CS 655: Computer Networks --Fall 2020

Distributed Password Cracker -- Mini Project Report

Team members and division of labour

Ziqi Tan	U 88387934	ziqi1756@bu.edu	System designer and frontend engineer
Xueyan Xia	U 82450191	xueyanx@bu.edu	Backend engineer (Spring boot)
Kaijia You	U 44518396	caydenyo@bu.edu	System performance tester and evaluator
Jingzhou Xue	U 10828768	jxue@bu.edu	Algorithm engineer

The GENI Rspec, source code, video is in Github Link: https://github.com/PhoenixTAN/Distributed-Password-Cracker
The file structure and the corresponding description is in **README.md**

1.Problem Statement

1.1 Definition

Distributed-Password-Cracker is a distributed system where a user submits the md5 hash of a 5-character password (a-z, A-Z) to the system using a web interface. The web interface with the help of worker nodes cracks the password by a brute force approach.

Our goal is to implement the web-interface as well as the management service that will dedicate jobs to worker nodes. Necessarily, the worker nodes need to be different machines. Moreover, the system should be scalable, where we can add or remove workers on the fly. mming to communicate with the worker nodes.

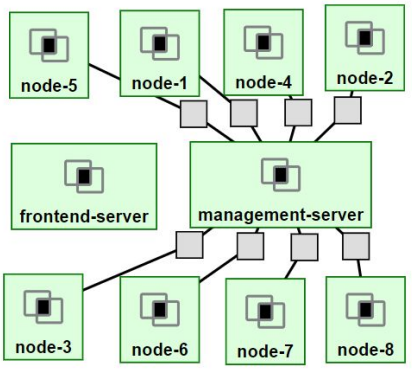
1.2 Motivation

The MD5 message-digest algorithm is a widely used hash function producing a 128-bit hash value, which has compromised security for password encryption. However, more scalable systems and advanced devices with larger memory or stronger computing power keep challenging the capability of this password hashing algorithm in recent years. Even some institutes develop systems and open them to the public, providing a cracking method for encrypted password. In this project, we develop a distributed system with a basic brute-force approach and cache to simulate their operations, evaluating the security of MD5 algorithm.

2.Design

2.1 Diagram

There are 8 workers and a management server, a front-end server in our project.



Management server (Routable IP: 192.86.139.65):
The management server receives a request from a front-end server, or handles multiple parallel requests from a Python crawler. Then, it dispatches password cracking jobs and forwards the request to workers based on the parameter from the request.

Front-end Server (Routable IP: 192.86.139.64):
It provides web service for clients to send password cracking requests and assign an arbitrary number of workers to solve this problem.

Node1-Node8 (do not have routable IP):
These are workers who crack passwords by a brute force approach with cache.

2.2 Environment

Operating System
4.15.0-121-generic #123-Ubuntu SMP Mon Oct 5 16:16:40 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux

Java version
JDK: Java 1.8, Spring Boot: 2.x

2.3 Resources

We only have limited resources in a worker node to crack passwords, which is even poorer than our own laptops.

Each node only has 1 GB memory in total, making it hard to cache enough information in the memory.

2.4 Implememtation

Clients can use the Web service we provide to send crack requests (via Restful API). We decouple the frontend and the backend. The crack requests are sent to the management server (backend service) from the browser. Then, the management server dispatches the jobs to different workers located on various machines. Once received the correct password from one of the workers, the management server will return it to the client/browser.

2.4.1 Front-End

We use React to build our frontend application. After building the project, we deploy it on the frontend server using the nginx as the Web server.

We can access the frontend service via 192.86.139.64 (both PC and mobile are supported).

User handbook:

- **Password field**: users can use this to generate the MD5 easily and do not have to copy back and forth, but the crack request will only send the MD5 and the number of workers.
- **MD5 field**: you can paste your own MD5 here.
- **Number of workers field**: you can change the number of workers on the fly, but we strongly suggest using 8 workers.
- **Crack button**: send the post request to the management server to get the password. It may take 1.5 minutes generally. After that, the correct password will be shown (e.g. "The password behind md5: AbCdE").
- **Clear button**: clear all the fields above.

Technology details

1. CORS(Cross-origin resource sharing) problem

The crack requests from the browser are cross-origin requests, because the service in the browser is in origin http://192.86.139.64:80, while the backend service provides Restful API http://192.86.139.65:8080/getPassword. The services are provided from different origins. The browser does not allow that. Thus, we need to mark the response header as cross-origin and allow origin http://192.86.139.64:80 to visit backend service to avoid the response being blocked by the browser.

2. The post requests are sent with the JSON format body.

2.4.2 Back-End

The Back-End service is implemented by Restful API with Spring Boot framework.

By assigning specific link for the MessageController and use Annotation @RestController and @PostMapping, the management server could handle a http request by http://server address:port number/getPassword.

Here we accept json data, in which we have the number of workers and an encrypted password waiting to be cracked. Then the server calls the dispatcher to distribute jobs ("aaaaa" - "ZZZZZ") to our workers. In this step, we use the CompletableFuture interface to inform the workers parallelly and asynchronously.

When any of the workers return the response, the *CompletableFuture.anyOf()* function would capture the result and get the correct password. In many cases, one of the workers would return the result in advance, which means we could directly return the password to clients instead of waiting for all workers to complete the jobs.
In the end, the *CompletableFuture.allOf()* function would ensure all the workers completed their jobs, then the main thread would be terminated and a specific password cracking task completed formally.

2.4.3 Dispatcher and Worker

In this project, we target all 5-character password combinations with all upper and lower case English characters. Since there are 26 + 26 = 52 choices for each character, in total there are 5^52 = 380204032 possible password combinations from "aaaaa" to "ZZZZZ". Here we know that all hashes are obtained from the MD5 hashing algorithm and no nonce is used. Thus, to crack the MD5 hashes, we can simply re-calculate hashes of all possible password combinations and match with the target to get the original password.

Our goal is to distribute tasks (hash all passwords from "aaaaa" to "ZZZZZ") evenly to the requested number of workers.

```
private String increment(String s, int step) {
    if (s.length() != LENGTH_OF_PASSWORD) {
        return "";
    }
    char[] chars = new char[LENGTH_OF_PASSWORD];
    int position = 0;
    for (int i = 0; i < s.length(); i++) {
        int charIdx = LENGTH_OF_PASSWORD - 1 - i;
        position += Math.pow(BASE, i) * (ALL_CHARS.indexOf(s.charAt(charIdx)));
    }
    position += step;
    for (int i = 0; i < s.length(); i++) {
        int charIdx = LENGTH_OF_PASSWORD - 1 - i;
        chars[charIdx] = ALL_CHARS.charAt(position % BASE);
        position /= BASE;
    }
```

Firstly, since our passwords contain both uppercase and lowercase characters and they are not consistent in the ASCII table, it is relatively hard to do calculations based on characters. Intuitively, we can map each character to integers respectively ('a' = 0, 'z' = 25, 'A' = 26, and 'Z' = 51) and calculate in base 52. Here we use it to provide the function to increment a string which we would use to loop through our problem set.

With the above helper functions, each worker can brute force the password by looping through its targeted range.

Then, it is the management server's job to distribute tasks to each worker. It divides the total number of tasks evenly. Then use the above Increment function to find the start string and end string for each worker.

2.4.4 Cache

In this project, we add two caches to our system. These two caches are implemented by static ConcurrentHashMap, which is a thread-safe map regularly used as cache. When the cache is full, it would discard the first element automatically. The size of our caches here actually is limited by the memory of the nodes on GENI. In these caches, the key is an encrypted MD5 password, the value is the corresponding real password.

The size of the cache on the management server is 500,000. When it returns a result to clients, it would store the result as an element in the ConcurrentHashMap.

In workers, the size of the cache is 300,000. While it is cracking the password, it would try to store each calculated result in the ConcurrentHashMap.

3.Executions

3.1 Configuration/Deployment/Usage

In this part, we talk about how to set up the environment and run our codes in GENI servers.

3.1.1 Frontend server

- **Run** the frontend project locally:

1. We use "create-react-app" as the tool to build the frontend.
2. Run "npm install" and then "npm start" to run the frontend locally.

- **Deploy** it on the frontend server.

1. In "package.json", add a field "homepage": ".".
2. Run "npm run build" to build to package the project.
3. Upload all the files in the "build" folder to the path "/var/www/html" in the server.
4. On the server, run "sudo apt-get update" and "sudo apt install nginx-full" to install the nginx.
5. Edit the nginx configuration file in path "/etc/nginx/nginx.conf".

- **Add** the following code into http configuration.

```
server {
    listen 80;

    location / {
        root /var/www/html;
        index index.html index.htm;
    }
}
```

6. Run "sudo service nginx start" to run nginx. Other commands are useful, such as "sudo systemctl status nginx" to check the nginx status, "sudo service nginx reload" to restart the nginx service.
7. Finally you can use your browser to access the web service.

3.1.2 Management server

We simply package the whole spring boot project into a jar file and then run this file along with a configuration file "application.properties".

1. Add "<packaging>jar<packaging>" into "pom.xml".
2. Run "mvn clean package" at the root of your project then it will generate a "target" folder where there is a jar file.
3. Upload the jar file to the server.
4. In configuration file (/src/main/resources/application.properties), we add the worker nodes IP --
list:10.10.1.2:8081,10.10.2.2:8081,10.10.4.1:8081,10.10.3.1:8081,10.10.5.1:8081,10.10.6.1:8081,10.10.7.1:8081,10.10.8.1:8081.
server.port = 8080
5. Upload the "application.properties" to the server.
6. Install Java 1.8. Run "sudo apt-get update" and "sudo apt-get install openjdk-8-jdk". Then, run "java -version" you will see
openjdk version "1.8.0_275"

OpenJDK Runtime Environment (build 1.8.0_275-8u275-b01-0ubuntu1~18.04-b01)
OpenJDK 64-Bit Server VM (build 25.275-b01, mixed mode)

7. Run “nohup java -jar management-server-v1.jar -Dspring.config.location=application.properties” to run the service.

3.1.3 Worker nodes

The deployment is basically the same as that of the management server node.

application.properties: server.port=8081

3.2 Metrics and graphs

3.2.1 Testing environment

In this project, we use Python to evaluate the performance of the server. In Python, we import gevent and requests packages to simulate concurrently requesting the server. Firstly, we define the request url and restrict the message format to be JSON.

Secondly, we randomly create a five-digit password and corresponding MD5 password to fill into the request body. Because there are 5^52 = 380204032 possible password combinations from “aaaaa” to “ZZZZZ”, so there are 47525504 possible password combinations for each worker. If the initial letter of the password is relatively big, the calculation time will be longer. For this reason, we just create the password whose initial letter and the second letter are between ‘a’ and ‘c’ and return the JSON format of the request body. Thirdly, we declare a function that sends the POST request to the server.

Finally, we define the number of concurrent requests in the main function then do concurrent requests in the call_gevent() function. In this function, we record the start time and append all requests in a list just like what we do by using threads. Then join these requests all and record the end time. By this way, we can concurrently send the requests to the server and calculate the response time in total.

3.2.2 Evaluate server computing power

To evaluate the server computing power, we should control the concurrent load so that we just request the server under 20, 30, 40, 50, 60 concurrent load to save time and the average of the corresponding single response time is 7.111s, 9.084s, 9.875s, 10.014s, 9.775s. Thus, we can draw a conclusion that the average response time is 9.888s when cracking a new MD5 password.

3.2.3 Evaluate server under high concurrent load

To evaluate the server under high concurrent load, we should ignore the server computing power so that we just request the server to crack the MD5 password of ‘aaaaa’ which has been cached in the concurrentHashMap in the management server. We tested 100, 500, 1000, 10000 and 50000 concurrent loads, the average of the corresponding single response time is 0.00946s, 0.00588s, 0.00506s, 0.00454s, 0.00494s. Thus, we can draw a conclusion that when the server is under a relatively high concurrent load, the average response time is around 0.00474s.

3.3 Conclusion, Analysis and discussion

To improve the performance of our model, we have tried to implement some cache. The idea is similar to the rainbow table a few years ago which maintains a large hashMap of common passwords and hashes. This methodology is leveraging computer memory against hash power. Since the GPU is crazily fast today and memory is quite expensive, the rainbow table is already outdated. However, under our circumstances, the geni node does not provide any GPU capability which limits our hash power. So we decided to adopt the idea and trade some memory space for faster response. As a result, users would get significant faster results if re-entering the previous queries.

Details: When we assign 8 workers, the jobs are dispatched as the following ranges.

start=aaaaa, end=gAaaa;	**start=gAaaa, end=naaaa;**
start=tAaaa, end=Aaaaa;	**start=naaaa, end=tAaaa;**
start=Aaaaa, end=GAaaa;	**start=GAaaa, end=Naaaa;**
start=Naaaa, end=TAaaa;	**start=TAaaa, end=ZZZZZ;**

On a single CPU computer, it takes 8 minutes or so to finish the job from “aaaaa” to “ZZZZZ”. Thus, theoretically, if the jobs are divided by 8, each request will be returned in 1 minute. The smaller (alphabetically) password we request, the faster we can get the response.

Error conditions:
As we send more and more requests, some worker nodes sometimes may become zombies. They just receive the jobs but not response. All we need to do is restart the node.