Report Of Lab3&4

刘书敏 2015303087

## USAGE:

To start the project, you need to have jdk1.8 and maven installed. Then type ***mvn package***in command line to build the project.

**mvn exec:java@server**

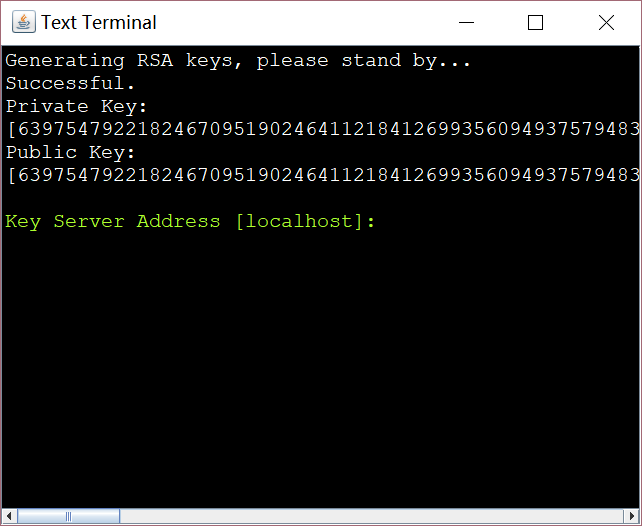
start the key server

**mvn exec:java@client**

start the client

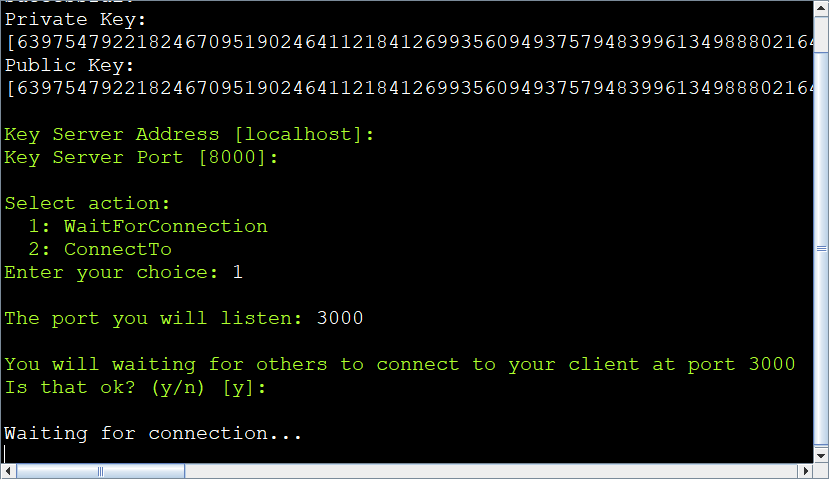
If you use windows, the client command may not work. Please click the **client.cmd** in project folder instead to run client.

Once you start client, you will see this:

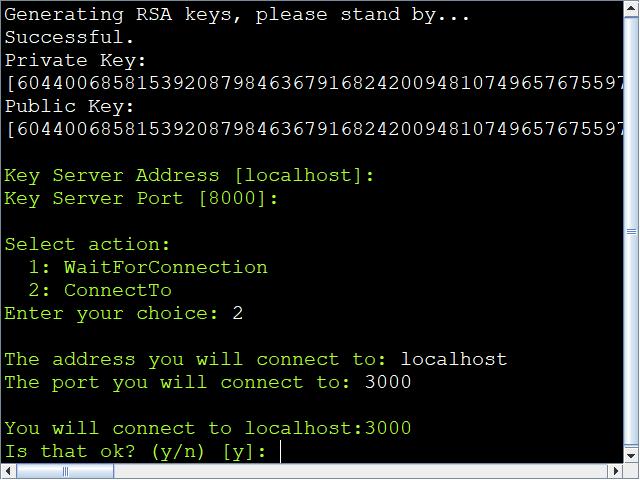


Just enter what it asks for.

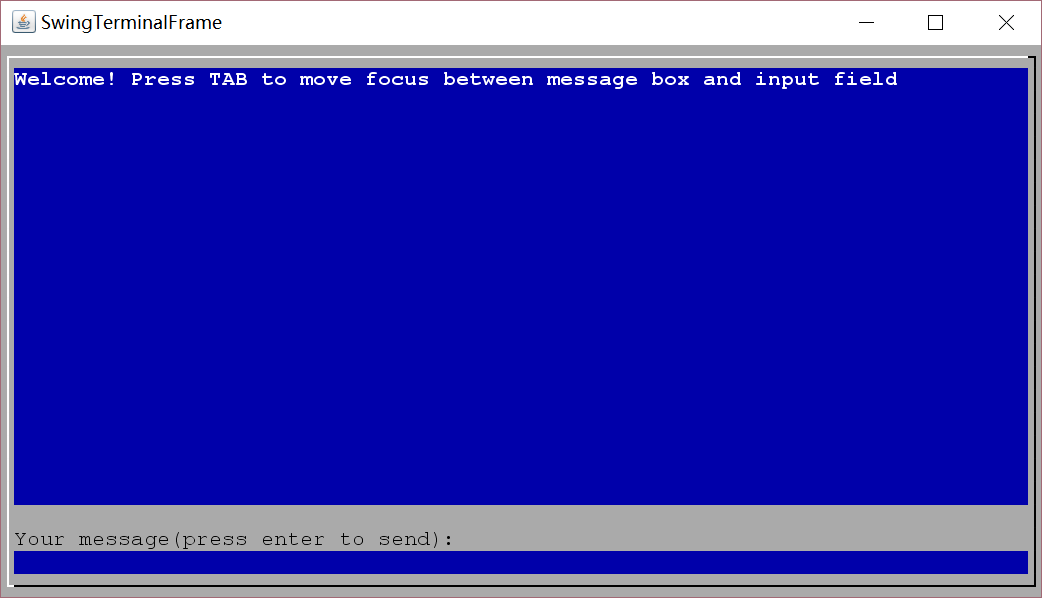
We select waiting for other to connect:



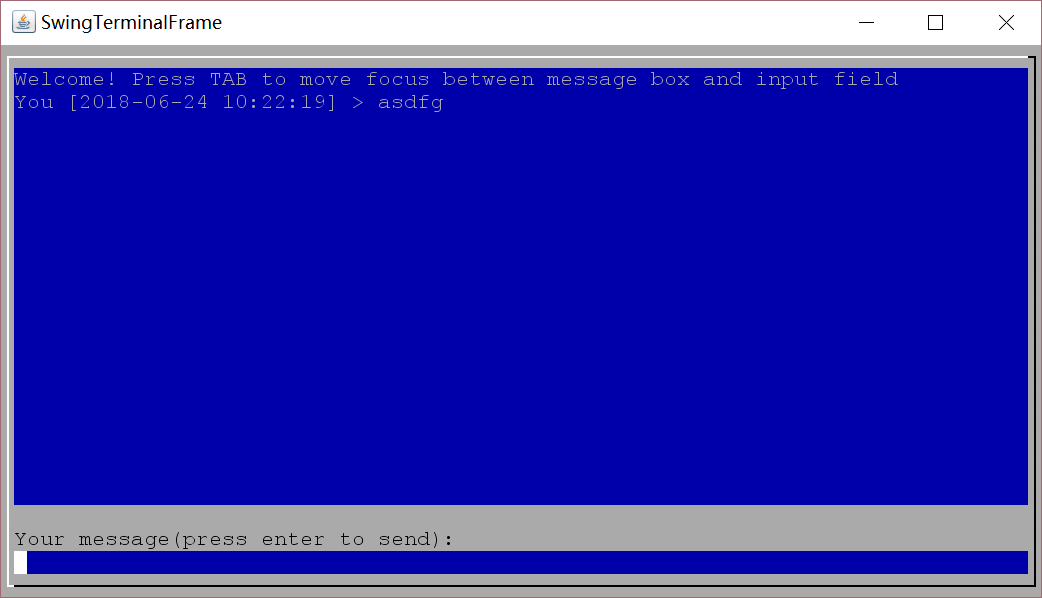
And we can open another client to connect to it.



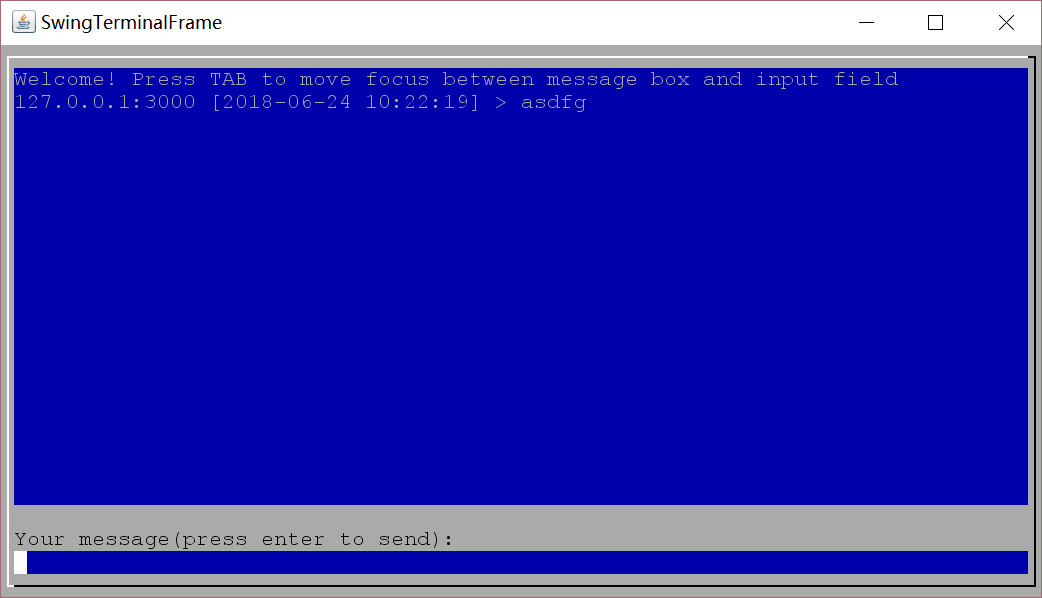
Then we get the chat room:



Send message:



In other client:

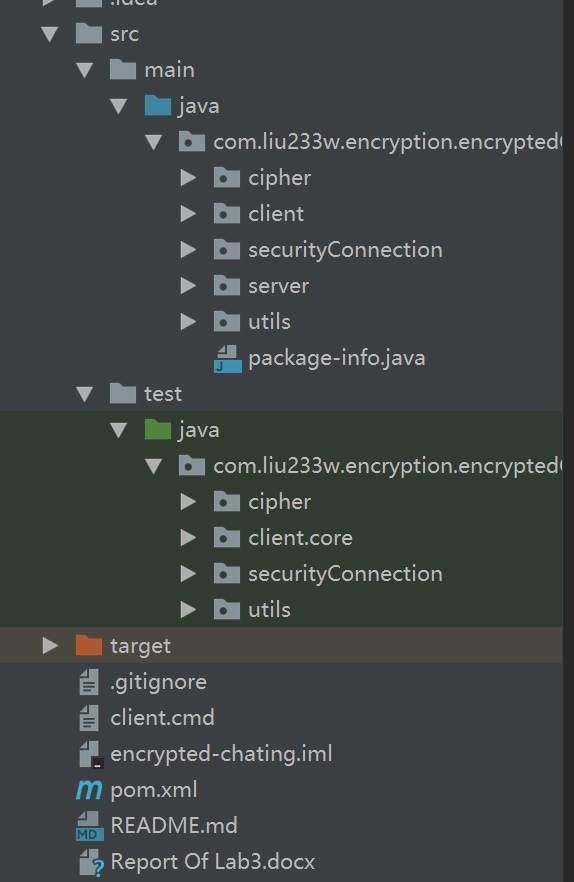


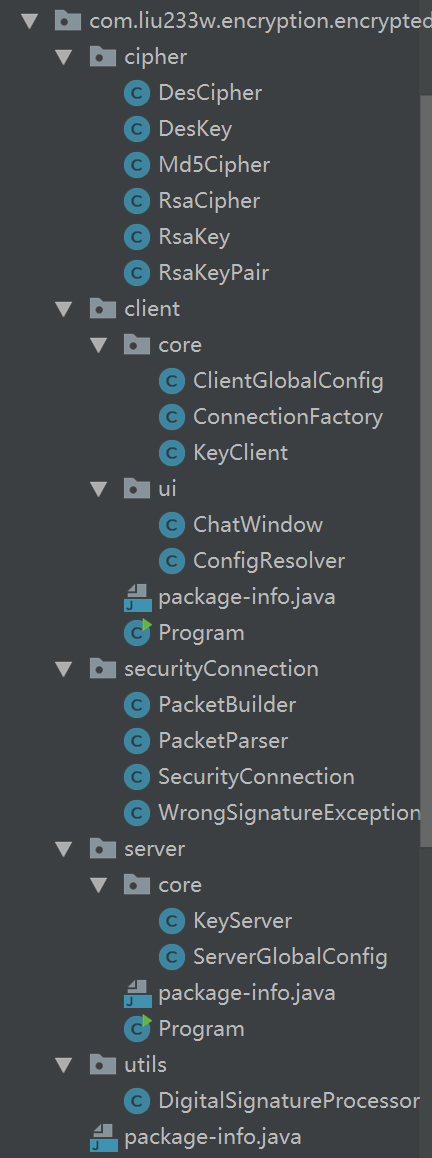
If the digital signature checks failed, it will display the message with an *(Unauthorized)* tag.

I’ve also packaged a binary file, you can start the server/client by clicking the cmd file in the binary folder. It only need jre installed.

## Architecture

The project is splitted into several layers.



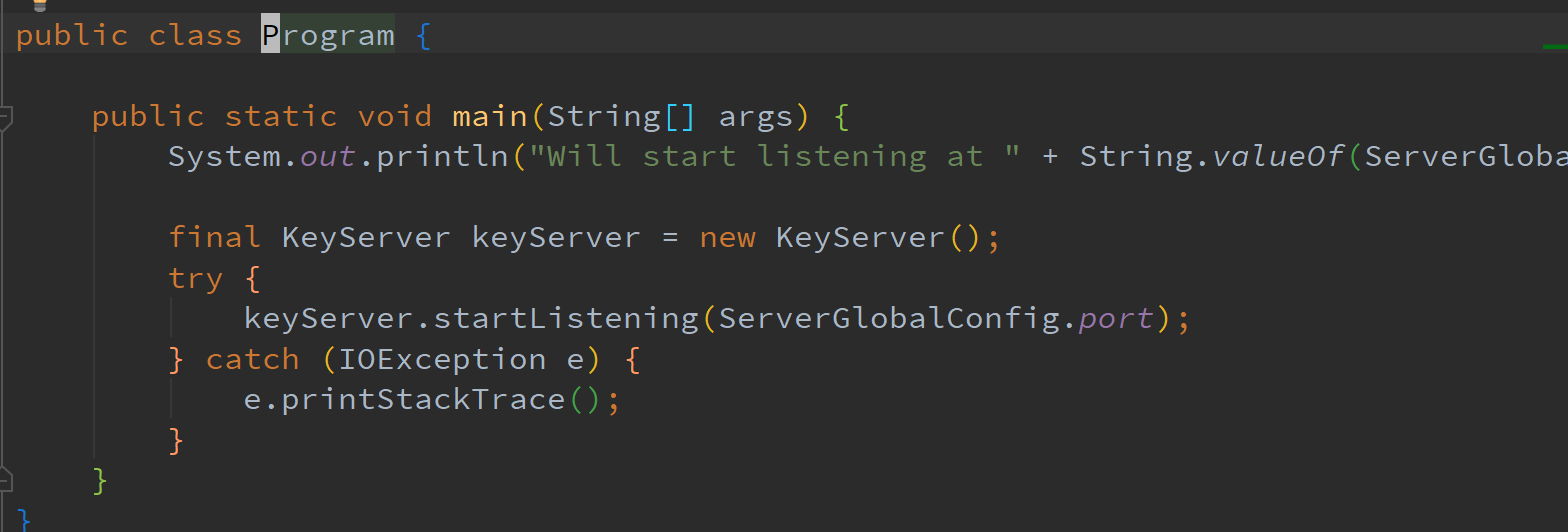


Ciphers are the core of the project. Other classes are all built on the top of the ciphers. For example, the **DigitalSignatureProcessor** class uses **Md5Cipher** and **RsaCipher** to generate digital signature. The **client.core.KeyClient** and **server.core.KeyServer** are designed to distribute rsa keys. They are the layer which built on top of the cipher layer.

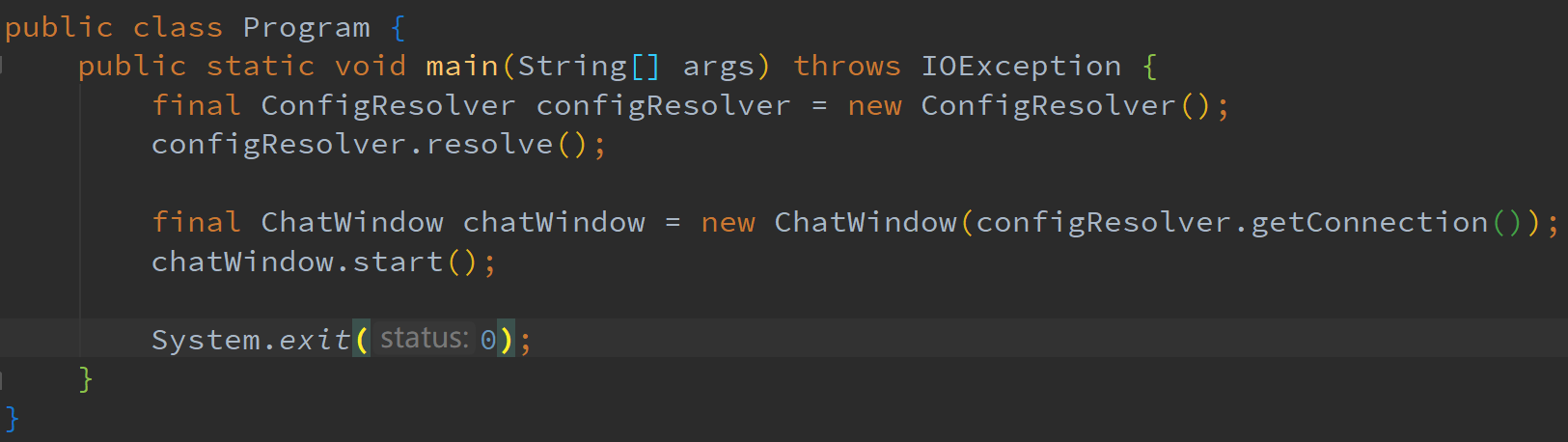
Then, the **SecurityConnection** (and its helper classes **PacketBuilder, PacketParser**) is built on the top of **DesCipher** and **DigitalSignatureProcessor**. I split the transfer process and the build process of the security connection. The **SecurityConnection** itself is only designed to handle the “transfer” process. It will just transfer the data with the help of **DesCipher** and **DigitalSignatureProcessor**.

The class **client.core.ConnectionFactory** handles the connection building process. It uses ciphers, sockets and **KeyClient** to build **SecurityConnection**.

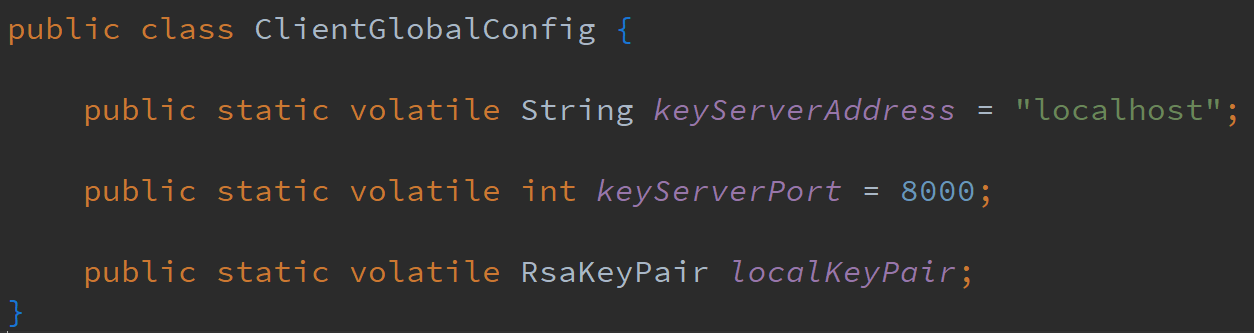
The ui layer is on the most top in the project. For key server, the ui is simple. It only displays a line of message. So, I just developed a Program class with a main method. Now the port which key server listen to is hard coded. But it’s easy to add an option to ask user for the port.



As for the ui of client, I developed two class to handle it.



The **ConfigResolver** will ask user what they want to do. Then it will fill the field in **ClientGlobalConfig**. (Note that they are not constant, so they can be changed in **ConfigResolver.**)

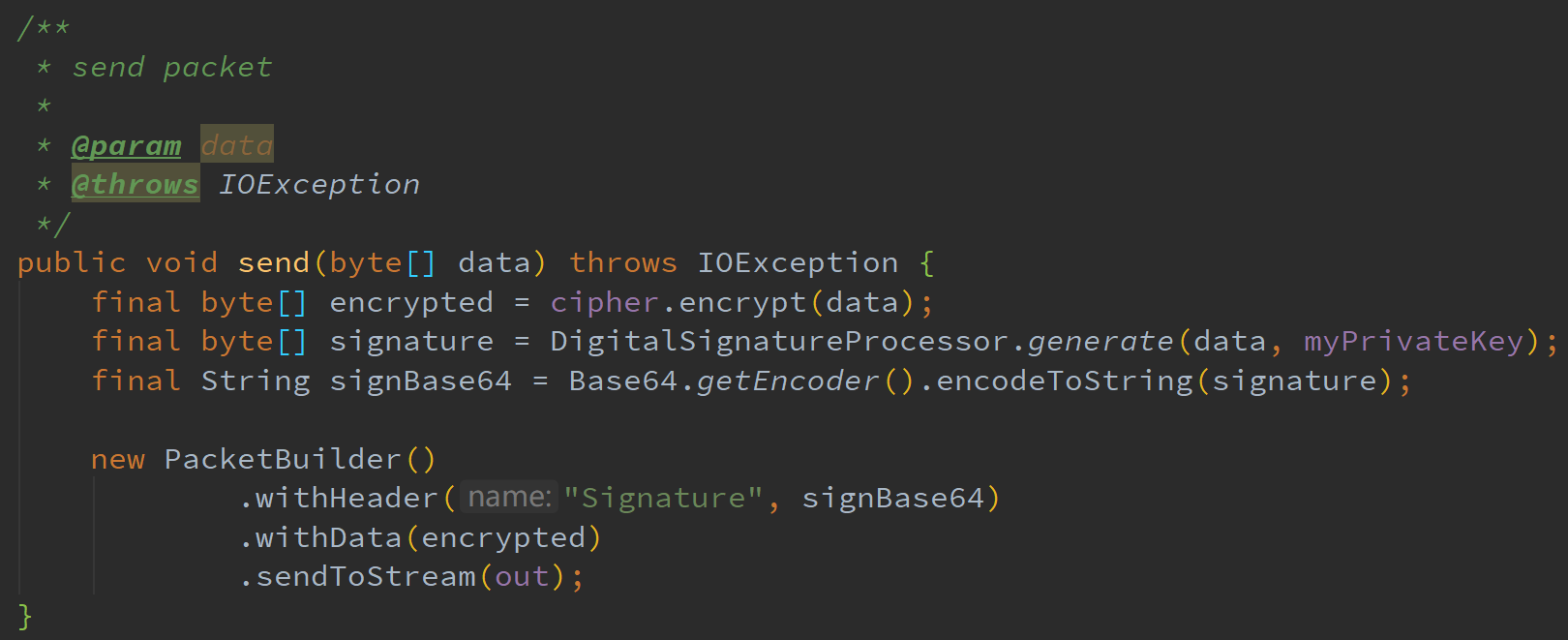


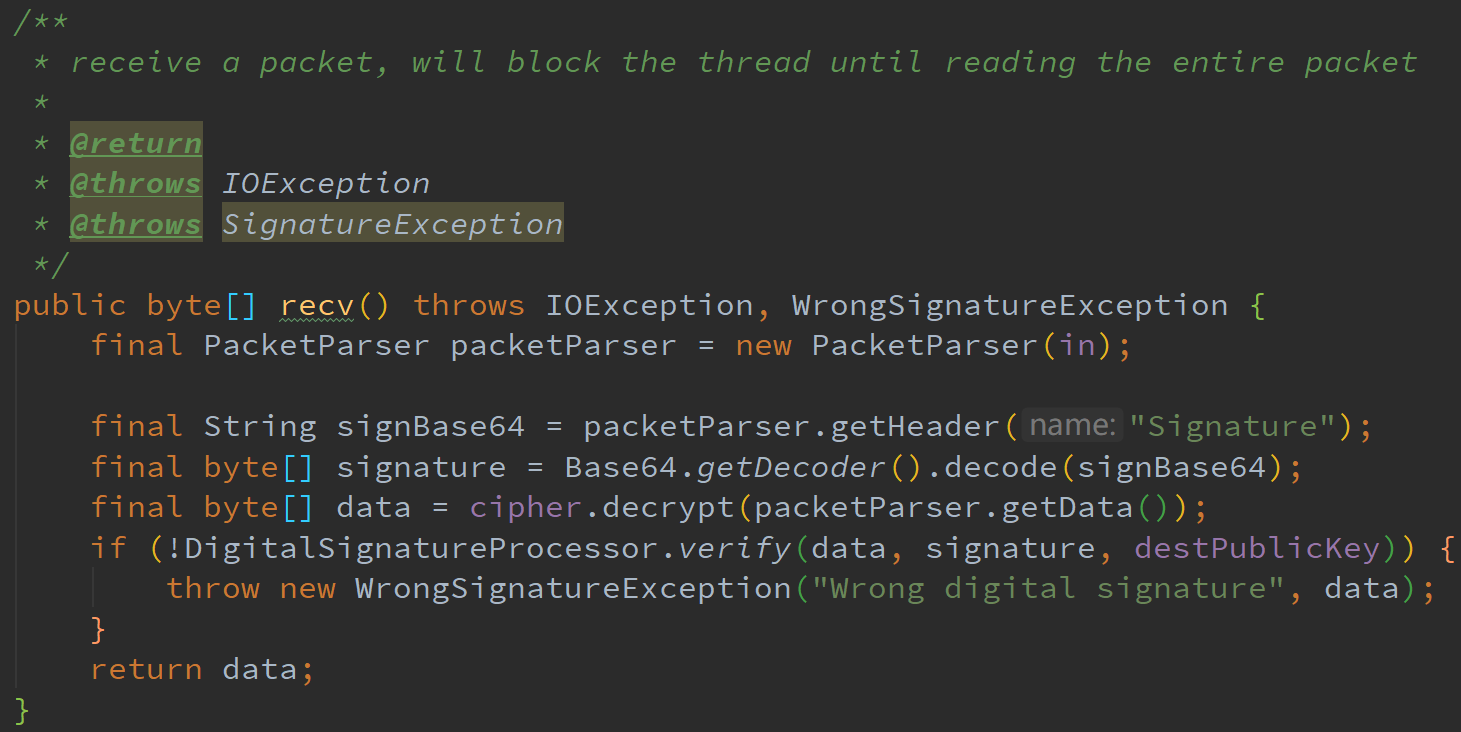
It will also generate the **SecurityConnection** by the selection of userusing **ConnectionFactory**. So, the main method can just get the connection from it and sends it to **ChatWindow**.

## The design of SecurityConection

In different project, the building of **SecurityConnection** can be different. But the data transferring by **SecurityConnection** can be the same. Because most of the security data transferring needs encryption and digital signature. So that I can use this class in other projects, such as final exam.

Every data I send by **SecurityConnection** are encrypted and have digital signature. **SecurityConnection** will group the encrypted data and digital signature in to a “Packet”, just like a http diagram, but with a simpler format. The class **PacketBuilder** and **PacketParser** are developed to handle the packet.

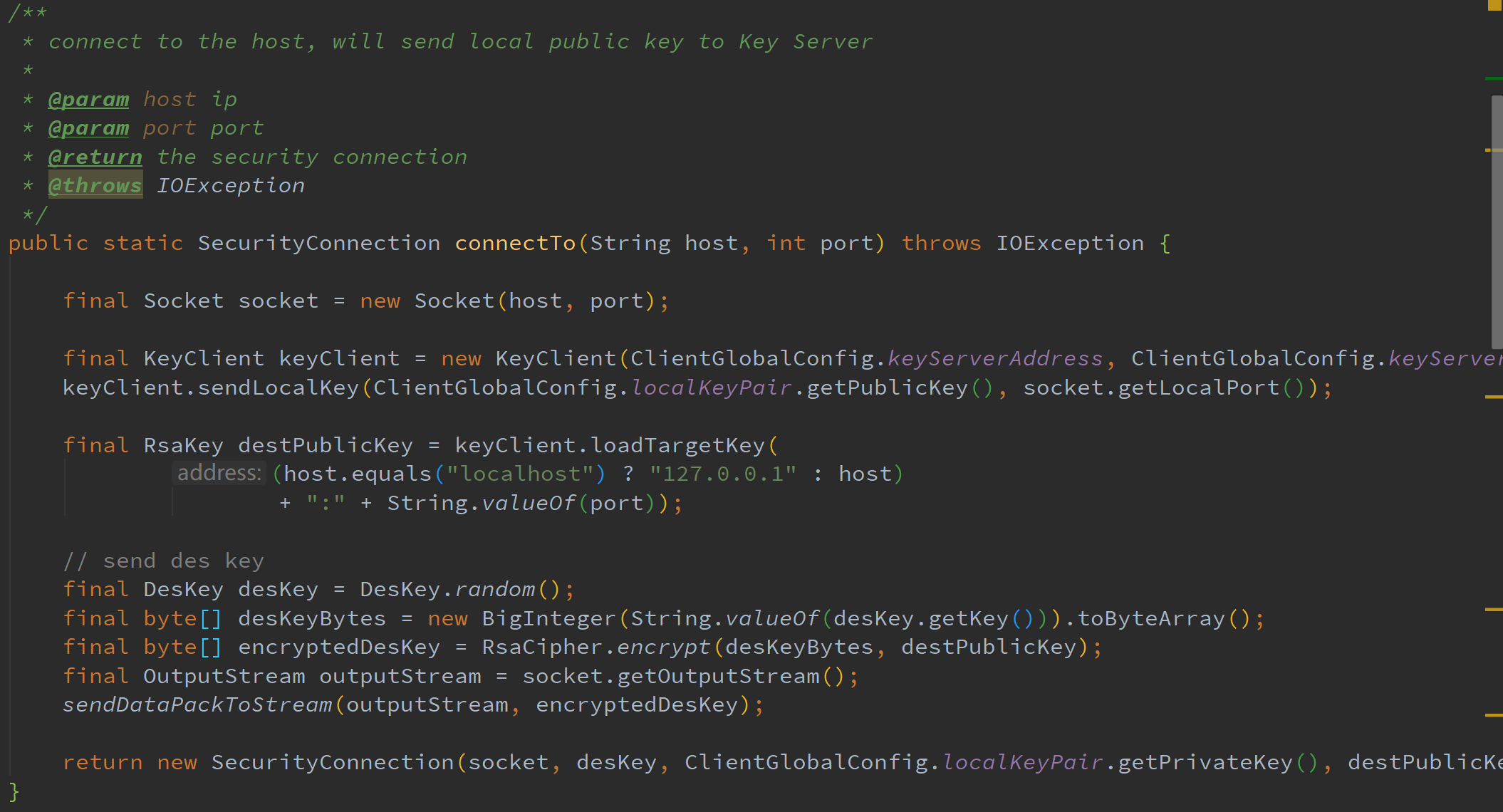




The two class will only transform the packet between object format and stream(bytes) format. The construction and consume of packet object is handled by **SecurityConnection**.

## The design of ConnectionFactory

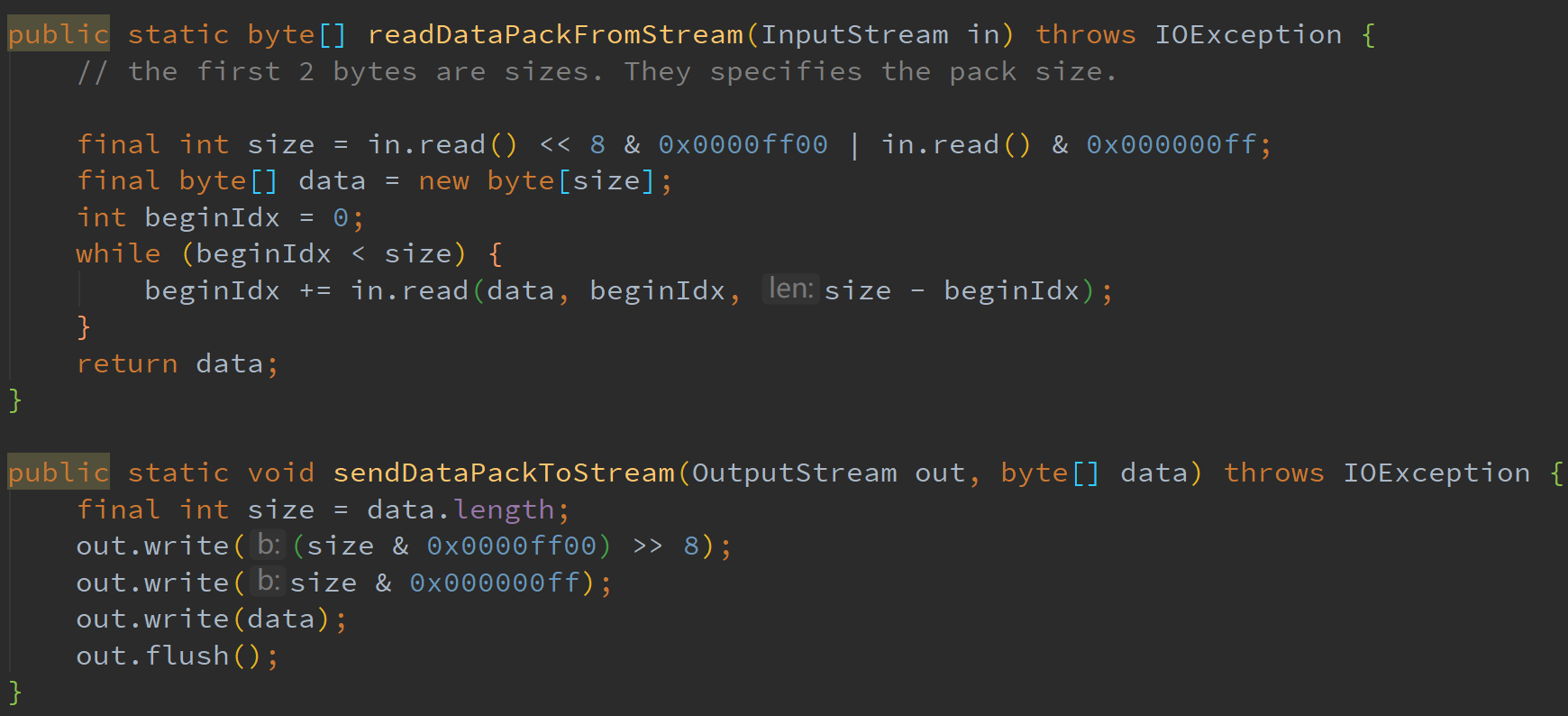
With the help of layers below, the connection building process can be easy to read.





When request a connection, it will send local rsa key to key server first and then send encrypted des key to target client. But when waiting for a connection, it will read des key first then rsa key. So, when target client finishes the reading of des key, the rsa key of requesting client have already sent to key server. If client read rsa key first then des key, the key server may not save the key yet, and the client would only get “None”.

The problem is that the building process need to send encrypted **DesKey**(session key) to the target client. I can’t use **SecurityConnection** here and must define an other format to represent the encrypted des key. Here is the *readDataPackFromStream* and *sendDataPackToStream* .



I use the first two bytes of the stream to represent the size of encrypted des key. Therefore, the size of encrypted des key can be at most 256\*256=2^16 bytes.