Parallel and Distributed Computing

Project Overview:

In this Project, you need to implement a distributed "password cracker". The project needs to be implemented for a clustered computing environment with multiple nodes running MPICH implementation of the MPI specification. 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/shadow** file containing the hashes of the password), he/she doesn't immediately learn the passwords. For our project we would use the Ubuntu Linux distribution and it uses SHA-512 algorithm to store the encrypted passwords.

The crypt( ) function:

In terms of implementation, crypt() function can be used to generate the cryptographic hashes. 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"

There are some further applications of the Salt used in the crypt and as per the man pages, "If salt is a character string starting with the characters "id" followed by a string optionally terminated by "$", then the result has the form:

id**salt**$encrypted

id identifies the encryption method used instead of DES and this then determines how the rest of the password string is interpreted. The following values of id are supported:

ID | Method

---

1 | MD5
2a | Blowfish (not in mainline glibc; added in some Linux distributions)
5 | SHA-256 (since glibc 2.7)
6 | SHA-512 (since glibc 2.7)

Thus, 5salt$encrypted and 6salt$encrypted contain the password encrypted with, respectively, functions based on SHA-256 and SHA-512.

The code segment below uses the mpiuser as the text to encrypt and **$6$4GfdWqHx$** as the salt to produce the hash as of our mpiuser user, as stored in /etc/shadow of the VM we are using.

```
#include <stdio.h>
#include <unistd.h>
#include <crypt.h>

int main(void)
{
  char id[] = "mpiuser";
  char salt[] = "$6$4GfdWqHx$";
  char *encrypted = crypt(id, salt);
  printf("%s\n",encrypted);
}
```

The code can be compiled using the commands below and you can notice the use of - lcrypt to compile the code.

```
$ gcc cryptS.c -lcrypt -o cryptS
$ ./cryptS
$6$4GfdWqHx$ye1ww.GiiIUWs7J028KKNU2KySHqhc9OBC5eNk6TNDVWHa8n70v2n/9cHfN
SKdSeK78gthZcyr2N4VdEsoH2o.
$
```

In terms of control flow, your project would ask the user to input the username and it then reads the **/etc/shadow** file to get the hash of the password. The application then 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 developed and programmed the cluster in our course, you will use a cluster-based system where multiple nodes will assist the password cracking process.

For simplicity, we assume that the maximum length for the passwords is only 8 characters and that it can only include lower case alphabets (a-z).