CSCE 662
Distributed Systems

Homework 1

Distributed Password Cracker

Johnu George
Sood Nitesh

2/22/2013

This document has been divided into two sections:
Section 0: Configuration and Usage
Section 1: LSP Protocol
Section 2: Distributed password cracker application

Section 0: Configuration and Usage:

LSP Configuration:

The server, worker and requester each read from a configuration file lsp.conf to read in certain system parameters. This enables a users to tune the system according to their needs without needing to re-compile the code. This file is a simple property file containing key value pairs. By default, the file contains the following parameters:

EPOCH_LTH=2.0
EPOCH_CNT=5
DROP_RATE=0.0
LOG_LEVEL=LOG_INFO
ENABLE_LOG_FILE=F

The system expects the file to be in this format ONLY. Please be careful while updating this file.

- EPOCH_LTH refers to the length of the epoch
- EPOCH_CNT refers to the epoch count
- DROP_RATE refers to the drop rate
- LOG_LEVEL refers to the required log level for logging to console. The available values are:
- LOG_DEBUG, LOG_INFO and LOG_CRIT
- ENABLE_LOG_FILE is a parameter that enables logs at all levels to be written to log files with a .log extension. The default value is F, i.e. false. To enable log files, this parameter can be changed to T.

If configuration is changed after starting the server, worker or requester, then a restart to the concerned process is required for the changes to take effect.

Usage:

Before starting any of the server, worker or requester, please modify the lsp.conf configuration

file if needed. The default values are already in the file provided along with the submission.

Examples-

To modify the drop rate from 0.0 to say, 0.2, change the DROP_RATE parameter in lsp.conf to 0.4, i.e.
DROP_RATE=0.4

To modify the epoch length to say, 10secs, change the EPOCH_LTH parameter in lsp.conf to 10.0, i.e.
EPOCH_LTH=10.0

To modify the epoch count to say, 10, change the EPOCH_CNT parameter in lsp.conf to 10, i.e.
EPOCH_CNT=10

To toggle the file logging change the ENABLE_LOG_FILE parameter in lsp.conf to T. By default it is F, i.e. server.log will be created, but nothing will be logged to it. If it is T, then logs at all levels in the system will get logged to the appropriate log file. Change the parameter as shown below
ENABLE_LOG_FILE=T.

After all configuration has been set, start the server, worker, requester as follows:

Server:
./server <port>
example: ./server 4000

Worker:
./worker <server_ip_address>:<port>
example: ./worker 192.168.1.112:4000

Request client:
./request <server_ip_address>:<port> <hash_string> <password_length>
example: ./request 192.168.1.112:4000 8077745fca09288c782f398c25ae2af77d0e4bea 5

Section 1: LSP Protocol

There are three main modules in LSP Protocol - API Handler, Network Handler and Epoch Handler. They are implemented in separate threads, which are started from lsp_server_create for server and lsp_client_create for client.

LSP Server Datastructures

```
struct lsp_server
{
  client_info_map client_conn_info;// map of all connected clients with key as conn_id
  Queue<inbox_struct> inbox_queue; // Server inbox queue
  pthread_mutex_t lock; // Server Mutex
  int next_free_conn_id;// Next free connection Id which can be given to a client
  int socket_fd; //server socket fd
}

typedef struct client_info
{
  int conn_id; //connection Id of each client
  int seq_no; //next sequence number which will be sent to the client
  struct sockaddr_storage addr;//client address
  conn_type conn_state; //defines the current state of the client
  pckt_fmt last_pckt_sent; // last sent packet
  Queue<outbox_struct> outbox_queue; //Outbox queue per client
  conn_seqno_map conn_map;//maintains ack status for each sequence no sent
  bool first_data_rcvd; //flag indicating whether any data is received
  bool first_data_sent; // flag indicating whether any data is sent
  int no_epochs_elapsed;//number of epochs passed since last message received
}
```

LSP Client Data structures

```
struct lsp_client
{
  pthread_mutex_t lock; //Client mutex
  int conn_id; //client connection Id
  int socket_fd; //client socket FD
  int seq_no; //next sequence number to be sent to server
  struct addrinfo* serv_info; //server address information
  conn_type conn_state;//indicates state of the client
  pckt_fmt last_pckt_sent;// Last packet which was sent
  Queue<inbox_struct> inbox_queue;//Inbox queue
  Queue<outbox_struct> outbox_queue;//Outbox Queue
```

```
  conn_seqno_map conn_map;// maintains ack status for each sequence no sent
  bool first_data_rcvd; //flag indicating whether any data is received
  bool first_data_sent; //flag indicating whether any data is sent
  int last_seq_no_rcvd;//sequence no of last packet from server
  int no_epochs_elapsed;// number of epochs passed since last message received
  bool closed;//indicates whether client is closed or not
}
```

Epoch Handler

Timeout is implemented using select system call and it is run in infinite loop. On every timeout, timeout functions are invoked and required actions are taken.

Client

1. If epoch_count is greater than configured epoch count(client hasn't received any packet from the server for last 'epoch count' intervals), take actions for disconnection by notifying the application. Push data message to inbox queue with payload size 0, in order to identify disconnection sequence.
2. If connection state is CONN_REQ_SENT (connection request is not yet acknowledged by server), Resend connection request.
3. If connection state is CONN_REQ_ACK_RCVD (connection request is acknowledged) and last_seq_no_rcvd is zero (no data messages are received from server) , acknowledgment with sequence number 0 is sent.
4. If connection state is CONN_REQ_ACK_RCVD (connection request is acknowledged) and last_seq_no_rcvd is non-zero (atleast one data message is received) , acknowledgment with sequence number of last received packet is sent.
5. If last data message which was sent, is not yet acknowledged, Resend the data packet which was cached in last_pckt_sent.
6. If last packet is acked and outbox queue is not empty, send out the packet

Server

For all clients, check for

1. If epoch_count is greater than configured epoch count (server hasn't received any packet from a particular client for last 'epoch count' intervals), take actions for disconnection of that particular client by notifying the application.
2. If connection state is CONN_REQ_ACK_SENT (server acknowledged the request) and last_seq_no_rcvd is zero (no data messages are received from that particular client), Resend the acknowledgement of connection request to the client.
3. If last_seq_no_rcvd is non-zero , Resend the acknowledgement for most recently data packet, pointed by last_seq_no_rcvd
4. If last data message which was sent, is not yet acknowledged, Resend the data packet

which was cached in last_pckt_sent

5. If last packet is acked and outbox queue is not empty, send out the packet

Network Handler

In this thread, recvfrom is called in infinite loop .On receiving a packet, recvfrom gets unblocked and packet is processed depending on the random value generated.

Client

1. If generated random value is less than configured drop rate, skip processing of received packet and continue for next recvfrm.
2. Unmarshal,Parse the data and validate the contents.

a)If packet received is Connection Response(Pkt-Conn-id != 0 and Pkt-Seq-No is 0 and pkt data is Null)

Store newly obtained connection Id in client structure

Change connection state to CONN_REQ_ACK_RCVD

b) If packet received is Data Acknowledgment (Pkt-Conn-id != 0 and Pkt-Seq-No != 0 and pkt data is NULL)

Set Acknowledgement status to true for that particular sequence no.

c) If packet received is Data Packet (Pkt Data != Null and Pkt- Conn_Id ! =0 and Pkt-Seq-No!=0)

Update last_seq_no_rcvd flag

Push to Inbox queue

Send Data Ack for same data packet

If received packet is classified as any of the above, reset epoch_count = 0.

Error Handling for Client

1. Discard duplicate Data acknowledgements, Connection Requests Acknowledgements(It can be identified from the packet sequence no)
2. Discard packets if connection Id in the packet does not match with already stored id on client data structure (Server has sent packet to a wrong client)
3. Discard packet if sequence number in packet is less than or equal to the last_seq_no_rcvd value. (if data messages received are duplicate or stale messages)
4. Discard any packet if packet parsed cannot be classified to any of the above valid lsp packet category. (Unknown message to the client)

Server

1. If generated random value is less than configured drop rate, skip processing of received packet and continue for next recvfrm.
2. Unmarshal,Parse the data and validate the contents.

a) If packet received is Connection Request (Pkt-Conn-id is 0 and Pkt-Seq-No is 0 and pkt

data is Null)

Create client structure and store client address
Get next available connection id and store in the data structure.
Change connection state to CONN_REQ_ACK_SENT
Send connection response to server.

b) If packet received is Data Acknowledgment (Pkt-Conn-id != 0 and Pkt-Seq-No != 0 and pkt data is NULL)

Set Acknowledgement status to true for that particular sequence no.

c) If packet received is Data Packet (Pkt Data != Null and Pkt-Conn_Id ! =0 and Pkt-Seq-No!=0)

Update last_seq_no_rcvd flag
Push to Inbox queue
Send data ack for same data packet

If received packet is classified as any of the above, reset epoch_count = 0.

Error Handling for Server

1. Discard duplicate Data acknowledgements and Connection Requests
   Duplicate Connection request can be identified by comparing address and port of received packet with stored values for all clients.
2. Discard packets if connection Id in the packet does not match with already stored Id on client data structure (Populated when connection Id is allotted to the client on receiving connection request)
3. Discard packet if sequence number in packet is less than or equal to the last_seq_no_rcvd value. (if data messages received are duplicate or stale messages.)
4. Discard any packet if packet parsed cannot be classified to any of the above valid lsp packet category. (Unknown message to the client)

API Handler

This is handled in main thread and actions are taken depending on the application requirement.
All Api functions are non-blocking functions.
LSP_Server

lsp_server* lsp_server_create(int port)

Creates and returns lsp_server structure.
If server cannot be created, returns NULL

It creates two threads which implements network handler and epoch handler.

int lsp_server_read(lsp_server* a_srv, void* pld, uint32_t* conn_id)

Validation: a_srv not NULL.

If server inbox queue is not empty

Pop the packet,
If it is a data message
Populate connection Id and payload from message
Return length of packet as return value
If it is a disconnection message
Populate connection Id from message and Payload as Null
Return -1
Else if server inbox queue is empty
Return 0

bool lsp_server_write(lsp_server* a_srv, void* pld, int lth, uint32_t conn_id);

Validation: a_srv is not NULL
Conn_Id must correspond to valid client

If last message which was sent from the server to the specified client, is not acknowledged,
Put into outbox queue of the client
Else
Send the packet to the client.

Return false if client with conn_id is not created in the map .ie, it returns false for invalid conn_id passed from the application

bool lsp_server_close(lsp_server* a_srv, uint32_t conn_id);
Validation: a_srv not null
Conn_Id corresponds to valid client

Free client structure for that specified conn_id and erase from the map. Once, it is erased from map, server stops receiving and sending messages from that client.

LSP Client

lsp_client* lsp_client_create(const char* dest, int port);
Client waits for configured epoch lengths to receive connection response from server. Return false if client doesn't get any reply otherwise return a pointer to a struct of type lsp client.
It creates two threads which implements network handler and epoch handler.

Int lsp_client_read(lsp_client* a_client, uint8_t* pld);
If client inbox queue is not empty
Pop the packet,
If it is a data message
Populate payload from message
Return length of packet as return value
If it is a disconnection message
Populate Payload as Null
Return -1
Else if client inbox queue is empty
Return 0

bool lsp_client_write(lsp_client* a_client, uint8_t* pld, int lth);
If last message which was sent from the client, is not acknowledged,
Put into outbox queue of the client
Else
Send the packet to the server.

bool lsp_client_close(lsp_client* a_client);
It -closes socket created for the client.

Section 2: Distributed password cracker application

2.1 Design goal of the server:

The server has been designed to provide a distributed solution to a password cracking request. It can accept multiple cracking requests, handle failures of different components such as death of a request client, death of a worker, etc. The cracking requests are served in order they arrive at the server, and each cracking request is fully assigned before a new one can be started.

The server is supported by the LSP protocol in the lower layers, and by the distributed password cracker in the application layer. This section describes the design and working of the distributed password cracker application in detail.

2.2 Terms used in this document:

1. Request, Cracking request - The cracking request sent by a request client to the server.

2. Application layer in server - The distributed password cracking application that runs on the server.
3. Task, Sub-task - These terms, used interchangeably, refer to the task given to a worker for evaluation. This may or may not be a sub-division of the original crack request, depending on certain conditions.

2.3 Design policy of the application layer:

The aim of the application layer at the server is to serve requests in the order that they arrive at the server. Each request is divided (see section 2.5: Job allocation at server) appropriately and then its sub-parts are dispatched to workers as and when they become available. A key point to note here is that only after **all** sub-parts of a crack request are **assigned** to some worker, a new request is broken up and served. Until any crack request is still partially un-assigned, no new request is served. If a request cannot be served at a particular time, it is cached, to be served later.

2.4 Task selection algorithm:

The application layer at the server follows the below algorithm to choose a new task to be dispatched to a worker, assuming a free worker exists:

```
1.  if any crack request in progress:
2.    for each crack request in progress:
3.      if any sub-task of current request is still unassigned:
4.        dispatch current sub-task to an available worker
5.        return
6.  pick the next request from request cache
7.  divide the picked request into its sub-tasks if required
8.  dispatch a sub-task to a worker
9.  mark this request as 'in progress'
10.  return
```

As evident from the above algorithm, requests which are 'partially completed' are given higher precedence over the new incoming requests. This is due to the server's policy of assigning **all** sub-parts of a request to some worker before processing a cached/incoming request. This, along with the use of a request cache ensures that crack requests are served in order of arrival at the server.

2.5 Job allocation at the server:

The server application divides the incoming crack requests and dispatches each division to a worker if it is free. Based on the length of the password, the range a*-z* is created. So for example if the length of the target password is 5, the range to be searched will be constructed as aaaaa-zzzzz.

Once the overall range is constructed, it is divided if needed. **A further division is not made if the password length is 3 or less**. So for ranges like aa-zz and aaa-zzz the task is not further subdivided, as this can be completed fast by a single worker itself. This optimizes the system as a whole because there are lesser number of messages exchanged between the server and workers if the password length is fairly small.

If the password length is more than 3, the range a*-z* is subdivided as follows. The aim of the division is that there are equal sized divisions created.

For example, if the range to be searched is aaaaa-zzzzz, we divide it into the following divisions:

aaaaa-bzzzz
caaaa-dzzzz
eaaaa-fzzzz
gaaaa-hzzzz
iaaaa-jzzzz
kaaaa-lzzzz
maaaa-nzzzz
oaaaa-pzzzz
qaaaa-rzzzz
saaaa-tzzzz
uaaaa-vzzzz
waaaa-xzzzz
yaaaa-zzzzz

This pattern is followed for all passwords that are more than 3 letters long. As mentioned before, if the password length is 3 or less, no such divisions are created. The range a*-z* as a whole is passed to a worker.

The 13 divisions above are then registered with the server, who will now distribute it among workers as and when they become available. The subdivisions above do not imply that we'll always need 13 or more workers. Even if there is only one worker, it will get each subdivision to work on consecutively. True to the distributed paradigm, the workers have no knowledge about which request they are serving. They only get a particular subdivision to work on, along with the hashed password they need to crack. It is upto the server application to keep track of the subdivisions passed to each worker, store the result returned by them, and combine the results once all subdivisions have been completed.

2.6 Job allocation at the worker:

The worker receives a message of the form:

c hash lower upper

Here, lower and upper could either be a* and z* or the lower and upper bound of a particular subdivision as described in the previous section. Whatever are the lower and upper bounds, the length of the target password can be found by examining the length of the lower or upper bound. This way, the worker is aware of the length of the password. Here the worker performs a similar optimization to the server. If the length of the password is more than 3, then the worker spawns 2 helper threads to do the cracking job. If length of the password is 3 or less, then the worker thread itself performs the cracking. We have kept the number of threads fixed to 2 because it provides enough speeding up for 5 letter passwords, rather than increasing the overhead of maintaining more threads in pursuit of more speed. Again, if the password length is small, no new threads are needed as the main worker thread itself can perform the cracking fairly quickly. Also, dividing the task by 2 gives us clean, equal sized divisions for each thread. This behaviour is illustrated by the following examples:

Input to worker: c hash aaa zzz
Password length is 3, so no threading takes place. Worker thread itself does cracking.

Input to worker: yaaaa-zzzzz
Password length is more than 3, so divide this task into 2 equal sized subtasks: yaaaa-yzzzz and zaaaa-zzzzz. Give each subtask to a new thread.

Once the threads have finished evaluating their respective subtasks, they update a shared memory area with their results. After both threads join, this memory area is examined to see if any of the threads have successfully cracked the password. **The access to the memory area is governed by the use of a mutex**. The consolidated result of the task (such as yaaaa-zzzzz) is then sent back to the server. The server is not aware of any kind of threading done by the worker. It only sends a task, and expects a reply of found/not found from the worker.
The following are the data structures/components used by the application layer:

- request_cache
- request_store
- requests_in_progress
- sub_task_store
- request_divided
- sub_tasks_completed
- sub_tasks_remaining
- worker_to_request

- worker_task

2.6.1 Role of each component:

1. request_cache: It caches requests. If all workers are currently busy serving a request, then the new incoming request is cached in this data structure.
2. request_store: as requests arrive at the server, each hash string is registered into this data structure against the request id. This is just a static store.
3. requests_in_progress: keeps track of the requests that have started but not yet complete.
4. sub_task_store: a static store that records how a request is sub-divided. Requests are divided in different manners based on the length of the password.
5. request_divided: records the division mapping of each request in progress. For example, it records the fact that for request 3, the sub-task 5 is being handled by worker 2.
6. sub_tasks_completed: records the sub-tasks completed for every request.
7. sub_tasks_remaining: records the remaining sub-tasks for each request.
8. worker_to_request: a mapping for worker id to request id.
9. worker_task: a mapping for worker id to sub-task number that it is working on.

2.7 Events handled by the server:

There are 4 high-level events that the application layer at the server handles:

1. Crack request

The server caches the request. It then checks if any workers are free. If they are, then we can be sure that there are no other requests that are partially unassigned, because if there were, then the free workers would have been assigned to them. Thus, if free workers are present, the server proceeds to serve the next request off the cache. If no workers are free, request is just cached and then the server goes back to waiting for the next events/messages.

2. Join request

Since this is a new worker, it must begin working on a new task immediately if such a task is available. The server chooses a new task according to the task selection algorithm of section 2.4. This task is then dispatched to the worker who just joined, and the necessary changes are made to the internal data structures that record this assignment. If no task could be chosen according to the algorithm, then the server just adds the current worker to the list of free workers.

3. Result returned by a worker

There is a fair amount of processing done for this event. First, the server checks if the request that this worker was working on is in progress or not. For example, if worker 2 was working on a sub-task of some request with id 5, and the server determines that request 5 is not in progress anymore. It could ‘not be in progress’ for a couple of reasons - the client who made request 5 could be dead, or some worker which was working in parallel has already returned a PASS result for this request. For these two reasons, a request can be marked ‘not in progress’ prematurely, i.e. before all sub-tasks have completed.

What happens if a request is determined to be ‘not in progress’? In such a case, whatever the result returned by the worker, it is ignored. The worker is then assigned a new task according to the task selection algorithm of section 2.4, or if no task is available, is put back in the free workers list.

What happens if a request is ‘in progress’? The server then checks if the result returned by the worker was a PASS or FAIL. If the result is PASS, then the reply of pass is immediately sent to the appropriate requesting client. The server also marks the current request as ‘not in progress’ in its own data structures. This is because it needs to just ignore the results of the other workers working on some other sub-task of this request. If the result was FAIL, i.e. the worker did not find the password in the sub-task it was assigned, then the server marks this event by updating a data structure (sub_tasks_completed) which records all sub-tasks that are completed. (This is also done for pass results). When the sub_tasks_completed data structure shows that all pre-divided tasks have completed, and a pass has still not been returned, it means that the request was invalid and no password can be found. So if this was the last sub-task to return and it is a FAIL, then the request needs to be marked as ‘not in progress’ and an appropriate reply is sent back to the requesting client. If this was not the last subtask, then the server will only send a reply back to the requesting client once all sub-tasks have been evaluated.

After the appropriate processing for a PASS or FAIL is done, then the server assigns a new sub-task or task to the returned worker according to the task selection algorithm of section 2.4.

4. Dead client

The server first checks if this client is presently in the list of free workers. If it is, then all this handler does is remove the worker from the free worker list, as no more tasks should be assigned to it now.

If it is not in a free worker list, the server checks if this client was previously assigned to some request. If yes, it means that this is a worker. In this case the footprint of this worker is removed from the server’s data structures. The sub-task that this worker was working on is now marked for evaluation again by putting it back into the sub_tasks_remaining data structure. This sub-task can now be taken up by some other worker immediately or later. This feature which gives robustness to the system is described in detail in section 2.8, Reliability and robustness.

If the dead client is a requesting client, then its footprint is removed from all data structures of the server, particularly requests_in_progress. This helps the server in ignoring all results that workers return pertaining to this particular requesting client.

2.8 Reliability and robustness:

The reliability and robustness exhibited by the server is described below. We consider the following scenarios, viz., the death of a requesting client, the death of worker(s), the addition of a new worker in the middle, and submitting requests while others are already running.

Please refer to section ----- (data structures) for the function of each data structure mentioned here.

Death of a request client:

When the server determines that a request client has died, it removes the footprint of this request client from the data structures. The important removals are from requests_in_progress, since this enables the server to ignore the results of sub-tasks pertaining to this request that will arrive in the future. It also removes the division mapping of this particular request from request_divided, since this is not needed anymore. It is possible that the server detected the death of the request client even before this request was popped off the request cache (request_cache). In this case, the server also removes the entry of this request client from request cache, if it is there.

Death of a worker:

The main job of the server in this scenario is to re-assign the task that the worker was working on to another worker. Our server implementation achieves this a little differently in that it does not re-assign the task immediately. It marks the particular sub-task into the sub_tasks_remaining data structure of this particular request. What this means is that, to complete a particular request, all sub-tasks in the data structure sub_tasks_remaining must be complete. So in effect, the task that was just added, is executed again, achieving reliability. This is illustrated in the example below:

At some time t, suppose the sub_tasks_remaining, and sub_tasks_completed data structures for say, request 5, are as follows. (assuming that request 5 was initially divided into 13 sub-tasks)

sub_tasks_remaining for request 5 = [8, 9, 10, 11, 12, 13]

sub_tasks_completed for request 5 = [1, 2, 3, 5]

Now let's say that the server detects that some worker has died and that worker was working on sub-task 4 of request 5. (The server knows what the worker was working on by examining the division mapping in the data structure request_divided). The server then marks the sub-task 4 as 'remaining' again by updating the above data structure as below:

[8, 9, 10, 11, 12, 13, 4]

So the sub-task 4 will eventually get re-assigned by the server to some other worker and get evaluated again. In some corner cases, the sub-task can also be immediately re-assigned. But it is guaranteed that the sub-task 4 will get evaluated again. If the password occurs in the range specified by sub-task 4, the password will eventually get cracked.

The server need not make any change to the data structure sub_tasks_completed, since the sub-task 4 is not completed. Even if all other sub-tasks were completed before it is detected that this worker is dead, the concerned request client will not be notified since the sub_tasks_completed data structure will not have all entries until the sub-task 4 is evaluated again. Until it does not have all entries, the request client is not notified. This of course, is assuming that the password exists in the range defined by sub-task 4. If the password exists in some other sub-task, then the server notifies the client as soon as that reply has come back from the concerned worker.

Addition of workers:

The server handles new workers which join while requests are already running. This is treated as a normal join event, and the server assigns a new task to this new worker according to the task selection algorithm of section 2.4. If no task can be found, then this worker is added to the list of free workers.

Submitting multiple requests:

The server can handle multiple cracking requests. If a new incoming request cannot be served due to no worker being available, then the request is cached, to be served as soon as possible, governed by the task selection algorithm of section 2.4. The point to note is that the server can accept many requests without failing.

2.9 Examples:

The following examples illustrate the flow of the distributed password cracking application:

Example 1: Password to be cracked - zzzzx

1. Server starts
2. Worker 1 joins
3. Worker 2 joins
4. Crack request accepted from requester 3
5. Password length is more than 3, hence sub-divisions are required
6. The search range of aaaaa-zzzzz is divided into sub-tasks as illustrated in section 2.7
7. The sub-tasks are then dispatched to the workers, depending on how many workers are available.
8. On the worker side, each worker spawns 2 threads to evaluate its sub-task. So the worker who gets the sub-task yaaaa-zzzzz, further divides it into yaaaa-yzzzz and zaaaa-zzzzz. One thread will report success while another will report failure.
9. Back at the server, it receives fail for each of the sub-tasks till it gets the reply from the worker who was assigned the sub-task yaaaa-zzzzz.
10. The server sends back a success reply to the request client.
11. The workers can begin a new task belonging to some other request which was submitted in the mean time, according to the task selection algorithm of section 2.4, or if no task is available, they can wait as free workers.

Example 2: Password to be cracked - eat

1. Server starts
2. Worker 1 joins
4. Crack request accepted from requester 2
5. Password length is equal to 3, hence no sub-divisions are required
6. The search range of aaa-zzz is sent as a whole to a free worker, if such a worker is available
7. The worker also detects that the password length is 3, so it does not spawn any new thread for password cracking.
8. The server sends back a success reply to the request client.
9. The worker can begin a new task belonging to some other request which was submitted in the mean time, according to the task selection algorithm of section 2.4, or if no task is available, it can wait as a free worker.

Testing done:

We have tested the distributed password cracker as follows:

- for a variety of test-cases, varying drop rates and varying epoch lengths.
- tested the system on a single host, as well as multiple hosts, using our personal laptops in tandem with the lenss-comp5.cse.tamu.edu host.
- tested the system's robustness by randomly killing and reviving workers, submitting

multiple cracking requests in parallel etc.

Team members and work done:

Johnu George: 50%
Sood Nitesh: 50%
