COMPUTER SCIENCE 213 FINAL PROJECT

DISTRIBUTED PASSWORD-CRACKER

May 15, 2018

Seth Ruiz
Tapiwanashe Zvidzwa
Myles Becker

Grinnell College
Department of Computer Science

Contents

OVERVIEW

In setting out to create our distributed password-cracker, our goal was to develop a system that allowed for multi-threaded MD5 hash comparisons across several computers in a network. To do this, we sought to establish a two-tier client-server architecture, leaving all of the distribution calculations to the main server, while the individual client nodes took care of the hash comparisons. Ultimately, we were successful in our endeavor, tackling the system on one end from the cracker perspective, and on the distributed network from another.

The cracker is fairly similar to its counterpart from lab five, with some considerable alterations, and a novel system for easing communication of ranges and dividing up said ranges in order to more efficiently parse through the range of passwords.

The distributed network is again, fairly similar to its counterpart in lab 9 with some minor and major alterations. From an architectural standpoint, the system utilizes a single server to which multiple clients connect, distinguishing itself from the distributed network we implemented in the lab. This allows the server to maintain the slicing of the potential hash ranges without the worry of having to update each node with the predetermined intervals, thus eliminating a lot of the calculations as distributed network would require and increasing the speed of the program. At a lower level, once the slices has been determined, the server uses a thread for each client to pass along a packet containing the start and end ranges for their hash comparisons, and waits to hear back once they've been completed. The client then receives the range, computes the hashes, and replies to the server with either a request for another slice, or the password, at which point the server will either offer another slice or alert the client that the password has been found.

Evaluating our implementation is fairly straightforward, as our tests proved the system's efficacy self-evident. We would supply the system with the MD5 encoding of the string 'zzzzzzz', which would require the cracker to iterate through all possible passwords, we would have a metric for observing the speed of the system and the benefit of multiple clients. We anticipate a relatively linear trend in time reduction, however the main advantage is that the system can have a potentially infinite number of clients as it operates on a linked list for client addition. Sheer number can make quick work of massively complex problems.

Lastly, we outline some of the improvements and alterations that could be made to

our system given more time and attention. We believe that our design and implementation accomplish our original goal, though in our pursuit we considered several design improvements that could make for a more verbose and developed system.

DESIGN & IMPLEMENTATION

For readability purposes, our design section has been split into two portions, the password-cracker itself and the network architecture it runs upon. Thought both are integral to the system, they were developed independently and nearly operate as such, making it easier to distinguish the design decisions and implementation choices that went into their formation.

PASSWORD-CRACKER

The password cracker is a more streamlined version of the cracker implemented in lab 5. This cracker does not need to read from a file a parse through a series of user names and hashes, rather, the cracker needs only 3 inputs: a hash represented as a string, a (double) starting number, and an (double) ending number. These are parsed into a series of functions, converting the string to a series of bytes of type uint8-t. First a thread array is created, as well as an array of structs used for storing starting and ending numbers, hashes, passwords, etc. The starting and ending number are used to make a series of threads within each client, further parallelizing the workload. The range of starting and ending numbers is divided up, giving each thread its own starting and ending number.

Each thread, with its own starting and ending number, iterates through that range, converting each number into a string via a novel base 26 conversion function. This string is then hashed with the MD5 function and then compared to the original hash. Should they be the same, the found password is saved into a field in the struct array. After all threads have finished searching and have joined, the program goes through the thread array, checking to see if a password had been saved into one of the structs in the array. If this is the case, then the client sends back the password to the server, otherwise it sends back "NOPE".

DISTRIBUTED NETWORK

As is outlined in the Overview, the network portion of the system is built upon a two-tier client-server architecture. This means that the server and the client operate as two distinct

entities, communicating with each other over the network, but ultimately two separate C files. The server accounts for one of these, charged with the task of determining the slicing of the 8,000,000,000 different potential hashes containing 7 lowercase letters. It accomplishes this by first establishing a port and opening a thread dedicated to establishing incoming connections, creating a thread for each individual client to communicate over.

It then uses this channel to pass a packet containing the starting and ending points for this specific range of slices, an array, the hash we are comparing against, and an instruction to the client. The client then switches on the instruction, either 'KEEP_LOOKING' or 'PASSWORD_FOUND', and hashes through the range and compares to our given hash or exits accordingly. If the client locates the password as it hashes through its predetermined range, it updates the instructional enum, copies the password to the array, and writes the packet over the socket and back to the server.

If the client doesn't find the password in its range, it replaces the enum with a request for more and sends it off the to server. In the former case, the server notes that the password has been found, reports that information to every client in the enum, enters a 15 seconds grace period to allow time for the clients to touch base, and promptly exits. In the latter, it merely repeats the process, allocating the client a new slice to run through and updating the enum accordingly.

There is a final case to consider about ending the search, that of multiple clients attempting to access the final range slice. If a global counter exceeds the number of available slices, any remaining requesting clients will be told that the search space has been exhausted and they will disconnect, to avoid clients making requests unnecessarily.

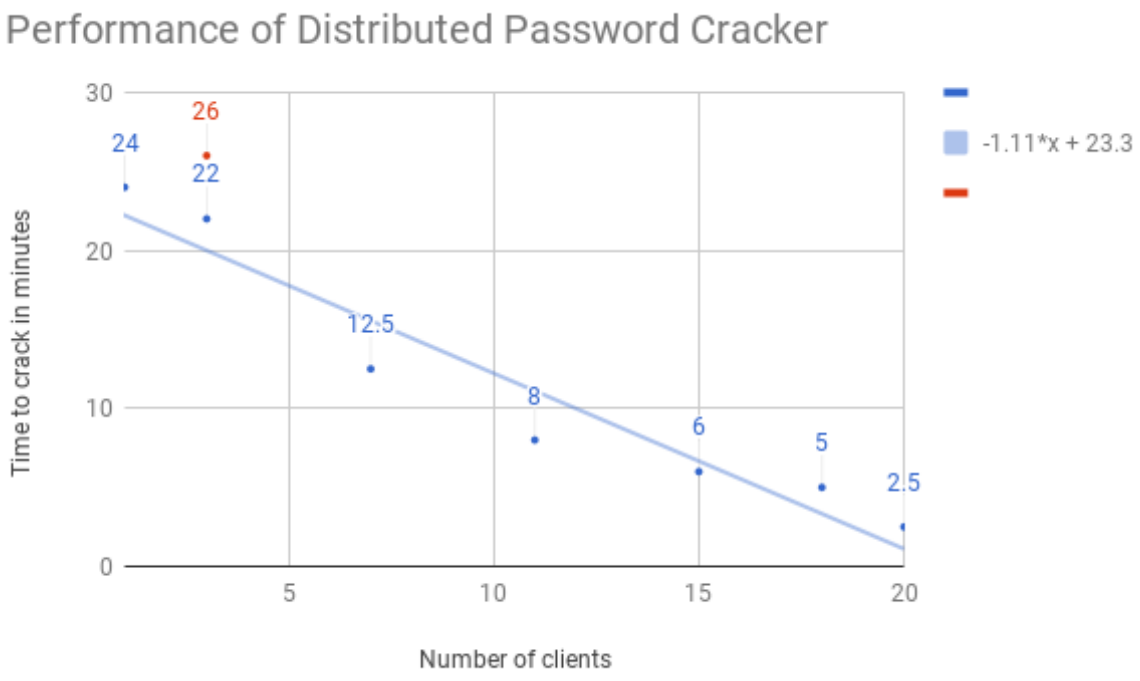
EVALUATION

To evaluate the performance of the distributed password-cracker we executed a series of experiments to get a time benchmark. There was always a 100% cracking rate for any password that fit the parameters of 7 characters, lowercase English letters. The tests and their results are as follows:

20 clients (each client ran alone, without any other applications open on a different machine) took 2.5 minutes
18 clients (3 on the same machine as the server, 4 on each of 3 other machines, 3 on a final separate machine) took 5 minutes
15 clients (3 on the same machine as the server, 4 on each of 3 other machines) 6 took minutes

11 clients (3 on the same machine as the server, 4 on another machine, 4 on a final machine) took 8 minutes
7 clients (3 clients on one machine and 4 clients on the other; like previously, no other applications were open) took 12.5 minutes
3 clients (all on same machine) took 22 minutes
3 clients (one machine held the server alone and another machine had all 3 clients running without any other applications) took 26 minutes
1 client took 24 minutes

Provided is a graphical representation of the results:



One thing that was notable was that the 1 client was nearly as efficient at parsing through the available passwords as the 3 clients running on the same machine, and more efficient than 3 clients running on a different machine. The 3 clients running on a machine separate from the server was slower than the 3 clients running on the same machine as the server, which is understandable with the large amount of information what would have to be communicated across the network rather than locally on the same machine. Burdening each machine with a multitude of clients greatly detracts from performance and so the most optimal method is to have as many machines running as few clients on each machine as

possible.Our hypothesis was mostly supported by the evidence, though a potential place for further measurement would be comparing workloads of individual clients against machines with multiple clients.

Provided are the specifications about the machines used:

Architecture: x86-64 CPU op-mode(s): 32-bit, 64-bit Byte Order: Little Endian CPU(s): 4 On-line CPU(s) list: 0-3 Thread(s) per core: 1 Core(s) per socket: 4 Socket(s): 1 NUMA node(s): 1 Vendor ID: GenuineIntel CPU family: 6 Model: 42 Model name: Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz Stepping: 7 CPU MHz: 1599.847 CPU max MHz: 3700.0000 CPU min MHz: 1600.0000 BogoMIPS: 6584.79 Virtualization: VT-x L1d cache: 32K L1i cache: 32K L2 cache: 256K L3 cache: 6144K NUMA node0 CPU(s): 0-3

Each test was conducted on the same hardware.

FUTURE WORK

A more dynamic allocation process is a potential next step for this system. This would involve a peer-to-peer network architecture rather than exclusively client-to-server. A client could then be allocated a broader range of password numbers and then operate as its own distribution center to other clients that would join to it. However the fluidity of this process is quite difficult and complex as there are several different approaches that could be implemented including many possibilities for suboptimal distribution. It would also require constant checking of the entire network in order to properly balance distribution, and the information would have to potentially pass through several clients until it reaches its destination. There is also no security that this process would be more efficient that client-to-server.

The system could also be tasked with handling not just lowercase, alphabetical passwords of length seven, but different cases, as well as integers. This would require a reworking of the string to double conversion with a much wider mapping function, however it would be a more feasible next step over dynamic allocation. This would require a large amount of time and several clients to be practical. Another addition would be the use of several different hash functions, not just MD5, as with true password cracking, one does not often know which hash function was used. This would be another condition added in the comparison function where a potential password is checked against the hash. Having a different hash function would also require the string to byte conversion to cover a wider range of possibilities. There is also the potential to give the cracker a file full of hashes for it to

crack. The cracker in its current stage is a great base for configuration and expansion, but should be handled with ethical consideration.
The code is accessible at :
https://github.com/SETHRUIZ/password-network

REFERENCES

Curtsinger, Charlie, 2018. Lab: Distributed Systems, *CSC-213: Operating Systems & Parallel Algorithms,* [lab] Available at:<http://www.cs.grinnell.edu/~curtsinger/teaching/2018S/CSC213/labs/distributed-systems.html> [Accessed 24 April 2018]

Curtsinger, Charlie, 2018. Lab: Password Cracker, *CSC-213: Operating Systems & Parallel Algorithms,* [lab] Available at:<http://www.cs.grinnell.edu/~curtsinger/teaching/2018S/CSC213/labs/password-cracker.html> [Accessed 24 April 2018]