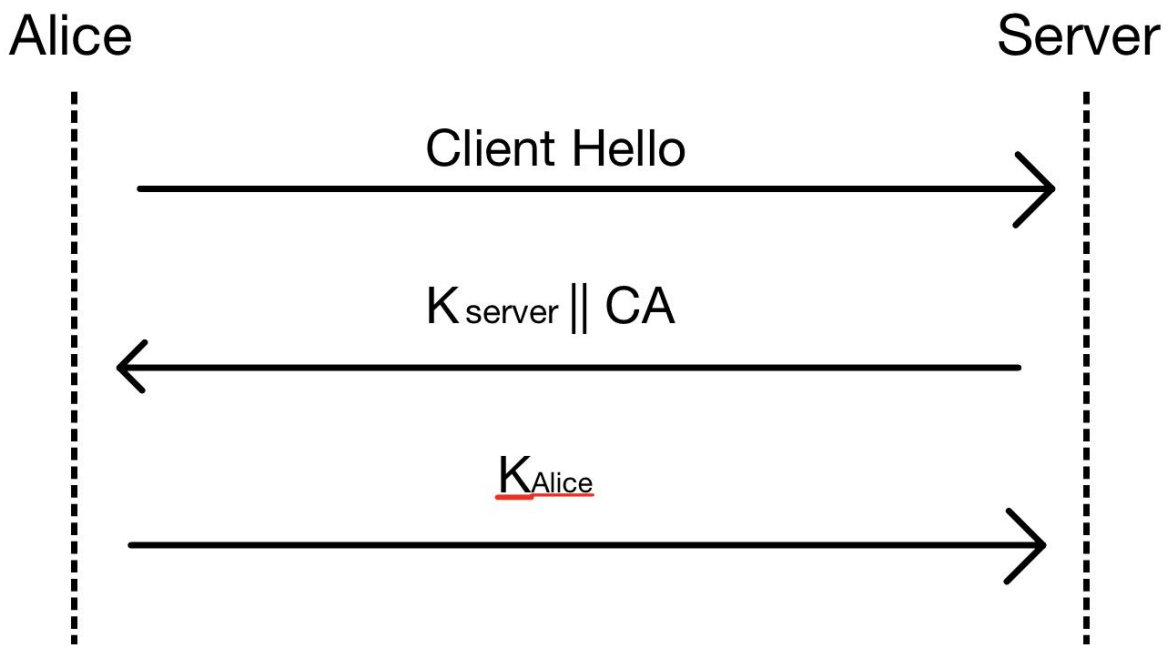
Secure Chat Service with File Sharing

How does the program meet the minimum requirements?

1. Our project uses a client server model. Our project takes command line input when running the .jar of the project (ServerDriver is main) and a configuration file can be specified. The configuration file is formatted in JSON. Protocol messages are all sent in JSON format. The client application (ClientDriver) can specify the port of the server in the clients configuration file.
2. Method of authentication users: Our project authenticates users using TOTP. The client establishes a TLS connection over SSL Sockets. The server has a keystore, and the client has a truststore.
3. Our project provides end to end confidentiality and authenticity. The messages are sent over a TLS connection. When a client connects, they establish an ECDH-KEX shared secret with each other client connected. When sending messages or receiving files, an AES key is generated to encrypt the message or file. That AES key is encrypted with the shared secret for each recipient. That AES key can only be decrypted by a recipient that knows the shared secret. The server does not have the ability to decrypt messages or files. Each message or file sent is sent with a nonce. That nonce is checked in the recipient's nonceCache to ensure message freshness.
4. Our project utilizes session keys: A secret key is derived from the ECDH-KEX shared secret for each client (Each pair has a shared secret). This is established when the connection begins and is not established again.

5. Our project resists replay and MITM attacks: The project resists replay attacks by employing nonces for each message or file sent. The project resists MITM attacks by deploying TLS with DH-KEX, because the connection with the server is secure; DH-KEX is not vulnerable to MITM and we can establish our shared key.

TLS Protocol Diagram:

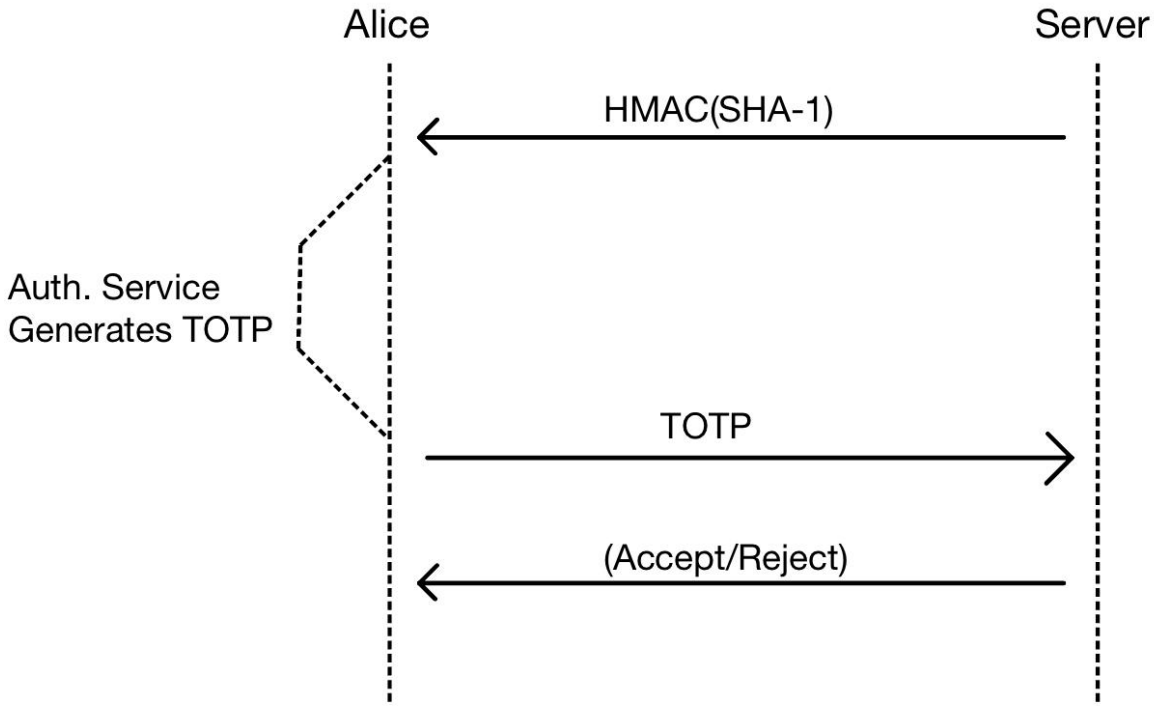


TLS/SSL Diagram Description:

ServerDriver reads in a keyStore from its configuration file and creates an SSL socket for each ClientDriver wishing to connect. When a ClientDriver wishes to connect with ServerDriver, it begins the socket handshake using TLS v1.3. ServerDriver sends its certificate to ClientDriver

from keyStore.jks and its public key. ClientDriver checks its trustStore.jks to see if the certificate was issued by a trusted CA. If the ServerDrivers certificate is trusted, ClientDriver sends its public key and they establish a shared secret which is used for secure communication.

TOTP Protocol Diagram:

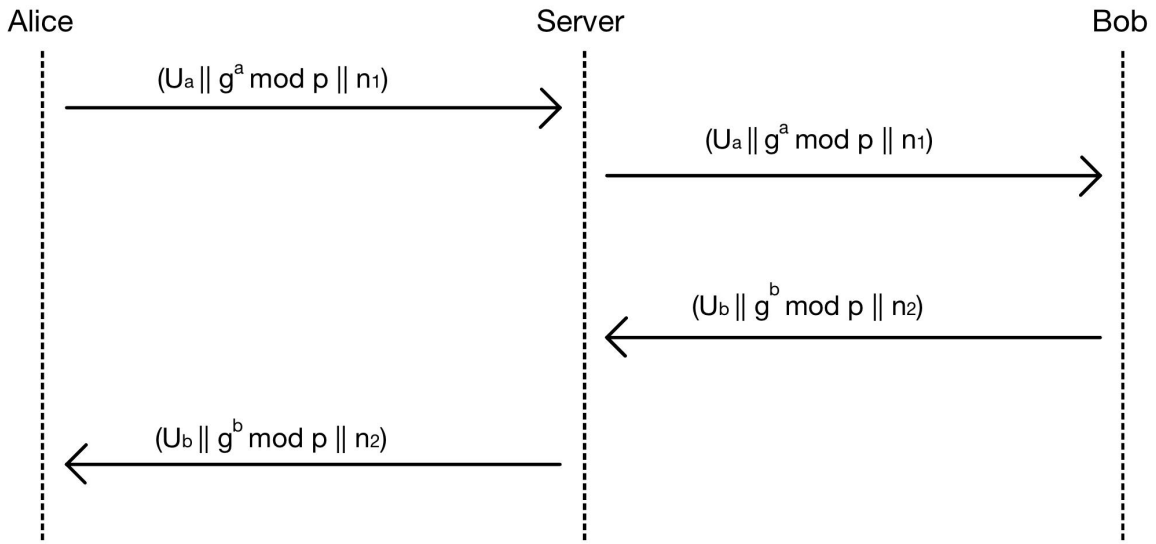


TOTP Create/Login Diagram Description:

Upon account creation, ClientDriver sends to ClientThread its username and password. The ClientThread class creates a TimeOTP object. The TimeOTP object's constructor initializes an HmacSHA1 SecretKey. ClientThread sends to ClientDriver a payload which contains the

Base32 representation of that SecretKey. ClientDriver constructs a QR as a buffered image using the Zebra Crossing library from the Base32 HmacSHA1 SecretKey and the clients username. The user scans that QR using Google Authenticator to receive access to the generated OTP's. Upon logging in to an account, ClientDriver sends to ClientThread its username, password, and an OTP derived from Google Authenticator. ClientThread verifies the OTP sent matches the expected OTP from the TOTP in from the account entry by: Creating a TimeOTP object with a basetime of 0 as well as the TOTP from the entry, and calculating (current time - basetime - 30 seconds). ClientThread verifies that the hash derived from the password in entry matches the hash derived from the pass sent from ClientDriver. If both the OTP and the password are verified, ClientThread sends a payload containing the status True to ClientDriver, and ClientDriver is subsequently logged into the service.

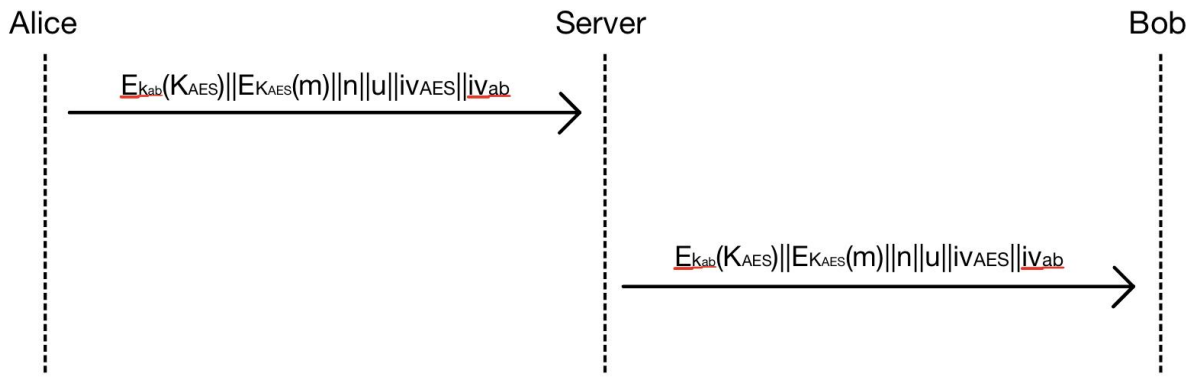
EC-DHKEX Protocol Diagram:



EC-DHKEX Diagram Description:

Once a user is authenticated with the server, an ECDH-KEX takes place for each other user authenticated with the server. ClientDriver constructs a SecObj. The SecObj constructs an ECDH-KEX key pair. ClientDriver sends to ClientThread a payload with status: exchange, a nonce, its username, and its public key (derived from the keypair in SecObj). ClientThread sends to all other ClientThreads a payload with the same information, which is received by each ClientDriver. Recipient ClientDrivers derive a shared secret from the received public key in SecObj. We use X.509 PKI to get an EC instance of the received public key which we compute with our private key to derive the shared secret. Recipient ClientDrivers send a payload which contains status: exchange2, their public key, their username, and a nonce to ClientThread. ClientThread sends to each ClientDriver the same payload. Recipient ClientDrivers of status:exchange2 derive a shared secret the same way. Each pair of ClientDrivers now share a secret.

Session-Key (Message/File-Share) Protocol Diagram:



Session-Key (Message/File-Share) Diagram Description:

When a ClientDriver sends a message or file, a payload is constructed in SecObj for each user which we share a secret with. An AES key and random IV are constructed. The AES key is used to encrypt the message or file using AES/GCM/NoPadding. A Session-Key and random IV are constructed from the shared secret using AES/GCM/NoPadding. The AES key is encrypted with the Session-Key. A payload is returned to ClientDriver which contains: The AES key IV, the encrypted AES key, the IV used to encrypt the AES key, the ciphertext of the message or file, a nonce, and the sender's username. ClientDriver sends to ClientThread this payload. For each authenticated user, ClientThread sends to all other ClientThreads this payload, which is passed to their respective ClientDrivers. Recipient ClientDrivers decrypt the encrypted AES key in the payload using their Session-Key derived from the shared secret with the sender. Recipient ClientDrivers decrypt the ciphertext using the decrypted AES-Key.