderati le altre operazioni no alla fine...

Hp: p and g are public known and also the server's public key is know to all clients

ALl in encrypted and authenticated the sequence diagram is just to show the main steps throught which the client and the server and user pass to register.

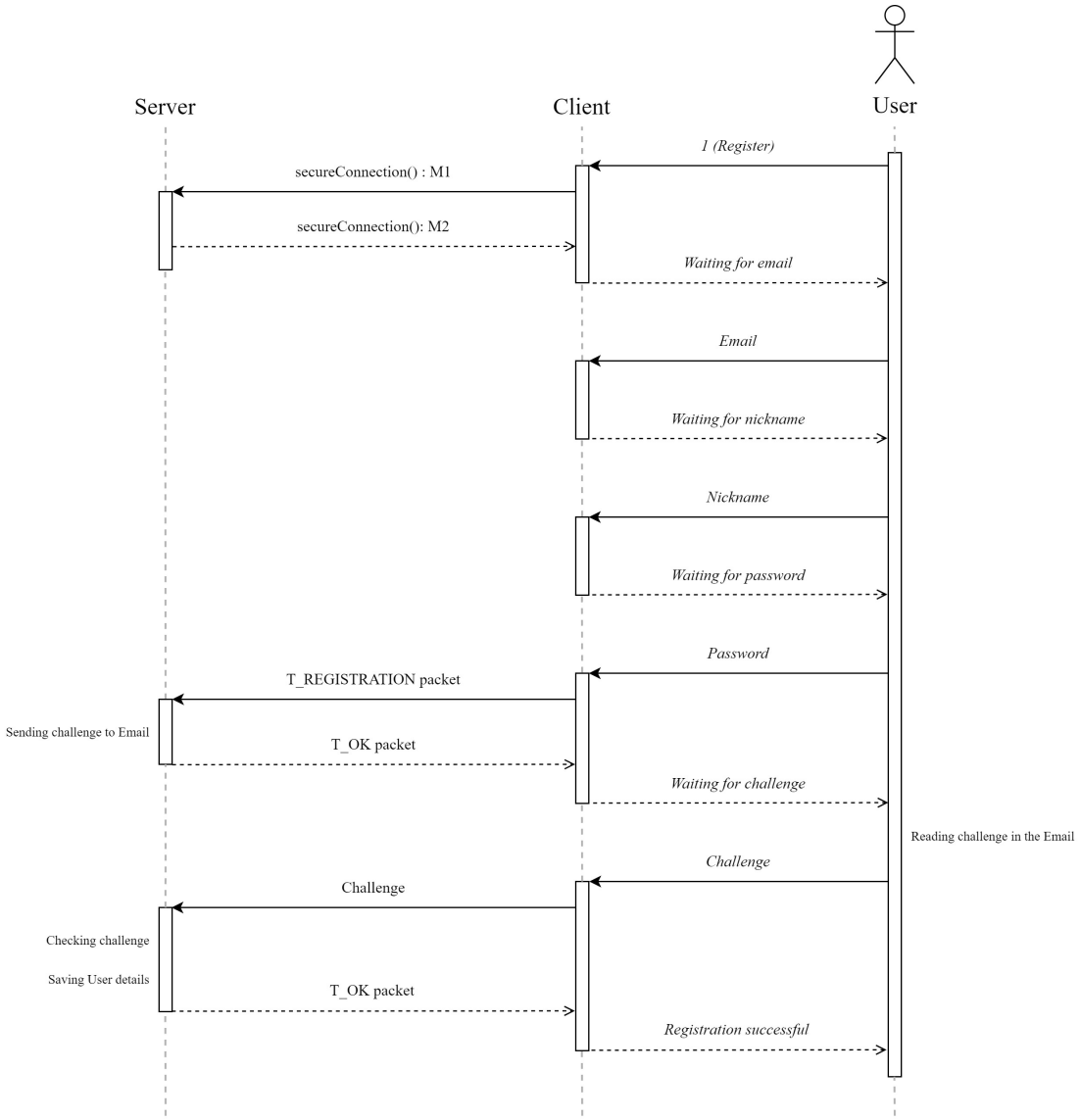


Figure 3: Registration of a User.

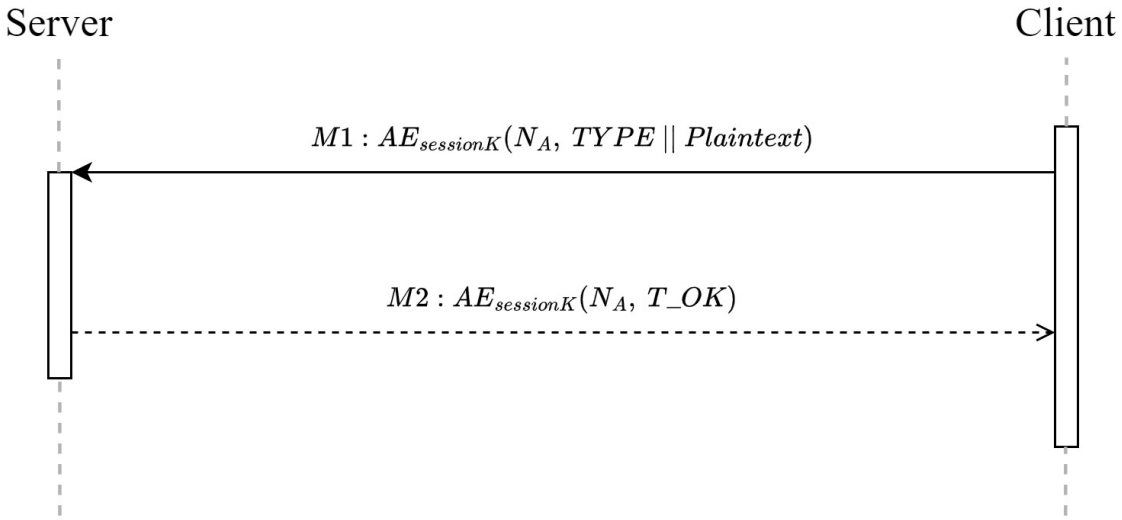


Figure 4: General Packet exchange.

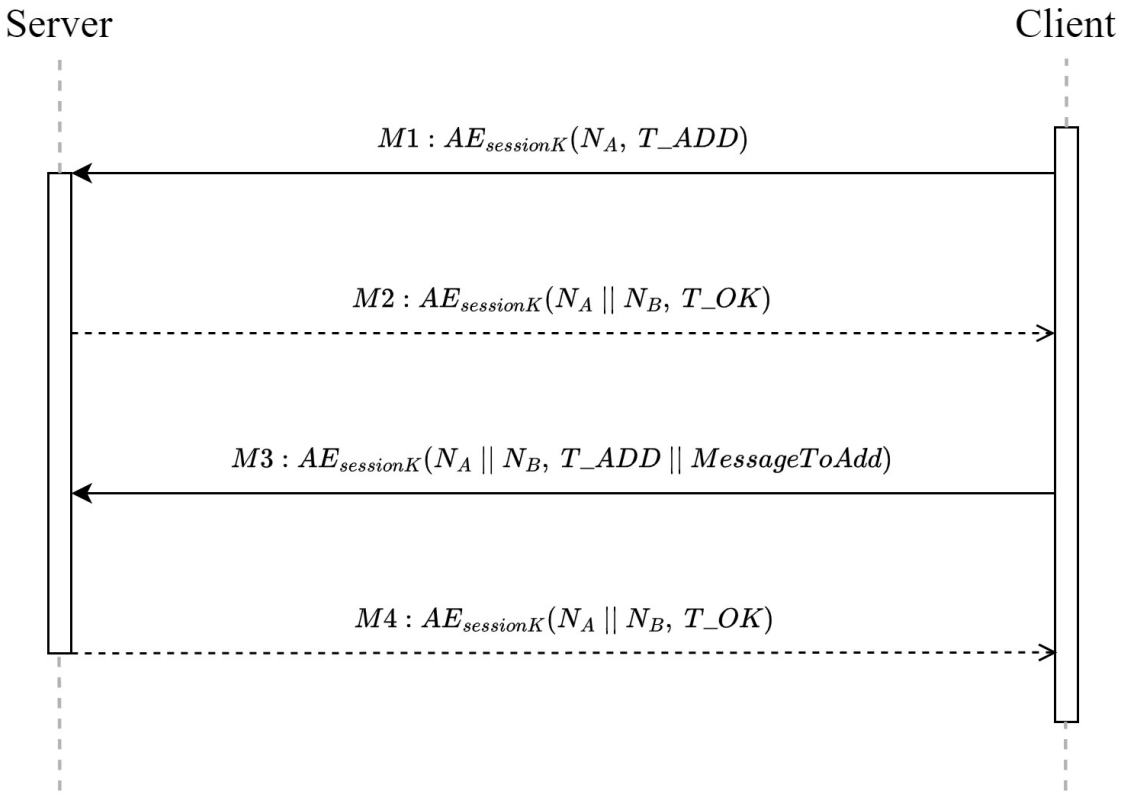


Figure 5: Add message exchange.