MPCS 52040 Distributed Systems
Homework 4: Password cracking

Due: 2/14/25 @ 10:00am

Link: `https://classroom.github.com/a/zt9SN4Zl`

Distributed Password Cracking

In this assignment you will create a distributed password cracking system. You will implement the system using REST (Flask) and run several instances of the REST service. You will develop a client that can distribute portions of the password search space to instances of the REST service. Each call to the REST service should attempt to brute force crack the password by checking each combination of letters against the hashed password. You will then enhance your service to improve fault tolerance and performance by caching previous attempts.

Background

Storing passwords in cleartext is an unsafe way of managing passwords due to the risk that passwords could be discovered. For example, if a server were to be compromised then attackers could retrieve all passwords stored on the server.

A better approach for storing passwords is to use a one-way hashing function to encode the password in an irreversible format. Note: while hashing algorithms are also used in hash tables, here we focus on the use of hashing to convert the password to a format that cannot be reversed. Hash tables use hashing algorithms to map keys to a fixed size table and uniformly distribute hashes across a key space.

The general idea behind a hashing function is to map input data (of any size) into data of a fixed size:

```
hash(string) => hashed_string
```

For example, using the MD5 hashing algorithm we can hash the string "mpcs" as follows:

```
md5('mpcs') => cce6b3fb87d8237167f1c5dec15c3133
```

With a hashing function the same input data will always be hashed to the same output data. However, there should be no way to reverse that process (i.e., going from output to input data). That is:

```
Unhash(cce6b3fb87d8237167f1c5dec15c3133)
```

does not recover the input data.

This is useful for storing passwords as it provides a way to compare the password entered by a user with their stored password (to confirm valid authentication) without the server ever needing to store or compare the cleartext password. THat is, we can compare 'cce6b3fb87d8237167f1c5dec15c3133' with 'cce6b3fb87d8237167f1c5dec15c3133' and know that the user entered their password correctly without ever seeing 'mpcs'.

There are many hashing functions with various properties and complexities. Here we will use the MD5 algorithm as it is simple and fast. **NOTE: MD5 is not secure** (and should not be used to store passwords) as it suffers from several vulnerabilities. We use it here as it is a good proxy for other (more computationally expensive) hashing algorithms. MD5 produces a 128-bit hash value.

When implementing a production authentication system, you should not simply store raw hashed values as it can leak information about users that use the same password. One method to avoid this issue is to add additional random data (called a salt) to the input such that there is no possibility of determining if passwords are the same. The salt can be stored in a database as it is used only to change the resulting hash. Further, best practices these days also extend the length of the hash to make it less computationally feasible for a brute force attack (this is important in the case that attackers obtain salts and because users often choose common passwords).

One final note: hashing is not equivalent to encryption. Encryption is a two-way function in which the value can be encrypted and then subsequently decrypted using a key. Hashing is a one-way function in which there is no way to reverse the hash to derive the original value.

Example code

The code below may be useful in this project. It will create hashes for combinations of lowercase characters (the commented out line above can be used for all printable characters, e.g., uppercase, lowercase, digits, and other characters) and compare the hash against a given hashed password. It will check each combination of characters up to the max_length and return the unhashed password. You are welcome to change this code to match your interface/messaging format.

```
import itertools
import string
import hashlib

def bruteforce_password(hashed_password, max_length):
    # chars = string.printable # all printable characters
    chars = string.ascii_lowercase # lowercase characters
    for password_length in range(1, max_length+1):
        print (password_length)
        for guess in itertools.product(chars, repeat=password_length):
            guess = ''.join(guess)
            if hashlib.md5(guess.encode()).hexdigest() == hashed_password:
                return guess

    return None

print(bruteforce_password('cce6b3fb87d8237167f1c5dec15c3133', 4))
```

For your client code, you should use the Python requests library.

Generating hashed passwords

You will need to generate hashed passwords to test your service.

The easiest way to do this with Linux is:

```
echo -n <password> | md5sum
```

Alternatively, you can generate hashed passwords in Python using the following code:

```
import hashlib
hashlib.md5('password'.encode()).hexdigest()
```

Part 1: Write the cracker web service

In the first part of the assignment, you will create a REST service using Flask for cracking passwords. The service should accept a hashed password (e.g., 'cce6b3fb87d8237167f1c5dec15c3133') and return the unhashed password (e.g., 'mpcs'). It is up to you how you design the interface in terms of endpoints and operations supported, however, it should follow best practice for REST services. The service must take JSON input and return JSON output. You should start the service as follows:

```
python cracker_service.py <port>
```

You will then need to create a Python client that can call your service to crack a given password. You should specify the number of characters in the password search space.

```
python client.py <port> <md5_password> <max_password_length>
```

Note: it is computationally expensive to brute force search the character space. You should start with very short passwords and with only a subset of characters (e.g., only lowercase characters).

Part 2: Distribute workload across services

Now you will need to devise a method to distribute the search problem across several instances of your service. First, think about how you can partition the search space so that a set of tasks can be submitted to each instance of the service. Once you have designed this approach, extend your service so that it can take a partial search space and return either success (with the cleartext password) or failure (not in the partial search space).

Extend your client so that it can break up the search space and send requests to many instances of your service. (you should start the services on a set of different ports before running your client, e.g., with a bash script). You will need to think about how your client can submit many tasks to many services concurrently. You should also think about how to size the work for each service and ensure that each service is processing a different "chunk" (i.e., don't send the entire space to every service). Finally, you should think about how you "chunk" the work in a way that does not lead to significant performance limitations.

The client should be started as follows.

```
python client.py <start-port> <end-port> <md5_password> <max_password_length>
```

Test your implementation to confirm that it indeed distributes workload across the connected services.

Document how you have tested this part of the project in your report (either txt or markdown).

Part 3: Fault tolerance

Failures are one of the most challenging aspects of distributed systems. In this part of the assignment, you should make sure that your client is resistant to service failures. Test your implementation by manually killing a service (e.g., via Ctrl-C) while running and ensure that the client continues to explore the search space and does not miss finding the password. Note: you do not need to deal with recovery of the service that failed.

Please document how you tested fault tolerance in your report.

Part 4: Performance

One important skill in distributed systems (and systems in general) is being able to analyze the performance of your system. In this part of the assignment, you will analyze the scalability of your system.

Create one or more plots that show the average time to crack a password as you increase the number of services, length of the password, and size of chunks. You should have at least two plots. The first showing number of services vs password length. The second, you can set a fixed password length and number of services, while scaling the chunk sizes. It is up to you how far you go, but you should increase the number of workers to be greater than the number of cores on your computer. You should crack password sizes that take a few minutes to crack. There is no need to consider password sizes that take many hours to crack. Remember to start small as you might be surprised how computationally intensive it is to crack a password.

Please describe your experiment in your report and include the scaling figures in the Git repository.

Part 5: Optimization

One problem with our architecture is that if a user were to request to crack the same password again it will result in the same workload being executed. We can use caching to optimize performance by storing results such that they will not need to be computed again. In your web service (**NOT the client**) add an in-memory cache for storing the results of cracking requests for the partial search space.

Please document how you implemented and tested the caching support in your report.

Submission

Please use the following link to create a repository for homework 4 in our GitHub classroom.

`https://classroom.github.com/a/zt9SN4Zl`

To assist the grading process please submit your final service and client. You do not need to submit separate files for each part.

Also submit a report (in either markdown or txt format) outlining how your solution deals with fault tolerance and caching, how you have tested your service, and the performance evaluation showing how your system scales.