Assignment 1: Plz Tell Me My Password

Part I Due: 11:59 pm, September 23, 2013
Part II Due : 11:59 pm, September 27, 2013

1 Introduction

The objective of this assignment is to gain experience in:

- socket programming
- writing a program with a simple logic
- writing a makefile
- version control with svn (We will learn how to use svn in class.)
- using a simple program checker
- non-blocking I/O and select (Part II)

The programming assignments in this course are designed to be progressive, meaning that we build one on top of previous assignments. Some assignments will reuse the code in the previous assignment, but not all of them will be based on the same code base. The assignments are designed to make you learn simple things so that you can use the basic components to build a larger system. The good side of this is that things start relatively easy. This is designed for students who do not have enough programming experience. The goal is to start from the basic material and build up to write a more sophisticated program.

The assignment is broken down to two parts:

- Part I: Simple socket programming (We will learn socket programming in class).
- Part II: Non-blocking I/O and time-out (We will learn about non-blocking I/O in class).

2 Lab Requirements

In this lab, you will implement a simple client that communicates with a server to guess your own svn password. As we have shown in the class, we have set up individual svn accounts for you to submit your assignment. However, we changed the password of the account, and your job is to guess the password in order to submit your assignment. Luckily, we made a server that gives you a hint. In particular, when your client asks whether the username and the password match, the server can tell you whether the password provided is less than, equal to, or greater than your real password. The password is an integer.

You should write a program that returns you the password by communicating with the server. Your client and the server should communicate with each other using the sockets API to send

UDP packets. They use a simple request and response protocol; client sends a request and server responds to the request. The protocol they must implement is detailed in the following sections. Using the protocol, your clients and our server must be able to interoperate with each other. Your job is to 1) correctly implement the protocol, 2) guess the password using the client, and 3) submit your code to the svn repository using the password you have guessed.

In the first part of the assignment, we assume that no packet gets lost between the server and the client. Therefore, you can use a blocking socket I/O. This part is designed for students who have not done any socket programming recently. However, the in the second part of the assignment, you must make the client robust to communication failures. In particular, when the request message to the server or the response from the server is lost the client must timeout and retransmit the same request up to 3 times.

The majority of the grade for this lab will be determined by the outcome. In order to submit the source code you must implement a client that talks to the server. Two versions of the server will be running continuously to help you write the code. The first version will operate normally without dropping any packet. In the second version, we will intentionally drop some requests from the client to emulate a communication failure (e.g., a packet drop). You should use the second one to test the code you submit for the second part of the assignment.

2.1 Implementation

You must write your client in C/C++. Because of the similarity between C and C++, you should be able to write your code in C (e.g., without using any custom class), but use the g++ compiler just for compiling. You are required to use g++ as the compiler. If you want to use standard data structures, we highly recommend that you take advantage of the C++ Standard Template Library, which provides data structures such as maps, lists, and queues, so that you don't have to reimplement the wheel. If you have no experience with C++, you may find the resources in Section 5 useful. Because of the similarity between C and C++, you should be able to write most of the code in C and just throw in the data structures from C++ that you wish to use.

For consistency, we require that projects follow the major aspects of the Google C++ Coding Standards:

`Http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`

We recommend *against* using C++ features such as exceptions, operator overloading, deep levels of inheritance, and multiple inheritance. With systems code, we urge: Keep it simple!

Lint and static checkers: You must run the two static code checkers on your *.cpp and *.h files.

- cpplint: http://google-styleguide.googlecode.com/svn/trunk/cpplint/cpplint.py cpplint enforces Google's programming style. You must try to minimize the warning.
- cppcheck: http://cppcheck.sourceforge.net cppcheck is static checker that can detect some bugs by looking at your source code.

```
Usage: cppcheck --enable=all *.cpp *.h
```

We will run cpplint and cppcheck on your source code and deduct points if there are too many warnings.

Makefile: You must provide your own makefile. Use g++ as your compiler. If your code does not compile with make, we will deduct points. See Section 6 for more details.

2.2 Client

Requirements The program must run like this.

```
./guess_password <server hostname> <port>
```

The output of the program will be a file named "password.txt". It must contain your username password printed as a string. For example, if your username is 20110001 and password is 1111, it should look like:

```
20110001 1111
```

Testing: The server is running on ina.kaist.ac.kr:3234. You can use it to test your code.

```
./guess_password ina.kaist.ac.kr 3234
```

Part I: You can assume that there is no failure in the communication. No packets are being dropped. However, you must check whether your socket function calls are successful. On socket() and bind() failure, you must terminate the program and print out the cause of the error using perror() and errno.

Use port number 3234.

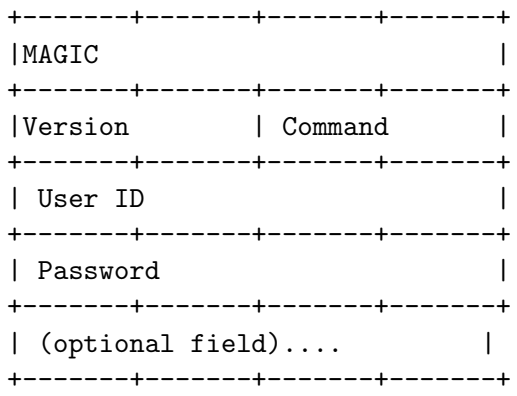
Part II: Your client must be robust against failures and exceptions.

- Handling dropped response/request packets. You should wait for 3 seconds and resend the request up to 3 times. After the 3rd failure, you terminate the client.
- Handling delayed response packets after the timeout. Packets may get sent after the time out. You should discard these packets and resend the request.
- Handling bogus packet. Your code must be robust to receiving bogus packets from the server (e.g., wrong magic number, wrong version number, or wrong ID/password). The client must retry after receiving bogus packets.

Use port number 3240.

2.3 Protocol

For this assignment, you will implement a binary format protocol. The protocol uses UDP. The various parts of the message will be aligned on specified bytes, such that there is less variation between messages:



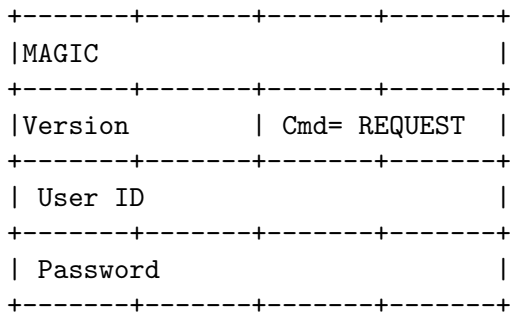
MAGIC refers to a magic number, so that we are able to identify whether a message is relevant to our application. Our MAGIC number is 0x323324. Version is 1.

The command is the command that is being sent. Remember that any numbers that you send that are longer than 1 byte MUST be in network byte order. A list of these commands can be found in msg.h. You must use that enumeration, or your code will not pass our tests.

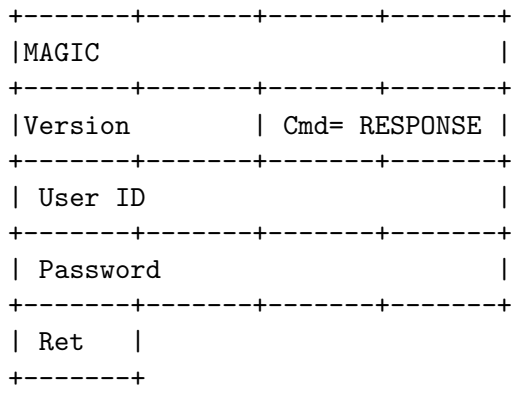
The following list of commands does explain what to do in every possible situation. In cases not covered by the following list or elsewhere in this document, do what you think would work best and would make the most sense.

- REQUEST: The client should generate a response message. In order to prevent it from overloading the network, make sure you only send one request per second. This can be achieved by inserting sleep(1) between every request. The User ID field should contain your student ID, and the password field should contain your guess. Note that both are 4 byte unsigned integers.
- RESPONSE: The server generates response to the request message. The response message contains a hint.

The request packet looks like this.



The response packet looks like this.



Ret indicates a return value and takes one of the following values:

- USER_NOT_REGISTERED: This means that the user id is not recognized by the server. If you can use the test svn repository it means that you are in the system. It is highly likely that your code is not functioning properly. If you have recently enrolled and suspect that you are not in the system, you must talk to the instructor immediately.
- LT: svn password < guessed password
- EQ: svn password == guessed password
- GT: svn password > guessed password

2.4 Failure (Part II)

In part II, your job is to make the client more robust by handling various network failures. You must use non-blocking I/O with select(). Do not use pthread or fork to spawn other process/threads. The timeout value should be 3 and you should retry up to 3 times.

2.5 Provided Files

- `msg.h` - contains some necessary constants

3 Technical Requirements

- Your project must function properly on the EE Linux machines. IP address of the machines are to be announced. Username is your student ID. The password will be sent to your email address.
- Your code must compile on the Linux machines.
- Your project must include a Makefile that builds your project
- Your code must be checked in to tags/pa1-final/ (see below)

4 Turning in your code

Your default svn repository is https://ina.kaist.ac.kr/svn/¡student ID¿. We suggest you make a trunk directory and put your working file there. For example, in the svn directory do the following:

```
# mkdir trunk
# cd trunk
;; create source files and copy msg.h
# cd ..
# svn add trunk
```

To turn in a working version of your code to us, you must make sure the files are committed to the **tags/pa1-final/** directory of your svn repository to avoid ambiguity. This is also required so that we can timestamp the code submission date and you don't have to worry about accidentally modifying the file after the due date.

One simple way to do this is to use the `svn copy` command. For example:

```
# svn copy trunk tags/final/
```

This will place a copy of everything in trunk into tags/pa1-final/.

VERIFY THAT ALL REQUIRED FILES ARE CHECKED IN TO SVN! You can do this by checking out your code into a different directory and making sure it builds (and passes the grading scripts in that different directory for the final turn in).

The Makefile must be a standard Makefile. For example, do not use cmake, as the Makefile it creates refers to a private directory that makes it extremely difficult for us to grade. The Makefile must not refer to any files outside the repository, and again, remember to turn in ALL the files needed to build or you will lose points.

5 C++ Tutorials and Resources

Most of your code may look like C. We don't expect to write an object oriented code for this class. However, you may use the STL library.

- C++ Tutorial
  `http://www.cplusplus.com/doc/tutorial/`
- C++ Reference
  `http://www.cppreference.com/wiki/start`
- C++ Primer
  by Lippman, Lajoie and Moo

6 Grading

The total number of points for this project is 100.

The breakdown of positive points is as follows:

- 10 points - Code quality, style (enforced with lint and cppcheck on all code submitted (Part I and Part II))
- 50 points - Code functionality and submission (Part I)
- 40 points - Robustness and Non-blocking I/O (Part II)

The grading breakdown for code functionalitity (Part I) is as follows.

- 15 points - Submission (right password)
- 15 points - Algorithm (don't do sequential search. Sequential search will get 0 points for this.)
- 20 points - Correct compiling with Makefile

Part II will look at the robustness of the code.

- 20 points - Handling dropped response/request packets.
- 10 pionts - Handling delayed response packets after the timeout. Packets may get sent after the time out.
- 10 pionts - Handling bogus packet. Your code must be robust to receiving bogus packets from the server (e.g., wrong magic number, wrong version number, or wrong ID/password). The client must retry after receiving bogus packets.