Table of Contents

Problem Statement

Design a key establishment protocol that will allow a Mutually Agreed Session key (K_{abc}) to be established between three entities A, B and C. This key can be used to secure the chat between these three entities.

The protocol must follow guidelines that are stated below:

1) The entities never communicate with each other directly. All messages must pass through the server 'S'.
2) The server 'S' cannot know the shared secret and must not be able to decrypt messages, or in other words end-to-end encryption is required between the entities.
3) Clients must establish shared secret to send encrypted messages.
4) Before the clients are allowed to use the services from server 'S' they must authenticate themselves with 'S'.
5) Each step in establishing the session key (K_{abc}) must provide an authenticated integrity check of the data transferred.

At the beginning

Entities have the following with them.
A has CA<<A>>, K_{ca}
B has CA<<B>>, K_{ca}
C has CA<<C>>, K_{ca}
S has CA<<S>>, CA<<A>>, CA<<B>>, CA<<C>>, K_{ca}

Legend
A = client A
B = client B
C = client C
S = server S
CA<<A>> = Certificate of entity 'A'
CA<<B>> = Certificate of entity 'B'
CA<<C>> = Certificate of entity 'C'
CA<<S>> = Certificate of entity 'S'
K_{ca} = RSA Public key of Certificate Authority

General Information
We are using RSA with key size of 2048 bits, AES with key size of 128 bits and SHA3-256 hash algorithm in our project.

On Server Start

Server must validate client certificates before it can use them.
Certificate contains the following:
Certificate of client 'A': CA<<A>> = A, K_a, $\{H(A, K_a)\}K_{ca}^{-1}$
Client 'B' and 'C' have similar information.

The server checks if the certificates of the clients are valid at startup as follows:
S: $\{\{H(A, K_a)\}K_{ca}^{-1}\}K_{ca}$ == $H(A, K_a)$?

If the above step results into true or values match, client 'A' certificate is valid.
Server 'S' must perform similar step for client 'B' and 'C' to validate their certificates.

Legend
K_a = RSA Public key of client 'A'
K_{ca}^{-1} = RSA Private key of Certificate Authority

Authentication

Authentication between A and S will be done in 3 steps which are described below.

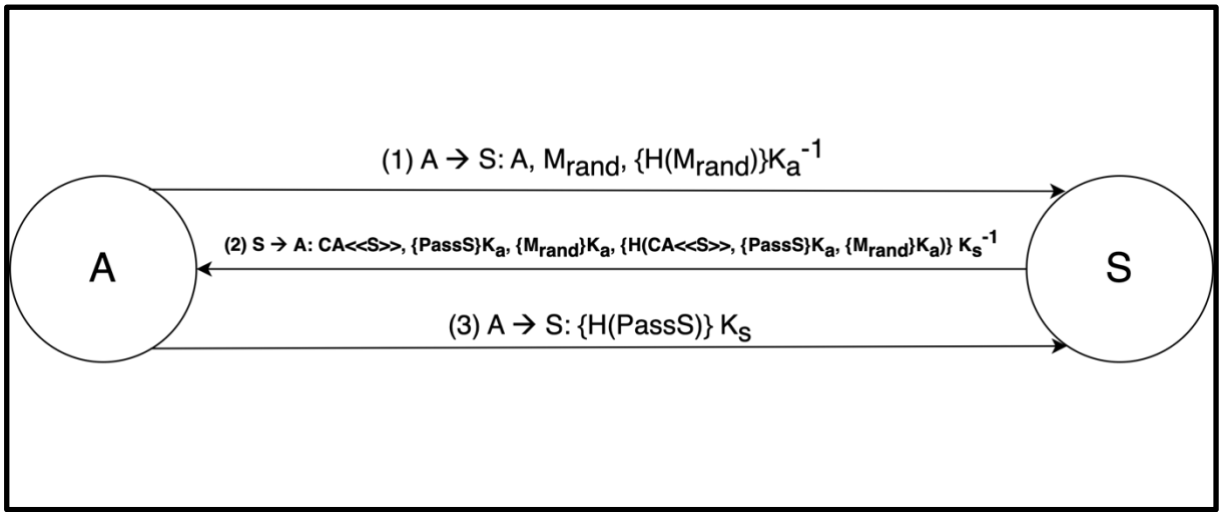

Figure 1: Authentication Protocol Diagram

Step 1:
A → S: A, M_{rand}, $\{H(M_{rand})\}K_a^{-1}$

Code: Search for **AUTH_STEP1** in **client.py** (**You can search for this TAG using any text editor or IDE by pressing CTRL + F or CMD + F, it will take you to the corresponding location in code**)

'S' now knows that 'A' has sent some random message M_{rand}. But it needs to verify that. Since 'S' already has CA<<A>> it can obtain K_a from the certificate. And perform the following:

S: $\{\{H(M_{rand})\}K_a^{-1}\}K_a == H(M_{rand})$?

Code: Search for **VERIFY_AUTH_STEP1** in **server.py**

If the hash matches, then 'S' knows that 'A' sent the message. (Authentication and Integrity done).

Step 2:
S → A: CA<<S>>, $\{PassS\}K_a$, $\{M_{rand}\}K_a$, $\{H(CA<<S>>, \{PassS\}K_a, \{M_{rand}\}K_a)\}\ K_s^{-1}$

Code: Search for **AUTH_STEP2** in **server.py**

Before 'A' can use the public of server it must validate the server certificate as follows:
A: $\{\{H(S, K_s)\}K_{ca}^{-1}\}K_{ca} == H(S, K_s)$?

Code: Search for **VALIDATE_SERVER_CERT** in **client.py**

'A' has access to its own private key K_a^{-1}. Hence, it can easily obtain PassS and M_{rand} from the message. 'A' can check the M_{rand} sent by the server to its own M_{rand} if they don't match, then there is an issue.
A: $\{\{PassS\}K_a\}K_a^{-1}$ => PassS
A: $\{\{M_{rand}\}K_a\}K_a^{-1}$ => M_{rand}

Another way to verify if the message was sent by 'S' is to perform:

A: $\{\{H(CA<<S>>, \{PassS\}K_a, \{M_{rand}\}K_a)\}\ K_s^{-1}\}\ K_s == H(CA<<S>>, \{PassS\}K_a, \{M_{rand}\}K_a)$?

Code: Search for **VERIFY_AUTH_STEP2** in **client.py**

If the messages match everything is okay.

Note:
K_s can be obtained from CA<<S>>.

Step 3:
Finally, the client sends the hash of PassS back to the server by encrypting it with 'S' public key.

A → S: $\{H(PassS)\}\ K_s$

In the last step, only 'A' will be able to read PassS from step 2, so only 'A' will be able to compute H(PassS) and then compute $\{H(PassS)\}K_s$. Hence, the last step has Integrity and Authentication combined into one.

Code: Search for **AUTH_STEP3** in **client.py**

'S' must perform:

S: $\{\{H(PassS)\}K_s\}K_s^{-1}$ == H(PassS) ?

If the operations match everything is okay.

Code: Search for **VERIFY_AUTH_STEP3** in **server.py**

In step 2 the server sent CA<<S>> to 'A'. Hence, 'A' now also has CA<<S>>.
Hence, at this stage entities have:
A has CA<<A>>, CA<<S>>, K_{ca}
B has CA<<B>>, K_{ca}
C has CA<<C>>, K_{ca}
S has CA<<S>>, CA<<A>>, CA<<B>>, CA<<C>>, K_{ca}

'B' and 'C' will also authenticate with 'S' in a similar manner.
Hence, at this stage entities have:
A has CA<<A>>, CA<<S>>, K_{ca}
B has CA<<B>>, CA<<S>>, K_{ca}
C has CA<<C>>, CA<<S>>, K_{ca}
S has CA<<S>>, CA<<A>>, CA<<B>>, CA<<C>>, K_{ca}

Legend
M_{rand} = Random message generated by client of 256 bits
PassS = Random message generated by server of 256 bits

Requesting Certificates of other Clients

Client 'A' needs certificates of 'B' and 'C' to get their respective public keys. For this, 'A' can request certificates from the server 'S' as follows.

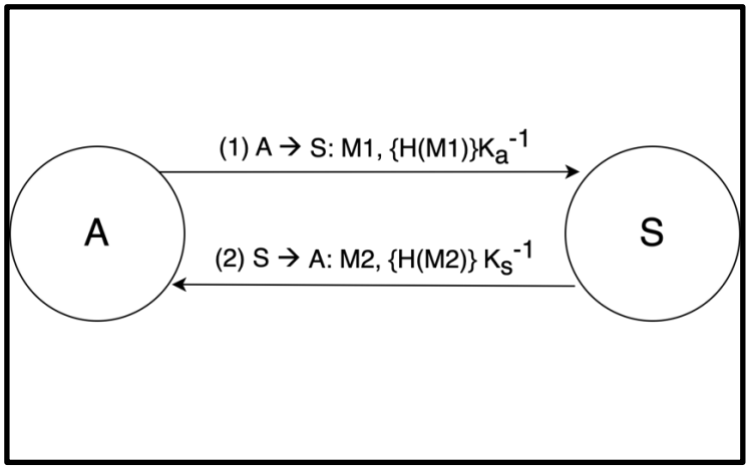

Figure 2: Client 'A' Requests Certificate from Server 'S'

Step 1:
A → S: M1, $\{H(M1)\}K_a^{-1}$

Code: Search for **REQUEST_CERT** in **client.py**
'A' sends a special control message "request/certificates" to the server for requesting 'B' and 'C' certificates.

'S' verifies if the request came from 'A' as follows:
S: $\{\{H(M1)\}K_a^{-1}\}K_a$ == H(M1) ?

If the values match, then the request came from 'A'.

Step 2:
S → A: M2, $\{H(M2)\}\ K_s^{-1}$

Code: Search for **SEND_CERT** in **server.py**

The server sends the certificates CA<<B>> and CA<<C>> to 'A'.

'A' verifies if the request came from 'S' as follows:
A: $\{\{H(M2)\}K_s^{-1}\}K_s$ == H(M2) ?

If the values match, then the response came from 'S'.

Before using CA<<B>> and CA<<C>>, 'A' must validate these certificates.
To validate certificate CA<<B>>, 'A' performs the following:
A: $\{\{H(B, K_b)\}\ K_{ca}^{-1}\}K_{ca}$ == $H(B, K_b)$? if the value matches, then certificate is valid.
'A' performs similar step for CA<<C>>.

Code: Search for **VALIDATE_CLIENT_CERT** in **client.py**

Hence, at this stage entities have:
A has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, K_{ca}
B has CA<<B>>, CA<<S>>, K_{ca}
C has CA<<C>>, CA<<S>>, K_{ca}
S has CA<<S>>, CA<<A>>, CA<<B>>, CA<<C>>, K_{ca}

Similarly, 'B' and 'C' can request remaining certificates from 'S'.
Hence, at this stage entities have:
A has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, K_{ca}
B has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, K_{ca}
C has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, K_{ca}
S has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, K_{ca}

Legend
M1 = request for CA<<B>> and CA<<C>>
M2 = contains CA<<B>> and CA<<C>>

Establishing K_{abc} for Clients A, B, C without Server S knowing K_{abc}

This part is divided into two steps, first all the clients 'A', 'B' and 'C' generate and share their random nonce to each other. In the second part, the clients generate a Mutually Agreed Session Key by hashing all the random nonces in a deterministic fashion.

Part 1: Sharing Random Nonce

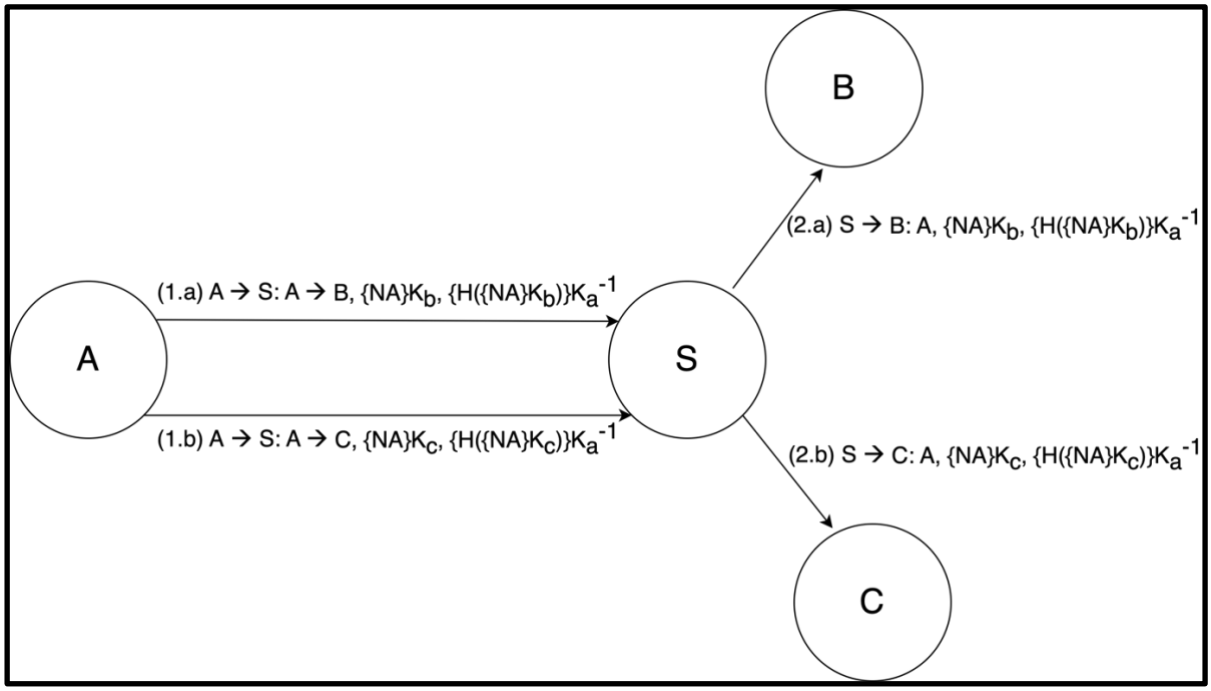

Figure 3: Sharing Random Nonce with other Clients

Client 'A' crafts a secured random nonce NA and sends this to 'B' and 'C' as follows:

Step 1:
(Step 1.a) A → S: A → B, $\{NA\}K_b$, $\{H(\{NA\}K_b)\}K_a^{-1}$
(Step 1.b) A → S: A → C, $\{NA\}K_c$, $\{H(\{NA\}K_c)\}K_a^{-1}$

Code: Search for **SHARE_NONCE** in **client.py**

A → B tells 'S' that this message is meant for 'B'.
Similarly, A → C tells 'S' that this message is meant for 'C'.

Step 2:
(Step 2.a) S → B: A, $\{NA\}K_b$, $\{H(\{NA\}K_b)\}K_a^{-1}$
(Step 2.b) S → C: A, $\{NA\}K_c$, $\{H(\{NA\}K_c)\}K_a^{-1}$

Code: Search for **FORWARD_UNICAST_MESSAGE** in **server.py** (The server does not care if the message is a nonce or simply a normal message intended for the user, it simply forwards it to the other client.)

'S' simply forwards the message to the correct client.

Client 'B' performs the following:
B: $\{\{H(\{NA\}K_b)\}K_a^{-1}\}K_a == H(\{NA\}K_b)$?

Code: Search for **VERIFY_NONCE** in **client.py**

If the values match, then NA was not tampered, and this message came from 'A'.

Client 'B' recovers NA as follows:
B: $\{\{NA\}K_b\}K_b^{-1}$ => NA

Code: Search for **RECOVER_NONCE** in **client.py**

Client 'C' performs similar steps to verify authenticity and integrity and then recovers NA.

Note:

1) 'S' cannot tamper with NA because 'S' does not have K_a^{-1} to sign the message.
2) 'S' cannot recover NA because it does not have K_b^{-1} or K_c^{-1}.

Hence, at this stage entities have:
A has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, K_{ca}
B has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, K_{ca}
C has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, K_{ca}
S has CA<<S>>, CA<<A>>, CA<<B>>, CA<<C>>, K_{ca}

Similarly, when 'B' crafts its random nonce NB and sends to 'A' and 'C' we have:
A has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, NB, K_{ca}
B has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, NB, K_{ca}
C has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, NB, K_{ca}
S has CA<<S>>, CA<<A>>, CA<<B>>, CA<<C>>, K_{ca}

Similarly, when 'C' crafts its random nonce NC and sends to 'A' and 'B' we have:
A has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, NB, NC, K_{ca}
B has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, NB, NC, K_{ca}
C has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, NB, NC, K_{ca}
S has CA<<S>>, CA<<A>>, CA<<B>>, CA<<C>>, K_{ca}

Part 2: Generating Shared Session Key from Random Nonces

Once all clients have their random nonces shared securely, each client can compute K_{abc} as follows:
$K_{abc} = H(NA, NB, NC)$

Code: Search for **GENERATE_SESSION_KEY** in **client.py**

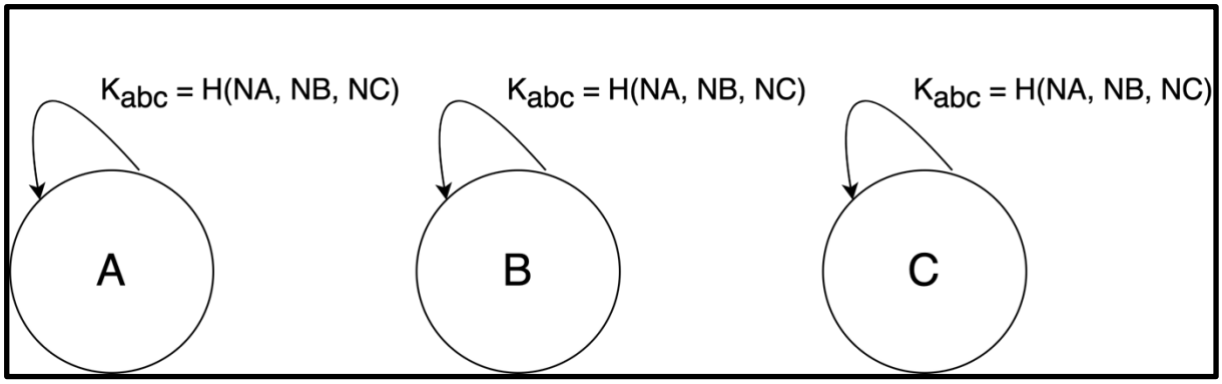

Figure 4: Each client has NA, NB, NC from the previous step, hence they can compute K_{abc}

Note:

1) The order in which the random nonces are hashed matters.
2) Since the server cannot read NA, NB, or NC it cannot generate K_{abc}.

Hence at this stage, each entity has:
A has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, NB, NC, K_{abc}, K_{ca}
B has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, NB, NC, K_{abc}, K_{ca}
C has CA<<A>>, CA<<B>>, CA<<C>>, CA<<S>>, NA, NB, NC, K_{abc}, K_{ca}
S has CA<<S>>, CA<<A>>, CA<<B>>, CA<<C>>, K_{ca}

Legend
NA = Random nonce produced by client 'A' of 256 bits in length
NB = Random nonce produced by client 'B' of 256 bits in length
NC = Random nonce produced by client 'C' of 256 bits in length
K_{abc} = Shared session key for use with AES algorithm in ECB mode

Sending message from one client to another via server S using K_{abc}

Now, the clients can use the Session key (K_{abc}) for securing the messages with each other.

The clients can operate in unicast and broadcast mode.

Unicast: Client 'A' shares message only with client 'B' (one other client), or client 'A' shares message only with client 'C'

Broadcast: Client 'A' shares message with both clients 'B' and 'C'.

The process is explained below:

Part 1: Unicast mode

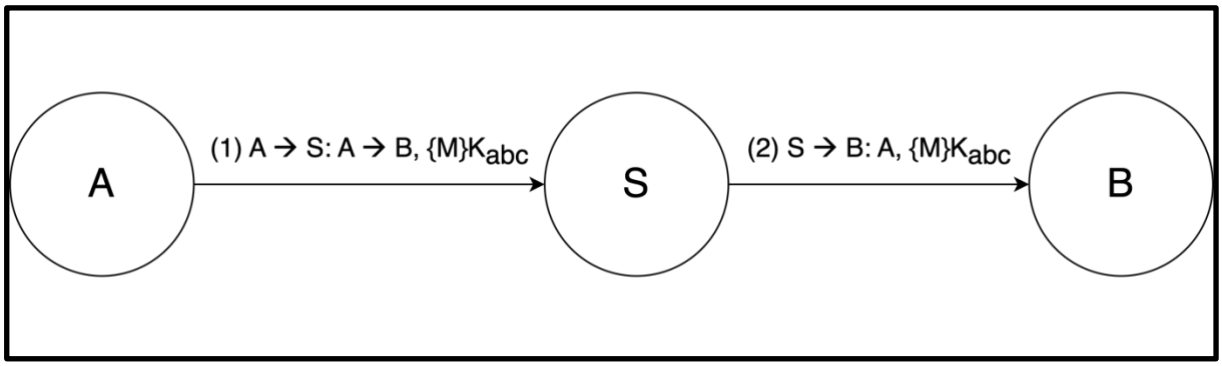

Figure 5: 'A' sending encrypted message to 'B' using shared symmetric session key

Step 1:
'A' sending message to 'B' via Server 'S' using K_{abc} as follows:

A → S: A → B, $\{M\}K_{abc}$

Code: Search for **SEND_UNICAST_MESSAGE** in **client.py**

Step 2:
Server reads the message direction A → B and understands that this message is being sent from 'A' to 'B' and does the following:
S → B: A, $\{M\}K_{abc}$

Code: Search for **FORWARD_UNICAST_MESSAGE** in **server.py**

Client 'B' decrypts the message as follows:

B: $\{\{M\}K_{abc}\}K_{abc}$ => M

Code: Search for **RECV_UNICAST_MESSAGE** in **client.py**

Part 2: Broadcast Mode

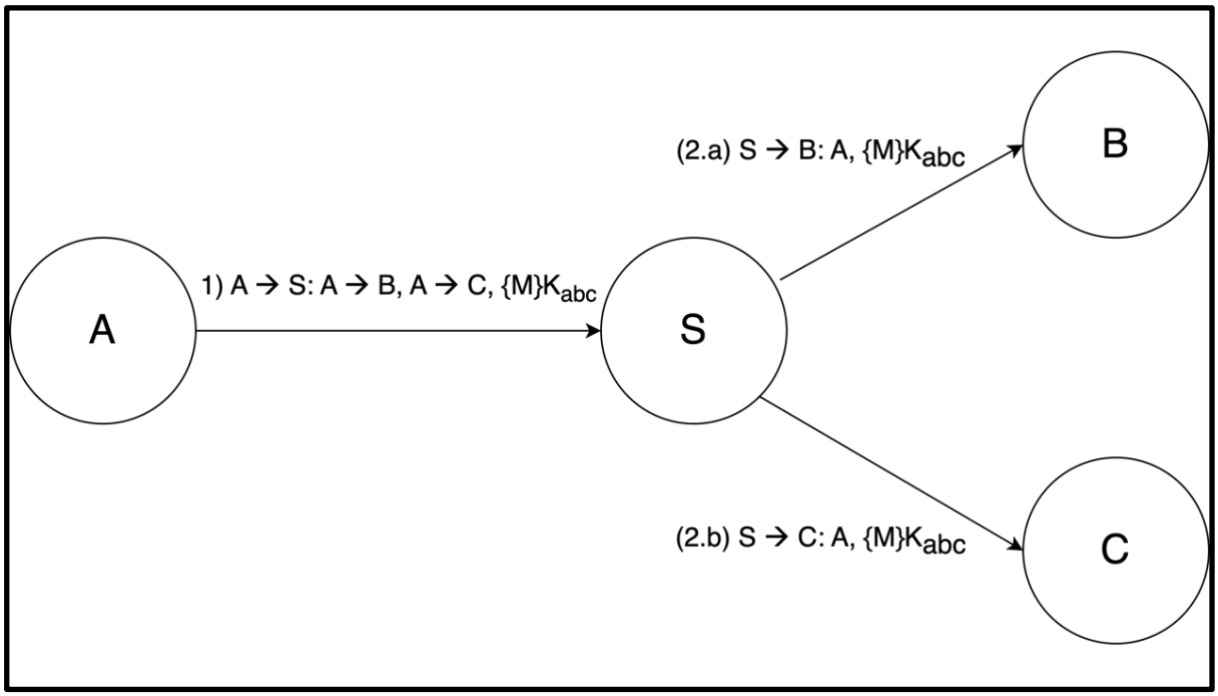

Figure 6: 'A' sending encrypted message to 'B' and 'C' using shared symmetric session key

Step 1:
'A' sending message to 'B' and 'C' via Server 'S' using K_{abc} as follows:

A → S: A → B, A → C, $\{M\}K_{abc}$

Code: Search for **SEND_BROADCAST_MESSAGE** in **client.py**

Step 2:
Server reads the message direction A → B and A → C and understands that this message is a broadcast message and does the following:

Step 2.a) S → B: A, $\{M\}K_{abc}$
Step 2.b) S → C: A, $\{M\}K_{abc}$

Code: Search for **FORWARD_BROADCAST_MESSAGE** in **server.py**

Clients 'B' and 'C' decrypts the message as follows:

B: $\{\{M\}K_{abc}\}K_{abc}$ => M
C: $\{\{M\}K_{abc}\}K_{abc}$ => M

Note about this Code: When the server broadcasts the message (example from 'A' to 'B' and 'C'), each client will receive the message only once. Hence, there is no presence of RECV_BROADCAST_MESSAGE in client.py instead the same RECV_UNICAST_MESSAGE code part is triggered. So, search for **RECV_UNICAST_MESSAGE** in **client.py**

Legend
M = Message

Note:
Since the server does not have K_{abc} it cannot recover the message M.

Project Working Screenshots

Since, we are not submitting a video demonstration for the project. We have provided the screenshot of the project working below. **The red boxes highlight important details.**

Part 1: Unicast, send message to one entity demo

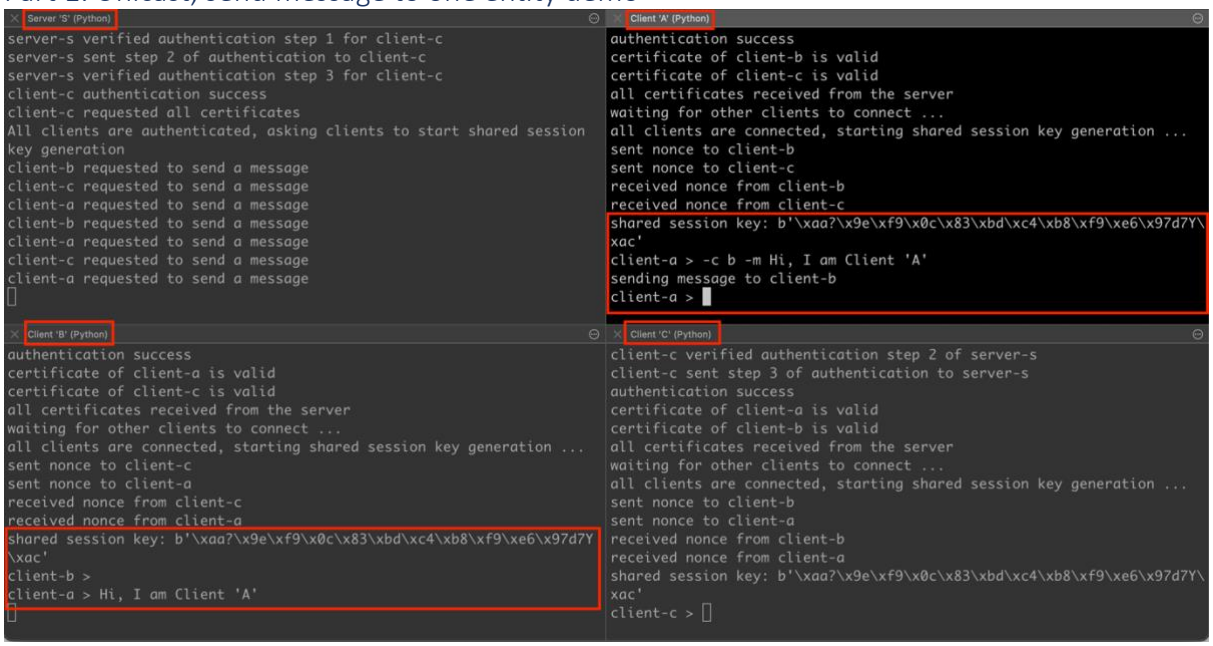

Figure 7: Client ‘A’ sending message to client ‘B’ only.

Part 2: Broadcast, send message to all entity demo

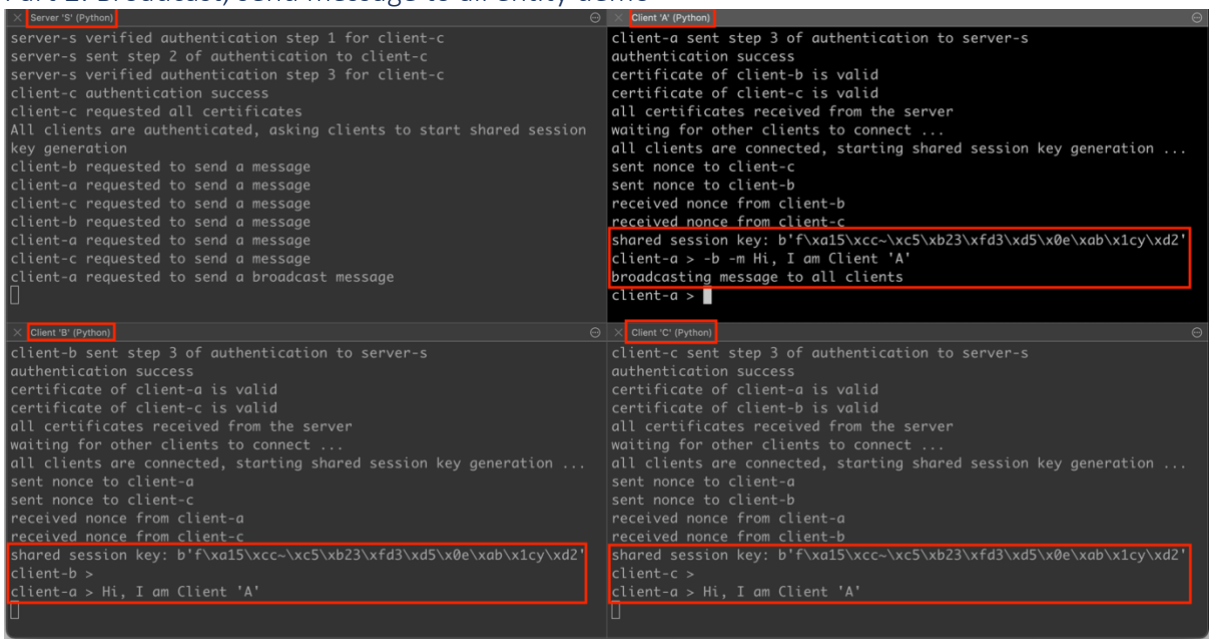

Figure 8: Client ‘A’ broadcasting message to clients ‘B’ and ‘C’.

From the images it is evident that all the clients have generated a Mutually Agreed Shared Session Key and it is possible to send messages to each other.

Examining Code

Please **cd into end-to-end-encryption-protocol** folder. This is the root directory for the project.

server.py: This file contains the server code.
client.py: This file contains the client code. (There is no separate code for 3 different clients since they do the same thing.)

algorithms.py: This file contains the functions which assist in:

- Encryption/decryption using AES and RSA
- Signing/Verifying using RSA
- Encoding/Decoding using Base64
- Hashing using SHA3-256
- Validating certificates
- Combining bytes to produce a shared key
- Generating random nonce

These functions are common in client.py and server.py hence we separated them out.
This file also supports independent execution which tests all the above functions.
To run tests in algorithms.py run:

main.py: This is the management module that helps to run the server-s/client-a/client-b/client-c. **Please use this file to run the server and client.**

readme.txt: Contains instructions to install dependencies of the project.

config.yaml: This file contains the settings for the project. The only setting that may require changes is the port number in case that port is busy.

make-certificate.ipynb: This is a Jupyter Notebook, this file contains the code that was used to generate the keypairs and certificate for the entities. (Note you may require an IDE that can open this Notebook, it is not a .txt file!)

certs folder: This folder contains all the certificates.

logs folder: This folder contains the logs written by server-s/client-a/client-b/client-c when sending message to each other. It is useful in inspecting what was sent on the TCP connection.