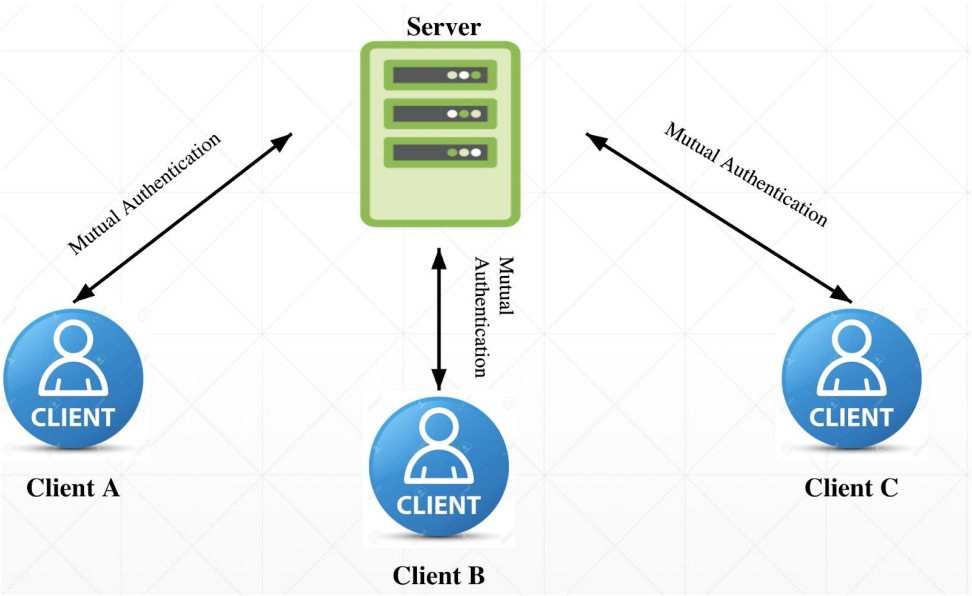
PS4

Yushen Ni, Ryan Joshua Dsilva

1. Architecture

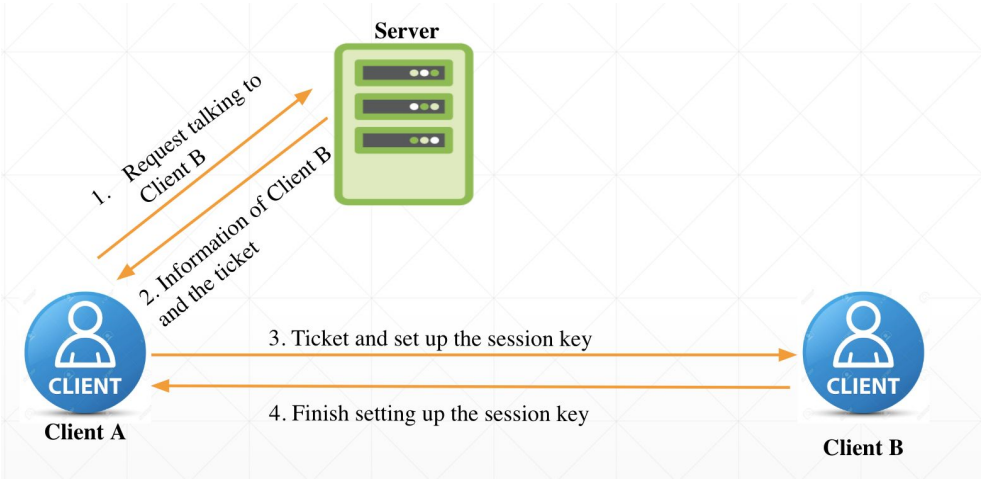
We set up a client-server architecture.

For client logging in the server:



We need mutual authentication between client and server.

For client talking to another client:



a. Client needs to inform the server who wants to talk to, let's say, B for example.
b. The server will send back the information of B (IP address, Port information) and the ticket to B.
c. Client will use this ticket to inform B "I want to talk to you and I am legitimate, let us set up our session key!"
d. B will send back his part of the session key and prove he is really "B".

After these steps, Client A and B finish mutual authentication and set up the session key.

2. Service

Users could enjoy these services:

a. Log in
b. List (Know all the users who have logged in)
c. Send message (Send any logged-in user messages)
d. Log out

Extra service:

a. Mutual authentication
b. Message integrity
c. Confidentiality
d. Some protection on weak password
e. Perfect forward secrecy
f. Resiliency to denial of service attacks
g. Identity hiding

3. Assumptions:

- User:
  - The user only requires his password for authentication.
- Client:
  - The client does not have its own public or private key.
  - The client knows the server's public key.
  - Each client maintains their own private constant, e.g, *a* for A, in the Diffie Hellman key exchange, for the duration for the session.
  - Client connections will terminate under non-abnormal conditions.
- Server:
  - The server has stored the salt used alongside the *h(password || salt)* in the serverside user database.
- All nodes:
  - The peers and server delete their established session keys at the end of each session.
  - Server and Client know g and p of Diffie Hellman.
  - Session keys cannot be cracked by brute-force attacks during the session.
  - All nodes are tightly time-synchronised.

4. Algorithms Used:

- Encryption: AES-256 with GCM mode

  AES is the current standard for reasonably secure block-cipher encryption, which allows for higher key sizes than its predecessor, DES, and is reasonably performant given these key-sizes. The Galois Counter Mode of parallel encryption and message code authentication allows for authenticated encryption, negating the need for message integrity verification, which supports block sizes of 128 bits.

- Key exchange: Built on Diffie-Hellman key exchange

  The mutual authentication mechanisms designed for this application are based on the Diffie Hellman symmetric key exchange scheme, with checks instituted in place to prevent man-in-the-middle attacks.

- Signature keys used: Pre-generated using RSA, 2048 bit size

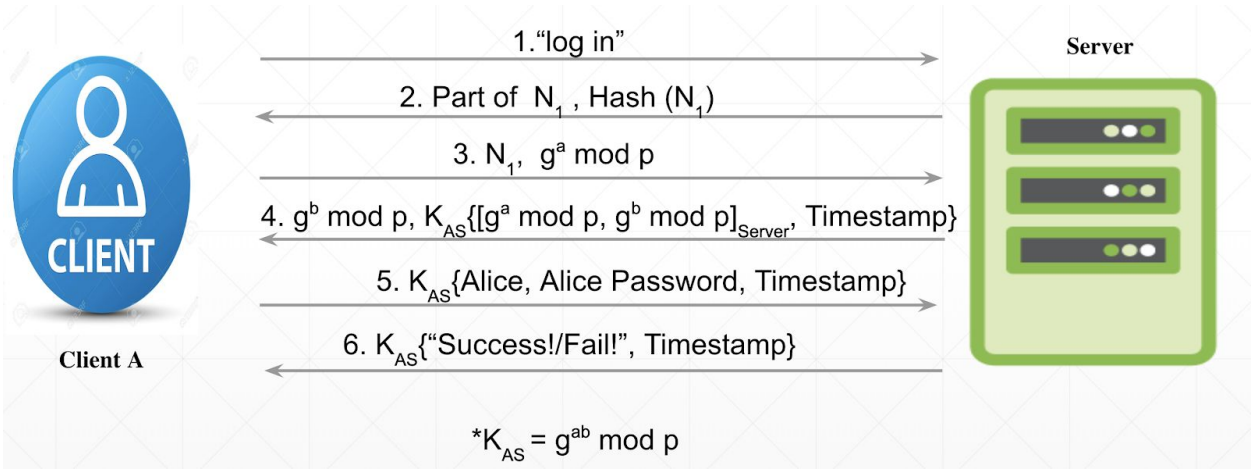
  RSA can be used to produce an asymmetric public-private key pair as the server's public-private keys, the 2048 bit key size is considered fairly resilient and future-proof and mathematically infeasible to crack in a reasonable span of time, and the keypair for the server will be pre-generated so the only computation time in program operation would be the verification time.

- Hashing algorithm: SHA256

  This hash function is the current standard and is fairly resilient to collision attacks.

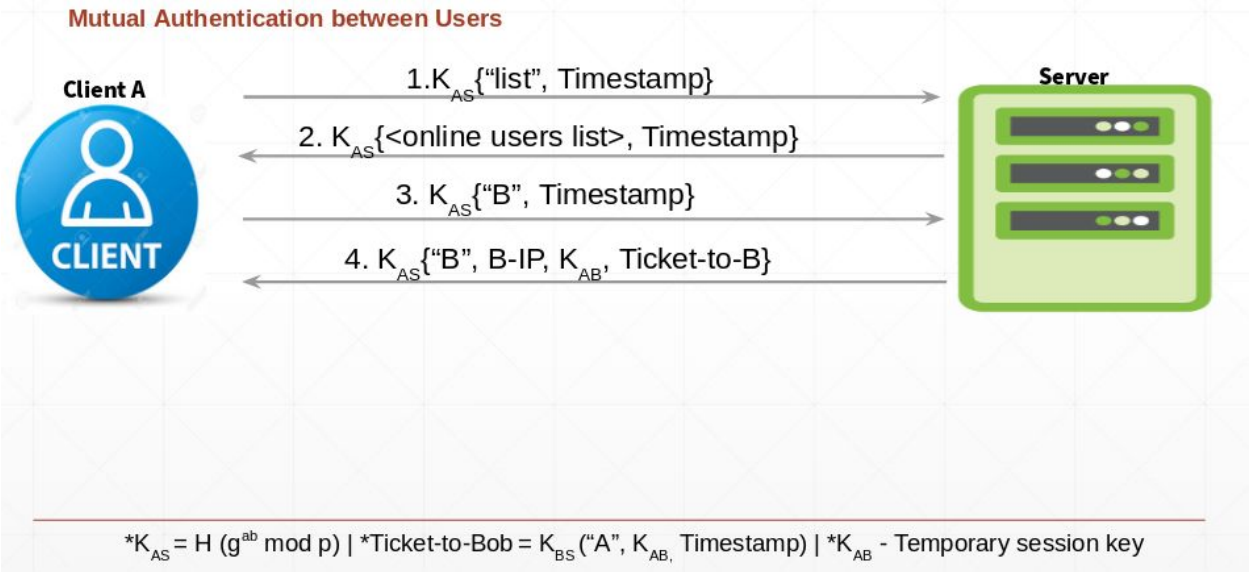
5. Protocols

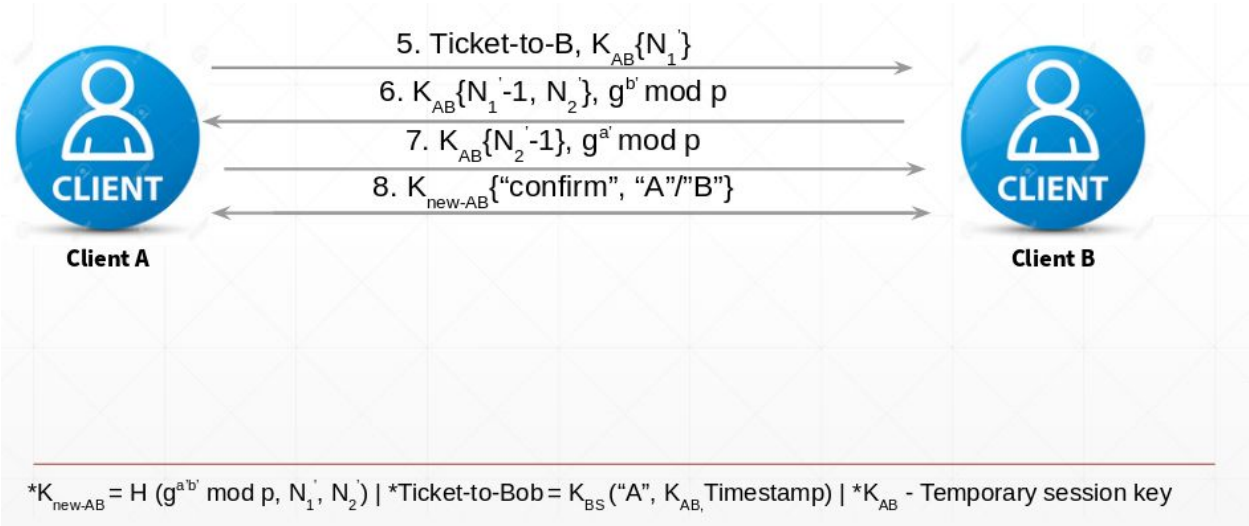
Mutual authentication between the user and the server



1. Client sends the "log in" request.
2. Server sends back a puzzle to protect it from denial of service attack:
    a. N_1 is a 128-bit nonce.
    b. Part of N_1 means from 27 bits to 127 bits of N_1
    c. Puzzle: You need to figure out 0 bit to 19 (originally 0 to 26) bits of N_1 to have the same hash value
3. Client sends the answer of the puzzle: N_1, and also sends g^a mod p to set up a session key to hide the identity.
4. Server sends back g^b mod p and they finish setting up the session key: $K_{AS} = g^{ab}$ mod p. Use this session key to encrypt [g^a mod p, g^b mod p]$_{server}$ and the timestamp. First one is the signature to prove "I am the server" and the second one is the timestamp to protect it from replay attack.
5. Now, Client knows that the other side is the REAL Server. He could send his information encrypted with the session key K_{AS} to prove himself, also, adding a timestamp to protect it from replay attack.
6. Server receives the username and password, and he will compare the salted hash result with his database. Later, he will send back the verification result and the timestamp.

Mutual authentication between the users



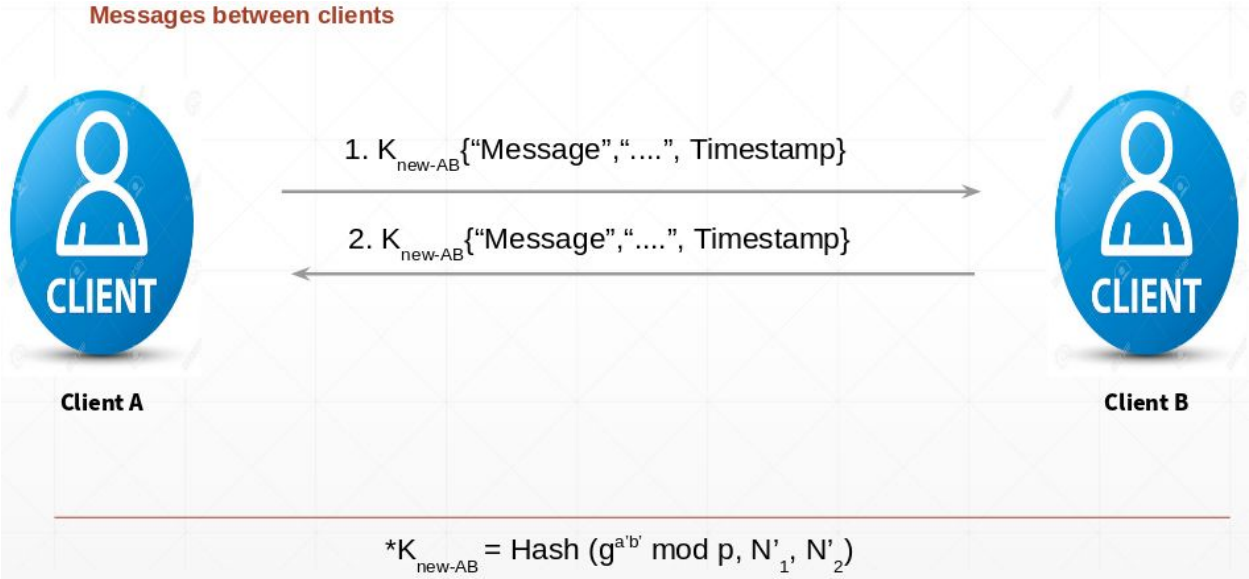


1. The Client requests a list of the users currently online users, timestamped to prevent replay attacks or denial-of-service scenarios.
2. The Server returns a fresh timestamped list of currently online users.
3. The Client makes a fresh request for another particular Client from the aforementioned list.
4. The Server returns the address details, a temporary shared key K_{AB} and a ticket to that user, which is, in this case, K_{BS}("A", K_{AB}, IV, Timestamp}. This is accompanied by a clearstring of the user whose details are being received, to prevent a bad actor replaying wrong contact credentials.
5. The Client then sends this ticket to the other Client it wishes to contact, with a 128-bit nonce N_1' encrypted with the shared key, for verification.
6. The other Client responds with the solved nonce challenge, and its own, N_2', encrypted

with the temporary shared key, along with its contribution to the Diffie Hellman key establishment.

7. The initiating Client responds in kind, with the solved challenge and its own contribution.
8. These messages are used to confirm the shared key established between the two clients, which in this case would be $K_{new\text{-}AB} = H(g^{a'b'} \bmod p, N_1', N_2')$, encrypted with this new shared key. The temporary key used until now is discarded at this point.
9. All these messages (messages 5 through 8) are now accompanied with timestamps to prevent possible attacks.

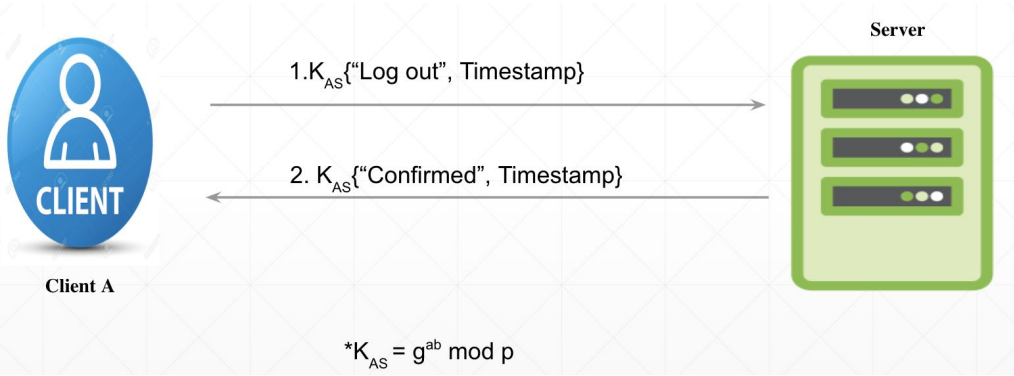
Messaging



1. Client A sends a communication of the 'Message' type with the message itself, timestamped to prevent replay attacks.
2. Client B responds in a similar fashion.

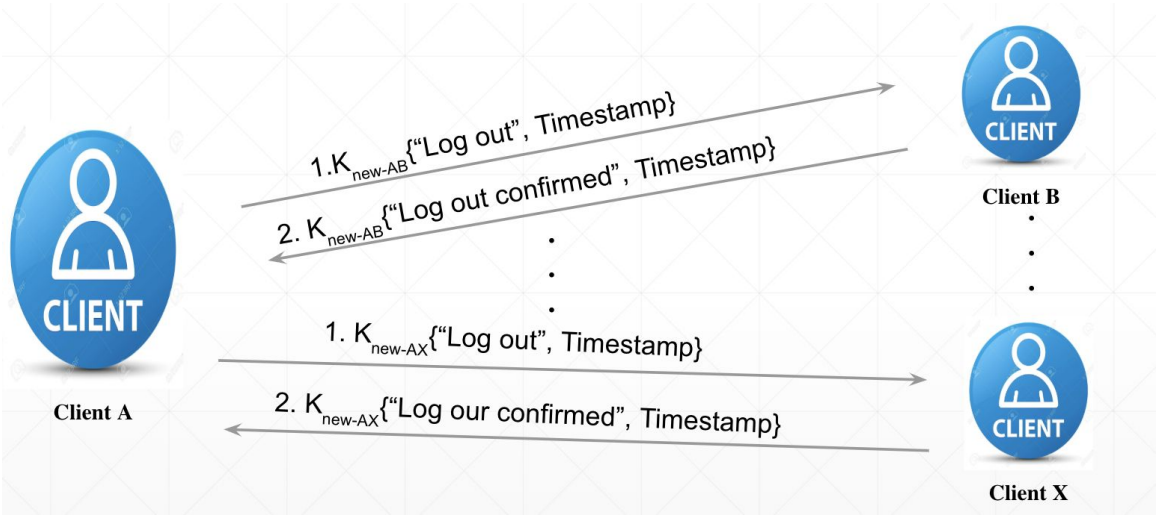
Log out

1. Log out from the Server



1. Client sends "log out" request and Timestamp encrypted with K_{AS} ($K_{AS} = g^{ab} \bmod p$, set up earlier between Client and Server).
2. Server sends back the confirm information with the Timestamp.

2. Log out from other Client



Client needs to log out from all the other clients that he has communication with.

1. Client sends "log out" and the Timestamp, both of which are encrypted with the new session key between Client A and Client B (K_{new-AB} = Hash ($g^{a'b'} \bmod p$, N'_1, N'_2))
2. Client from the other side sends back "log out confirmed" and the timestamp, using the K_{new-AB} encrypted. After this, both Client will forget K_{new-AB} session key, a', b', N_1', N_2'

6. Discussion

Discuss the following issues:

1. Does your system protect against the use of weak passwords? Discuss both online and offline dictionary attacks.
   Yes.
   Online attack: We will limit the trying time to 3. If the user has failed 3 times, he needs another 30 minutes to try again.
   Offline attack: In our database of the server, we do not only store hash of the password, instead, for every password, but it also has his own salt and we store hash(password|salt). It needs the adversary to generate a new precomputed table for all possible password values hashed with that specific salt, meaning that even two users with the same password but different salts would require two different tables to attempt to crack the same password.

2. Is your design resistant to denial of service attacks?
   Yes.
   For Client logged in the server, we set up a puzzle to figure out to resist denial of service attack.
   For Client talking to another Client, Client needs to talk to Server first. We use the Timestamp to resist replay attack and protect it from the denial of service.

3. To what level does your system provide end-points hiding, or perfect forward secrecy?

   End-points hiding: When Client wants to log in Server, they set up the DH session key to hide the end-point. Also, Client needs Server to prove himself first and later sends the user name and password, which also provide more protection on end-points hiding.

   Perfect forward secrecy: For every Client-Server and Client-Client conversation, they set up a DH session key first, which provide perfect forward secrecy.

4. If the users do not trust the server can you devise a scheme that prevents the server from decrypting the communication between the users without requiring the users to remember more than a password? Discuss the cases when the user trusts (vs. does not trust) the application running on his workstation.

   For our design, what the Server needs to do is give the two Clients participating in the conversation a temporary key. Later, the two Clients will set up their own session key。Server has no idea what it is because it is DH session key, and Client will have conversation with this new session key. This scheme requires no additional information from the user aside from his initial password entry, and does not require any certificates on behalf of either Client in the communication.

   Cases of trust:

   - The user may trust the communication between self and the other user to stay unknown to the only other actor legitimately involved in the authenticated

process, the Server.

Cases of mistrust:

- The only case where the application cannot be trusted is if the Server's public certificate and IP address on the client application has been replaced with those of an adversary.