Scalable distributed cracking program

Skye Im

Introduction

The Distributed Cracking Program, or DCP, is an application divided into two segments, a server and a client. The server sends out password guessing jobs to clients, and clients calculate passwords on multiple threads and returns the answer, if applicable. DCP can theoretically scale up to 1023 clients, each with as many worker threads as it wants. The server communicates with each client using TCP packets, which allows multiple machines to be connected over local area networks for maximum processing power.

DCP is written entirely in C, because DCP was designed for a final project for an Operating Systems course using C. It is entirely possible that writing the networking and task distribution in a higher level language such as C++/Python/C# and calling a C library for the actual calculation may have been more maintainable and leak less memory, but this is out of the scope of the course.

Because DCP serializes tasks into packets consisting of characters for exchanging data, it is possible for a client process to be written using a different language and interface with DCP as long as the language supports raw sockets.

Quick start

Building

DCP is only confirmed to compile on Unix systems, particularly Debian.

In the following section, everything before `$` is the current directory, and everything after `$` is a terminal command.

First you need to compile the `pwchecker` library. Navigate to `DCP/PwChecker` and run:

```
DCP/PwChecker$make all
```

Next, the library must be moved to the appropriate location, DCP/lib. Move it manually, or type the command:

```
DCP/PwChecker$cp lib/libpwchecker.a ../lib
```

Return to the DCP directory and type the command:

```
DCP$make crackerapp
```

The server and client will be produced in the `DCP/bin` folder as the executables `server` and `client`, respectively.

Running

This is the syntax for running the server:

```
DCP$./bin/server [INPUT FILE] [OUTPUT FILE]
```

For example, given an input `input.txt` and output `output.txt` in the `DCP` directory, the command will be:

```
DCP$./bin/server input.txt output.txt
```

The server always runs on port 7777, so make sure it is free.

This is the syntax for running the client:

```
DCP$./bin/client [IP ADDRESS] [NUMBER OF WORKER THREADS]
```

For example, if the server is at IP address 127.0.01 and we want 3 worker threads, the syntax would be:

```
DCP$./bin/client 127.0.01 3
```

Once the server is up it goes into an idling state. During this state, any number of clients may connect. Type any key into the server console and press enter to start running the cracks.

A client may connect to the server at any time, even while running. The server will gracefully accept the connection request and include the client on the next job. However, if a client disconnects the server will enter an infinite loop. The server does not know which client disconnected, and therefore cannot redistribute its workload. The server does not need to finish all jobs before outputting the found passwords. The answers for the passwords found before termination will be found in the specified output file.

Note that DCP is untested for large password lengths (>5) because of technical limitations. DCP is a prototype and leaks memory, so larger operations may overflow and crash. For a reliable password cracking experience, the author suggests using a mature, FOSS password cracker rather than one coded by one person.

System design

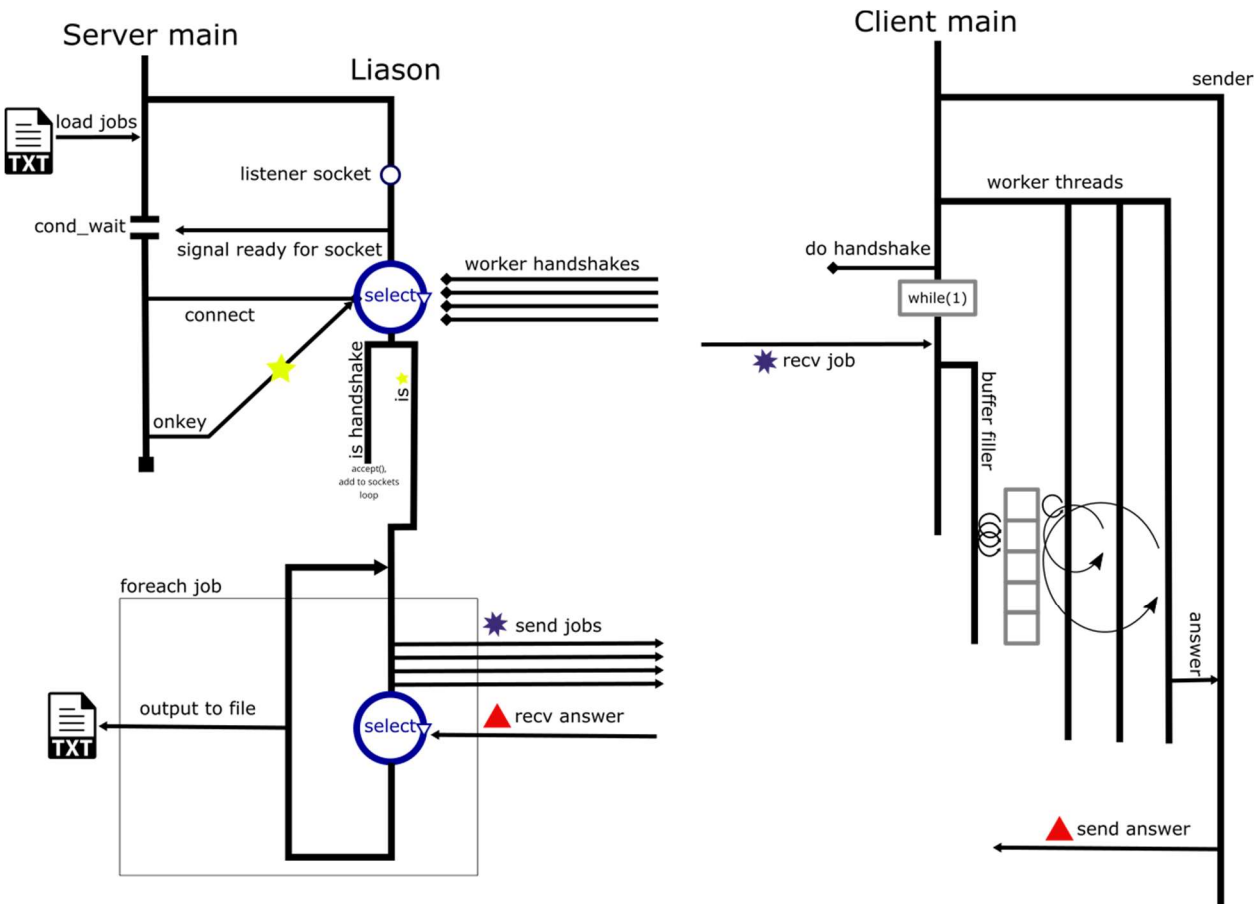

Figure 1: Entire system architecture

DCP is a networked, distributed program. Its system design is similarly complicated. First we will define some terms. Each term has full descriptions later on.

job: A single hash to brute-force the password for.

work index: Each job has a separate work index.

task: A range of passwords to check the hash against. Tasks are encoded in work packets.

main thread: The first thread of a program.

> **server main thread:** Spawns a liaison thread and loads job info from a file, then waits for user input to instruct cracking to begin.
>
> **client main thread**: Spawns worker and sender threads, and receives tasks from a server. Upon reception of a work packet, spawns a buffer-filler thread.

liaison thread: The thread of a server which handles task distribution and networking.

handshake: A client establishes a connection to the server, which accepts the connection. This is a handshake.

password index: The enumerated representation for a password.

work unit: A work unit is a smaller range of passwords to check a hash against. Tasks are usually received by a client and broken up into work units to be distributed to worker threads.

buffer: A 100-long threadsafe ring buffer that stores work units produced from a buffer-filler thread.

worker thread: A worker thread consumes work units from a buffer and executes the unit. If it discovers the password, the answer is transmitted to the sender thread.

sender thread: A sender thread receives an answer from a worker thread and transmits it to the server as an answer packet..

buffer-filler thread: A buffer-filler thread receives a task and divides it into work units, which are then filled into the buffer for worker threads to consume.

packet: A *packet* is a minimum five byte string encoding specific types of data to be transmitted over TCP sockets.

> **packet header:** The packet header is a five byte-long string that informs the receiver of the length of the packet to follow, as well as the type of packet.
>
> **start packet**: Transmitted from the server main thread to the liaison thread. Instructs liaison to begin password cracking.
>
> **work packet**: Transmitted from the liaison thread to a client thread. It includes info such as the hash to compare against, the password length, the work index, and the range of potential passwords to check.
>
> **answer packet**: Transmitted from the sender thread to the liaison thread. The answer packet simply holds discovered password info.

Job

Each job is encoded in the input file with the following format:

`[HASH] [PASSWORD LENGTH]`

An example job is:

`4b3bed8af7b7612e8c1e25f63ba24496f5b16b2df44efb2db7ce3cb24b7e96f7 4`

A job represents a password to be cracked.

Work index

Each job has a work index to facilitate work distribution, as described in *Client main thread*.

Task

A job is divided into multiple tasks, one for each client connected to the server. A task is simply a range of passwords to check against. Passwords may be comprised of one or more characters 0-9, a-z, A-Z. Therefore, it is trivial to treat passwords as base-62 numbers and enumerate them as ranges.

Server main thread

The only purpose of the main thread is initialization. Time must be allowed for clients to connect, and it is easier to have the user manually start the cracking.

Because there is already a socket for the liaison thread, the main thread connects to the liaison thread's socket and sends a start packet when the user signals to begin cracking.

Client main thread

The client main thread establishes a connection to the server and waits for a work packet. It then spawns a buffer-filler thread which handles filling the buffer.

The reason the client spawns a buffer-filler instead of filling the buffer itself is in order to receive new work packets. Each work packet has a work index. If the work index is higher than the current work index, the client attempts to cancel the buffer-filler thread and spawns a new buffer-filler thread with the new information. It also updates the current work index.

Before doing each work unit, a worker checks the work unit's work index against the current work index, and skips doing work if the current work index is higher. The current work index does not need to be protected by a mutex because only one thread writes to it, and it does not matter if the worker thread reads an old work index because it is in a while loop.

Handshake

Each client sends a handshake to the liaison thread when initializing, in order to receive work packets.

Password index

As described in *task*, a password can be treated as a base-62 number and be converted into a base-10 number, which serves as its index.

Work unit

A work unit includes the following info:

`work index, password index, work range, hash, password length`

It is represented as the `workunit` struct.

When the client main thread receives a work packet, it spawns a buffer-filler thread which then generates work units.

Buffer

The buffer is a 100-work unit-long ring buffer structure with semaphores and put/take functions. It is threadsafe.

Worker thread

Spawned by a client main thread. Worker threads attempt to take work units from the buffer and crack passwords. As described in *client main thread*, worker threads compare the current work index to the work unit's work index and skips doing work if the work unit's work index is smaller than the current work index, ie, if the work unit is outdated. This allows worker threads to quickly empty a buffer that is still filled from a previous task and receive new work units.

When a worker thread discovers a password, it constructs an answer packet and sends the answer packet to the sender thread using a semaphore-protected field.

Sender thread

The sender thread blocks on a semaphore-protected field, then takes the answer packet and transmits it to the liaison thread.

Buffer-filler thread

The buffer-filler thread is spawned by a client main thread. It divides a task into smaller work units, which are by default have a range of 10000. Define the macro `BATCH_SIZE` when compiling `client.c` to change this range. Too large a batch size will cause worker threads to do a lot of unnecessary work, while too small a batch size will incur significant overhead from filling and emptying the buffer.

Once the entire task has been converted into work units and inserted into the buffer, the buffer-filler thread exits. The buffer-filler thread may block on `put`-ing into the buffer, and is not guaranteed to exit before a new task arrives.

When the client main thread receives a new work packet, it cancels the buffer-filler thread via `pthread_cancel` then creates a new buffer-filler thread. Since the `put` function may block on `sem_wait`, either the cancel will instantly return or cause the buffer-filler thread to cancel, and the old buffer can be safely cleaned up via `pthread_join`.

Packet header

The format of the packet header is:

```
MMMMT
```

where `MMMM` is the length of the packet, excluding the header and up to 9999, and `T` is the *type* of packet. Valid values for T include `s`, `w`, `a`.

This is an example of a packet header:

```
  73w
```

Notice the two leading spaces.

Start packet

The start packet holds no extra information and is always five bytes long. This is the only valid form of start packet, including the header:

```
   0s
```

The start packet simply signals the liaison thread to start cracking. The liaison thread also marks the socket that sent the start packet as not a client.

Work packet

A work packet encodes a task and is sent by a liaison server to a client. The format for a work packet is:

```
MMMMwW+ N+ B+ L+ H(64)
```

where `MMMMw` is the header, `W+` is at least one digit of work index, `N+` is at least one digit of number of workers, `B+` is at least one digit of task index, `L+` is at least one digit of password length, and `H(64)` is the 64 character-long hash.

The number of workers and the task index is relevant to a client because these two numbers, along with the password length, allow the client to determinate the range of passwords it is checking the hash against. Sending only task index and number of workers therefore allows packets to be much shorter.

This is an example of a valid work packet:

```
  73w3 4 3 8
1709bc8ccb278eaf47ce59c271217866da96be38f52bd95ec798c29ba71c96c2
```

Note that there is a space after the *8*. This packet describes a task of work index 3, 4 workers, and the 4th worker, cracking a password of length 8 as described by the hash.

Answer packet

An answer packet holds info for a cracked password. It contains work index, password, and password length information. The format for an answer packet is:

```
MMMMaW+ L+ P+
```

where `MMMMa` is the header, `W+` is at least one digit of work index, `L+` is at least one digit of password length, and `P+` is `L+` digits of password.

This is an example of a valid answer packet:

```
  15a3 10 helloworld
```

This packet describes an answer for work index 3, a length 10 password `helloworld`.

Answer packets are sent to the liaison thread by a sender thread. When the liaison thread receives an answer packet, it records the packet in an output file and distributes the next job.

Brief description of some functions

During the process of creating DCP, many helper functions were required. Some helper functions are exceptionally interesting and are introduced here.

```
serialize_packet(void* packet…), deserialize_packet(void** packet…)
```

The function signature is truncated for brevity. These two functions delegate to various functions named `serialize_work`, `serialize_answer`, and so on, hiding the messy pointer-casting logic behind a simple interface. This is an interface pattern, a form of proto-OO design implemented in C.

Unfortunately, the serialize family of functions all use `asprintf`, which is **not** a POSIX-compliant string function, rather being a GNU function. In order to achieve full POSIX compliance, DCP needs its an `asprintf` shim.

```
recv_all(…), send_all(…)
```

Neither `recv` nor `send` are guaranteed to actually send or receive the entire message. `recv_all`, `send_all`, and the `select()` pattern were adopted from the fantastic resource for networking in C, *Beej's Guide to Network Programming*, freely available online. The _all functions simply keep track of the number of bytes received from the system call and loop until all bytes are received.

```
calc_pwrange(int nworkers, int rangeindex, short pwlen…)
```

```
nth_pwd(unsigned long long index, short pwlen…)
```

This functions allow packets to be very small, rarely longer than 100 bytes. Using some simple math, clients can reconstruct the correct range of passwords to check against from three integers. As described above, this method works by treating each password as a base-62 number.

Whole picture

This is a typical example of DCP operation:

Server starts, loads jobs and waits.

Server creates liaison thread, starts listening using `select`.

Liaison signals main thread.

Main thread connects to the liaison thread socket and waits for user input.

Client1 starts, spawns sender and worker threads, then handshakes with server.

Client2 starts, spawns sender and worker threads, then handshakes with server.

Main thread receives user input, sends a start packet to liaison.

Liaison sends two work packets, one each to Client1, Client2.

Each client receives the work packet, spawns buffer-filler threads.

Buffer-filler threads insert work units into buffer. Worker threads consume work units and do passwork cracking work.

Client3 starts, spawns sender and worker threads, then handshakes with server.

Client2's worker thread discovers the password, signals sender thread. Sender thread sends answer packet to liaison thread.

Liaison thread receives answer packet, outputs to file and sends out new work packets. Three total, one each for Client1, Client2, Client3.

Repeat until all jobs are done.

Liaison thread exits.

Client1, Client2, Client3 gracefully exit by segfaulting because the author was not able to track down the reason for the segfault.

Challenges faced

The various design decisions for DCP have been justified above.

Many challenges were faced while creating DCP. Because C provides nearly no user-friendly libraries for networking, there is a large amount of boilerplate code before any data can even be sent. The example code provided during class (`tserver`, `tclient`) was adapted into the header `simplesockets.h`.

Another issue was *using* sockets. Sockets provide no information at all of message length when receiving. A custom packet protocol (described in detail above) was implemented, as well as serialization and deserialization functions. The packet has a fixed-length header which describes the length of the packet body. The receiver then simply needs to `recv_all` the packet length to precisely receive the entire packet.

Synchronization was, surprisingly, the least difficult part of this project. In Operating Systems class, we exhaustively covered the various methods of synchronization, be it via named semaphores, in-memory semaphores, mutexes or conditional variables. The buffer provides the vast majority of the synchronization for the client, while the server is largely single-threaded. DCP uses threads instead of processes, because threads provide many of the advantages of processes, but also share memory among threads. Sharing memory means easier setup and tear-down, because `shared_mem` is very difficult to use and dirty to clean up. The stack cleans itself up on exit.

Generating password combinations was a problem solved via the methods described above.

One on-going challenge that has yet to be solved is memory management. It is commonly said that C gives a programmer a gun which more often than not ends up pointed at her own foot. Using Valgrind on DCP reveals that many hundreds of bytes of memory are lost, but tracking down where and when exactly is an arduous, thankless task. Since DCP is not intended for commercial use, the lost memory is not a critical issue.

Another challenge that wasn't considered critical enough to solve is how the server fails to accept a client disconnecting gracefully. DCP (perhaps naively) assumes that clients only connect and never drop connection. The solution would involve keeping track of which client has which task, and upon connection drop redistributing the

task to the remaining clients, as well as removing the dropped client from the select `fd_set`. The problem gets more difficult when one of *those* clients subsequently drops, and really isn't part of the final project description and so was left unsolved.

Conclusion

DCP was a valuable learning experience. I have not had much experience with using C to code real-world applications, and I believe that I learned much from DCP, and that I have a much more solid grasp on C, including generic functions, makefile structure, string usage and synchronization. I also re-discovered my burning hatred for C as a language—anything related to strings was especially painful—and in the future I will most likely *not* use C unless it is a situation where the choice is between C or ASM. However, the semaphore and mutex knowledge gained from DCP is broadly applicable to other languages, too, since most languages provide bindings to system calls.

DCP has many aspects that require improvement. The server does not gracefully handle client connection drops, and the client segfaults when the server drops. Memory is leaked left and right with reckless abandon. But DCP is also a distributed, networked computing application, and I believe it fulfills the final project criteria.