Network-Centric Computing - CS 382

Assignment 3: Distributed Password Cracker

Hassan Shahid Khan, Wajih Ul Hassan, Gohar Irfan Chaudhry

Due Date: 9th May 2015

Submission Instructions:

- Please use the following naming convention: <Roll#>_Assignment3.zip
- **We will NOT accept any submissions via e-mail.** There will an LMS tab for late-submissions as well. There are no exceptions to this rule.
- Be prepared to justify the code you have submitted in vivas. We take plagiarism concerns very seriously in this course.
- **This is an individual assignment.**
- Please direct all questions related to the assignment to the discussion forum at Piazza.
- Late submissions will be penalized 10% per day.

Introduction:

For a few decades, UNIX-based systems (and now OS X and windows) all store the hash of a user's password alone. When you type your password to login, the system hashes the password you type and compares it against the stored hash value. In this way, an attacker who takes over a computer and steals its /etc/passwd file doesn't immediately learn the clear text passwords belonging to the system's users. A cryptographic hash function is a hopefully one-way transformation from an input string to an output string, where it's impossible to go backwards from the output to an input that will generate the output:

Hash(***cleartext***) --> hash value

This system is a great start, but it's not unbreakable. The original, and still somewhat commonly used, way of hashing passwords uses a hash function called ***crypt()***. When it was initially designed in 1976, crypt() could only hash four passwords per second. With today's (drastically) faster computers, incredibly optimized password cracking code can attempt over three million hashes per second on a single core. While the hash function is still one-way, you can find a lot of passwords using a brute-force approach: generate a string, hash it, and see if the result matches the hash of the password you're trying to find. Still, three million per second isn't all that fast. Using just the upper, lowercase, and numeric characters, there are 628 possible eight character passwords. That would take 842 days to crack using a single core. But why stop there? There must be a few hundred idle cores sitting around campus (On Rustam perhaps?)

Concepts

In first assignment you will create a distributed password cracker. You will use the concepts of concurrency, threading, and client/server communication protocols to solve this task.
The goal of this assignment is to create a distributed system that can run across different machines. Password cracking is an "embarrassingly parallel" application—it consists of a set of expensive operations on small chunks of data, and there's no data that needs to be shared between different password cracking nodes as they crack. They just receive a work unit allocation, try all of the passwords in that unit, and tell the server if any of them was a match. The challenges in this assignment are twofold. First, the system is designed to run on the wide-area network. The Internet is not a nice place: it can drop, mutilate, or delay your packets. Your remote worker nodes may become unavailable temporarily or permanently. You'll have to deal with packet loss and node and communication failures. Second, since you're harvesting the work of a bunch of "volunteer" computers, the nodes may run at widely disparate speeds. Some may be high-end servers; others may be cheap notebooks or ten-year-old desktops. You'll have to make sure that your server properly balances the load across these nodes so that everybody's busy and you may not accidentally allocate 10 years of work to an ancient 486!

To know workers speed server would have to send ping command to workers. In response to this command worker will inform server about its existence plus the number of computations worker have performed. By this server can maintain the computation power of its workers and hence it can distribute work for next client accordingly.

Design and Implementation

The cracker consists of three programs:

- Worker client
- Request client
- Server

A request_client sends a password cracking job to the server, and the server is responsible for dividing the job into parts and allocating those parts to worker_clients. The clients and server communicate with each other using the sockets API to send UDP packets. The protocol they must implement is detailed in the following sections; the clients and server must be able to interoperate with other clients and servers that correctly implement the protocol. The server must be robust to communication failures (clients never receive a message) and to client failures by eventually assigning uncompleted jobs to other clients.

A note for Java: Use MD5 or SHA1 hashes

All messages are to be transmitted using UDP

Server

The server should recognize the two different clients as they contact it. It should keep track of the clients and which jobs they're currently working on or what job they've requested that the server solve. It should handle timeouts. You must use select to do so. Using select provides an efficient and clean way to handle timeouts (please look up the select UNIX command). The server may allocate jobs as it sees fit. It is your job to ensure that the jobs are of reasonable size and that the load is balanced across the worker_clients. The server should do this by having the number of workers assigned to each request be as equal as possible, without pre-empting workers. For example, there are 10 workers, which are all working on one request, and a new request comes in. As the workers finish their jobs, they are assigned to work on the new request, until each request is being handled by 5 workers. Your code should produce an executable file named ”server” and should use the following command line arguments:

```
./server <port>
```

Example:

```
./server 2222
```

For ensuring that the port number of the server process is unique for each student, please use the following formula to compute the port number of the server process. Port = 10000 + (last four digits of your registration number) *20

For example if you registration number is 10080013, the port number that you should try first is: Port = 10000 + (0013)*20=10000+2600=10260.

In case this port number is already used, try the next port number (10261), until you found an unused port.

Worker_client

The worker_client should be multi-threaded to facilitate communication with the server. One thread should be used to do the actual work of the client (the cracking) and a second thread should be used to listen for messages from the server. The client should be independent and should be able to find out everything it needs to know through the protocol defined below. The client should not remember jobs it’s worked on in the past. Your code should produce an executable file named “worker_client” and should use the following command line arguments:

```
./worker_client <server hostname> <port>
```

Example:

```
./worker_client compute-0-2 10260
```

Request_client

The request_client does not need to be multi-threaded, but it certainly can be if you like. Its job is to request that a password be cracked by sending a hash to the server. It is also required to ping the server every 5 seconds to remind the server of its existence. Since we do not know how long it will take the server to find the answer, this approach is more feasible than defining an arbitrary timeout value. Your code should produce an executable file named “request_client” and should use the following command line arguments:

```
./request_client <server hostname> <port> <hash>
```

Example:

```
./request_client compute-0-3 10260 aaHLWHfiLg
```

Note: Do NOT run your processes on head rustam node (which you use to login).

Protocol

For this assignment, you will implement a comma separated protocol. The various parts of the message will be aligned on specified bytes, such that there is less variation between messages:
`Magic, Client_ID, Command, Key_Range_Start, Key_Range_End, Hash`

HASH is a string of length less than or equal to 32 bytes.
Data Types
Magic: unsigned int (4 bytes)
Client_ID: unsigned int (4 bytes)
Command: All commands specified in the next section are represented as number. In protocol command number is an unsigned int (4 bytes)
Key_Range_Start: Char[6]
Key_Range_End: Char[6]
HASH: Char[33] (HASH value can have at most 32 characters)
MAGIC refers to a magic number, so that we are able to identify whether a message is relevant to our application. Our MAGIC number is 15440.
The **Client_ID of worker_clients** is the ID of the client randomly generated by the server when the worker sends a REQUEST TO JOIN. When the worker sends a REQUEST TO JOIN, it sets the client id field to 0. When the server responds (with a JOB), the server sets the client id field to a randomly generated number. The worker_client should use that number from that point on.
The **Client_ID of request_clients** is the ID of the client randomly generated by the server when the request_client sends a HASH. When a request_client sends a HASH, it sets the client_ID field to 0. When the server responds with an ACK JOB, the server sets the client ID field to a randomly generated number. The command is the command that is being sent. Remember that any numbers that you send that are longer than 1 byte must be in network byte order.

Commands

The following list of commands does explain what to do in every possible situation. In cases not covered by the following list or elsewhere in this document, do what you think would work best and would make the most sense.
REQUEST_TO_JOIN: The worker_client starts the relationship with the server by sending this command. There is no other info for this message.
Command Code: 1

JOB: When the server wants to assign a job to a worker_client, it sends this command, along with the range and the hash. For example: XXXX YYYY HHHHHHHHHHHHH” where the XXXXs represents the value of Key_Range_Start and YYYY is the value of Key_Range_End, and the client needs to check all the values in the range [XXXX, YYYY] including the end points. The Hs is the hash of the password.
For example:

JOB aaaa 9999 aas4dfBX3efadfe
Command Code: 2

ACK_JOB: When the worker_client receives a JOB, it should acknowledge it with an ACK JOB with these contents: “XXXX YYYY HHHHH…” where the Xs, Ys and Hs are the same as discussed above. The server should also use this command to communicate to the request_client that it received the request. The other info field does not need to contain anything, but the client id field should be filled in (by the server).
Command Code: 3

PING: Used for request_clients to ping the server, and for the server to ping worker_clients.
Command Code: 0

DONE_NOT_FOUND: When the worker_client completes its job and has not found the password, it sends a DONE_NOT_FOUND with parameters: “XXXX YYYY”. If the server is not able to find the password, it sends the request_client a DONE_NOT_FOUND with : HHHHH… (only hash values are included in the message)
Command Code: 4

DONE_FOUND: When the worker_client completes its job and has found the password or the server wants to tell a request_client the password, it sends a DONE_FOUND “PPPP HHHHH” where PPPP is the password and HHHHH is the hash.
“PPPP” is included in the Key_Range_Start field and the Key_Range_End field is empty.
Command Code: 5

NOT_DONE: When a worker_client has received a PING, but is not done processing its job, it sends this command to the server with parameters: XXXX ZZZZ HHHHHHH…,where
“XXXX” is the start of the range the server assigned and ZZZZ is the latest value the client has tried with the hash “crypt()” function. This means that the client has so far checked the range [XXXX, ZZZZ], but didn’t find a match. The reason for providing the latest value “ZZZZ” is that if the client fails and the job is assigned to a new client by the server, the new client does not start the job from the beginning of the range, rather it tarts from the last “ZZZZ” value received by the server from the old client before its failure.
Command Code: 6

CANCEL_JOB: This command is explained in the following two scenarios.
Command Code: 7
<u>Scenario 1</u>
1. Request client sends CANCEL_JOB command to the server along with its Client_ID.
2. The server forwards the CANCEL_JOB command to all the worker clients working on the job associated with the given Request client.
3. Upon receiving the CANCEL_JOB command each worker client cancels the current job and wait for the new request.

Scenario 2

1. A worker client founds the password and sends the DONE_FOUND message to the server.
2. The server after sending the password to the request client sends CANCEL_JOB command to all remaining worker clients working on that job to stop working and wait for new request.
3. Upon receiving the CANCEL_JOB command each worker client cancels the current job and wait for the new request.

HASH: When a request_client wishes to make a request, it sends this command to the server with parameters: “HHHHH”.
Command Code: 8

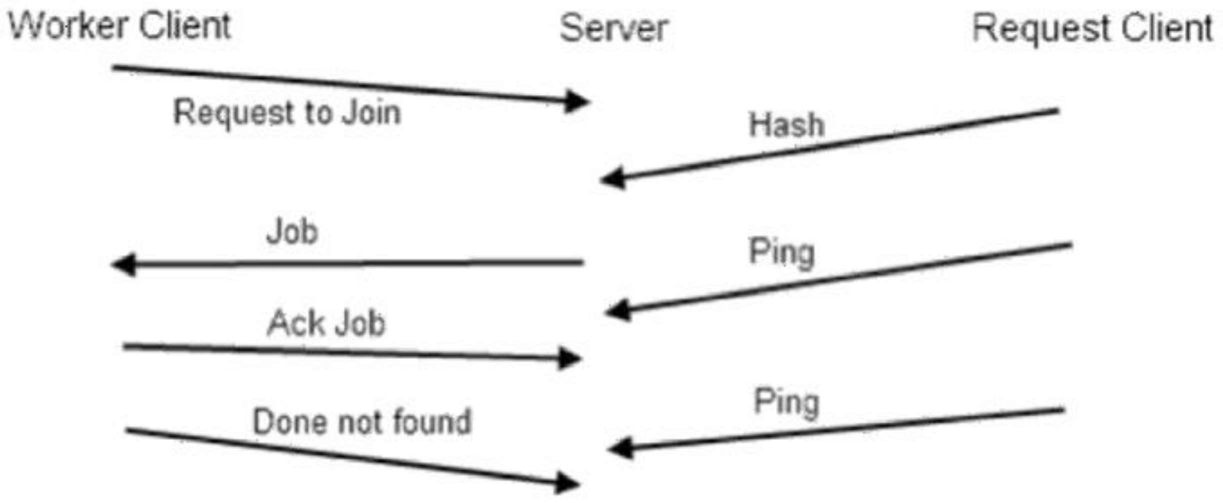

Figure 0. Assume the central server never fails.

Failure

If a worker client fails to acknowledge the receipt of a job (times out), the server should attempt to send the job twice more. Once it has tried 3 times and the client has not acknowledged the job, the server should remember the job and assign it to the next available client and forget all about the previous client. If the worker client fails to send a message, once it has completed the job (times out), the server should send a PING to the client. The worker should respond with its previous DONE message (whichever variant it might have been). If the worker is currently processing a job, it should respond with NOT_DONE. If the server receives NOT_DONE response, it should reset the number of pings it sent to the worker, and wait for timeout again. If the PING times out, the server should try twice more and then the reassign the job to another worker client and forget about the previous client. The request client is responsible for sending a PING message to server every 5 seconds. The server should monitor these pings and if it doesn’t receive a ping within 15 seconds, it should abort that job. The timeout for all worker client/server messages is 3 seconds. The means that server after assigning a job to a worker client should wait for 3 seconds to for ACK_JOB. If the server doesn’t get the ACK_JOB within 3 seconds, the client has died out and server should send the job to next available client. Once the server receives the ACK JOB, it should wait for 3 seconds to send the PING

If the server fails, all state should be saved to a file, and when restarted it should be able to resume operation without any problems.

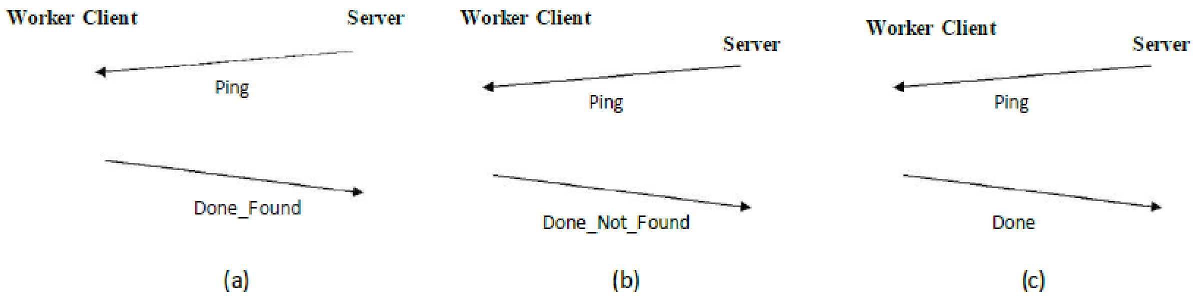

Figure 1. Worker_client's response to server's ping when: (a) the worker_client has completed processing and has found the password; (b) the worker_client has completed processing and has not found the password; (c) the worker_client has not completed processing.

Password length and Salt

For this assignment, the passwords are **ALWAYS** 5 characters in length. The characters will be alphanumeric with no special characters such as $, %, &. Moreover, the passwords are assumed to be case-sensitive.

Ordering: Assume the following ordering between characters when searching for passwords.

[a-z] < [A-Z] < [0-9].

Other Notes

A significant percentage of the grade of this assignment will be given based on efficiency of the protocol. Optimizations will be given extra credit.