Lab 1: Distributed Password Cracker

Due: 11:59 pm, February 10, 2011

1 Introduction

For a few decades, UNIX-based systems (and now OS X and windows) all store only the hash of a user's password. 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 cleartext passwords belonging to the system's users. Recall that 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:

$$Hash(cleartext) \rightarrow hashvalue$$

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 62^8 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...

Many hands make light work.
–John Heywood

In this lab you will create a distributed password cracker. You will use the concepts of concurrency, threading, and client/server communication protocols to solve this task.

1.1 Concepts

The goal of this lab is to create a distributed system that can run across the entire Internet. Password cracking is an "embarassingly 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, multilate, 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 netbooks 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, but so that you don't accidentally allocate 10 years of work to an ancient 486!

2 Lab Requirements

In this lab, you will implement a distributed password cracker. The cracker consists of three programs, the worker client, the request client and the 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.

The majority of the grade for this lab will be determined by the included grading scripts. Because of this, your server and clients *must* be compatible with our sample binaries. Luckily, the sample binaries are included for you to test against, along with some testing scripts. In order to encourage you to do your own testing, there will be some hidden tests used by the staff to grade. It is your responsibility to ensure that your code is well-tested.

2.1 Implementation

You must write your server and clients in C++. We highly recommend that you take advantage of the C++ Standard Template Library, which provides data structures such as maps, lists, and queues, so that you don't have to reimplement the wheel. If you have no experience with C++, you may find the resources in Section 7 useful. Because of the similarity between C and C++, you should be able to write most of the code in C and just throw in the data structures from C++ that you wish to use.

20% of the project grade is for code quality and style. For consistency, we require that projects follow the major aspects of the Google C++ Coding Standards:
`Http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`

We recommend *against* using C++ features such as exceptions, operator overloading, deep levels of inheritance, and multiple inheritance. With systems code, we urge: Keep it simple! See the *Software Engineering for Systems Hackers* notes for more thoughts on this matter.

2.2 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.

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

2.2.1 Using `crypt()`

The `crypt()` function has a great man page (man 3 crypt). It takes two arguments, the key and a “salt”:

```
char *crypt(const char *key, const char *salt)
```

The key is the password you want to encrypt. The salt is a constant string that can be used to produce different encrypted versions of the same key. That is:

```
crypt("hello", "aa") -> "aaPwJ9XL9Y99E"
crypt("hello", "ab") -> "abl0JrMf6tlhw"
```

You’ll note that the salt appears in the hash value. The purpose of the salt is to deter pre-computed dictionary attacks in which an attacker, in advance, computes the hash of tons and tons of strings and then simply looks up the result.

For this assignment, you’ll always use the salt “aa”.

Note For The Curious: Crypt is *not* a recommended hash function to use today for most applications. Many implementations of crypt have a deadly flaw when used outside of simple unix passwords: they will discard characters after the 8th! Those curious about the use of hash functions for applications such as Web security should take a look at the paper “Dos and Don’ts of Client Authentication on the Web”, which discusses these vulnerabilities and best practices in more detail. For more general cryptographic hashing, consult the Web—there are several alternatives depending on your security and speed needs and the way you want to use the hash function. But don’t use crypt!

2.3 Clients

2.3.1 Worker Client Architecture

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 unix34.andrew.cmu.edu 2222

2.3.2 Request Client Architecture

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 unix34.andrew.cmu.edu 2222 aaHLWHfiLg

2.4 Protocol

For this assignment, you will implement a binary format protocol. The various parts of the message will be aligned on specified bytes, such that there is less variation between messages:

```
+-------+-------+-------+-------+
|MAGIC                          |
+-------+-------+-------+-------+
|Version|Command| Client ID     |
+-------+-------+-------+-------+
|Other Info (any length)        |
+-------+-------+-------+-------+
|Other Info (continued)         |
+-------+-------+-------+-------+
```

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. Version is 1. 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. A list of these commands can be found in msg.h. You must use that enumeration, or your code will not pass our tests. Other info should be a NULL-terminated string and can be any length.

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.
- JOB: When the server wants to assign a job to a worker client, it sends this command, along with the range and the hash in the other info field: “XXXXX XXXXX HHHHH” where the XXXXXs represent the range the client is to check. The HHHHH is the hash of the password.
  For example:
  JOB AAAAA ABAAA abs4dfBX3
- ACK_JOB: When the worker client receives a JOB, it should acknowledge it with an ACK_JOB with these contents in the other info field:“XXXXX XXXXX HHHHH”
  where the XXXXXs and HHHHHs are the same as before. 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).
- PING: Used for request clients to ping the server, and for the server to ping worker clients.

- DONE_NOT_FOUND: When the worker client completes its job and has not found the password, it sends a DONE_NOT_FOUND with other info:"XXXXX XXXXX"
  If the server is not able to find the password, it sends the request client a DONE_NOT_FOUND with other info: "HHHHH".

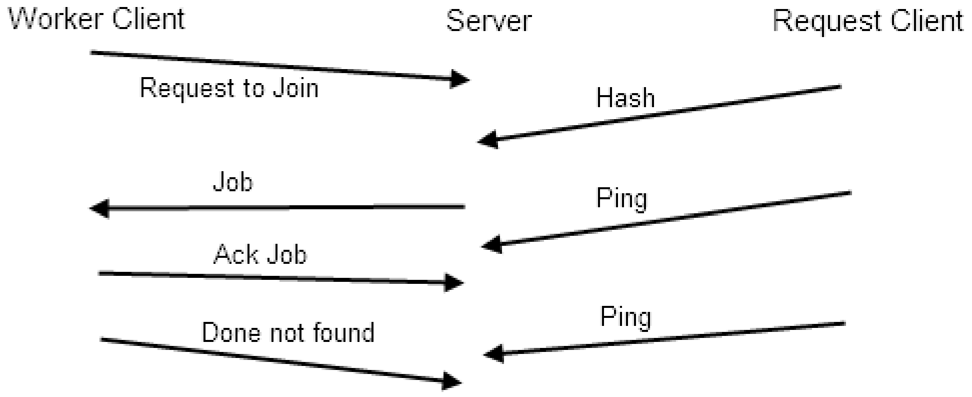
- DONE_FOUND: When the worker client completes its job and has found the password or the server wants to tell a request client the password, it sends a
  DONE_FOUND "PPPP HHHHH"
  where PPPP is the password and HHHHH is the hash.

- NOT_DONE: When a worker client has received a PING, but is not done processing its job, it sends this command to the server.

- SHUTDOWN: When the server is asked to terminate, it sends every worker client this command. There is no other info for this message.
  The worker clients must terminate.

- HASH: When a request client wishes to make a request, it sends this command to the server with other info: "HHHHH".

This protocol is better represented using a diagram:



You will notice that the protocol is centered around the server. We assume that the server will not fail. That is why the server does not need to respond to pings from request clients.

2.5 Failure

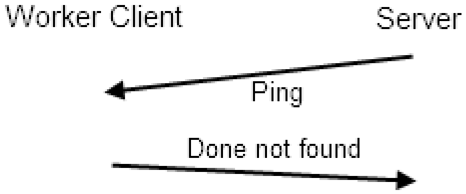
If a worker client fails to acknowledge the receipt of a job (times out), the server should attempt to send the job twice more. Once it has tried 3 times and the client has not acknowledged the job, the server should remember that job and assign it to the next available client and forget all about the previous client.

If the worker client fails to send a message once it's completed a 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 a server receives the 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 reassign the job to another worker client and forget about this previous client.

The request client is responsible for sending a PING message to the server every 5 seconds. The server should monitor these pings and if it does not receive a PING from a request client in 15 seconds, the server should assume that the request client died, and remove its request from the queue.

The timeout for all worker client/server messages is 3 seconds. This means that the server should assign a job, and wait up to 3 seconds for the ACK_JOB. If it does not receive an ACK_JOB within that time period, the client has timed out and the server should try to send the job again. Once the server receives the ACK_JOB, it should wait 3 seconds before sending a PING.

To better test failure, we've provided `dropper.cpp`. To use it, simply include "dropper.h" in your client and server. You can set the LOSSY environment variable to reflect the degree to which you want to simulate a lossy network. Your code should be able to function up to a LOSSY value of 10.



2.6 Provided Files

The following files are provided in the course handout directory, /afs/andrew/course/15/440-sp11/handout/lab1/ :

- `msg.h` - contains some necessary constants
- `cracker_checker.sh` - checks the correctness of the cracker
- `dropper.cpp` - aids in testing packet drops
- `server_tester.rb` - tests the server
- `worker_client_tester.rb` - tests the client

3 Deadlines

Rome was not built in one day.
– John Heywood

3.1 Part 1: Local password cracking

This part is optional, but you should do it if you are unsure about how to make progress for this lab, or if you would like to make sure that parsing and cracking work before attempting the implementation of the protocol.

Implement a test client that just runs from the command line. The executable should be called `cracker`. It will accept a range of characters and a hash value, and attempt to crack the hash using all strings in the range of characters:

```
./cracker <begin> <end> <hash>
```

For example:

```
./cracker AAAAA ABAAA sdfj325SWE
```

It should output NO if the password is not within the specified range and should output YES if it is.

To test your cracker:

```
./cracker_checker.sh
```

The tests will print out a message if they failed.

The ordering for the range is as follows:
A B ... Z a b ... z .. 0 ... 9

This means that if the range is AAAAA ABAAA, your code should check:

```
AAAAA
AAAAB
...
AAAAZ
AAAAa
AAAAb
...
AAAAz
AAAA0
AAAA1
...
AAAA9
AAABA
AAABB
...
ABAAA
```

3.2 Part 2: Communication Protocol and the rest

Implement the communication protocol and the clients and server as described in this handout.

We have provided several scripts to test your code:

- The functionality of the server can be verified with:

  ```
  ruby -W0 server_tester.rb
  ```

- The functionality of the worker client can be verified with:

  ```
  ruby -W0 worker_client_tester.rb
  ```

3.3 Additional Requirements

When a request client receives the password, it should output it to stdout and terminate. There should be no other debugging output from the request client. Also, your server is required to handle SIGTERM. When it receives a SIGTERM, it should send SHUTDOWN messages to all of its worker clients and then terminate. Also, please note that the passwords we will be giving your cracker will always be 5 characters.

4 Technical Requirements

- Your project must function properly on the Unix Andrew machines
- Your code must compile on the Unix Andrew machines
- Your project must include a Makefile that builds your project
- Your code must be checked in to tags/final/

5 Turning in your code

AFS will be used to hand in work. We'll talk about this a bit closer to the deadline.

The Makefile must be a standard Makefile. For example, do not use cmake, as the Makefile it creates refers to a private directory that makes it extremely difficult for us to grade. The Makefile must not refer to any files outside the repository, and again, remember to turn in ALL the files needed to build or you will lose points.

6 Scalability and Graphs

As the lab is about distributing computation across many nodes, observe how the runtimes required to solve the problem vary both with different password lengths and with different numbers of clients.

Along with your project, submit the following graph (and any associated writeup textfile). The axes for the graphs we expect are shown below.

1. **Graph of Work Unit vs. Throughput**: For 4 clients, a fixed password length of 5 and a password of "AZURE", vary the work unit size from 3 to 5 and plot the length of time required to find the password. Work units are defined as the size of the range. So a work unit size of 3 would be represented by AAAAA AABAA or AAAAAA AAABAA. Talk to us if this isn't clear.The completion time is defined as the time between when the sender sends the initial job to the time it receives the password from a client.

7 C++ Tutorials and Resources

- C++ Tutorial
  `http://www.cplusplus.com/doc/tutorial/`
- C++ Reference
  `http://www.cppreference.com/wiki/start`
- C++ Primer
  by Lippman, Lajoie and Moo

8 Grading

The total number of points for this project is 100.

The breakdown of positive points is as follows:

- 20 points - Code quality, style
- 10 points - Graphs
- 70 points - Code functionality

The grading breakdown for code functionalitity is as follows.

- 15 points - Implements password cracking (Part 1).
- 25 points - Implements the password cracker assuming no communication failures. E.g., successfully speaks the protocol, distributes work, and finds the password.
- 15 points - Implements the password cracker while being robust to communication failures.
- 15 points - Works with multiple request clients.

The ruby testing scripts (worker_client_tester.rb, server_tester.rb) we provide will test most of the components of the project we will use for grading. If you pass those ruby testing scripts, you will likely get all the points for functionality.

Tread a woorme on the tayle and it must turne agayne.
– John Heywood