Password Cracker

Ngu (Nathan) Dang, Ludmila Glinskih, Ta Duy Nguyen, Fabian Spaeh

GitHub link: https://github.com/285714/cs655-pw-cracker
Name of the GENI slice: Project-Spaeh
Link of the web interface: http://pcvm1-22.geni.it.cornell.edu:8080/
Demo: https://drive.google.com/file/d/1ZHwmtlB8w65M8YygPGNvgSdwQZOd09JN/view

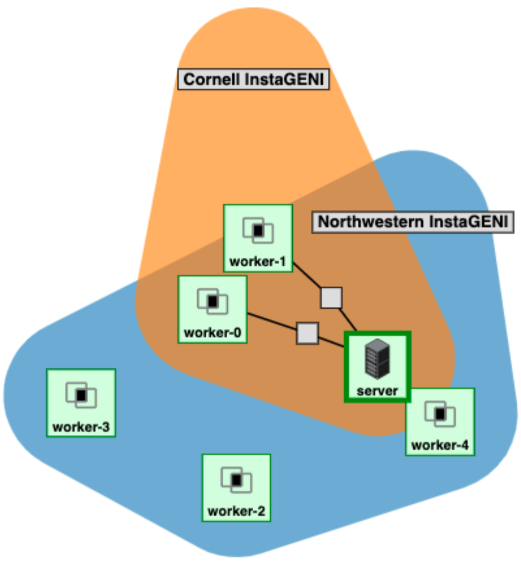
Introduction and Problem Statement

We design and implement a simple 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 using the brute force approach. During our work on this project, we plan to learn:

- how to model workers, the communication delay and request arrival
- how to work with asynchronous communication between the server and clients via sockets programming;
- how simple map-reduce algorithms work;
- how to create GENI `rspec.xml` configs;
- methods for implementing a simple interactive web application.

In the following sections, we explain how we designed our system in detail, then we present the experimental methods as well as the results, and finally we conclude with some possible extensions.

Experimental Methodology



Worker: a unit that receives requests of a form `(hash, range)`, and checks whether there is a string of length 5 between `aaaaa` and `WWWWW` that has the same MD5 hash as hash.

Server: main component of the system, runs web-interface and server application. Communicates with 5 workers. Each server creates multiple processes for communicating with each of 5 workers, because of that (and limitations on the used resources on GENI), each server cannot serve too many simultaneous requests.

System Design

The web-interface and the main server application, that dedicates jobs to worker nodes, are on the same machine. The system is scalable: through the web-interface a user sets the number (between 1 and 5) of workers they want to use, and then they can restart the password cracking with a new number of workers. Each worker gets an MD5 hash of a password and a fixed range of strings for which the worker computes an MD5 hash of all strings in the range and compares it with the hash of the password. The management (server) service uses socket programming to communicate with the worker nodes.

All workers are on different machines: one machine, one worker. In our GENI instance, we use 6 virtual machines: 1 machine runs a web-interface and a server application, 5 machines run 5 workers. 5 workers are accessible to the user. We support the following failure of a worker, that we detect through the termination of the socket connection:

- machine dies,
- connection problems.

Results

Usage Instructions

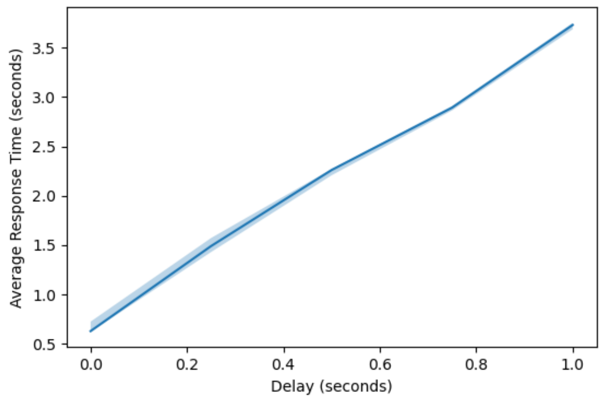
Requests to the currently running password cracking service are supported through the web interface: http://pcvm1-22.geni.it.cornell.edu:8080/. For our experiments we used this web interface, and direct calling of methods of Server service on a server machine on the GENI instance `Project-Spaeh`. Similar Geni instances can be created using `rspec.xml` file provided in the github repo of the project.For the detailed usage instruction, please, refer to the README.md file in the github repo.

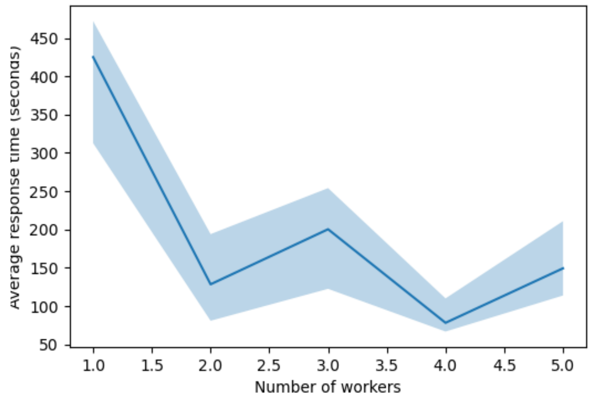
Experiments

We analyze the time required for cracking a password depending on the number of workers, simultaneously served users, and different failures in the system.

One user, one worker and multiple delays

In this experiment, we assume that there is only one request and for simplicity, the server uses one worker. We change the network delay between the worker and the server and measure the time between the moment a user presses the “Crack-it” button in the web-interface and the moment the web-page shows the password. We compute the average computing time over multiple runs. Since the server stores computed passwords in the past (to speed up computing time), we measure the time to crack passwords of similar complexity (with respect to the server’s algorithm). Below (left) is the plot of average computing time as a function of the delay between the server and the worker.

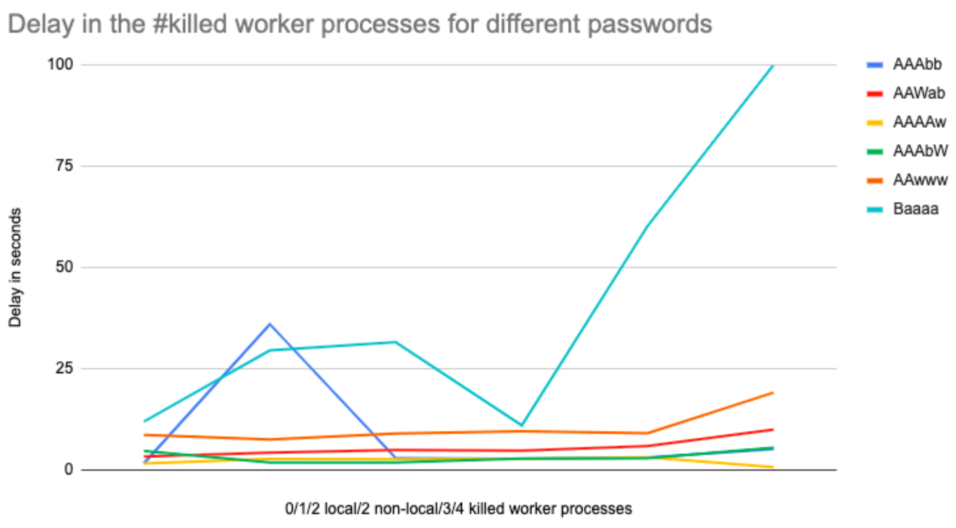




One user, different number of workers, average over different passwords

In this experiment, we query the server on passwords of different complexity, with different numbers of workers (1 through 5), and compute an average cracking delay depending on the number of workers. Above (right) is the plot of average computing time as a function of the number of workers.

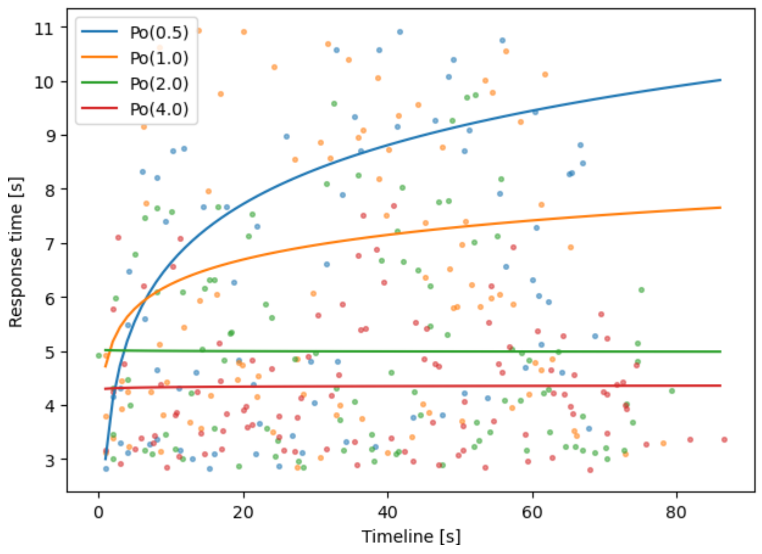
One user and different number failing workers



In this experiment, one user queries the web-interface on different hashes requesting 5 workers, but kills (manually) 1 through 5 different workers. Average delay is shown for each number of killed workers for 6 distinct hashes. Note, that in our GENI instance two workers are located in the same rack as the server, and three are located in another rack (in another state), we also ran the experiment, checking whether killing of a worker in the closer network affects the cracking delay. The resulting delays were almost the same in both experiments for passwords of different complexity, which shows us that network delays are insignificantly small compared to the time required for cracking the password.

Average response time to Poisson arrival streams

We simulate Poisson arrival streams over a timeline of 100s of random hash requests to analyze the average response time per request and to evaluate the response time of our system under different request rates. The experiment was conducted using two workers and a subset of around 1.000.000 passwords chosen to be processed relatively fast on each worker. However, based on our experiments we expect the average response time to scale proportionally on an increased range of passwords and inverse proportionally to the number of workers. For each mean arrival time, we average the values over 5 runs. In the figure to the right, we observe that for a mean gap of 2s, 4s, and 1s, the response time roughly approaches a constant value, while for a mean gap of 0.5s, the response time diverges. Extrapolating the results to the full range of $52^5 \approx 4 \cdot 10^8$ passwords and any number of w workers, we expect an average response time that is about $4 \cdot 10^8 \, / \, 10^6 \cdot 2/w = 800/w$ times higher. In particular, we expect average response times of about $1/w$ hours for mean gaps of 2s and 4s, and $2/w$ hours for a mean gap of 1s.



Analysis

1. The optimal number of workers needed to handle a single user request. From the results of the Plot 1 we get that the average time required for cracking the password

in multiple runs for a request on different numbers of workers is decreasing with the increasing number of workers. So our system behaves appropriately, and extra resources indeed help the workers to solve the problem faster.

2. Increasing the number of users served simultaneously increases the average time for handling one request. Again, that is consistent with the intuition that the more users we serve, the fewer resources we can use for serving each user. In our experiments we also tried to avoid using the same hashes, as the server saves all pairs of previously cracked passwords with their hashes, which could make the delay from the increasing number of simultaneously served users smaller (e.g. when different users requested cracking the same password).
3. Lastly, if we terminate worker processes, then we see significantly increased time for cracking each password. That means that our system recovers, but not very fast after each worker failure.

Conclusions

We implemented a simple distributed system, where each component communicates through sockets, showed that scaling this system improves the performance, and showed that our service is reliable against failures in a network connection and failures in worker machines.

Possible extensions

- Our system supports only one server application on one machine. If the server machine dies, then the password cracking service becomes unreachable. Simple possible extensions are supporting server-processes on multiple machines with selection of a new main server process if the old one becomes unresponsive.
- Currently we do not detect slow execution of one worker, for example: if all workers finish their job and one worker takes much more time, we just wait for this one worker. Alternatively, we could maintain a global average runtime of each worker (that we can recompute, based on the formulas for average RTT we discussed in class), and reassign the task of a slow worker (that runs 10 times longer than on average) to another one.
- Currently, if a connection with a worker becomes flaky, we completely stop communication with this worker, even if, possibly, it almost finished cracking the password. More flexible approach would be to re-establish failed connections with the workers and restart their cracking process from the previously saved state. As we want clients to be stateless, we can communicate intermediate progress of each worker to the server every 10 seconds.

Division of Labor

Ta Duy Nguyen implemented the backend functionality for cracking passwords, experiments, and helped with the report;

Fabian Spaeh implemented the web-application and scripts for deploying the project on GENI, ran experiments, and helped with the report;

Ngu (Nathan) Dang and Ludmila Glinskih wrote this report, ran experiments.