Lab 1: Distributed Password Cracker

Due: 11:59 pm, February 10, 2010

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 two programs, the client and the server. The server is responsible for dividing the job into parts and allocating those parts to clients. The client 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; both the client 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 client *must* be compatible with our sample binaries. Luckily, the sample binaries are included for you to test against, as is the full grading script. There are no "hidden" testcases.

2.1 Implementation

You must write your server and client 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 clients as they contact it. It should keep track of the clients and which jobs they're currently working on. It should handle timeouts. You must use `select` to do so.

The server should divide the jobs in the following way. Each client will receive a work unit, defined by a range from one string to another. The size of the work unit should be variable for graphing purposes. For example, this is a work unit with a size of 4:
AAAAA ABAAA

This indicates that the client is responsible for checking all the strings from AAAAA to ABAAA (inclusive). This is a work unit of size 5:
AAAAA BAAAA

Your code should produce an executable file named "server" and should use the following command line arguments:

```
./server <port> <hash> <size of work unit> <password length>
```

Example:
./server 2222 Badj45sWjgiE 4 5
The hash is obtained by calling the crypt function (man crypt).

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 Client

2.3.1 Client Architecture

The 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 "client" and should use the following command line arguments:

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

Example:
./client unix34.andrew.cmu.edu 2222

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)         |
+-------+-------+-------+-------+
```

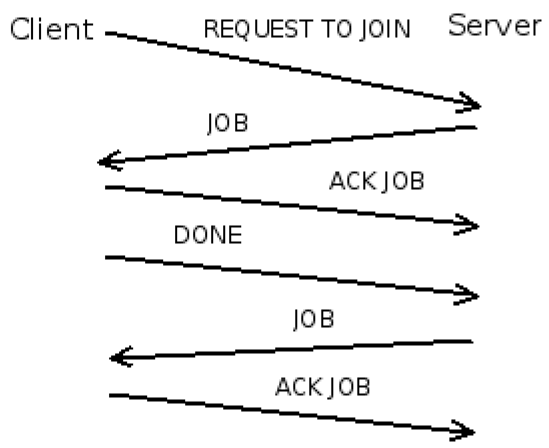
MAGIC refers to a magic number, so that we are able to identify whether a message is relevant to our application. Our MAGIC number is 15440. Version is 1. The client ID is the ID of the client randomly generated by the server when the client sends a REQUEST_TO_JOIN. When the client sends a REQUEST_TO_JOIN, 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 client should use that number from that point on.

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 client starts the relationship with the server by sending a
  command: REQUEST_TO_JOIN. There is no other info for this message.
- The server acknowledges this with
  command: JOB other info: "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
- The client acknowledges this with an
  command: ACK_JOB other info:"XXXXX XXXXX HHHHH"
  where the XXXXXs and HHHHHs are the same as before.
- When the client completes its job and has not found the password, it sends the server
  command: DONE_NOT_FOUND other info:"XXXXX XXXXX"
- When the client completes its job and has found the password, it sends the server
  command: DONE_FOUND "PPPP"
  where PPPP is the password.
- When the password is found, the server sends every client a
  command: SHUTDOWN. There is no other info for this message.
  The clients must terminate.

When the password is found, the server should print the password on standard output.

This protocol is better represented using a diagram:



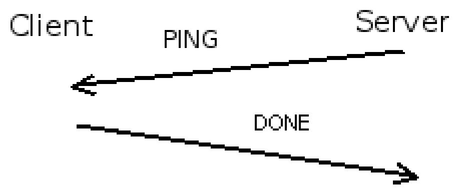
2.5 Failure

If the 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 client fails to send a message once it's completed a job (times out), the server should send a PING to the client.The client should respond with its previous DONE message (whichever variant it might have been). If the client 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 client, and wait for timeout again. If the PING times out, then the server should reassign the job to another client and forget about this previous client.

The timeout for all 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

- `protocol_checker.rb` - checks the protocol used by the server/client
- `cracker_checker.sh` - checks the correctness of the cracker
- `dropper.cpp` - aids in testing packet drops
- `server_tester.rb` - tests the server
- `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 client and server as described in this handout.

We have provided several scripts to test your code:

- The majority of the protocol can be verified with:

  ```
  ruby -W0 protocol_checker.rb
  ```

  The results will be outputted to your screen.

- The functionality of the server can be verified with:

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

- The functionality of the client can be verified with:

  ```
  ruby -W0 client_tester.rb
  ```

3.3 Additional Requirements

You must ensure that the server exits properly after outputting the password. This is to ensure that the script makes progress and does not hang indefinitely. **You should also turn OFF all debugging output - except for printing the password at the end - or you will lose points!**.

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

To turn in a working version of your code to us, you must make sure the files are committed to the **tags/final/** directory of your svn repository to avoid ambiguity. This is also required so that we can timestamp the code submission date and you don't have to worry about accidentally modifying the file after the due date.
One simple way to do this is to use the `svn copy` command. For example:

```
# svn copy trunk tags/final/
```

This will place a copy of everything in trunk into tags/final/.

VERIFY THAT ALL REQUIRED FILES ARE CHECKED IN TO SVN! You can do this by checking out your code into a different directory and making sure it builds (and passes the grading scripts in that different directory for the final turn in).

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.

You should be able to follow these steps:

```
//Check out the repository in a different directory
% cd tags/final
% make
...
% ruby -W0 client_tester.rb
...
% ruby -W0 server_tester.rb
...
```

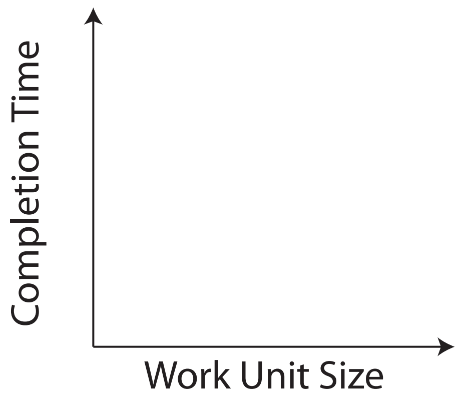
6 Scalability and Graphs

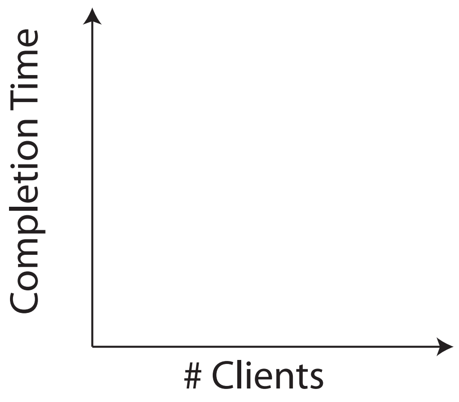
As the lab is about distributing computation across many nodes, observe how the runtimes required to solve the problem vary both with different password lengths and with different numbers of clients.

Along with your project, submit the following graphs (and any associated writeup textfile) by commiting them into your svn repository's tags/final/ directory. 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 from 3 to 5 and plot the length of time required to find the password. 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.

2. **Graph of Clients vs. Completion Time**: For a fixed password length of 5, a password of "AZURE" and a work unit size of 4, vary the number of clients communicating with the server from 1 to 4 and plot the password cracking completion time (i.e., how long it takes to crack the password). If you do not observe an improvement in completion time beyond a certain number of clients, please try to explain why you think this happens.





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.
- 20 points - Implements the password cracker while being robust to communication failures.
- 10 points - Works with multiple clients.

The ruby testing scripts (client_tester.rb, server_tester.rb) we provide will test all 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. The protocol checker scripts are provided for your use, but will not be used by us to grade.

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