PRIVACY PRESERVATION IN AN E-COMMERCE SYSTEM

Design Report

Sai Santhosh Jella – SXJ220035

Sai Tharun Reddy Mulka – SXM220351

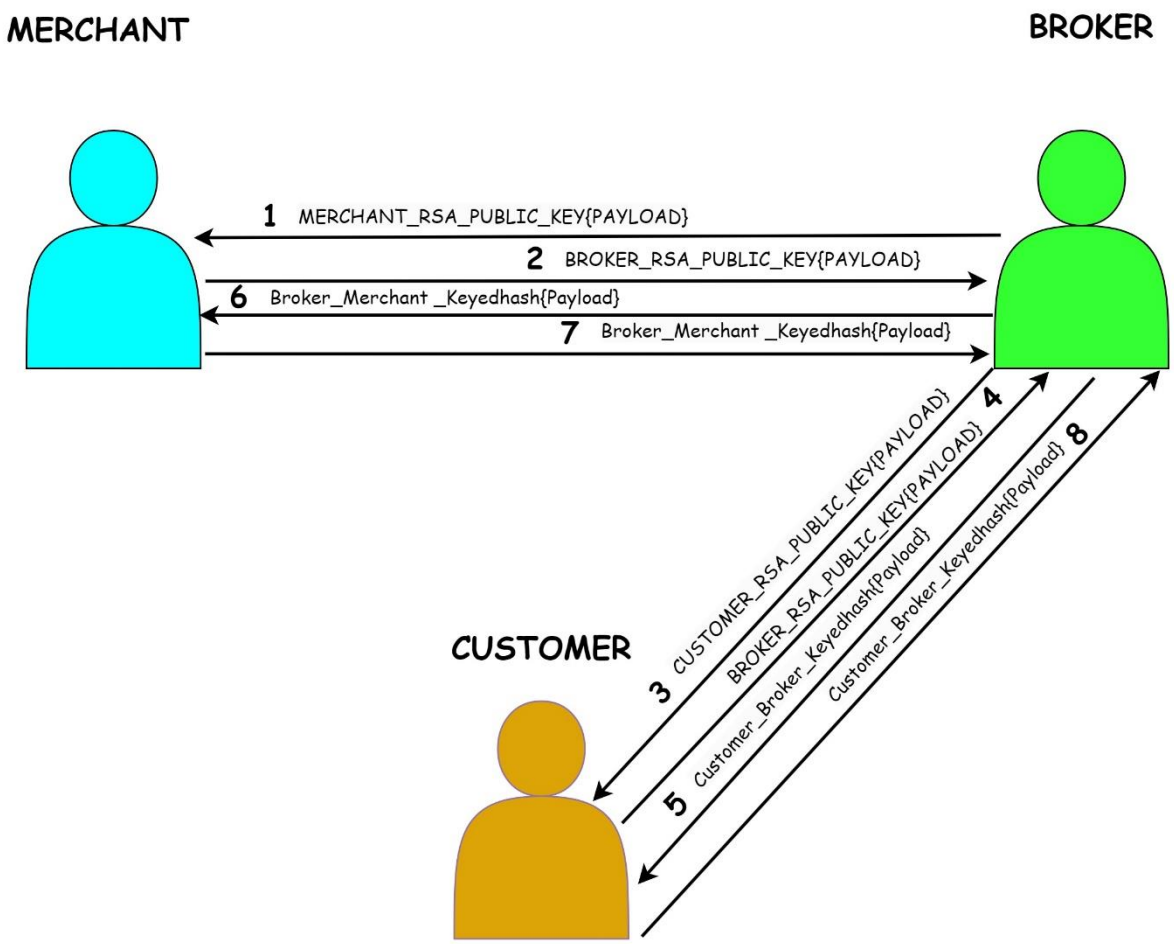
Teja Zangam – TXZ230002

Shivashankar Chandrashekaraiah – SXH220091

Supported Functionalities in Authentication:

Broker and merchant mutual auth –

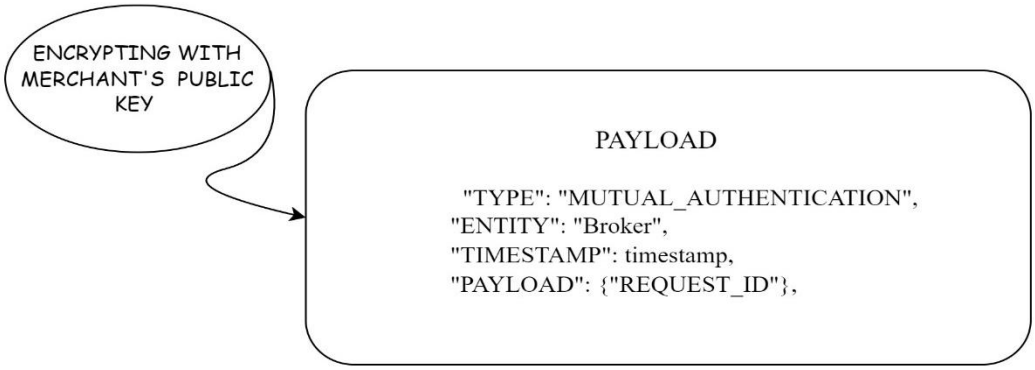
For This project, we have used fast API, Uvicorn (ASGI application) to develop a dynamic and responsive server architecture. Fast API provides a high-performance API with automatic request validation, where uvicorn handles asynchronous tasks and socket connections ensuring efficient request handling, implementing sockets is our key aspect of the project allowing real-time data exchange between entities.



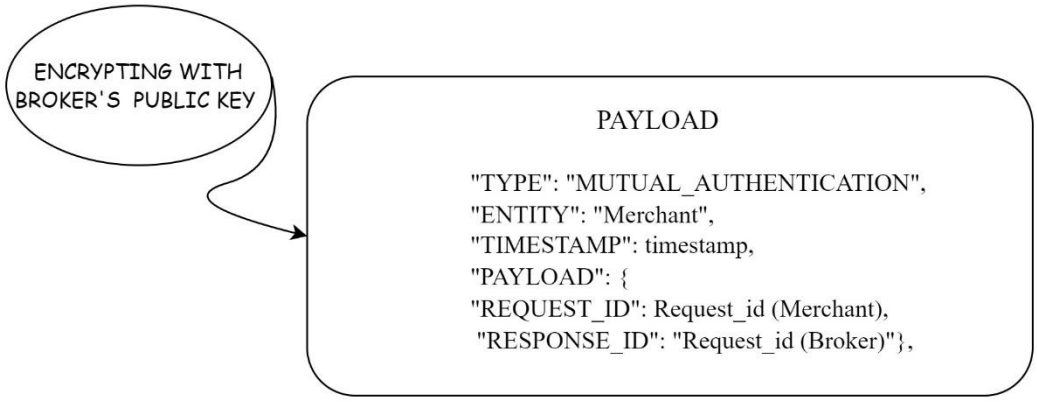
WORKFLOW OF AUTHENTICATION

1 - Broker Sends Authentication payload to Merchant
2 - Merchant sends back validation payload
3 - Customer sends login Creds for authentication
4 - Broker validates it and send back the payload
5 , 6 - Customer sends authentication payload to Merchant through Broker
7 , 8 - Merchant sends response payload to Customer Through Broker

- Initially, the broker sends an authentication payload (using the RSA public key), i.e., the payload is encrypted with the merchant's public key so that only the merchant can decrypt it “with his private key.” The payload contains message type: Mutual_authentication, Entity: Broker, Current timestamp, and a request id (This ensures that if the merchant decrypts the message and responds with this request-id, then authentication is done) as shown below:

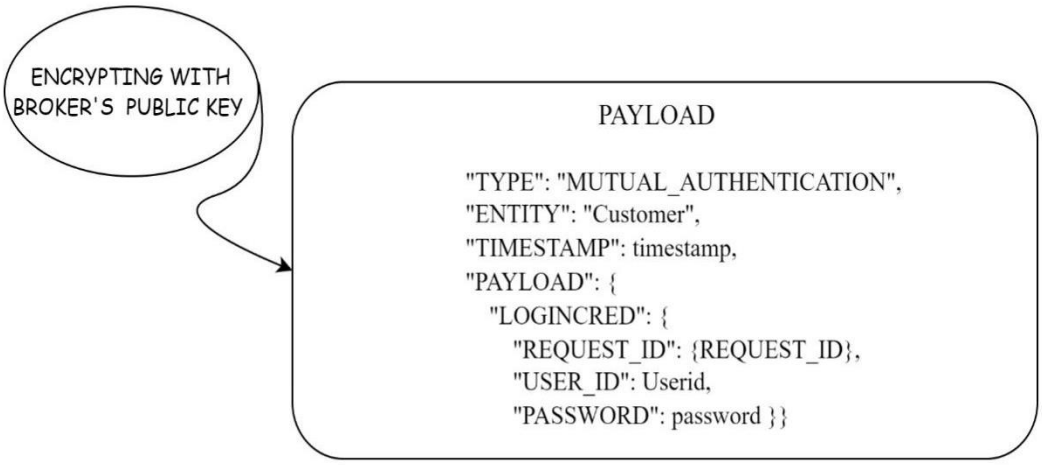


- Later, the merchant validates the message by decrypting the payload with his private key and sends a message to the Broker, encrypting it with the “Broker's public key” so that only the Broker can decrypt it “with his private key.”
- The Payload contains the message type “MUTUAL_AUTHENTICATION,” Entity as Broker, Timestamp, “REQUEST_ID of Broker,” and “Response_ID of Merchant,” as shown below.
- Broker validates the payload sent by the merchant by decrypting it with his private key, checks for the valid timestamp, and also checks the request-id, if it matches with the before-sent request ID, then the broker sends Mutal authentication is successful with the response-id sent by the merchant, this ensures the merchant that the broker is a legitimate user. This is how Broker and Merchant mutually authenticate.

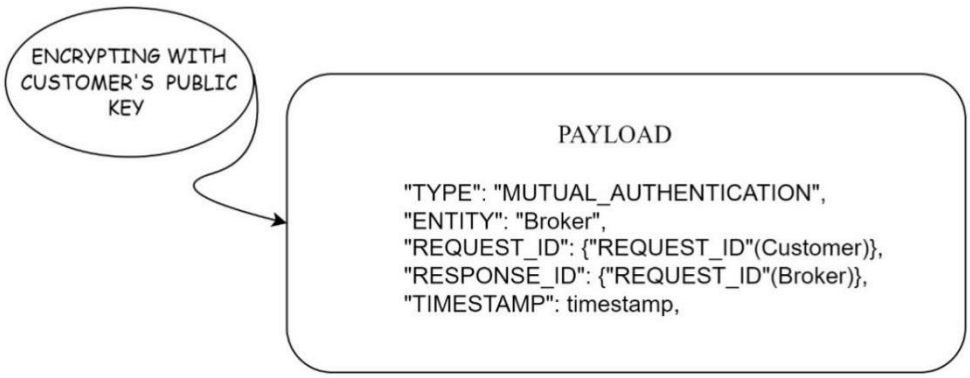


Customer and broker mutual auth –

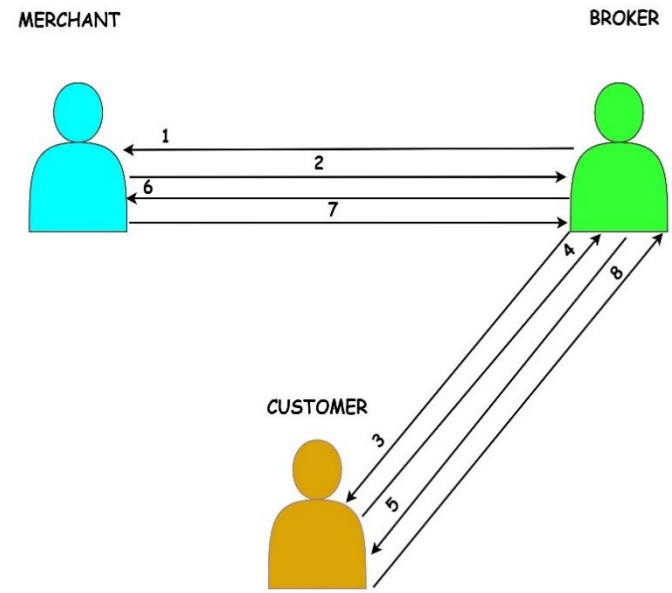
- Firstly, the Customer provides login credentials: username and password, these details are appended to the authentication payload and sent to the broker, encrypting it with the “Broker's public key” so that only the Broker can decrypt it “with his private key.”
- In the payload part, the Customer includes Type “MUTUAL_AUTHENTICATION,” Entity Customer, a timestamp, Login Credentials, and Request_id of the customer.



- Later, the Broker verifies the message by decrypting the payload with his private key. Then, the Broker verifies the login credentials of registered users. If the credentials get validated, the broker responds with an authentication message to the customer, if the credentials sent by the customer are invalid, then the broker aborts the authentication transaction and notifies the customer the credentials are invalid.
- The validated response payload contains the message type mutual authentication, entity: broker, Request_id of the customer, response_id of Broker, and current timestamp are shown below. Then, the customer decrypts with his private key validates the timestamp, and request ID.
- If the request matches the response with authentication, it is successful with the response ID of the broker. This is how Broker and Customer mutually authenticate.



Diffie Hellman Key Exchange -



WORKFLOW OF DIFFE HELLMAN(AFTER MUTUAL AUTHENTICATION)

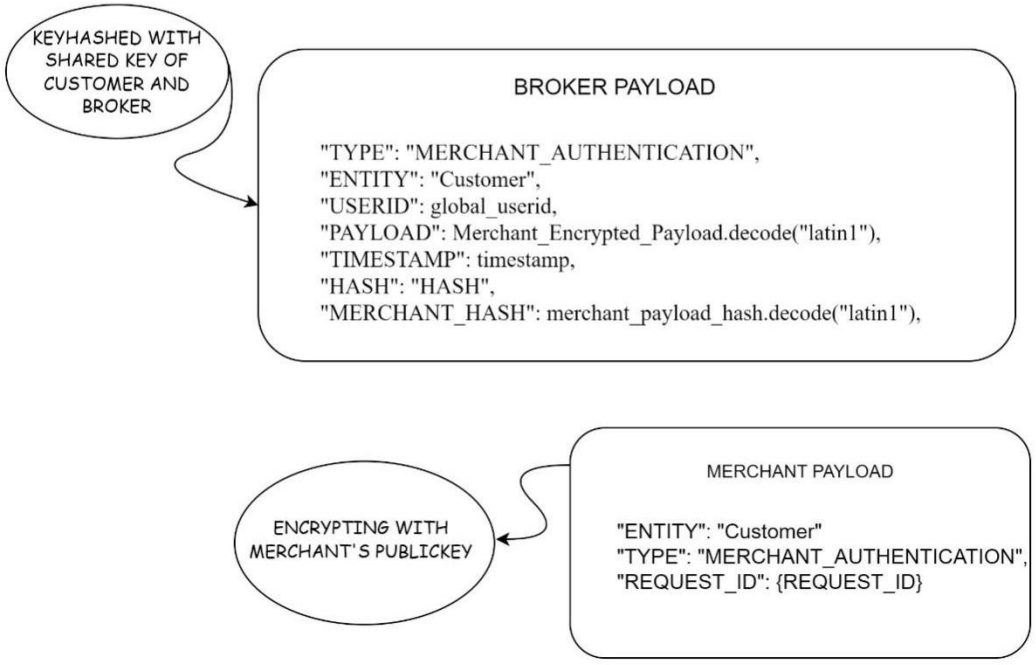
1 - Broker sends his genrerated key to Merchant(futher calculates shared key)
2 - Merchant Sends his generated key to Broker(further calculates shared key)
3 - Broker sends his genrated key to Customer (further calculates shared key)
4 - Customer sends his generated key to Broker (futher calculates shared key)
5 , 6 - Customer sends his generated key to Merchant through Broker
7, 8 - Merchant sends his genearted key to Customer through Broker

- After mutual authentication is done, entities share a secret key using the Diffie Hellman Key exchange method, as per the project statement, let us assume both Broker and Merchant, Broker and Customer, Merchant and Customer agree on a prime number and generator, and each entity generates a random public and private key pair using that prime number and generator. The public key is shared between both entities, and entities further use this public key to calculate the shared key. The key between the merchant and customer is shared after the customer authenticates with the merchant.
- The Diffie-Hellman workflow is shown above.

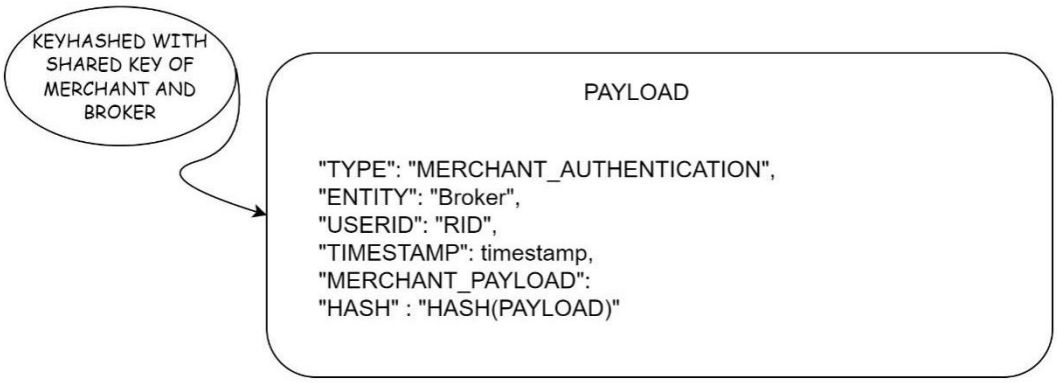
$$\boldsymbol{For\ shared\ key} == (\boldsymbol{Other - publickey})^{\boldsymbol{self}_{privatekey}}\ \boldsymbol{mod\ prime}$$

Customer authenticates merchant and establish secure communication–

- After mutual authentication with the broker-–merchant, broker –Customer, and keys are shared, the customer wants to authenticate with the merchant and establish secure communication between the customer and merchant, as the broker should not be able to read the communication.
- So, to authenticate with the merchant, the customer sends a auth payload to the broker, and the broker forwards the message to the merchant.



- Here, it is not mutual authentication; only the customer needs to be authenticated.
- So, to authenticate with the merchant, the customer sends an authentication payload, where the payload contains Message type: Merchant authentication, Entity: Customer, user_id, merchant payload, current timestamp, hash of merchant payload (which works as integrity check). Merchant payload contains an entity, message type, and Request id of the customer; merchant payload is encrypted with the merchant's public key and adds to the broker payload (as shown above) and Key hashing the whole message with a shared key between the broker and customer and sends it to broker.
- The broker decrypts it with a shared key and sees the message type Merchant authentication verifies the user and replaces the user id with a random id, and then the broker forwards the message to the merchant by implementing a shared key hashed of payload.



- Broker forwards the payload with contents: message type as "MERCHANT_AUTHENTICATION," entity as a broker, userid as RID, timestamp and Merchant payload, and "HASH (merchant payload)." As shown above.
- Then, the merchant decrypts the payload using the shared key and then decrypts the merchant payload with his private key and validates it.

Key hashed operation :

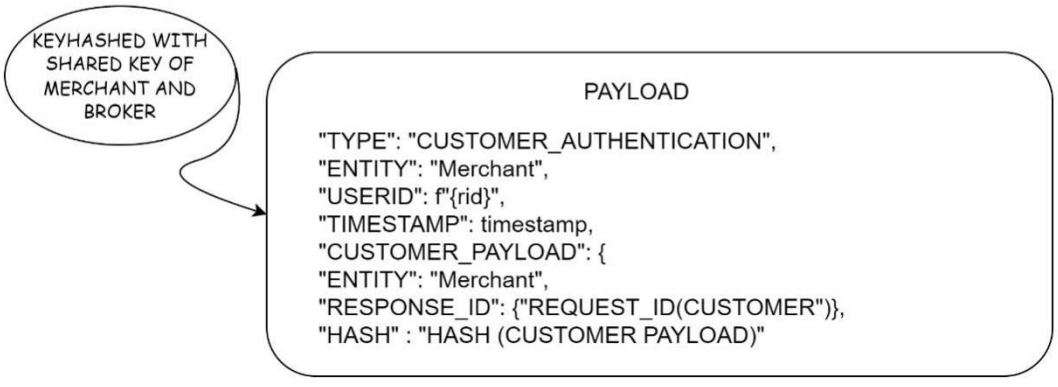
$$A = MD(shared\ key, IV)$$

- Message = {"Timestamp": "T", "MSG": "SUCCESS"}

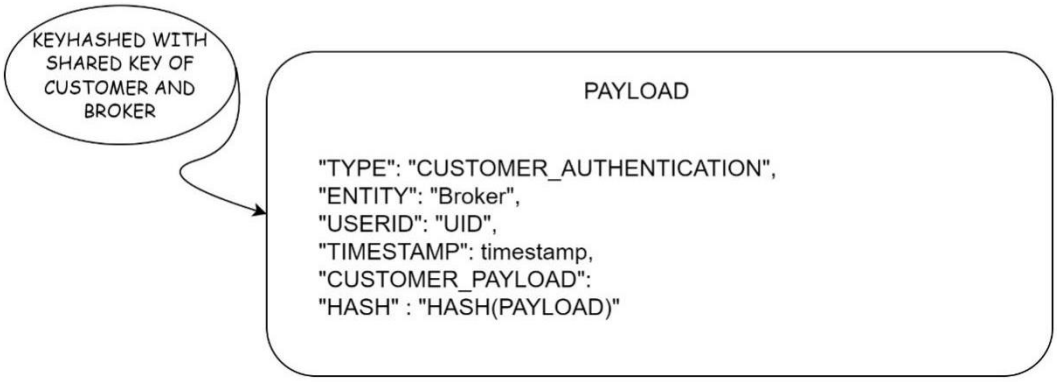
$$C = Message\ XOR\ A$$

Hash(C): MD(Message | shared key)

- Then responds with a payload, encrypts the payload with shared key and sends the key hashed payload to the broker.
- The payload send to broker includes message type as "CUSTOMER_AUTHENTICATION",
  entity as Merchant, Userid as rid, timestamp and Customer payload as"{Entity:"MERCHANT", RESPONSE_ID:"REQUEST_ID(CUSTOMER)", HASH: "HASH(CUSTOMER_PAYLOAD)"} as shown below:



- The Broker decrypts the message with “his shared key” and sees the message type: “CUSTOMER_AUTHENTICATION”.
- Then the broker forwards the message to the customer by replacing RID with user id encrypting it with the shared key between broker and customer as shown below.



- Then, the customer decrypts with the shared key.
- Customer authenticates merchant by validating the timestamp and the request-id he has sent and comparing the hash value of the payload and the timestamp he has received and checks the integrity of the message by comparing the hash value of customer payload. This is how the customer authenticates the merchant.

Possible attacks that are mitigated in authentication -

- **Spoofing:** By using a unique and random user ID for each customer, the broker ensures that even if an attacker learns a user ID, they can't easily guess the IDs of other users.
- **Replay Attacks:** There is no chance of a replay because the messages are encrypted. Only the user with a proper key can decrypt it. And the timestamp is used to mitigate the replay attack.
- **Man-in-the-Middle Attacks (MitM):** MITM can be possible only if keys are accessible, but as Keys are stored in a password-protected DB server database reading is not possible thus MITM is not possible, and RSA encryption is tough to break.
- **Rainbow table attack:** We mitigate the rainbow table attack on the Database’s password as we are using the salt technique to store the password in a hash format.

Transactions after Authentication

- Once authentication is successful and each stakeholder has exchanged the session keys, we can use the following encryption scheme to send data across the stakeholders.
- Here are the keys and IV value with each stakeholder:
    - K1-i, IV1-i with Customer-i and Broker (as there are n number of customers, i denotes ith customer)
    - K2, IV2 with Broker and Merchant.
    - K3-i, IV3-I with Customer-i and Merchant.

- Each stakeholder encrypts the message they are sending by using the secret key hash method as explained below:

ENC- 1:

- Let the message being sent be plain text P, and IV-AB is the initial vector which is a random value concatenated with the Key KAB which was exchanged between the parties A and B during the authentication process. This scheme is used whenever the parties need to send a message to each other using the keys and IV values specific to them.
- We break the message into MD-length chunks p1, p2,p3….pn. Let the cipher text be c1, c2, c3…cn, and the intermediate values b1, b2, b3…bn from which we will computer each cipher text block.

b1 = MD(KAB | IV-AB)	c1 = p1 XOR b1
b2 = MD(KAB | c1)	c2 = p2 XOR b2
…..	……
bi = MD(KAB | ci-1)	ci = pi XOR bi
……	……

DEC-1:

- For decryption, the other side B does the following –

  So if bi = MD(KAB | pi-1), we would decrypt in the following manner:
  So, KAB is available with A and B, IV (Initialization Value) was already shared between A and B during authentication and Ci will be sent over the network to B. We can decrypt Ci to get Pi in the following manner:

b1 = MD(KAB | IV)	P1 = b1 XOR C1
b2= MD(KAB | C1)	P2 = b2 XOR C2
b3 = MD(KAB | C2)	P3 = b3 XOR C3
….	
bi = MD(KAB | Ci-1)	Pi = bi XOR Ci

  The stakeholder B will get all the Pis and concatenate and take the action based on the contents of the plain text.

INT-1:

- The integrity of the messages being sent is checked using the keyed hash. Each party sending the message P to other party will include a hash MD(P|K), here K is the Key they share between each other.

- For every message sent across the sockets the hash of the message will be sent too along with the message, the hash of the message will be Hash(M|K), where M is the message being sent and key K is the shared key between the two parties.

- The HASH is bring sent along with the encrypted message, once the receiving entity gets the payload the entity will calculate the hash of the message by decrypting it and compare this hash with the hash they received.

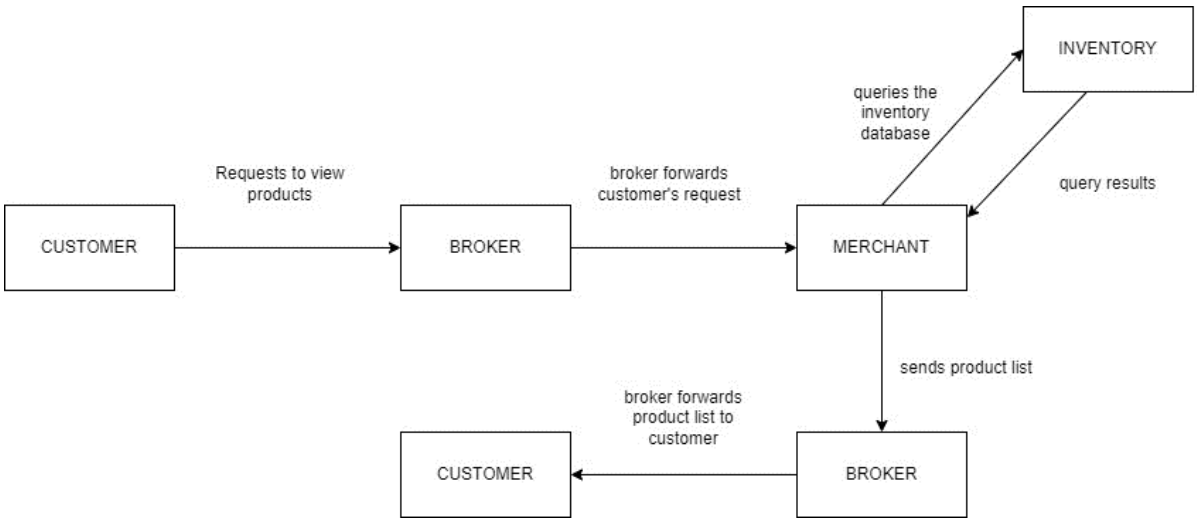
Assumptions:

- Each entity has a separate route maintained by the uvicorn application, so there is no ambiguity which entity is communicating. A route will differentiate and sends corresponding messages to those entities. This helps us in maintaining multiple customers.

- Broker also generates a random ID for each customer for every customer and maps it to the User ID and the socker ID he is talking to, this random ID is used to send the message to the merchant whenever he wants to send the message from the customer, this Random ID helps in safeguarding the real identity of the customer from the Merchant. This random ID is exchanged with the merchant during the authentication process between customer and merchant.

- Merchant whenever wants to send back a message back to the customer he re-uses the Random ID which was sent by the Broker. This also helps Broker in sending a reply to the Customer based on this Random ID as we have lot of customers.

Use case 1: Customer wants to view the products available at the merchant end.



Customer to Broker: Customer sends the request for viewing the products by generating appropriate payload to the broker.

Broker and Merchant: Once broker receives the message, decrypts it checks for the type of message since it's a message for merchant broker changes the UID to RID and remaining entities accordingly. Then forwards it to the merchant.

Merchant: Once Merchant receives the message and identifies as VIEW message after decrypting the payload, he queries the inventory and appends the products to the payload.

Merchant to Broker: Merchant encrypts the inner payload with the customer shared key and outer payload with the brokers shared key. Then sends to the broker.

BROKER: Broker upon identifying the message is for customer, he appends the merchant payload as it is inside the customers message and changes entity to FROM_MERCHANT.

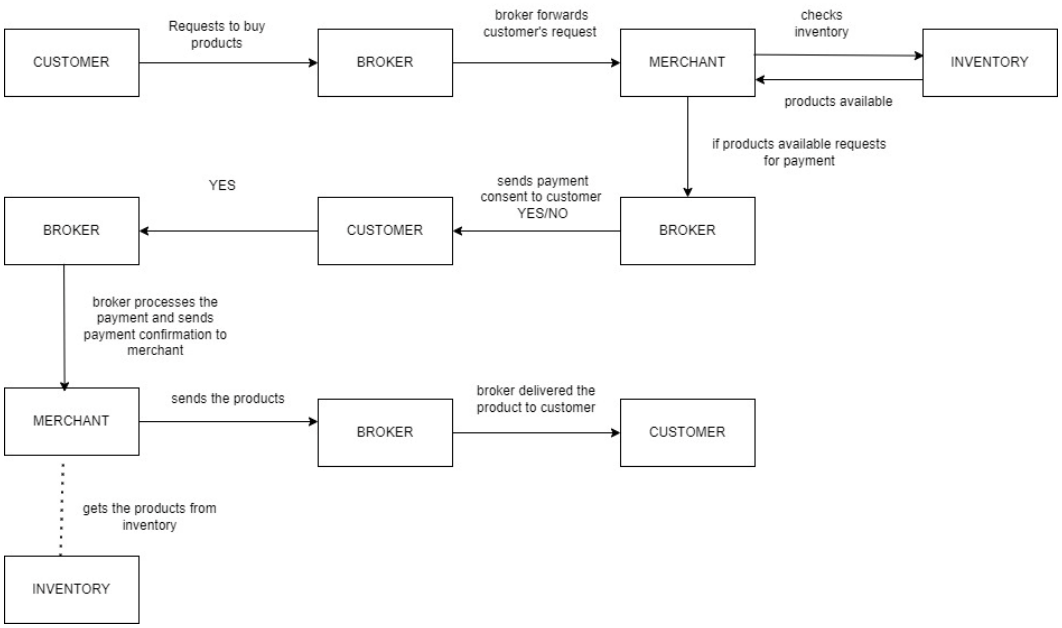
Broker to Customer: After changing the appropriate content in the message, broker encrypts the whole message with he share secret key between the broker-customer and forwards it to the customer.

Customer: Customer on receiving the message from broker, decrypts the whole message and decrypts the inner payload from merchant with the merchant-customer shared key. Then the products gets listed.

*****For every transaction, the integrity is maintained by appending the hash to the message, as well as integrity check is done at every stage by re calculating it. *****

Use case 2: Customer wants to buy products from the merchant.

Flow if products are available.



Customer to Broker: After viewing all the products customer sends a request to broker that he wants to buy some products.

Broker to Merchant: Once broker receives the message, decrypts it checks for the type of message since it's a message for merchant broker changes the UID to RID and remaining entities accordingly. Then forwards it to the merchant.

Merchant: Based on the Customer request merchant checks whether the specified products are available in the inventory or not. If the products are available in the inventory, then the customer sends that the products are available and request for payment to the broker.

Merchant to Broker: The broker sends the payment consent as YES/NO to customer saying that the products are available and requests for payment.

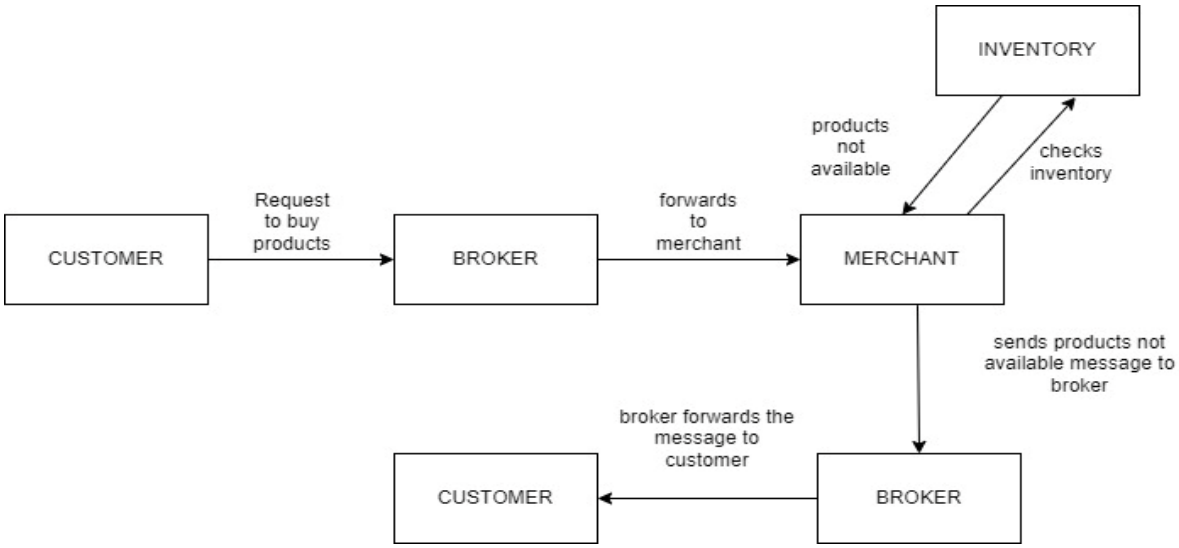
Broker to Customer: If the customer agrees on making the payment and then makes the payment.

Customer to Broker: The payments are handled by broker and the broker processes the payment and sends the payment confirmation to Merchant.

Broker to Merchant: After Receiving the payment the delivery of products starts.

*** Transaction_ID is added for security purpose to make sure that customer pays only for the requested products, this prevents payment replay_attacks.***

Flow if Products are not available.



Customer to Broker: After viewing all the products customer sends a request to broker that he wants to buy some products.

Broker to Merchant: Broker forwards the customer's request to the merchant that the customer has requested to buy the products.

Merchant: Based on the Customer request merchant checks whether the specified products are available in the inventory or not. If the products are not available in the inventory then the customer sends that the products are not available.

Merchant to Broker: The broker receives the message from Merchant and then forwards the merchant's message to customer.

Broker to Customer: The Broker sends the merchant's message that the products are not available in the inventory and the transactions ends.

Merchant delivers the product to customer through broker.

Merchant to Broker: After receiving the payment from the customer the merchant gets the products from the inventory and then updates the inventory and sends the products to the broker.

Broker: The Broker receives the products from the merchant and then sends the products to the customer.

Customer: The products are delivered by broker from merchant and the transaction ends.

*** We have added the random number of bytes product delivery payload, so that the delivery size is different for each product payload ***

```
customer_payload = {
        "TIMESTAMP": str(datetime.now()),
        "PRODUCTS": PRODUCTS,
        "RANDOM_BYTES": (random.randint(0, 1000) * b"x").decode(ENCODING_TYPE),
        "TRANSACTION_ID": cust.transaction_id,
    }
```

Attacks Mitigated in the above transactions:

- **Replay attacks:** Using the Timestamps and Transaction IDs we mitigate the replay attacks, the customer/merchant/broker make use of transaction id to keep track of the status of the transaction, if during these transactions there is any replay messages, they would know that there is some issue and discard the messages.
- **Man in the middle attacks:** The message digests "HASH" value in each of the message delivered helps in maintaining the integrity of the messages thus mitigating MITM attacks.
- **Eavesdropping:** We mitigate passive eavesdropping by encrypting the messages with Initial vectors IVs. And, we also mitigate the active eavesdropping as we are encrypting the messages as explained in ENC-1.

Contribution:

Teja Zangam – Customer – Broker, Broker - Merchant Authentication, DH_Key Exchange

Shivashankar Chandrashekaraiah – DH_Key Exchange, Customer – Merchant Authentication, Viewing Products, Buying Products, Integrating the code End-to-End.

Sai Santhosh Jella - DH_Key Exchange, Report

Sai Tharun Reddy Mulka – Viewing Products, Buying Products, Few Authentications Part, Integrating the code End-to-End. Report.