Homework 2: Distributed Password Cracker

600 points

1 Overview

In this assignment you will construct a simple distributed system that can harness the power of many processors to speed up a compute-intensive task. In your implementation, you will incorporate features required to create a *robust* system, handling lost or duplicated Internet packets, as well as failing clients and servers. You will also learn the value of creating a set of layered abstractions in bridging between low-level network protocols and high-level applications.

Your system will implement a distributed password cracker. Most systems minimize the cases where a password is stored or transmitted explicitly, since a minor breach of security would enable an adversary to get a copy of the password and then have full access to many system resources. Instead, a *secure hash function* h is applied to the password P to give a hash signature $H = h(P)$. Function h is designed so that it cannot easily be inverted. That is, knowing the value of H provides no real information about the value of P. Rather than storing and passing passwords around explicitly, systems strengthen the security of their password management by only keeping their hash signatures.

As an example, in early Unix systems, information about all user accounts, including the hash signatures of their passwords, was stored in an unencrypted and unprotected file `/etc/passwd`. When a user logged in and gave password P, the system would compute $h(P)$ and see if it matched the signature stored in the file for that user. The security of this approach was premised on the assumption that an attacker would be unable to determine the value of a password given its hash signature. Nowadays, Unix systems store the hash signatures in less accessible locations, because it became practical to crack passwords using brute-force approaches, similar to the one you will implement.

One commonly used secure hash function is SHA-1, developed by the National Security Agency. It generates a 20-byte hash signature of an arbitrary sequence of bytes. Here are some examples of applying SHA-1 to text strings:

P	$h(P)$
cat	9d989e8d27dc9e0ec3389fc855f142c3d40f0c50
bat	9b64d4ad2ee2fb847c0d7c96a9480b183cafd31b
bar	62cdb7020ff920e5aa642c3d4066950dd1f01f4d

As these examples illustrate, despite the similarity among text strings, their hash values are very different.

One simple attack on a scheme that relies on the difficulty of inverting a secure hash function is to run a brute-force search, in which we enumerate possible passwords and see if they hash to H. Our measurements show that a typical Andrew Linux machine can compute SHA-1 hashes at a rate of around 10,000 per second. Running sequentially, a brute force cracker would require around 9 hours to try all possible passwords consisting of at most 6 lower-case characters. But, if we could harness the power of 100 machines, then we could reduce this time to around 5 minutes.

Any time we want an application to run across tens or hundreds of machines, we must devise ways to make the system robust. Inevitably, some of these machines will stop working, or we will encounter cases where machines become disconnected. Thus, writing a distributed password cracker gives you a chance to hone your skills in designing reliable distributed systems.

The project is split into two parts:

A: Implement the *Live Sequence Protocol*, a homegrown protocol for providing reliable communication with simple client and server Application Program Interfaces (APIs) on top of the Internet UDP protocol.

B: Implement the password cracker application. You will find that the abstract interface provided by LSP will make this part go much smoother than it would if you were just hacking UDP code directly.

2 Part A: Live Sequence Protocol

The low-level Internet Protocol (IP) provides what is referred to as an "unreliable datagram" service, allowing one machine to send a message as a packet to another, but with the possibility that the packet will be either dropped or duplicated. In addition, as an IP packet hops from one network node to another, its size is limited to a specified maximum number of bytes, known as *the Maximum Transmission Unit* (MTU). Typically, packets of up to 1500 bytes can safely be transmitted along any routing path, but going beyond this can become problematic.

Very few applications make use of the IP service directly. Instead, they are written in terms of one of the following protocols:

UDP: Also an unreliable datagram service, but it allows packets to be directed to different logical destinations on a single machine, known as ports. This makes it possible to run multiple clients or servers on a single machine.

TCP: A reliable streaming service, in which a series of arbitrary-length messages is transmitted by breaking each message into multiple packets at the source and then reassembling them at the destination. TCP handles such issues as dropped packets, duplicated packets, and preventing the sender from overwhelming both Internet bandwidth and the buffering capabilities at the destination.

The Live Sequence Protocol (LSP) provides features that lie somewhere between UDP and TCP, but it also has features not found in either protocol.

- Unlike UDP or TCP, it is specialized to support a client-server communication model
- The server maintains *connections* between a number of clients, each of which is identified by a numeric *connection identifier*.
- Communication between the server and a client consists of a sequence of discrete messages in each direction.
- Message sizes are limited to fit within single UDP packets (around 1000 bytes).
- Messages are sent *reliably*: exactly one copy of each message is received, and messages are received in the same order they are sent.
- The server and clients monitor the status of their connections and detect when the other side has become disconnected.

2.1 LSP: Network Perspective

We will first describe LSP in terms of the messages that flow between the server and one of its clients. Each LSP message contains three values:

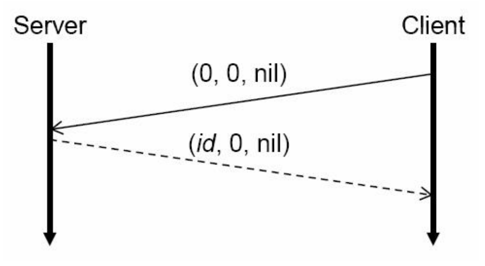

Figure 1: Establishing connection. The server assigns the connection ID and responds to the client with an acknowledgment message (shown as a dashed line.)

Connection ID: A unique, 32-bit, nonzero number assigned by the server to identify this connection. Your implementation may chose any scheme for assigning IDs. Our implementation simply assigns IDs sequentially, starting with 1.

Sequence Number: A 32-bit number that is incremented with each data message sent, starting with number 1.

Payload: A sequence of bytes, with a format and interpretation determined by the application.

We will use the notation (id, sn, D) to describe a packet with connection ID id , sequence number sn, and payload D, where an empty payload is denoted nil.

There are three categories of message:

Connection Request: Having form (0, 0, nil), sent by a client to the server to establish an initial connection.

Data: Having the form (id, i, D) where the payload is nonempty, and this is the i^{th} data message sent in this direction for this connection.

Acknowledgment: Having the form (id, i, nil), sent to acknowledge either a connection request ($i = 0$) or a data message ($i > 0$).

Figure 1 illustrates how a connection is established. In this figure, the vertical lines with downward arrows denote the passage of time on both the client and the server, while the lines crossing horizontally denote messages being sent between the two. The client initiates the connection by sending a connection request to the server. The server assigns a connection ID and

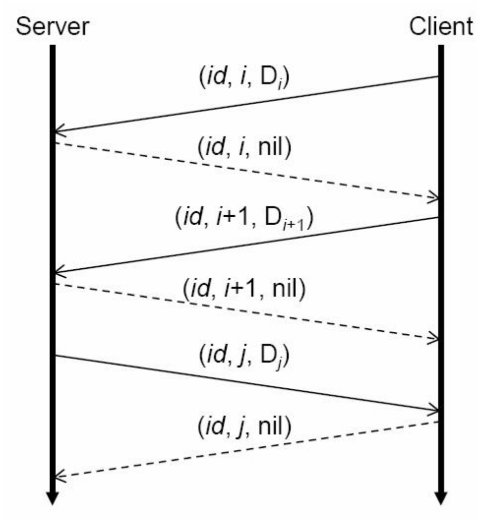

Figure 2: Normal communication. Message number i must be acknowledged before message $i + 1$ can be sent. Acknowledgment messages are shown as dashed lines.

responds to the client with an acknowledgment message containing this ID, sequence number 0, and an empty payload.

Figure 2 illustrates a normal communication sequence between the server and a client. Data messages can be sent in either direction, and different series of sequence numbers are maintained for each direction. The figure illustrates the transmission of data values D_i and D_{i+1}, having sequence numbers i and $i + 1$ from the client to the server. These are followed by a transmission of data value D_j , having sequence number j, from the server to the client.

Observe that every data message is followed by an acknowledgment message in the opposite direction. Each side must wait for the acknowledgment to be received before it can send another data message. Note, however, that it is entirely possible for one side to receive a data message from the other while waiting for the acknowledgment of a data message it has already sent - the two sides operate asynchronously, and IP does not guarantee that packets arrive in the same order they are sent.

We can see that the basic protocol illustrated in Figures 1 and 2 is not at all robust. On one hand, the presence of sequence numbers would make it possible to detect when a message has been dropped or duplicated. However, if any message - connection request, data message, or acknowledgment - gets dropped, the linkage in either one or both directions will stop working, with both sides waiting for messages from the other.

To make LSP robust, we incorporate a simple time trigger into the servers and clients. That is, timers fire periodically on the clients and server, dividing the flow of time for each process into a sequence of *epochs*. Let us denote this time interval δ. Our default value for δ is two seconds, although this can be varied.

Each time the epoch timer fires a client takes the following actions:

- Resend a connection request, if the original connection request has not yet been acknowledged.
- Send an acknowledgment message for the most recently received data message, or an acknowledgment with sequence number 0 if no data messages have been received.
- If a data message has been sent, but not yet acknowledged, then resend the data message.

The server performs a similar set of actions for each of its connections:

- Acknowledge the connection request, if no data messages have been received.
- Acknowledge the most recently received data message, if any.
- If a data message has been sent, but not yet acknowledged, then resend the data message.

Figure 3 illustrates how the epoch events make up for failures by the normal communication. We show the occurrence of the epoch timer as a large black oval on each time line. In this example, the client attempts to send data D_i, but the acknowledgment message gets dropped. In addition, the server attempts to send data D_j , but the data message gets dropped. When the epoch timer triggers on the client, it will send an acknowledgment of data message $j - 1$, the last data message received, and it will resend data D_i.

Assuming the server has an epoch event around the same time (there is no requirement that they occur simultaneously), we can see that the server will send an acknowledgment for data D_i, and it will resend data D_j.

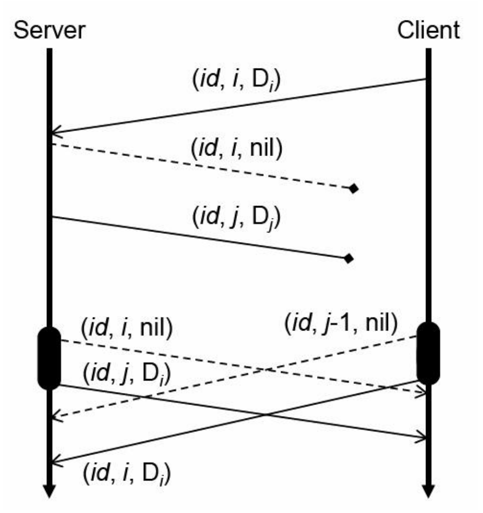

Figure 3: Epoch activities. Both sides send acknowledgments (shown as dashed lines) for the most recently received data and (possibly) resend any unacknowledged data.

We can see in this example that duplicate messages can occur: for example, the server received two copies of D_i. For most cases, we can use sequence numbers to detect any duplications. That is, each side maintains a counter indicating which sequence number it expects next and discards any message that does not match the expected number. One case of duplication requires special attention, however. It is possible for the client to send multiple connection requests, with one or more requests or acknowledgments being dropped. The server must track the host address and port number of each connection request and discard any for which that combination of host and port already has an established connection.

One feature of our handling of epochs is that there will be at least one message transmitted in each direction between client and server on every epoch. As a final feature, we will track at the endpoint of each connection the number of epochs that have passed since a message was received from the other end. Once this count reaches some number K, we will declare that the connection has been broken and notify the application that it must take some action. Our implementation sets this limit K as 5. Thus, if nothing is received from the other end over a total period of $K \times \delta$, having a default value of 10 seconds, then the connection will be terminated.

2.2 LSP: Application Program Interface (API) Perspective

We will now provide a view of LSP from the perspective of a C programmer. We require you to implement the exact API shown here to facilitate automated testing.

2.2.1 Formatting Packets

Since raw C structs sent over sockets is not ideal, we will be using a serialization/marshaling library to facilitate worry-free networking between server and client. The following struct which defines a LSP packet will need to be specified in the format required by the marshaling library.

```
typedef struct
{
        uint32_t connid;
        uint32_t seqnum;
        uint8_t payload[];
} lsp_packet;
```

```
message LSPMessage
{
        required uint32 connid=1; // Connection ID
        required uint32 seqnum=2; // Sequence Num
        required bytes payload=3; // Payload
}
```

For marshaling, we will be using the "Google Protocol Buffers for C" library (`protobuf-c`). Use the above text as a `.proto` specification for the LSP protocol. Generate the required files using the `protobuf-c` compiler.

2.2.2 Client API

An application calls the function `lsp_client_create` to initiate the activities of an LSP client. This function is declared as follows:

```
lsp_client* lsp_client_create(const char* dest, int port);
```

where `host` is the domain name of the server, and `port` is the port on which the server is operating. If the client cannot be initiated, e.g., because the server is not available, the function returns FALSE. Your implementation is supposed to return a pointer to a struct of type lsp_client. Beware of malloc.

The application client simply reads and writes data messages. It can also close a connection, causing the client to cease transmitting or receiving messages. All of the details of establishing a connection, acknowledging messages, and handling epochs is hidden from the application programmer. The following function declarations define this interface:

```
// Client Read. Returns NULL when connection lost
// Returns number of bytes read
int lsp_client_read(lsp_client* a_client, uint8_t* pld);

// Client Write. Should not send NULL
bool lsp_client_write(lsp_client* a_client, uint8_t* pld, int lth);

// Close connection. Remember to free memory.
bool lsp_client_close(lsp_client* a_client);
```

As the comment indicates, the `Read` function returns -1 when the client has become disconnected from the server. Furthermore, the application should never call `Write` with a payload of NULL or perform other funny invocations.

2.3 Server API

The API for the server is similar to that for a client, except that every message is tagged by its connection ID. The server is set up and initiated by calling the function `lsp_server_create`, defined as follows:

```
lsp_server* lsp_server_create(int port);
```

Its parameter is the `port` number that the server should use. The function returns a pointer to a struct if starting a server was successful. If the server cannot be started, the function returns NULL.

The server application can also read and write messages and close connections. In its case, however, every one of these operations includes a connection ID:

```
// Read from connection. Return NULL when connection lost
// Returns number of bytes read. conn_id is an output parameter
int lsp_server_read(lsp_server* a_srv, void* pld, uint32_t* conn_id);
```

```
// Server Write. Should not send NULL
bool lsp_server_write(lsp_server* a_srv, void* pld, int lth, uint32_t conn_id);

// Close connection.
bool lsp_server_close(lsp_server* a_srv, uint32_t conn_id);
```

The `Read` operation returns two values: the connection ID indicating the client from which the message was received (`conn_id`, an output parameter), and the number of bytes read (as a regular return parameter) and stored in the array referred to by the `pld` pointer. If the return value is NULL, this indicates that the client has become disconnected. The Write and Close operations include a connection ID as an argument, indicating which connection should be written to or closed. Closing a connection indicates that the server should stop transmitting or receiving messages for this connection.

2.3.1 Setting LSP Parameters

In addition to the basic operations of the client and server, your LSP implementation must support the following operations to facilitate automated testing:

```
// Set length of epoch (in seconds)
void lsp_set_epoch_lth(double lth);

// Set number of epochs before timing out
void lsp_set_epoch_cnt(int cnt);
```

These functions set the parameters δ and K for the epoch duration and the maximum number of epochs that are allowed to pass without receiving any packets from the other side of a connection.

```
// Set fraction of packets that get dropped along each connection
void lsp_set_drop_rate(double rate);
```

Setting this parameter to a value r with $0.0 < r \leq 1.0$ should cause program to randomly "drop" packets with probability r. That means that your code that either sends or receives packets should generate a random number x between 0.0 and 1.0 and, if $x < r$, then your program should either fail to send the packet or proceed as if the incoming packet had never been received.

2.4 LSP Application Example

As an illustration of how applications can be written to use LSP, we show the core functions for a simple echo client and server. The client takes as an argument the port number of the server it connects to. The following is the echo client code (`sample_client.c`):

```
lsp_client* myclient = lsp_client_create("127.0.0.1", atoi(argv[1]));

// payload of the LSP packet
char message[] = "ilovethiscoursealready";
lsp_client_write(myclient, (void *) message, strlen(message));

// receive buffer
uint8_t buffer[4096];
int bytes_read = lsp_client_read(myclient, buffer);

// Print the received LSP protocol payload
puts(buffer);

lsp_client_close(myclient);
```

The client uses the client API to initialize a client, send some data to the server using the LSP protocol format, then reads the data and prints it. **Note that marshaling and unmarshaling is NOT to be done by the echo client, but by the program which implements the client API** (i.e, in `lsp.c`).

The following is the server code (`sample_server.c`):

```
lsp_server* myserver = lsp_server_create(atoi(argv[1]));

uint8_t payload[4096];
uint32_t returned_id;
int bytes_read;

for(;;)
{
        //      Wait for echo client to send something
        int bytes = lsp_server_read(myserver, payload, &returned_id);

        //      Echo it right back
        lsp_server_write(myserver, payload, bytes, returned_id);
}
```

The echo server starts off by using the server API to initialize a server. It then goes into an infinite loop, reading data and echoing it right back. Note that the implementation is incomplete since `lsp_server_close` has not been used. This gives you an idea of what an echo server which uses the server API looks like. **Note that marshaling and unmarshaling is NOT to be done by the echo server, but by the program which implements the server API** (i.e, in `lsp.c`)

2.5 Requirements and Style Issues

Use the socket API to send LSP packets back and forth. Make sure that you handle (un)marshaling correctly. Because the `bytes` example code for `protobuf-c` is marked as TODO on the webpage, the following code shows you how to interact with the `bytes` data type. Here it is, for marshaling only:

```
uint8_t* buf;
int len;

LSPMessage msg = LSPMESSAGE__INIT;
msg.connid = 0;
msg.seqnum = 0;
msg.payload.data = malloc(sizeof(uint8_t) * ...);
msg.payload.len = ...;
memcpy(msg.payload.data, pld, lth*sizeof(uint8_t));

len = lspmessage__get_packed_size(&msg);
buf = malloc(len);
lspmessage__pack(&msg, buf);

sendto(...);
free(buf);
free(msg.payload.data);
```

Now, your client/server implementation will have to handle the following three things: a network handler which takes care of the LSP protocol specifications (and keeps the protocol abstracted to the API user), an application handler which interacts with possibly multiple programs which want to use the client/server API, and an epoch handler which handles timeouts when a timer fires.

Beware of malloced/calloced pointers and structs, as this is an easy way to run into problems.

It is upto you whether you want to implement blocking or non-blocking I/O.

What you put into the LSP packet's payload is upto you. Make sure your client and server use the same language to talk to each other via the LSP payload. Part B will require you to support certain client-server primitives.

You will be graded for this part of the assignment both on how well your program can pass a set of automated and manual tests, as well as the quality of your design and implementation. You should think carefully about an overall architecture for your system: how you will structure the different components in your client and server, how they will communicate and coordinate with one another, and what data structures you will require. Include in your code documentation on this architecture as a set of comments. The quality of your documentation will be considered in grading.

3 Part B: Password Cracker

Your password cracker will involve implementing three different components:

Request: An LSP client that sends a user-specified password cracking request to the server, receives and prints the response, and then exits.

Worker: An LSP client that continually accepts cracking requests from the server, exhaustively checks for a password over a specified range of strings, and then responds to the server with the search result.

Server: An LSP server that manages the entire cracking enterprise. At any time, it can have any number of workers available, and it can receive any number of requests. For each request, it splits the request into smaller jobs, and farms these out to workers. It monitors the returning results and ultimately replies back to the request client.

We will consider only passwords consisting of lower case letters. We can limit the amount of effort by any cracking job by specifying a range of possible passwords, giving the alphabetically smallest and largest passwords to attempt. So for example the range aaaa to zzzz designates all 4-character passwords. (Our jobs will always involve passwords of some fixed length.)

Format	From-To	Use
j	W-S	Join request
c *hash lower upper*	R-S, S-W	Crack request
f *pass*	W-S , S-R	Password found
x	W-S , S-R	Password not found

Table 1: Message types for password cracker. In the "FromTo" column, "W" denotes a worker, "S" denotes the server, and "R" denotes a request client.

Table 1 shows the types of messages that are sent among the different system components. These are sent as strings (formatted with spaces between the fields) as the payloads of LSP packets. The overall operation of the system proceeds as follows:

- The server is started using the following command, specifying the port number for the server to use:

  ```
  ./server port
  ```

- One or more workers are started using the following command, specifying the address and port number of the server:

  ```
  ./worker host:port
  ```

- When a worker starts, it sends a join request message to the server, letting the server know that it is available.
- The user generates a cracking request by giving the following command, specifying the address and port of the server, the hash signature to be inverted, and the length of the password

  ```
  ./request host:port hash len
  ```

- The request client should generate a crack request message giving lower and upper values aa...a and zz...z, where the number of a's and z's is based on the length of the desired password.

- The server breaks this request into more manageable-sized jobs (You should choose a suitable maximum job size.) It then starts farming the jobs out to workers by sending them crack requests with a smaller range of possible passwords.
- A worker responds to the server that it has either found a password or that it has not.
- Once a worker finds a password, the server can relay this result to the request client and cease any further effort for this request.
- If the workers exhaust the full range of the original request without finding a password, then the server responds back to the request client that it failed to find a password.

The request client should print its results on standard output as follows. You must match this format precisely in order for our automated testers to work.

- If it finds password pass, it should print

  ```
  Found: pass
  ```

- If it does not find a password, it should print

  ```
  Not Found
  ```

- If the client loses the connection to the server, it should print

  ```
  Disconnected
  ```

We will assume that the server operates all the time, but it is quite possible that a request client or some of the workers can drop out. You should take the following actions when different system components fail:

- When a worker loses contact with the server it should shut itself down.
- When the server loses contact with a worker, it should reassign any job that the worker was handling to a different worker.
- When the server loses contact with a request client, it should cease working on any cracking request being done on behalf of the client. (You need not forcibly terminate a job on a worker; just wait for it to complete and ignore the results.)
- When a request client loses contact with the server, it should exit with an error message.

Your server will need to implement a *scheduler* to assign workers to incoming client requests. You should design a scheduler that balances loads across all requests, so that the number of workers assigned to each outstanding request is roughly equal. Your code should contain documentation on how your scheduler achieves this requirement.

3.1 Testing and Evaluation

You should test your code on actual hash signatures, varying the number of workers, and handling multiple requests simultaneously. You should also experiment by terminating active workers and starting new ones to demonstrate the robustness of your system. With multicore processors, you should be able to see performance gains even when adding more workers on a single machine. You should be able to set the drop rates for the different components to nonzero values (say 0.10 or 0.20) and still have the system operate successfully.

We will provide automated tests for your system components closer to the due date.