PortKnocker

by Jan Witkowski s18749

This project consists of two programs:

- Server
- Server's client

Server is responsible for authentificating clients based on listening for UDP packets (treated as port knocks) on ports given as parameters.

When a packet arrives on one of the ports, server check its origin against the list of earlier noticed knocks. Each knock from the same origin must be noticed in the proper order in order to authenticate the origin.

If any of the knocks is incorrect, server clears the knocks table and ignores rest of the knocks.

After succesfull authentification server spawns a new thread for the authenticated origin. This thread notify the client about the sucessfull authentification by just starting the transmission of the file.

Protocol decription

File transmission is implemented on the basic UDP sockets. File is first loaded into the chunks table implemented as below:

```
[
    chunkID: [
        packetID: [
            UDPPacket: [byte, byte ...]
            ...
        ]
        ...
    ]
    ...
]
```

The chunk and Packet sizes are controlled by configuration.

When the file is loaded, the transmission begins. First a Metadata packet is sent. The Metadata Packet is of this structure:

```
filename::expected_bytes::chunk_ize::packet_size::expected_chunks
```

The Metadata packet is sent as as a separated chunk.

Then, the data packets are sent in chunks, and in between of sending chunks, the server is waiting for the acknowledgment for each chunk.

If the chunk will not arrive in 5 seconds, the chunk is re-sent. If the ack contains a single letter 'n', the chunk is re-sent.

After the transmission server thread ends. If the client will receiveall packets and chunks correctly, it will reconstruct and save the file in the received/ folder If the client cannot receive some pactets, it will send an ack containing single letter 'n' and it will wait with the 10s timeout for the re-transmission. If the missing packets do not arrive before the timeout, the transmission is aborted.

After succesfull transmission Client is checking the chunk map against missing chunks and packets.

Troubleshooting

Sending rate

The first problem is the correct choose of the protocol constants like:

- packets per chunk
- timeouts on client and server threads

The implemented protocol is able to send and receive about 100 packets per chunk running on localhost, whithout any packets getting lost. But with the greater speed and no dynamic speed controll, the protocol becames useless since no re-transmission is possible to be delivered.

The next problem is whether to controll the speed of the protocol dynamically or not. When a lot of packets are lost, the protocol may slow down too much, and if no packets are lost, it may speed up causing the oposite. The speed controll is a great topic but it is not implemented due to the great complexity.

Also the choose of the "correct" number of re-transmissions is a bottleneck, because every packet may be lost, so endless re-transmissions are out of consideration. This problem also with the problem of the dynamic speed controll would make the implementation really complicated so the chunks are resent only once after a timeout or on a client call (nack).

What works, what doesn't

When running the client and the server on the same machine, I was able to sucessfully transmit files of the size of 1GB in 30s without any packet getting lost.

However when you increase the chunk size, the protocol is useless because it does not contoll the speed, so it wont adjust.

So probably, if a server was running on the different machine somewhere over the seas, It would not be able to send any files at all.

Also the size of the file is limited by the implementation, because it is loaded to the one chunk map. So this is a huge bottlenect that could by simply fixed by spawning a number of senders and assigning them to the some file parts.

How to compile / run

Linux

Make the compile_linux and run_linux scripts in the main directory executable by:

```
chmod +x compile_linux run_linux
```

run compile script:

```
cd main/dir
./compile_linux -t client
# or
./compile_linux -t server
```

option -t decides if compile the server program or the client program

If you choose server, the script will first make 4 files in the main dir, 2Mfile, 16Mfile, 24Mfile and 64Mfile. you can use them to test the protocol. They will not contain any usefull information, but they have a size specified in the file name.

The script uses the same build directory for both server and client so do not compile server and client at the same time.

Then you can run the programs with the run_linux command:

```
./run_linux -t client localhost port port ...
# or
./run_linux -t server port port ...
```

you can also prepend the comand with **time** command to count the time of transmission:

```
time ./run_linux -t client ...
```

Windows

I did not have a time to prepare the scripts for windows, but the process is very simple.

```
cd main\dir\Server\s18749
javac -d ..\..\bin *.java
cd ..
```

```
copy manifest.mf ..\bin
cd ..\bin
jar cmf manifest.mf server.jar *
copy server.jar ..
```

and to run:

```
java -jar server.jar params...
```

the same applies to the client program.