# Operating Systems, Assignment 6

1. (40 pts) For this problem, we're going to write a multi-threaded TCP server in Java, with synchronization, challenge-handshake authentication using RSA and using a session key for efficient communication after authentication and key exchange. The program will do the same job as the scrabble server program from homework 5. This one, however, will be a lot more interesting because of the addition of authentication and encryption. Encryption will require us to implement our own client to communicate with the server. We won't be able to use a generic client like we did with `nc`.

## Supported Commands

After authenticating, the user will be prompted for commands. They can type in any of the commands we used in the previous assignment, query the score for a word, submit a new word, report all the submitted words or terminate the client.

We'll use the same limitations on string length that we used with the previous assignment. A username can be up to 10 characters and can consist of only letters. A word can be up to 24 letters.

If the user enters an invalid command, the server will respond with "Invalid command". If the user enters a bad username, the server will just close the TCP connection with the client, without sending any response.

## Partial Implementation

I'm providing a partial implementation of the client and server. At startup, the server reads the list of users with public RSA keys from a file named passwd.txt. Then, the server repeatedly accepts client connections, and supports limited interaction with each client. At startup, the client reads the user's username and sends it to the server. The server gets the client's username and responds with a challenge string (an array of randomly-chosen bytes). Authentication isn't supported yet, so, right now, the server just starts processing client commands immediately. It processes commands until the client enters the "quit" command, but it doesn't yet do anything for any of the commands; it just sends a copy of the client's command back as the response.

You'll need to add code to support the remaining three commands. You'll also need to complete the code to authenticate users, have the server send the client a session key (which is already generated for you), use AES for encrypting messages with this session key and implement multi-threading and synchronization in the server.

## Users and Passwords

The server supports a set of known users. At startup, it reads the list of users, each with a public key from the `passwd.txt` file. A client proves their authenticity by knowing the private key that goes with this public key. Each user's private key is stored in base64 in files matching the username. So, for example, anton's private key is in a file called `anton.txt`.

Most of the users have a private key that goes with the public key in the password.txt file. However, this isn't true for the user, giuseppe. His private key doesn't match his public key, so he won't be able to successfully authenticate. This is to make it easy for you to test authentication. Everyone should be able to authenticate with the server, except giuseppe.

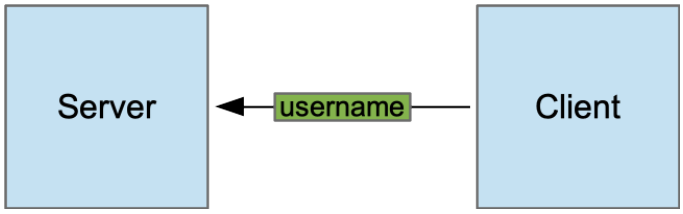
At startup, the client prompts the user for a username. If the client types in an invalid username (e.g., one that's too long or contains something other than letters), the client prints "Invalid username" and

terminates. If the username is valid, the client sends it to the server and waits for a challenge string. If the client types in a username that doesn't have a file with a private key, or if the client fails to authenticate with the server, the client just throws an exception and terminates.

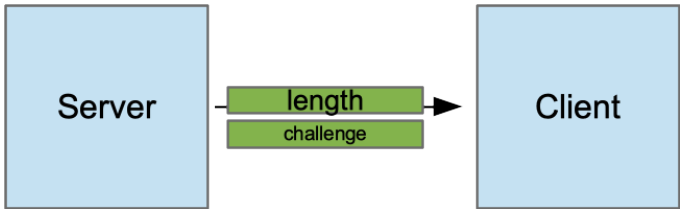
Admittedly, it's not very realistic to just store all the users' private keys in files in the same directory. This arrangement simplifies development, but, if we wanted to use this as a real application, we'd want to make sure the private key for each user was only visible to that user. You saw this when you were creating your arc accounts on assignment 4. Whey you made a key pair with ssh-keygen, it was stored in the '.ssh' directory under your home directory. The permissions on this directory should be set up so that only you can access these files.

## Client/Server Communication

The client and server communicate using DataInputStream and DataOutputStream objects wrapped around the streams for the socket connection. These classes make it easy to encode strings, integers, arrays of bytes and a lot of other things. Initially, the client will prompt the user for a username and send it to the server as an unencrypted, UTF-encoded string. This part is already done for you.

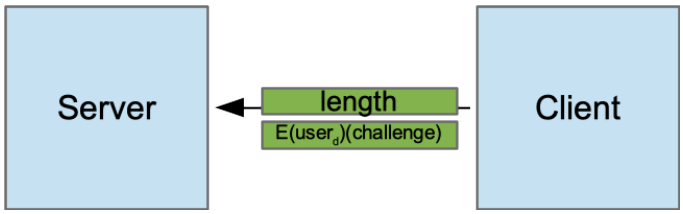



In response, the server will generate a random 16-byte challenge string (really, just an array of bytes) and send it to the client. For this and all subsequent client/server communication, we'll be exchanging arrays of bytes rather than strings. This makes sense, since encryption produces an array of bytes. Since TCP doesn't automatically respect message boundaries, we'll need a way to tell where messages start and end. In the starter, the Server includes two static methods for this. The getMessage() and putMessage() methods send a message by first sending an unencrypted, 4-byte length. This is followed by the sequence of bytes in the message. The server sends the challenge to the client in this format.



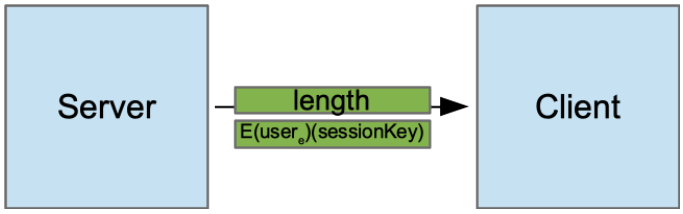

The first two messages are unencrypted (although the second one is mostly a random string of bytes). All subsequent messages between the server and a client will be encrypted, initially with RSA and then with AES, once a session key has been shared between the Server and Client.

After receiving the challenge, the client encrypts it with RSA using its user's private key. It sends the result back to the server, thus demonstrating that the client knows the private key without actually sending a copy of the key. Encrypting the challenge correctly is enough to prove that the client has the key.

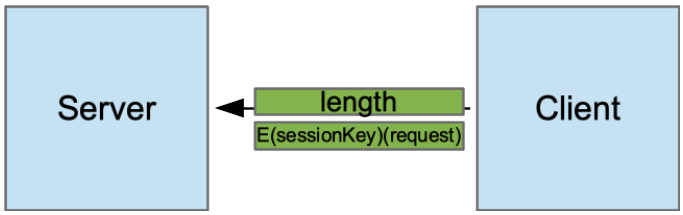



After receiving the client's response, the server decrypts it with the user's public key and verifies that it matches the original challenge string. Be careful how you check to make sure these arrays match. I've seen some students make mistakes in this part of their implementation.

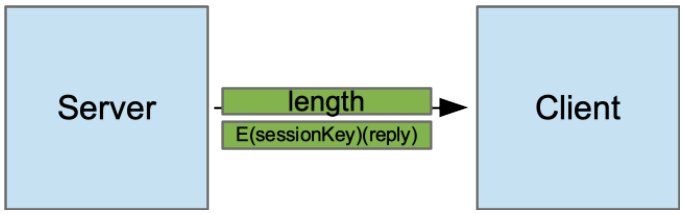
Then, the server sends the client a randomly-generated session key, encrypting it with the user's public key, so that only that user can decrypt and recover the session key.




Up to this point, encryption has used RSA (to encrypt/decrypt the challenge and to encrypt/decrypt the session key). Henceforth, encryption will be done using AES, with the server-generated session key. This will make encryption less computationally expensive, and it will make it possible to encrypt messages of arbitrary size. Whenever the user enters a command, the client will encode it as a string, turn that string into a byte array, encrypt the byte array using an AES cipher object and send that to the server.




The server will decrypt the request, perform the requested action and send back a reply. The reply will also be encrypted via AES with the session key. The client will be waiting for the server reply, to see if the command was successful, so your server will need to send back a response of some kind for every request, even if the client doesn't need to print anything before the next prompt.




## RSA Encryption

The starter already includes code to load the user's private key on the client and all the public keys on the server as the passwd.txt file is read. There's also code to create cipher objects to encrypt and decrypt with these keys. You'll need to add code that encrypts/decrypts the challenge string and the session key using these cipher objects.

## AES Encryption

You'll need to add code to create Cipher objects that encrypt and decrypt with the server-generated session key using AES. For our in-class example, we created an AES cipher object using "AES/ECB/NoPadding" for the transformation string. This was a nice example, since it helped to illustrate how AES is a block cipher, but it's a little more difficult to work with, since encrypted messages had to be a multiple of 16 bytes.

When you make the Cipher objects for AES encryption and decryption, use the string "AES/ECB/PKCS5Padding". This will automatically pad messages before encryption, so you won't have to worry about whether or not they are the right size.[1]

## Multi-Threading and Synchronization

In the partial implementation, the server is single-threaded. It can only interact with one client at a time. You will fix this by creating a new thread to interact with each client that connects. When the client disconnects, its thread will terminate.

If there are multiple threads running in the server, there's a possibility for race conditions in accessing shared state. This can be a problem for the `submit` and `report` commands. If two users try to modify or access the list of submitted words at the same time, the data structure in the server could be corrupted. Add synchronization code to prevent race conditions in these commands. Make sure one client can't access the list of submitted words while another client is also accessing it. This will be easy in Java; you can use Java's built-in synchronization support (i.e., synchronized methods, synchronized blocks and wait()/notify() if needed).

Your server should never try to send or receive a message while it's holding a lock. This is just a countermeasure against malicious clients. Let's say you try to send a report of the word list to a client while you're inside a synchronized method. An evil client might refuse to read from the server. In the server, that client's thread would eventually block, waiting to send a report. If it blocked like this in a synchronized method, no other threads in the server could acquire the same lock, so the server could be incapacitated by one badly behaved client.

To send a message to the client, you'll need to prepare the message in memory (possibly using some synchronized methods or blocks), then actually send the message when the client's thread isn't holding any locks.

## Port Selection

Your server will listen for connections on the same port you used on assignment 5. This will prevent us from fighting over port numbers if more than one student is using the same system to develop.

## Authentication Failure

You can test your authentication implementation by using the giuseppe user. For all of the other users, the public key in the server's passwd.txt file matches the user's private key, but not for giuseppe If you try to log in with this account, authentication should fail. This is demonstrated at the end of the sample execution below.

[1] Really, it would be better to encrypt with "AES/CBC/PKCS5Padding" (but don't do it). The CBC in the middle stands for Cipher Block Chaining, like we talked about in class. Without this, every block gets encrypted independently, exactly the same way by AES. So, every time the client sends the "report" command, it gets encrypted to exactly the same random-looking array of bytes. Of course, we could fix this, but I'd rather leave this security weakness in our program and give you a chance to think about it. An eavesdropper could tell something about the client's messages even if she couldn't actually decrypt them.

If the client fails to authenticate properly, the server should close the socket connection to this client and let the client's thread terminate without sending the client any response. From the sample output below, you can see that failure to authenticate will probably cause an exception in the server. The partial implementation already contains code to catch this exception and close the client's socket.

## Sample Execution

Below is a sample interaction between the server and two clients, each running in a different terminal window logged in on the same host. The server is run on the left, and the two clients are illustrated in the columns to the right. I've included some comments to explain what's going on. Don't actually type these in when you try out this example.

```
# Start up the server.
$ java Server
```

```
                # Run one client and query a word
                $ java Client localhost
                username> anton
                cmd> query counterrevolutionary
                28

                # Submit a word
                cmd> submit skillfully
```

```
                                # Run a concurrent client and submit a word
                                $ java Client localhost
                                username> bennie
                                cmd> submit monopolize

                                # The report contains words from both
                                cmd> report
                                     anton               skillfully  20
                                    bennie               monopolize  23
```

```
                # Disconnect as anton and connect
                # as a different user
                cmd> quit
                $ java Client localhost
                username> carly

                # Submit a word for this user and
                # view the (sorted) report
                cmd> submit The
                cmd> report
                     carly                      The   6
                     anton              skillfully  20
                    bennie              monopolize  23

                # Chagne my word and get a new report
                cmd> submit Theoretically
                cmd> report
                     anton              skillfully  20
                     carly           Theoretically  21
                    bennie              monopolize  23
```

```
                                        # Try some invalid commands.
                                        cmd> query InvalidWordBecauseItIsTooLong
                                        Invalid command
                                        cmd> fskdsljdfk fjsdkslfsj
                                        Invalid command
                                        cmd> submit invalid-chars-123
                                        Invalid command

                                        # Quit both of the clients
                                        cmd> quit

                cmd> quit

# Try authenticating with a bad
# key pair.
$ java Client localhost
username> giuseppe
IO Error: java.io.EOFException

# Exception thrown in he server when authenticationf fails.
Encryption error: javax.crypto.BadPaddingException: Decryption error

# Kill the server.
ctrl-C
```

## Submitting your Work

When you're done, submit an electronic copy of your `Client.java` and `Server.java` using the assignment link for Homework_6 on Moodle.