CSCE 662 HW2

Distributed Password Cracker

Ke Wang **Xintong Xia**

Part I: LSP Implementation

1. LSP Server

a) There are two separate threads associated with a newly created server. One is network handler and the other is epoch handler. The server has a connection list for managing the clients, an inbox queue and an outbox queue for recording the receiving/sending message.

b) The network handler is responsible for receiving message from clients and taking measures corresponding to the contents of message.
If the message is a connection request, the handler will assign a connection id and create a new connection to record receiving and sending data message for each client (connection id).
If the message is a data message, the handler will immediately send an acknowledge, and at the same time put the message in inbox and update statistic information (i.e. sequence number).
If the message is an acknowledge, the handler will check the outbox and remove the message with the same id and sequence number.

c) The epoch handler serves as (re)sending data message and acknowledge.
Every fixed epoch time ends, epoch handler will check the outbox, if there are any, the handler will resend all the message. The handler will also send acknowledge for the most recently received message and update the connection state of each client.

2. LSP Client

There are also two separate threads associated with a newly created client. The network handler and the epoch handler works almost the same way as those of server. The main difference is that the client does not need to maintain a connection list.

Part II: Password Cracker

1. Server.c

a) The server has a main thread for listening and reading message from the clients. Two separate threads for handling different types of message and monitoring the connection state of workers.

b) Handling data message
If 'j' message (worker joining request) received, the server will put the worker at the back of the worker queue.

If 'c' message (cracking request) received, the cracking handler will be called, cracking handler will check the HashTable to see if the request is already exist (or finished).

If not, the handler will evenly divide the cracking task into 26 small parts and initialize the state of all the tasks as “unknown”. It will dispatch each partial assignment to each worker in the worker queue. With task assigned, each worker will be removed from the front of the queue and put back at the end of the queue.

If the request exists but has not been finished, the handler will check the result (Found or Not Found) and give it back to client.

If 'f' message (password found) received, the F handler will give the password back to the client and mark the corresponding task as “True” in the HashTable.
If 'x' message (password not found) received, the X handler will check the state of each assignment in Hashtable, if there is no “Unknown or True” state found, then it will send a “Not found” message to the client.

c) Monitoring workers
Server will send the divided job to available workers and remove worker from worker queue if it has not received the data message from worker for several epoch times.

d) The server also maintains a HashTable to map the key (hashed password) and the value (a structure containing request id and task list). This table will facilitate the cracking procedure by taking hashed password as searching key to get the corresponding information.

e) Cracking request
Since the server will not delete the requested tasks, clients can get the result by rerunning the commnad line if they lost connection with server.

2. worker.c
Each worker has a main thread to receive and analyze the task from the server. And it has a requesting queue to record all the tasks. The worker also has another separate thread to keep cracking the password. Each assignment contains a start alphabet (like 'a', 'b', ...'k'...), and a password length. The cracking handler will use brute force algorithm, which is each time generate a password alphabetically, compare it with the target hashed password, and then return the result to the server.

3 request.c
The client need provide a length for desired password and the hashed password.

Part III: Build And Run
The code has been tested on a single core x86_64 Ubuntu 12.10 machine, with Linux Kernel version 3.5.0-23-generic, GCC 4.7.1. Compiled executable (server, worker, request) has been attached with the source code.

To build:

```
$ make clean
$ make
```

To run:

```
$ ./server host
$ ./worker host:port
$ ./request host:port 40-bit-sha1-hashed-password len
```

Note:
1. In our application, we use the sha1.c for generating hashed password. To get the correct result, you need hash your desired password using this file. There is a shatest.c in our folder, which could help you generate the hashed password.
2. client may lost connection with server if the password is relative long and has not been cracked for a while. You can just rerun the command line and get the result.

Part IV Self Test Result

Simple:
=======
0. Checking for 100% compliance with hw2.pdf. For example, things like the capitalization of 'N' and 'F' in "If it does not find a password, it should print 'Not Found'" checked
1. Connection initiation and simple LSP protocol working test: checked
2. Simple 4 letter lowercase password cracking test: pass
3. Setting packet drop rates [0,0.5] : pass
4. Single character password cracking : pass
5. 10 clients, 1 server, 1 worker: pass
6. 1 client, 1 server, 10 workers: pass
7. 10 clients, 1 server, 10 workers: pass
8. Sending a randomly generated string as the password hash (you are supposed to return a password not found message): pass
9. Sending non-LSP protocol traffic to your server: pass
10. Changing the epoch count to something other than 5: pass
11. Sending a crack request before the server has any workers available: pass

Intermediate:
=============
0. Checking your implementation for memory leaks etc.
1. Changing the epoch timer to medium-large values : pass
2. More than 2 simultaneous crack requests to server (PER CLIENT): pass
3. Creating a large number of workers and setting the epoch time very low (5*epoch < time taken to crack a password): pass
4. High packet drop rates, > 0.75 (artificial): not pass
5. Arbitrary worker termination in the middle of password cracking: pass
6. Creating a large number of workers to overwhelm the server's job farming algorithm: pass

7. Out of order packet transmission, on purpose: <u>pass</u>
8. Creating a large number of clients who send connection requests but any replies from the server are dropped on purpose (firewall simulation): <u>pass</u>
9. Random termination of clients: <u>pass</u>
10. Checking your code for handling counters (what if the 32 bit sequence number counter rolls over?)

Expert:
=======
1. Sending random connection ids from client to server, as a protocol test
2. Sending crack requests with payload very close to 1500B (~750 character passwords)
3. Purposefully replying with out of order sequence numbers close to 32 bits (non compliant with LSP protocol, but it can blow your buffers)
4. Very high/very low epoch times
5. Very high packet drop rates (in the network, not artifical): <u>not pass</u>
6. passwords of 5 characters or longer (up to 10 minute execution time) : <u>conditionally pass</u>