CSCE 438 - Homework 2 Design Document

John Keech and Philip Van Ruitenbeek

February 15, 2013

1 LSP Protocol Design

1.1 Overall Design

The LSP Protocol implementation follows the guidelines laid out in the original handout. It allows for reliable ordering of packets and retransmission to guarantee delivery. It does not compute checksums of the packets, so there is no guarantee on the validity of packets that arrive. The LSP Protocol is split into two components: the server and the client. Each client maintains a single connection to a server, while a server can have zero or more connections from clients. Therefore, the server must keep track of all of the individual clients and communicate with each one separately.

The general design is to send and receive messages as fast as possible using user-level read and write threads in conjunction with an epoch thread. Therefore, each client and server will maintain three additional threads. The epoch thread is built as outlined in the protocol description. The read and write threads utilize inbox and outbox queues and use mutexes for thread synchronization. The underlying network IO is performed asynchronously so that we can make sure the connection stays valid while data is trying to be read. The LSP API, however, is designed so that all calls are blocking, since this makes it easier for the user to implement common programs.

1.2 Common Data Structures

The client and server have many things in common. They both read and write messages in basically the same way, and they both have an epoch thread that provides the reliability. The main difference between the two is that the client only has a single connection, while the server has multiple connections. The details about each connection are identical.

A connection includes a socket to the other end, a status indicator showing the state of the connection at any given time, as well as some other variables for keeping track of the last sequence number sent out and received, the number of epochs that have passed with no communication from the other end, and so on. Each connection also maintains an outbox queue of messages that should be sent to the other end. Since this is common to both the client and the server, all of these details were split into a separate Connection struct that could be used by both the client and the server. The client has a single connection to the server, while the server maintains a map of connections to clients that can be looked up by their connection ID.

In addition to the common connection data structure, each client and server maintains an inbox queue of messages that have been received and parsed. This inbox queue only contains data messages. The connection requests and acknowledgement messages are used internally within the LSP implementation, but the inbox queue stores the messages that are waiting to be processed by the consumer of the LSP API.

1.3 Read Thread

The Read Thread acts the same way for the client and the server. They both listen for incoming messages on the socket, parse them, and handle them according to if they are a connection request, and acknowledgement, or a data packet. If it is an acknowledgement that has not been received yet, the write thread is signalled so that it can send the next message that is pending for that specific connection. since the read thread is always running in the background, messages are received and processed almost as soon as they arrive. In addition, since the write thread is signalled, the next message can be written immediately, so the user does not have to wait until the next epoch occurs for the message to be sent. If you have long epoch times, this design will allow for much faster communication.

1.4 Write Thread

The Write Thread, like the Read Thread, is always waiting in the background and polling the outbox queues for new messages to send. The Write Thread must also keep track of which messages it has previously sent, because we only send each message once (and retransmission is handled within the epoch thread). The Write Thread must also verify that the message at the front of the outbox queue can actually be sent, meaning that we must have already received an acknowledgement for the previously sent message. The speed at which the Write Thread polls for new messages is on the order of milliseconds, which should be much faster than the epoch times in most cases.

1.5 Network

A separate Network class was made to handle the low-level network IO operations, and since the majority of the functions operate on the common Connection objects, this class was accessible to both the LSP Client and LSP Server. Things such as reading and writing to sockets, marshalling and unmarshalling using the Google Protobuf library, and simulated packet dropping were handled by this library.

2 Password Cracker Design

2.1 Overall Design

The design of the distributed password cracker was to split each job into small, fixed-size chunks that can be handled by single-threaded, lightweight worker processes. Each worker only works on one chunk at a time, but it should be very efficient at computing all of the hashes for that chunk. One benefit of the lightweight worker processes are that it gives the user much more flexibility in how they want to run the distributed system. For example, if you have a quad-core processor on a computer that you use for everyday work and internet browsing, you probably don't fully utilize your processing power. Each of our worker threads is designed to fully utilize a single core, which allows the user to choose how much of their processing power they will devote to the password cracking jobs. If they want to use 50% of their processing power, they could spawn two worker processes.

The server handles the bulk of the work distributing and managing the pending jobs. Jobs are partitioned into fixed-size chunks, and chunks are handed out to available workers one at a time. If multiple crack requests are currently pending, one chunk is taken from each job in a round-robin order so that all jobs can be processed concurrently. Lost chunks (ones where the worker who is working on a chunk losses connection or gets killed) are placed at the end of the queue and picked up by another worker later on.

2.2 Server (server.cpp)

The server employs a queue to keep track of crack requests, a queue to keep track of inactive workers, and a map to keep track of each active worker's current crack request. When a crack request is sent to the server, a new crack request is added to the crack request queue. When there are inactive workers and the crack request queue is not empty, the next crack request is popped from the request queue and split in two when the size of the request is bigger than the max request chunk size (hard coded to 500000); the next request chuck is sent to the next inactive worker and added to the active request map, and the other request chuck is added back to the request queue. When a new worker joins the server, it is added to the inactive worker queue. When a worker disconnects from the server, if it was working on a crack request, then its request will be added back to the next request queue. If a request client disconnects from the server, then all of its requests will be deleted and workers will not be sent any more crack requests from the disconnected request client.

2.3 Worker Client (worker.cpp)

The worker client uses the openssl SHA1 library for cracking password attempts. It will check every SHA1 hash in the range (inclusively) that is received from the server until it finds a match or it reaches the end of the range. If it find a match, then it will send the cracked password back to the server, otherwise it will send back a request not found message.

2.4 Request Client (request.cpp)

The request client sends a password crack request to the server and then waits for a response. If the password was found, then it will show the password to the user. If the password is not found, then it will alert the user that the password was "Not Found". The request client does not allow password crack requests for passwords longer than 6 characters.

2.5 Failsafes

We implemented all of the failsafes identified within the LSP Protocol. In addition, we use some NULL return values to indicate that other other party has become disconnected. In specific, the lsp_client_read function will return as soon as the connection becomes disconnected, with a return value of 0 (NULL). This way the client program can gracefully terminate. Likewise, the lsp_server_read method will return NULL when a client disconnects, and it will set the conn_id output parameter's value to the connection ID of the client who disconnected. This will allow the server to gracefully handle that client's departure.

We also implemented some error checking when sending messages. Both lsp_client_write and lsp_server_write verify that the input buffer is not NULL and that the length is not 0. These methods return true on success, and false on failure.

Additionally, when the internal Read Threads in the LSP implementation for the client and server receive an incoming message, it is first unmarshalled into the Google Protobuf LSPMessage object. If unmarshalling fails (due to a malformed buffer in the packet), the message is dropped and the clients never see it.

Finally, even though the LSP API methods implement synchronous (blocking) IO for simplicity, the interals all use asynchronous IO so that the connection state can be continuously monitored. If the connection ever drops during one of these blocking API operations, the method returns the appropriate error code (usually NULL).

Mutexes were used throughout the back-end implementation of the LSP Protocol to ensure thread safety of shared variables.

3 Building and Running the Code

For direction on how to build and run the code, please refer to README.md.