Distributed MD5 Hash Cracker

Tyler Reece, U29363025

GitHub Link: https://github.com/reecetyl/Distributed-MD5-Cracker
Project on GENI and Public Link: Slice name geni-miniproject-tr, public facing IP at tlreece@pcvm1-31.instageni.colorado.edu

I. Introduction/Problem Statement

Distributed processing, in which more than one computer is used to run an application, is a powerful tool that can be used to run a large computation more efficiently. Computer networks make distributed processing even more powerful, as a huge computation can be parceled out to computers all around the world, who work in parallel to solve a problem that would otherwise be intractable for a single computer. In the modern day, distributed processing is becoming ubiquitous, with uses in distributed databases, scientific computing, and Bitcoin mining. This project uses the practice problem of brute-force cracking an MD5 hash to investigate the challenges of implementing a distributed system over a network.

The learning outcomes of the experiment are to understand the challenges associated with implementing architecture that allows for a distributed process to run smoothly and correctly. While the breaking of MD5 hash is an interesting problem in itself, it is simply used as an example problem that can benefit from distributed processing. The real motivation for the project, as it pertains to CS655, is to learn to build a scalable, distributed client-server architecture, where an arbitrary number of clients can connect to work in parallel to brute-force crack the hash. Most importantly, the architecture must be (1) *distributed*, meaning the server and worker nodes must exist on different computers; (2) *scalable*, meaning the system should be able to accommodate multiple worker nodes, even adding nodes during the computation; and (3) *fault-tolerant*, meaning even if some worker nodes fail during the computation, their work can be picked up and finished by other nodes. These three key attributes are desirable in nearly every distributed processing architecture, and are thematic to many concepts learned in CS655.

II. Experimental Methodology

A. Architecture Diagram

The distributed system consists of three main components: (1) the web portal, which is responsible for displaying a user-friendly page for to input the target hash and display the results of the brute-force attack; (2) the web server, which is responsible for receiving the target hash and parceling out computational chunks to worker nodes, and receiving the correct broken hash when the computation is complete; and (3) the worker node(s), which receive a range of possible passwords to brute-force and report their findings back to the web server upon completion. See the diagram below for a visual depiction of the system architecture.

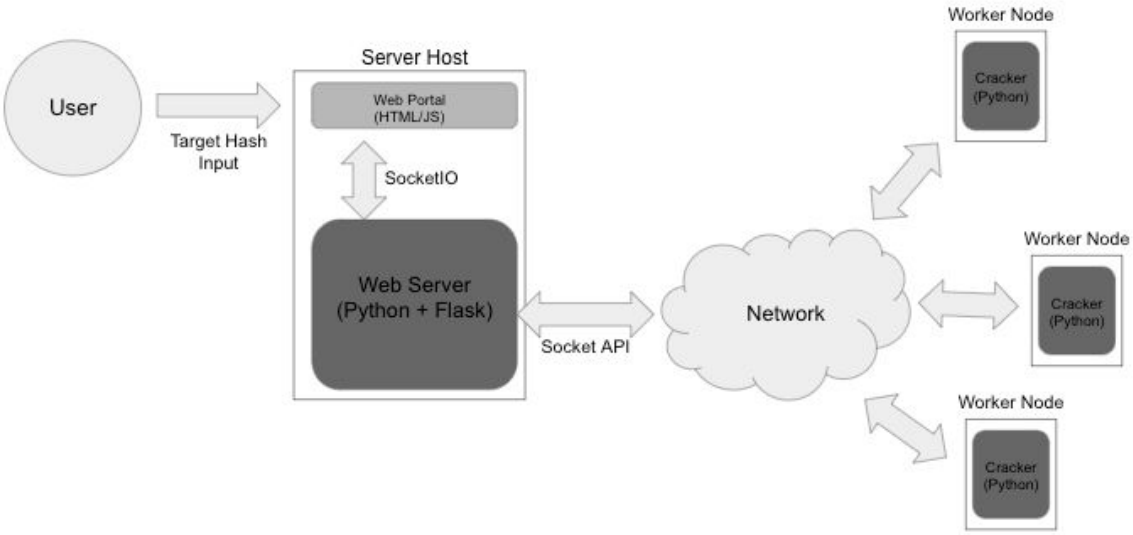


Figure 1: Architecture Diagram

B. Key Assumptions

There are a few key assumptions inherent in this project that are necessary to appropriately scale the workload. While a real-world system would need to address each of these complications, I assume:

1. The MD5 hash inputted into the web portal is made up of a 5 letter password only using capital and lowercase letters (a-z, A-Z). Upon other input, after all possibilities are tried, an error message will be published on the portal.
2. The main server program and web portal will not crash (only simple error handling is implemented). However, worker nodes are free to "crash" (by terminating their processes during computation).
3. The target hash is input into the web portal before worker nodes are activated to start.

III. Results

A. Usage Instructions

The instructions for the usage of the project follow. Please refer to the video demo for a demonstration of the usage. Note: The steps below are for a setup "from scratch". If running directly from the slice provided, everything is already downloaded on the machines. Simply log onto the server node and run `python3 server.py` to start the server. Then complete steps 7-10 to conduct a brute-force attack.

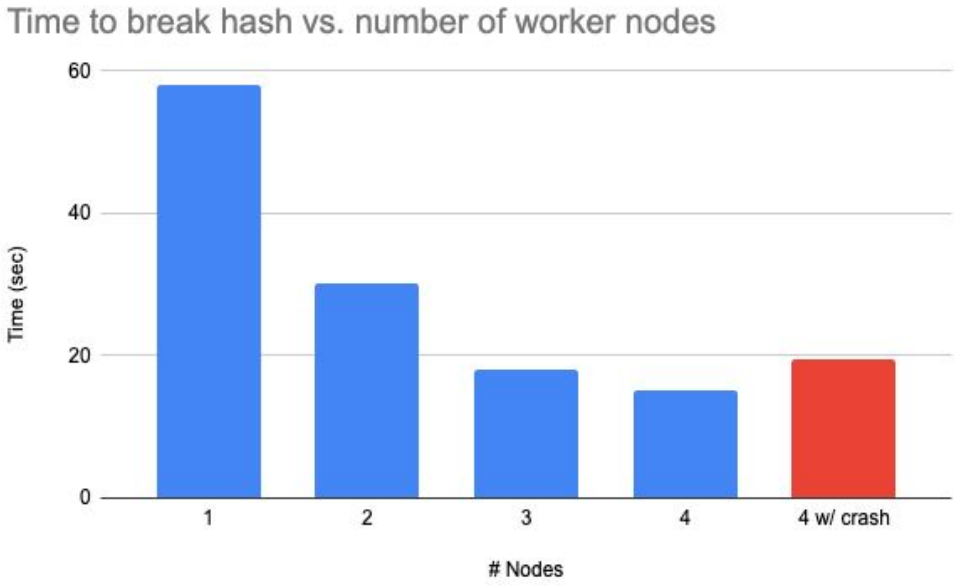
1. Set up GENI resources in accordance with the RSpec included in the GitHub repository. This will include a server and four worker nodes (more can be added if desired).
2. SSH into the server node, and clone the repository via the command `git clone https://github.com/reecetyl/Distributed-MD5-Cracker.git`. Enter the new directory via `cd Distributed-MD5-Cracker`.

3. Run the server setup script via `sh serverSetup.sh`. This will download the necessary Python modules needed for the server, and start the server running.
4. SSH into a worker node. Again, clone the repository via the command `git clone https://github.com/reecetyl/Distributed-MD5-Cracker.git`. Enter the new directory via `cd Distributed-MD5-Cracker`.
5. Run the worker setup script via `sh workerSetup.sh`. This will download the necessary Python modules, as well as creating the dictionary needed in the cracking process. This may take a few minutes. Wait until you see the "done" message printed to continue. The file "dictionary.txt" should appear in the directory.
6. Repeat steps 4 and 5 for each worker node.
7. Navigate to the site of the web portal using a browser. If you use the default setup provided in the GitHub repository, this will be the link `http://192.12.245.165:5000/` . You should see a simple web portal with an input field.
8. Input an MD5 hash of a 5 character passcode into the input field and click the submit button. A "processing hash…" message will appear upon submission.
9. Begin as many worker nodes as you wish by running the command `python3 md5_cracker.py` on each of the worker nodes. You will see the guessed passcodes appear as the worker nodes attempt to brute-force the hash. This may take several minutes, depending on how far down in the dictionary your provided hash is.
10. The web portal will display the broken hash message when a worker node breaks the hash. You can reload the page and repeat steps 7-10 to break additional hashes.

B. Analysis

Using the distributed system that I created, I chose to study the relationship between the number of worker nodes and time to crack a particular hash. My hypothesis was that increased parallelization leads to increased efficiency, in a roughly linear relationship. I chose to use the hash of "afaaa" as the target hash, because one worker alone could break this in about 45 seconds, which was a reasonable time frame without making repeated trials too long, or trials with many worker nodes too short. Of course, this could be generalized to other hashes that are farther down the permutation dictionary, but this is for simple experimentation purposes. I measured the time the nodes took to break the hash and plotted them below. I also included one run where I began four worker nodes breaking the hash, but then aborted a worker after five seconds, to see the effect a worker node crashing had on the process, as well as to prove that worker nodes can be deleted on the fly without interrupting the overall computation. All trials run are showcased in my demonstration video on the GitHub repository. Each trial was run three times and averaged for a final result.

The results of the experiment roughly adhered to my hypothesis: that there is an approximately linear relationship between the number of worker nodes and the time it takes to break the hash. The experiment where I ran four nodes with a crash five seconds in was roughly comparable to the three node trial, which was also to be expected. Again, a demo of each of these experiments is shown in the video.



IV. Conclusion

This project served as a good illustration of the challenges associated with engineering a distributed system. Not only does the algorithm for doing the computation have to be sound, but the demands inherent in distributing computations over a network make the architecture much more complicated. However, for a simple problem such as brute-force attacking an MD5 hash, I was able to create a distributed system to run over a custom network on the GENI testbed. The server is able to run along with the worker nodes to break any 5 letter MD5 hash in a reasonable period of time, and adding worker nodes reduces computation time in a predictable fashion. Finally, the learning goals of developing a distributed, scalable, and fault tolerant system were achieved.

There are several possible extensions I would propose to this project. First, there is the clear extension of increasing the range that the hash cracker works on to include numbers, special characters, and longer passwords to make it more useful in a real world security application. In terms of networking, future work could do additional experimentation on increasing the number of nodes to find the optimum number of workers that balances network usage with increased parallelization capabilities. Finally, it would be interesting to see how effective such a system is in generalizing to more modern, secure hash algorithms.

V. Division of Labor

I chose to do this project solo so I had 100% of the labor.