CSCE 662: Distributed Processing Systems

Homework 2

By

Sanutha Venketraman [521-00-5441]
Suvodeep Pyne [619-00-0741]

Table of Contents

Distributed Password Cracker

Objective

To build a failure resistant distributed password cracker system using a Live Sequence Protocol operating on top of UDP

Application Design

Following are the main facets of the application from a design perspective. Each of these are explained in the coming sections.

The Application is divided into the following layers each of which is described in the sections below.

- Socket Layer
- Serializer Layer
- Message Receiver
- Message Sender
- Message Processor
- Worker (This layer only exists in workers)
- Epoch (Keep alive)

The following data structures were used to manage the application

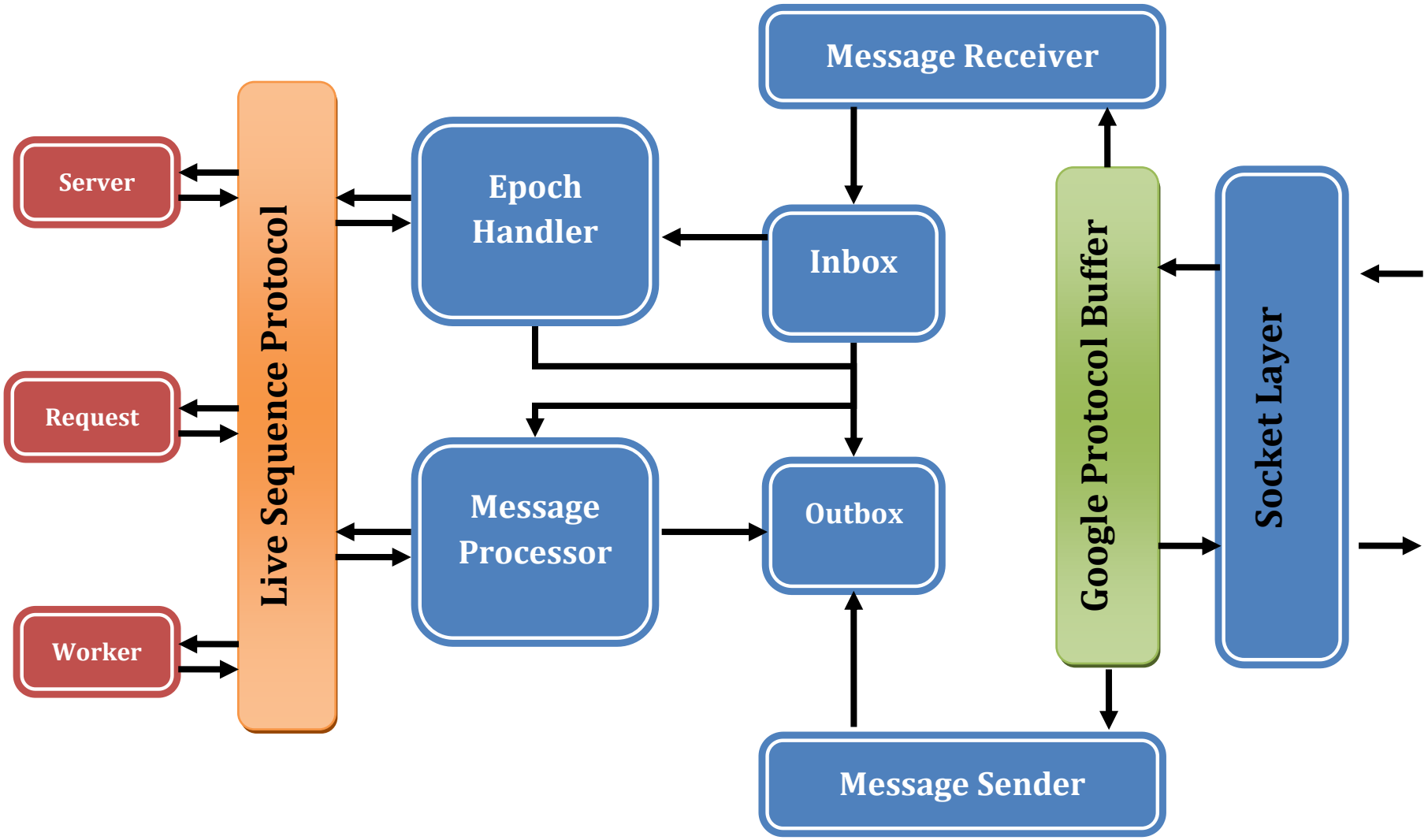
- Inbox queue
- Outbox queue
- Connection Information
- Worker Information

The application is split into the following threads.

- Talker Thread
- Listener Thread
- Processor Thread (main thread)
- Epoch Thread
- Worker Thread (only for workers)

The Distributed Password Cracker is designed as shown in Figure 1.

Figure 1: Application Architecture



Application Layers

Socket Layer

This is responsible for the actual network connection using UDP and IPV4. Receiving and sending messages over the network between the different applications is handled by this layer. This functionality is implemented in code using the Connector class.

Serializer

This layer uses the Google Protobuf Library to serialize the data for transmission across the network. This layer forms the bridge between the socket layer and the Message Receiver and Message Sender Layers

Message Receiver

This layer is responsible for fetching an incoming packet from a socket recvFrom and placing it in the inbox for processing. The Message Receiver module runs in a thread of its own to facilitate asynchronous handling of incoming messages.

Message Sender

This continuously polls the Outbox and checks if any LSPPacket has been added. If yes, it gets the topmost packet and sends it to the socket layer to be sent to the respective client/server.

Epoch Handler

This implements the heart-beat mechanism and keeps the connection alive. After a certain count has passed, it notifies the application that the connection is dead.

At the end of an epoch, it checks if there are any messages in the outbox. If there is, it sends the topmost message again. This is because no acknowledgement has been received from the other side, and it is assumed that the last sent message did not reach.

On the other hand, if there is no message in the outbox, the epoch handler sends a packet of the following format to ensure that the connection is kept alive.

Table 1: Epoch Alive Packet

Connection ID	Sequence Number	Length	Buffer
Client ID	0	0	NULL

Live Sequence Protocol

The LSP provides the interface between the applications and the other layers. The different threads are controlled by and spawned off here. Also synchronization issues are handled using pthread mutexes. The different threads used are:

Listener thread

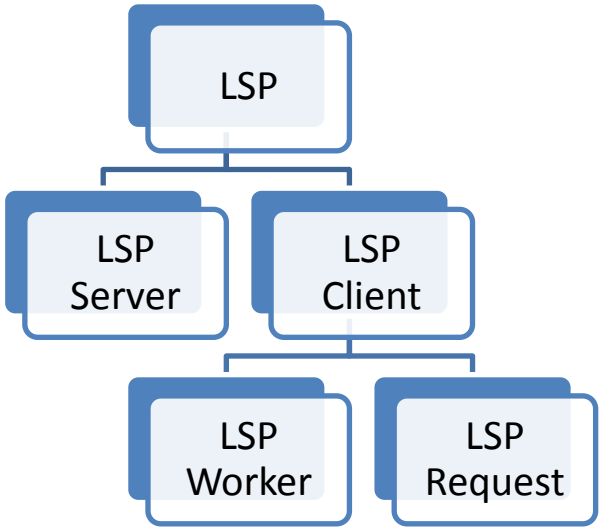
Thread that will keep polling to check if any message comes in over the connection. This connects to the socket layer.

Talker thread

Thread that will keep polling the outbox to check if there is any message that can be sent. This connects to the Message Sender.

Epoch Thread

Thread that will help check the heartbeat, and inform if a connection has terminated.



Message Processor (Per Application)

This part handles the message handling for each of the applications. The class hierarchy is as shown in the figure below. It handles the different incoming packets and takes actions as described below.

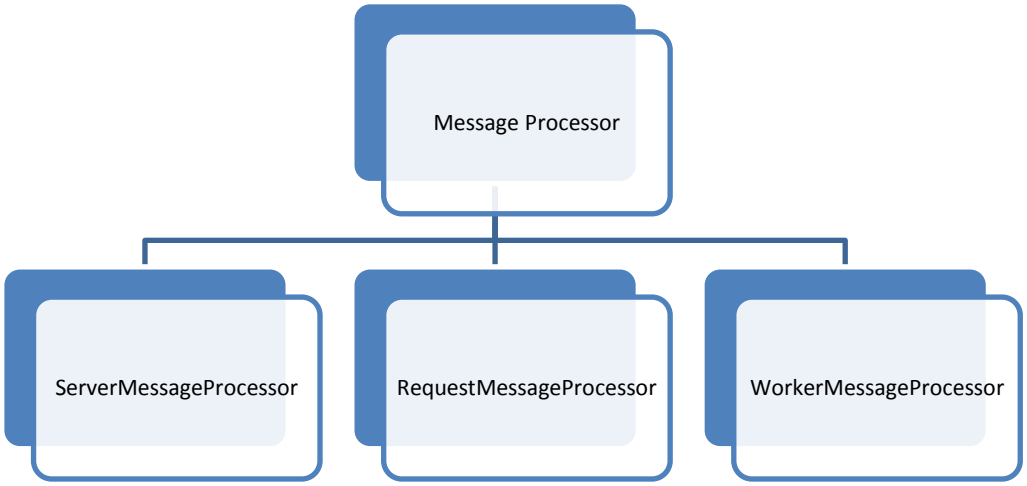


Table 2: Message Processor Hierarchy

Table 3

On receipt of Packet Type	Server	Worker	Request
Connection Request	Store information about the client in ConnectionInfo	NA	NA
Acknowledgement	Check against last message in outbox for that client. If they match, pop message off outbox and set msgSent as False.	If ack is for a connection request, send a join request to server. Check against last message in outbox. If they match, pop message off outbox and set msgSents as false.	If ack is for a connection request, send a crack request to server. Check against last message in outbox. If they match, pop message off outbox and set msgSents as false.
Join Request	Asserts it is from worker Update ConnectionInfo to indicate it is a worker.	NA	NA
Crack Request (Request) - This refers to the crack request that comes to the server from the request.	Assert it is from requester. If length of password < 4: Assign to single least-busy worker If 7 > length of password > 4: Split equally among all least-busy workers. Else: Respond to client as password cannot be processed. Update map with requester Id, and workers assigned for it. Update ConnectionInfo of requester with the request.	NA	NA

Crack Request (Server) - This refers to the crack request that comes to the worker from the server.	NA	Add the request to the queue. The polling thread will find the new request and process it. Send the result of cracking to the server after processing.	NA
Found	Assert message for worker. Remove map entry for particular requester. Send result to requester.	NA	Display the message to the user. Disconnect.
Not Found	Assert message from worker. Update map entry for particular worker. Check if all workers have responded. If yes: Send message to requester. Remove entry from map	NA	Display message to the user. Disconnect.

Data Structures

Inbox

This is a queue into which packets received are put in as soon as received over the connection. There is a single Inbox associated with each of server, worker and request.

Outbox

This is a queue into which packets that are to be sent over the connection are put. The outbox messages are not popped off after sending. They are popped off when one of the following happens:

1) An acknowledgement is received for the top most message in the queue.
2) A data packet with seqNo 1 greater than the top most message is received.
3) A data packet is being added to the queue, and the message before that are ACK messages.

A client has only one outbox for its communication with the server. The server has an outbox for each of the clients it is connected to.

ConnectionInfo

This object stores information about each connection. In the case of a server, this is a vector of objects, where each object stores information about the different clients that connected with the server. In case of clients (request and workers), this is a vector of a single object having details of its connection with the server. The typical details include

1) Connection status – alive or dead.
2) Hostname
3) Port
4) connectionId
5) Outbox for each client, etc.

API Handling

All the API methods have been implemented to support automation testing as instructed. However, the main code of the project does not use these APIs for internal processing. This includes support for the 3 main types of APIs

- Lsp_server side API
- Lsp_client side API
- Epoch API

Finer Details

Sequence Numbers (SN)

Sequence numbers are maintained to make sure that the two ends of a communication receive all messages and also in the correct order. This is done as follows:

- The connection request initiated by the client has SN = 0.
- The join/crack request sent by the client has SN = 1.
- Every acknowledgement has sequence number equal to that of the message it is acknowledging.
- Every time a data packet is sent, the sequence number is incremented.
- Every packet created by epoch has sequence number as 0.

Connection ID

The connection ID is a unique number that identifies each client. This number is in sequence from 1, and they are never repeated.

Optimizations

- When the server receives a crack request, it splits the work between the workers as follows:
  - If 0 < length of password < 4:
    - Assign the task to a single worker as the number of entries to be compared is < 17576.
  - If 3 < length of password < 7:
    - Find upto 5 least busy workers.
    - Split the task equally and assign to the 5 workers.
- Algorithm to find n least busy workers
  - Every worker has a queue of requests which are assigned to it. The workers are sorted based on the length of their queues. The top n ones are chosen as candidates for the task.
- The hash given in the crack request from the request to the server must be of length 40 since it is a SHA1. Therefore, a check is made to make sure this before allocating to workers.

Compiling the Application

$make clean
$make

Using the Application

On compilation, the following binaries are created – server, worker, request.

Running server

```
$ ./server port
Example: ./server 5000
```

Running worker

```
$ ./worker host:port
Example:./worker localhost:5000
```

Running request

```
$ ./request host:port hash len
Example: ./request localhost:5000 81fe8bfe87576c3ecb22426f8e57847382917acf 4
```

Sample Output

```
./request localhost:5000 81fe8bfe87576c3ecb22426f8e57847382917acf 4
Found: abcd

Some random string for SHA
./request localhost:5000 zzzqqqaaasssxxxbbbhhhqqqpppsssiiinnnddkkk 4
Not Found

L > 6.
./request localhost:5000 2fb5e13419fc89246865e7a324f476ec624e8740 7
Not Found

Len = 0
./request localhost:5000 81fe8bfe87576c3ecb22426f8e57847382917acf 0
Not Found

No workers are available
./request localhost:5000 81fe8bfe87576c3ecb22426f8e57847382917acf 4
Not Found

Hash length > 40
./request localhost:5000 81fe8bfe87576c3ecb22426f8e57847382917acf1234 4
```

Efforts by each member

Suvodeep Pyne	**50%**
Sanutha Venketraman	**50%**