Socket Programming Assignment: Distributed Computing

Due Date: April 3rd , 2016 : 11pm

This lab is to be done in groups of two

Every single code of line should be your own. Note evaluation of this lab will be both via a demo session and as part of a lab final.

Description

In this assignment, we will build a client-server system in C/C++. No other programming language is permitted: this is best for learning the internals of socket programming. The assignment is based on distributed computing i.e. leveraging the processing power of 'worker' machines. As a case-study of distributed computing, we will undertake 'password cracking', needless to say for noble causes :-)

Most servers store user passwords after hashing the password and not in plain text for obvious security concerns. When a user (client) types a password to login, the serve hashes the password and then compares it with the stored hash value and if it matches, lets the user in.

The idea here is that a user has access to the hash and wants to retrieve the original password. The hash function is one-way, so to retrieve the password, one has to resort to some form of brute-force approach: generate a string, hash it and see if it matches the hash; then the string is the password. If the password was 8 characters and can take upper/lower case as well as numeric characters, then we have 62^8 combinations. For a machine that can process 1 million hashes per sec, it will take worst case 6.9 years to crack it. That is where distributed computing can help. If you employ 100 machines, you can do the same task in 25 days.

You will implement one such system but of course at much smaller scale and lot simpler.

Implementation details:

At a high level, the password cracker system you build will consist of 3 programs: User, Server and Worker. A user sends a hash of a password to the server for cracking it. The server will divide the job among a group of workers, who get back to the server with the status of the allocated job. After completion of the job, the server gets back to the user with the password.

1. User: A user passes a hash to the server and will print out the password once it hears back from the server as well as the time taken for cracking the password. Note server may take a long time in getting back, your code should work given this.
   A User passes three pieces of information to the server
   - hash of the password
   - no of characters in the password
   - a 3 bit binary string, where the three bits indicate the use of lower case, upper case and numerical characters in the password. For example, 001 indicates the password is made up of

only numbers; 111 indicates the password can contain lower, upper as well as numerical characters

Your user code should produce an executable 'user' and should use the following arguments:
./user <server ip/host-name> <server-port> <hash> <passwd-length> <binary-string>
Example: ./user osl-9.cse.iitb.ac.in 5000 aBBHPfIg 6 001

The user should also print the password and time taken for cracking the password on the console once it hears back from the server.

2. Server: The server can be contacted by multiple users and workers at a given port (use Select call to handle this). In this lab, we will restrict maximum number of users to 3 and workers to 5. The server can allocate jobs as it sees fit. The logic is yours but ensure that the load is balanced and it does an efficient job. For this, the server needs to keep track of who is working on which job and the current status. As a new user requests come, server should assign these jobs to workers without prempting them.
   Your server code should produce an executable 'server' and should use the following arguments:
   ./server <server-port>
   Example: ./server 5000 (basically server is listening on port 5000)

3. Worker: A worker registers with the server as soon as it is started. Then it waits for a job from the server and gets back to the server with a status of the job once it completes it.
   Your worker code should produce an executable 'worker' and should use the following arguments:
   ./worker server ip/host-name> <server-port>
   Example: ./worker osl-9.cse.iitb.ac.in 5000

4. Miscellaneous Stuff:
   - To ensure reliability of your communication in the system, you can go with TCP. If you want to use UDP, you need to implement reliability within your application.
   - Crypt() is the function you should use to obtain a hash from a given string. See its man page for more details.
     char *crypt(const char *key, const char *salt);
     The key is the password you want to encrypt, salt is a two character string that can be used to produce different encrypted versions of the same key.
     crypt("abc","aa") gives aaMjTET7rKV4Y
     crypt("abc","ab") gives abFZSxKKdq5s6
     (crypt() is not recommended these days due to some security flaws, but for this lab it is ok)
   - Use loopback interfaces for initial testing (i.e. all processes run on the same machine). Once it works fine, you should run the processes on different machines and make it work. In the demo, some processes will run on the same machine and some on different machines. As an example, I may ask user1, worker1 to run on machine-1; server, user-2 to run on machine-2 and worker2 on machine-3 and worker 3 on machine-4.

5. Evaluation: You will need to generate a single figure with 3 plots (in jpg format). Each plot corresponds to a given length “numeric” password. The length of the password will be 7, 8 and 9 numeric characters. The plot measures the average time taken to crack the given length password

as you vary the number of worker. The x-axis will be the number of workers ranging from 1 to 5 (have to be different machines, else plot may not make sense). The y-axis will be the execution time. Each data point in the plot should be averaged over 10 runs (different passwords of same length).

__General Instructions:__

1. This lab is to be done in **groups of two students.**
2. You are **not allowed to exchange code snippets or anything** across groups. In case of any doubts, you are allowed to consult any Internet resource, books, or the instructor. While it is alright to read on general socket programming on the Internet, and look at code examples, this should be only for the purpose of understanding. Do all the coding yourself, do not copy or cut-paste code from any website. You can use only socket programming for these projects. Do not use any sophisticated libraries downloaded from the web. Please understand the spirit behind all these and follow them strictly.
3. You can use C/C++ for coding, **no other language**. Although Java/Perl and other languages may provide a better interface, C/C++ are bare-bones, and hence appropriate for a networking course.
4. Give sufficient time for testing the code, do not cram everything to the last day. Remember that in industry, people spend more time testing than coding!
5. Commenting and indenting are important. Comment *during/before* coding, not at the end. For every variable/function you should have a comment. Use appropriate code comments for explaining the logic where necessary.
6. Provide documentation for your code. This should explain the code structure in terms of directories, files, how to compile etc. Detailed instructions for submitting the documentation along with code are at the end of the document.
7. Pay attention to the variable names, function names, file names, directory names, etc. Make sure they are intuitive.
8. Refer to http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html for more instructions on coding conventions. Although this link is for Java, many of the principles are applicable for any programming language.
9. We will evaluate all of the above aspects of your code, not just whether or not the code runs.

__Submission guidelines__

Read the below instructions *carefully* and follow them meticulously.

Organizing your submission

- All relevant files should be under one directory. T**his directory should be named after the roll numbers of the two students.**
- Within the directory, you should have a file called "**README.txt**". The following contents are required in the README file:
  - **List of relevant files:** Give the list of relevant files including all source files and configuration files which *you* have written. Of course you need not include things like stdio.h which someone else has provided for you. Specifically *do not* have any irrelevant, old, or temporary files which clutter the directory. Clean-up before submission.

    - **Configuration file(s):** If your code uses any configuration file(s), describe the format of such file(s) clearly. Also include in the directory some example configuration files.
    - **Running instructions:** You may be generating more than one executable. Describe logically what each executable does. In addition, for each such executable file, how should one run it? What are the command line arguments (describe each argument)?
- Within the directory, you should have a “**Makefile**”. The default targets of the Makefile should be the executables user, server and worker. It should also have a "clean" target so that I can do "make clean". If I type:
  make clean
  make
  I should get the executables user, server and worker in that directory.
- The directory should have a **results.jpg** file (see evaluation)
- Apart from the readme, makefile and results.jpg, you can have any number of sub-directories and code files.
- In the final submission, you have to tar-gzip or zip the main directory and submit a single file. Make sure to tar-gzip or zip from the *parent directory* of this directory, not from within this directory itself.
- Submit the tar.gz or zip via BodhiTree1 before the deadline.

The real test

- Save your submission somewhere and look at it after say, a semester. Does your README help? Do your comments help in understanding what you did in the code? This is the real test! Of course, since we have to grade you much before that, we will look at your code, comments, and documentation as described above.

<u>Demo & Evaluation</u>

Details to be provided later.