Concurrent and Distributed Systems – Course Project

Deadline (no extension possible!): 28th April, 2016 – 11:55 PM.

Project Overview:

In this Project, you need to implement a distributed "password cracker"[1]. The proposed project will be internet-based i.e. the server and computing nodes will be able to use hostname/IP address to communicate with each other. In order to decipher a password, we will use brute force (testing for all possible combinations) and as the number of combinations increases exponentially with the size of password, we would need a distributed system to assist us in this task. The project would come into its own for problems of significant complexity and you need to demonstrate your approach as such.

One way to encrypt a password is using a cryptographic hash function. As the name suggests, it is a hash function that takes an input and returns a fixed-size alphanumeric string. Ideally, it should be extremely computationally difficult to regenerate the password given only the hashed text. One example of the usage of such cryptographic hash function is password management in Operating Systems. Instead of storing the password in plain text, the OS encrypts (computes hash) and saves it. As a result, even if someone steals the password information (for instance /etc/passwd file containing the hashes of the password), he/she doesn't immediately learns the passwords.

In terms of implementation, crypt() function can be used (however not recommended in general) to generate the cryptographic hash. It takes two arguments, the password to encrypt and a constant string, called salt, that can be used to produce different encrypted versions of the same password. That is:

```
crypt("ez", "aa") -> "aaIeGWlWFikfg"
crypt("ez", "ab") -> "baFxrQlh02Qxw"
```

For this project, you'll always use blank salt i.e. "", crypt("ez", "") -> " kmscp9jrzH2". To summarize the objective is that given a hashed password, for example, kmscp9jrzH2, you need to have a distributed system that is able to tell us the plain text password, that is, ez. The user provides the hashed password to your application and it uses brute-force (testing all possible combinations) to identify the password. As you can imagine, trying all possible combinations on a single machine is not feasible, and having studied the Distributed Systems course, you will use a fully distributed system where multiple slave nodes will assist the password cracking process.

For simplicity, we assume that the maximum length for the passwords is only 5 characters and that it can only include characters (both lower and uppercase alphabets).

[1] Inspired from https://www.cs.cmu.edu/~dga/15-440/F10/lectures/pass_cracker.pdf

Application Architecture and Components:

The overall application architecture consists of three core components (and corresponding programs): Client, Server and Slaves.

- The client is responsible for making the request, to the Server, to decipher the password.
- The server program on receiving the request, welcomes the client, analyzes and divides the task into parts and allocates these parts to Slaves, which have already registered with the server.
- The slaves are the workhorses and they register with server and are assigned the tasks. The tasks include the original password to crack and the range assigned to the specific client.

Example Scenario:

Our client is a golang program that uses command line arguments to get the encrypted password and other related information, see details below. Assume that the user has provided the encrypted password *kmscp9jrzH2* (*encrypt*ed hashed value for ez). Let us also assume, to explain the process, we know in advance the password length is only 2 and can only contain lower case alphabets. All the possible combinations (aa-zz) are 676 i.e. 26x26.

Although for this trivial example, there is no need for a distributed system but we are looking at the restricted version, just to elaborate the process. Once the server receives the request, it partitions the problem in parts (in relation to the number of slaves which have registered with it). Lets assume three slaves have already registered with the server and thus the server needs to partition the overall job in three parts, one for each. Lets assume, it allocates 234 possible combinations to check to each to both slave1 and slave2 and remaining 208 to slave3. Thus slave1 is responsible for checking passwords from aa-iz, slave2 is responsible for chunk ja-qz and slave3 is responsible for the chunk ra-zz.

The slave program/node on receiving the chunk tries all possible combinations. For slave1 it starts with aa, encrypts it, and checks the encrypted hash code with the one provided by the client. It then moves on to ab, then to ac and so on.

The Client Program

The client program is responsible for making the request to the Server, to decipher the password. It does not take user input but rather uses the command line arguments for the required information from the user. The command line parameters include:

- ***-cipherText****=someCryptedPassword* – This argument is the compulsory for the user to provide and represent the cipher text that needs to be decrypted.

- ***-hostName****=serverHostName* – This argument represents the host name for the server, to whom the request is being sent. This argument is optional and if omitted, defaults to 127.0.0.1
- ***-hostName****=serverHostName* – This argument represents the port number on which the server is listening This argument is optional and if omitted, defaults to 2600

Example usage:

- *go run client.go -cipherText=kmscp9jrzH2 -hostName=127.0.0.1 -port=2600*
- *go run client.go -cipherText=kmscp9jrzH2*

Once the request has been sent to the server, the client program displays the information to the user and waits for the response from the server. Once the response arrives, it displays it to the user and exits.

The Server Program

The server is the core component and listens on two different ports; one it exposes to Clients to receive jobs (password cracking requests). The other port is for Slaves to register themselves with the server. Specifically it has following core attributes:

-
- The ports on which it is listening are provided using command-line arguments and should default to some values if arguments are missing.
- It should be a multi-threaded server and should be able to handle more than one client and their requests simultaneously.
- It maintains a lit of Slaves to whom it can delegate computation. This list is populated by the messages it receives from the Slaves. See the Slaves description below for details. It should be both fault tolerant (if the Slaves go down) and also should be able to handle new Slaves being added.
- It serves as the job allocator and based on the size of the job (request) it splits the job in pieces and distributes them to Slaves.
- It also serves as the load-balancer and equally distributes the load amongst Slaves and manages different clients and slaves.

The server is the core component and carries significant weightage.

The Slave Program

The Slaves are the workhorses and register with the Server, assigned the tasks and report the success/failure status and associated information to the Server. Specifically the core attributes of the Slave component are as follows:

- The hostname for the server and the port is specified by command line arguments as with the Client program.
- On startup, the Slave program connects to server, on the specified hostname and port, to register itself with the server.
- Once registered, it listens on a port (whose number can only be provided using command line argument) for the tasks assigned from the server.
- Once the task is assigned, they perform the task and send the response back to the Server.

Evaluation Criteria:

The evaluation criteria would be communicated later in a separate post.