CS 655

NETWORK SECURITY
MINI PROJECT FINAL REPORT

Team: Palak Jain
Geni Slice: mini-project
Github: https://github.com/thepalakjain/gradnetworks/tree/main/mini-project

Introduction/Problem Statement

My objective is to design a distributed system where a user can submit the md5 hash of a 5-character password(a-z) to a management. The web interface, with the help of worker nodes will then crack the password using a brute force approach. The management system that assigns jobs to the worker nodes uses socket programming to communicate with the server and with the worker nodes. I'd like to analyze the extent to which parallelization between worker nodes speeds up the computation. I ended up modifying the scope of the project from the one in the proposal because I ended up working on the project alone instead of with the two other people in the initial proposal due to time constraints and unexpected personal circumstances.

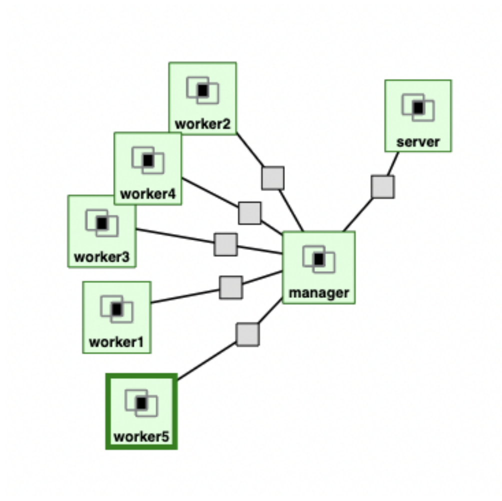
Given a system with many available worker nodes, the question I try to answer is: how many workers should be assigned to a single client to ensure that the expected wait time across clients is below some threshold δ? For this question I assume that the md5 hashes are chosen at random from the space of hashes of 5 character strings with alphabet set (a-z).

More workers always allow for more parallelization thereby reducing the time taken for the server to handle requests. However, in reality there is a cost associated with additional worker nodes and a real system needs to balance these costs with the expected benefits in order to make decisions. For example, the question answered by this project is an interesting albeit rudimentary way a server could make the decision of how many worker nodes it should 'hire' to balance its expected workload. Client wait time is a good proxy for the worker time used each client request because the maximum time spent by the worker nodes on trying to crack a particular password is equal to the client wait time plus the maximum amount of extra time a worker node spends trying to crack the password after it has already been found by a different node. This extra time should be a constant product of the amount of time it takes for the manager to remove unassigned jobs related to the hash once the password has been cracked and we can assume that this value will not change much with the maximum available nodes. Additionally, my management system splits the work of cracking one password into many small 'jobs' to prevent workers from putting in 'too much' wasted effort after a password has already been cracked. It also splits the work up in this way so that when multiple md5 hashes are entered into the system, the jobs corresponding to them get interleaved instead of being carried out one by one. This way, the variance of expected wait time for any particular job scales slightly better with the volume of clients and makes the assessment more relevant to the system if it were to actually be deployed.

I leave out analysis based on other dependencies such as propagation delay and the reliability of the network to keep the length/scope of the project manageable.

Experimental Methodology

Architecture



On GENI, I reserved one node to host the website for the client to use, one node to run the management service that assigns jobs to worker nodes, and 5 worker nodes to do the actual brute force checks for cracking the passwords. The number is 5 is arbitrary and based purely on ease of implementation. Although none of the worker nodes dropped offline during the experiment, the management system was designed to be robust to such an occurrence and can operate with any arbitrary number of connected workers.

Assumptions

In order to conduct the analysis I assume that only one client can interact with the program at a time. In doing this I vacuously end up assuming that the server will not receive multiple requests for the same hash at one time. The manager doesn’t currently handle this case because the job queue identifies jobs based on the hash to which they relate; the manager could be extended to handle this case by storing jobs in association with the hash as well as the time that the request was received. I do not implement this because it is unnecessary for my analysis.

Methodology

First, I conducted a theoretical analysis of what I expect based on the assumption that cracking each password takes the same amount of time. Then I ran an experiment for 1-6 workers, running each set on the hashes of 15 randomly chosen passwords and averaging the individual wait times for the passwords. I then compare my theoretical expectation with the experimental results.

Results

Usage

See the github README for detailed instructions on interacting with the project code. I provide a web interface for a client to provide different input hashes and the management server uses the number of currently connected worker nodes to find the passwords associated with the hashes. I run the web server, the management service, and workers manually through command line prompts. For the experiment, I connect worker nodes one by one, and for each number of workers I query all the hash values used for the experiment one by one. The management node currently returns the found password to the server who prints the password, the time taken to find the password, and the number of workers to be via the command line. It would be interesting to write a program to display the password on the website or to privately transfer the password to the client some other way but I leave out this implementation since it isn't relevant to the experimental analysis. I did the data collection manually using the outputs to the terminal.

Theoretical Analysis

For this calculation I assume that the workers all take exactly the same amount of time to run through a 'job', where a job is to check all passwords with the first two characters fixed to a provided value. I also assume that the manager takes no time to provide the workers with a new job as soon as they are done with the last one. This means that all n workers get assigned a job each at once and they all get done at once and so on till the password is found.

Let each set of assignments to all n nodes be called a 'round'. Let p_i be the probability that the job is found in round i and let r be the number of rounds. The probability that the password will be found in a round is the same for all the rounds. Let T_n be the RV for expected time taken to crack the password with n workers. Let t be the time taken for one round.

Number of jobs = $26^2 = 676$ (since a job fixes two characters and there are 26 possible chars)

$r = \frac{676}{n}$

for all rounds i, j, $Pr[p_i] = Pr[p_j] = \frac{1}{r} = \frac{n}{676}$.

$$
\begin{aligned}
E[T_n] &= (p_1 \times t) + (p_2 \times 2t) + \dots + (p_{676/n} \times \tfrac{676}{n} t) \\
&= \frac{n}{676}(t + 2t + \dots + \frac{676}{n} t) \\
&= \frac{t(676/n + 1)}{2}
\end{aligned}
$$

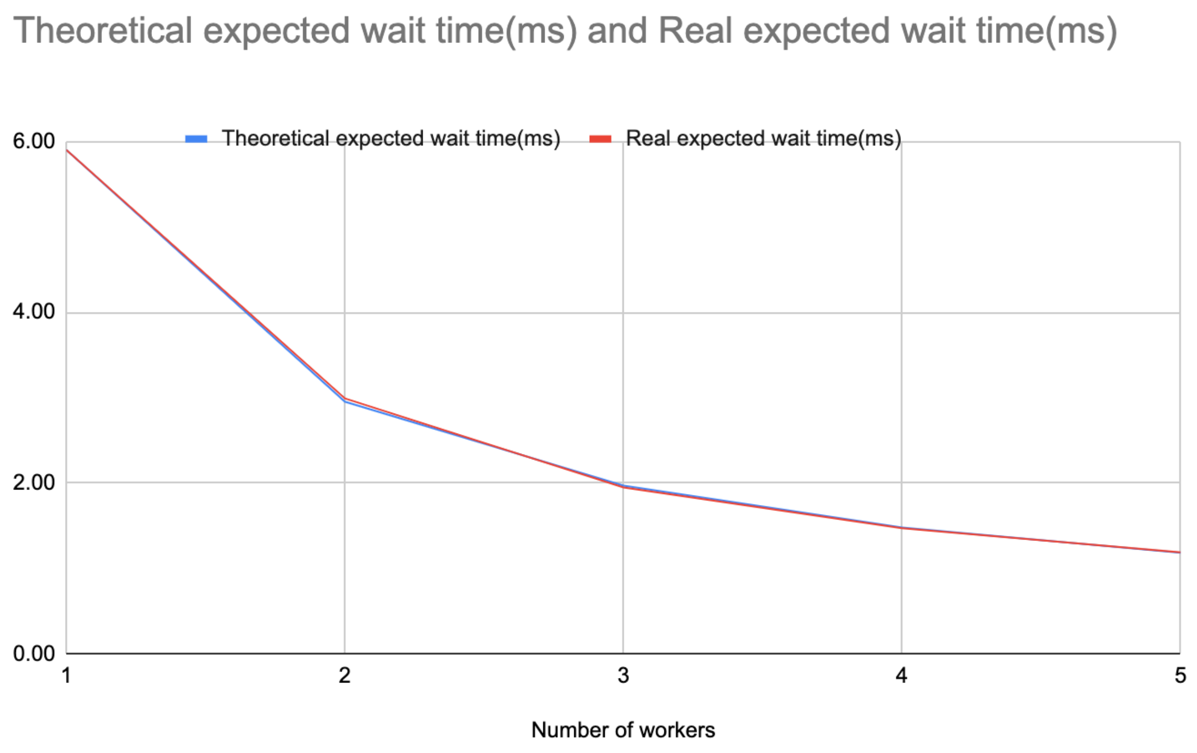
In particular, for one worker:

$$E[T_1] = \frac{t(676+1)}{2} = 338.5t$$

So, we will measure the average time taken for one worker to crack the password and use it to approximate t. We will then use this value to predict the avg time taken for 2,3,4,5, and 6 workers and then compare these to our experimental values.

Experimental Analysis

I measured the average wait times as described in the methodology section for 1-5 worker nodes. The graph of theoretical values vs experimental values below shows that the experimental results were pretty much exactly as we expected.



Conclusion and Possible Extensions

The experimental results I obtained were exactly in line with the expected values! We can see the diminishing returns for additional worker nodes from the graph in the results section. Since the application is multithreaded and can deal with several clients at once, one could analyse the wait times when requests are made in parallel by writing some automatic application for making requests from the server. However, I leave this out for ease of implementation. One could also further extend the analysis by analysing the behaviour of the system based on different volumes and frequencies of client requests. Finally, it may be interesting to run these experiments with even more worker nodes to better understand the diminishing returns of additional worker nodes.