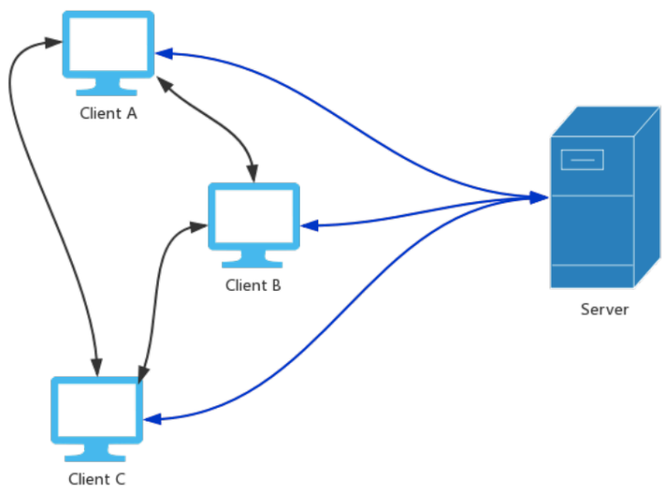
Secure IM Design

Chi Zhang & Yuyang Zhang

1. *Architecture:*

The system is based on client-server architecture.

- Server listens to a port with a TCP server socket. It can manage online and offline users, and help them to authenticate each other.
- Client has a TCP connection with the server. Based on the connection, the client can login, logout, get user list from the server and get authorization to talk to other clients.
- Client also listens to a port with a UDP server socket. Based on it, clients can exchange messages with each other.
- Services like PFS, identity hiding, DoS Protection, etc. are provided

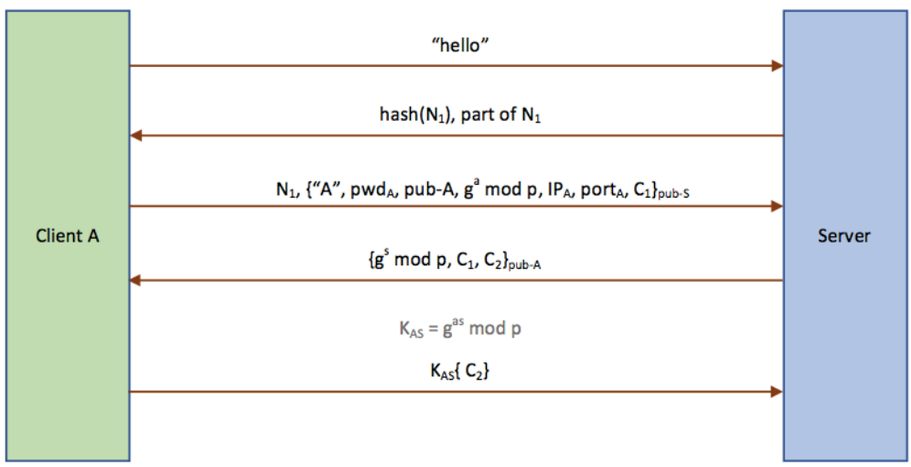


2. *Assumptions*:

- Each user knows his/her password
- All clients know the server's public key
- Server knows all users' names, along with salted hashes of their passwords and salts
- Everyone knows g and p, which are used for Diffie-Hellman key exchange

3. *Protocols*:

a. Authentication protocol (login)



When client A tries to login the server, it will start the following process:

1) Client A sends a "hello" message to the server.
2) Server generates a 128-bit nonce N_1, sends back part of it (the 21-128 bits) and its hash value.
   N_1 is used to avoid the DoS attack.
3) Client A finds N_1 through computations and generates a DH component a.

Then it generates a RSA key pair (pri-A and pub-A), sends N_1 and ("A", pwd_A, pub-A, g^a mod p, IP_A, $port_A$, C_1) encrypted with the server's public key to the server.

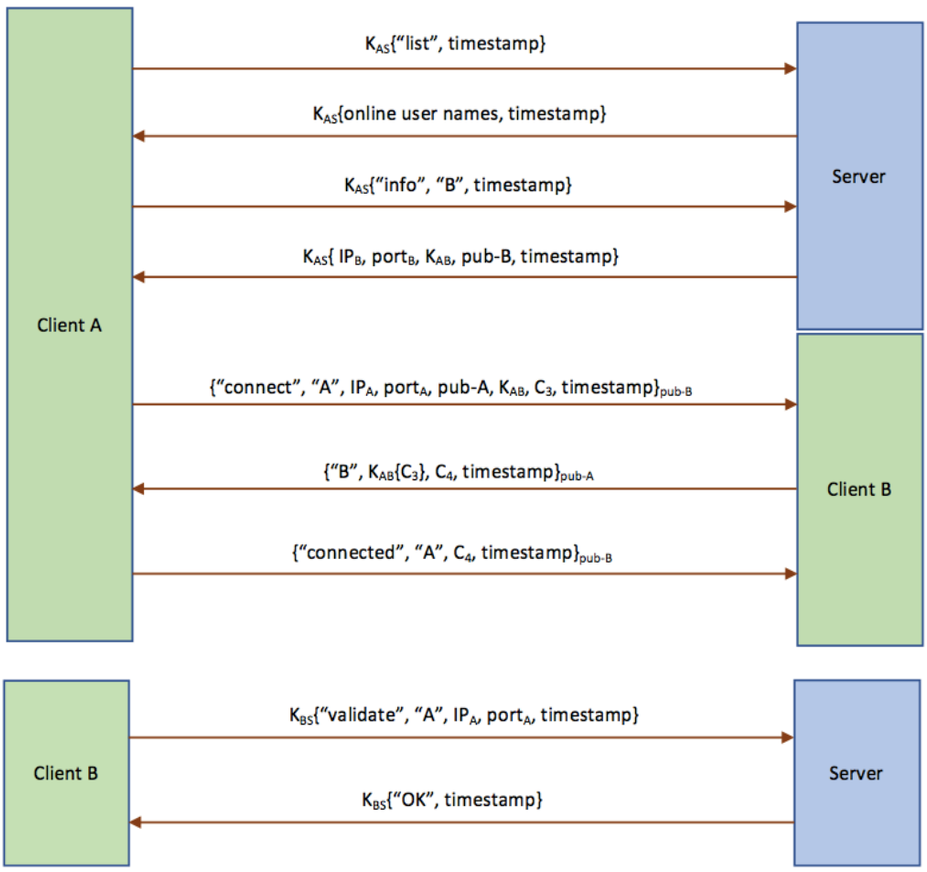
"A" is encrypted in the message so its identity is hidden.

C_1 is a nonce, which is used to avoid replay attack and to confirm that A is talking to the right server.

4) The server first checks if the value of N_1 is right. If it's right, it decrypts the data with its private key.
   Then it uses the $salt_A$ and pwd_A to compute the hash value, and compares with the value stored in the database to check if the pwd_A is right.
   If pwd_A is right, it generates a DH component s, and sends back (g^s mod p, C_1, C_2) encrypted with the public key of A.
   C_2 is a nonce, which is used to confirm client A gets the right session key.
5) Client A uses its private key to decrypt the message.
   Now client A and server shares the session key $K_{AS} = g^{ab}$ mod p.
   Then it sends C_2 encrypted with K_{AS} to the server.
6) The server decrypts the message with K_{AS}, and confirms C_2 is right.
   By now, client A successfully logins to the server.

Note: since we are using TCP socket, to avoid invalid connections, the server will close the client when any of these steps fails or there is no response for a long time.

b. Key establish protocol (and authentication with peers)

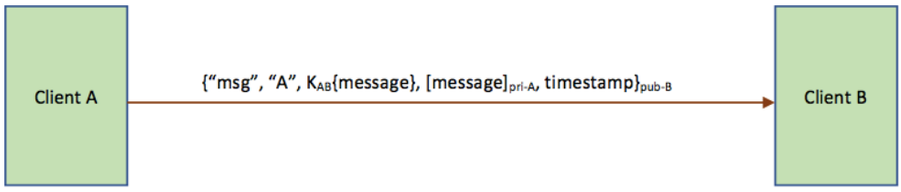


When client A wants to authenticate with client B, it will start the following process:

1) Client A sends a "list" message to the server, encrypted with K_{AS}
   Every message includes a timestamp, which is used to avoid replay attack.
2) Server responds with all online user names, encrypted with K_{AS}
3) Client A sends a "init" message specifying "B" to the server, encrypted with K_{AS}
4) Server generates K_{AB} for A and B, and send it back together with (IP_B, $port_B$, pub-B), encrypt with K_{AS}
5) Client A creates a "connect" message including ("A", pub-A, K_{AB}, C_3), encrypts it with pub-B, and sends it to B
   C_3 is a nonce, which is used to confirm A is talking to the right B.

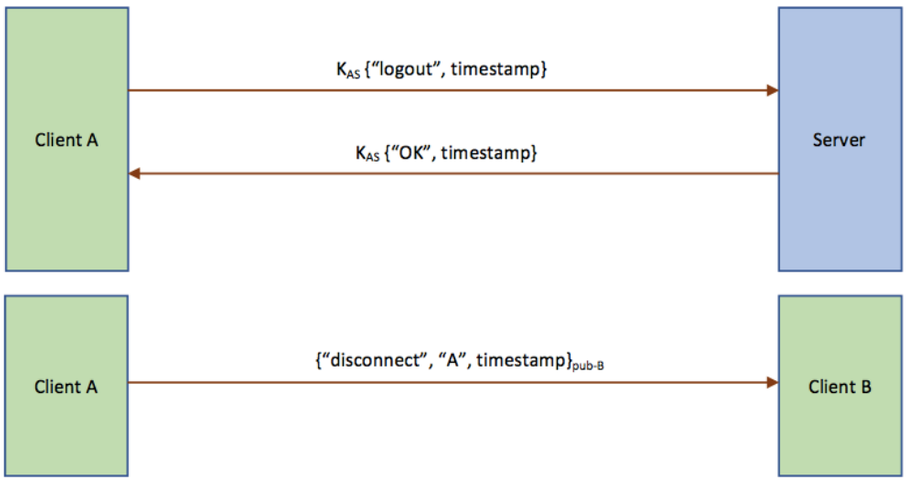
6) Client B decrypts the message with its private key, generates C_4 and encrypts with K_{AB}, and sends it back along with C_3.
We encrypt C_3 with K_{AB} to prove B has gotten the right session key.
7) Client A decrypts the message with its private key, confirms C_3 is right, and adds B into the authenticated client list. Then it sends back a "connected" message including its information and C_4 back to client B.
8) Client B decrypts the message with its private key, confirms C_4 is right. Note that now client B cannot make sure it's the real client A, since other clients like C can also claim itself to be A and initiate the above connection request. So, client B sends a "validate" message containing A's information to the server and confirms its identity. If it is real, then B adds A into the authenticated client list.
This is a two-way authentication, but sometimes maybe only one-way authentication can be built. In such cases, a client cannot send messages to or receive messages from non-authenticated clients (strangers), even if the other side has been authenticated.

c. Messaging protocol



The messaging protocol is relatively easy:
Client A specifies the type as "msg", add "A" and the message encrypted with K_{AB}, encrypts the whole message with pub-B, and sends it to client B.
"A" is encrypted in the message so it's anonymous.
Signature of the message is sent to ensure its integrity.

d. Logout protocol



When client A tries to logout from the server, it will start the following process:
1) Client A sends a "logout" message to the server, encrypted with K_{AS}
2) Then the server removes A from the online user list, deletes all A's information including K_{AS}, pub-A, etc., and sends backs "ok"
3) Client A notifies all connected client (in this case only client B) a "disconnect" message and then exits. Client B receives this message and remove A from the authenticated client list.

4. Questions:

1) *Does your system protect against the use of weak passwords? Discuss both online and offline dictionary attacks.*
Yes, our system can protect against the use of weak passwords.

A client can only validate its password after solving the server's challenge, and the password is encrypted with the server's public key, so it's not feasible for the attacker to do the online attack.
For a password, we choose a salt for it, and then stores both the salt and a hash of the combination of the salt and the password. In this way, even if the attacker steal the password file, it would be not possible to perform a single cryptographic hash operation and see whether a password is valid for any of the users. Therefore, the system can prevent from offline dictionary attack.

2) *Is your design resistant to denial of service attacks?*
Yes, the design is resistant to denial of service attacks.
First of all, any clients must solve the server's challenge before successfully build TCP connections with the server. And the server will actively close connections if any step fails or timeout, to avoid invalid connections initiated by malicious attackers.
Secondly, each message contains a timestamp, which helps to avoid denial of service attacks by replaying messages.

3) *To what level does your system provide end-points hiding, or perfect forward secrecy?*
Our system can provide end-points hiding for users, since user names are always encrypted in the messages. But the IP addresses of source and destination can be known by eavesdroppers.
It can provide perfect forward secrecy, since session keys are used in the communications of two end points, and will be forgotten as soon as they are disconnected.

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.*
If the user trusts the application running on his workstation, we can let the application to generate a Diffie-Hellman key which will then be used to encrypt the communications between two users. Since the key will not be known by the server, the communications are secured.
If the user does not trust the application, we can build an extra application upon it. The new applications authenticate with each other through a new trusted server. Then we use the new application to encrypt/decrypt messages before sending them.

Modification Since PS4

Since UDP is stateless, we need to send the sender's information to the receiver every time so that the receiver can know whom to reply.
So we changed the three client-client messages in the key establish protocol to include the sender's information, like user name, IP and port number.