Version: 0.3

EE122 Project 2A: Scalable *tiny* World of Warcraft

Instructor:

Prof. Ion Stoica (istoica@eecs.berkeley.edu)

TA's:

Junda Liu (liujd@eecs.berkeley.edu)

DK Moon (dkmoon@eecs.berkeley.edu)

David Zats (dzats@eecs.berkeley.edu)

Due: 11:50pm on November 16, 2009 through BSpace

Clarifications

- *Please modify your server to save/read player files in the same format as that used by the provided project 1 reference server.*

Clarifications (Version 0.2)

- *The standard output of the tracker will be used for grading. So, please call the functions provided in tracker_output.h and do not output any other text.*
- *The standard output of the server will be ignored as in project 1.*
- *Players should not move to a new location when respawning.*
- *Please disable the periodic increase in hp.*
- *In STORAGE_LOCATION_RESPONSE, the UDP port is 2B long.*

Overview

The *tiny* World of Warcraft game you developed in Project 1 has become *wildly* popular! However, this popularity has lead to frequent server overloads. To overcome this problem, you decide to revise your client and server to achieve the desired scalability.

Now, multiple servers will be used, where each server is responsible for a certain area in the dungeon. To avoid data consistency issues, the client will fetch the player's state from a specific location and provide it to the appropriate server upon login. As the player moves from one area of the dungeon to the other, the client will ensure that it is always connected to the correct server. Finally, when the player logs out, the client will store the updated state back to the specified location. A tracker will be used to provide the client the information necessary to perform these tasks.

In this project, you will make the server and client modifications required to achieve the previously stated goals. You will also implement the tracker.

Detailed Interactions

Instead of passing the server's IP address and port as command line arguments to the client, we now pass the tracker's IP address and port to the client. Now, when the client receives the login command, it will contact the tracker to obtain the location (IP address/port) where the player's state is being stored. The client will then connect to this address to obtain the player's state. If the specified location does not have the player's state, it will generate it as described in Project 1.

The client will then contact the tracker with the player's coordinates (obtained from the player's state) to get both the address of the server responsible for the appropriate area and the boundaries of that area. The client will then login to the specified server, providing it the player's state.

Whenever a player moves, the client will check to see if the player will stay within the boundaries of the area. If so, the move command is processed in the same way as in Project 1. If not, the client will logout of the current server, contact the tracker to obtain the server responsible for the new location, and login to the appropriate server (again, providing the new server the player's state).

Upon receiving a logout command from the user, the client will logout of the current server. It will then contact the tracker to determine the location where the player's state should be stored and contact that location to update the player's state.

Note: Communication with the Tracker as well as the storage and retrieval of player state will occur over UDP. All other communication will occur over TCP.

Tracker Operation

Upon initialization, the tracker reads the configuration file to determine the number of servers and the areas they are responsible for. Each line in the configuration file has the following format:

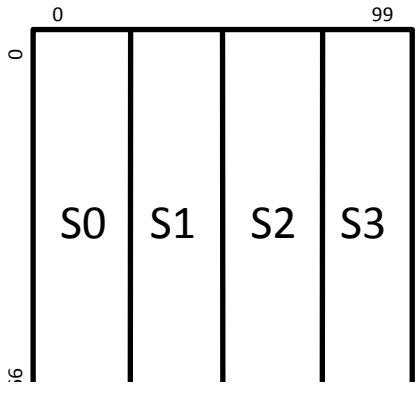
server_ip_address server_tcp_port server_udp_port

For example, the first two lines of the file may look as follows:

132.64.128.56 5454 7777

116.15.34.12 2323 7878

Partitioning will occur by assigning each server responsibility for a portion of the x-axis. So, each server will be responsible for all of the values in the y-axis. For example, if there are four servers, the responsibilities will be assigned as follows:



In the above figure, S0 – S3 represent Server IDs. The specific range of x values for each server will be calculated as follows:

```
Min X = (99/(Total # of Servers))*Server_ID

If (Server_ID == (Total # of Servers – 1)) {

        Max X = 99

} else {

        Max X = (99/(Total # of Servers))*(Server_ID + 1) - 1

}
```

Note: we assume that all of the variables in these calculations are Integers.

In the above calculations, we have assumed that each server is uniquely identified by a Server_ID. Server_IDs are assigned starting from 0 up to (Total # of Servers) -1.

As mentioned in the previous section, the Tracker will also be responsible for informing the client about the location of a certain player's state. The tracker will use the following hash function to calculate the Server_ID (and thus the server) that will be used for storing/retreiving each player's state. This hash function will take the player's name as input.

```
unsigned int hash (char *s)

{

        unsigned int hashval;

        for (hashval = 0; *s != 0; s++)

                hashval= *s + 31*hashval;

        return hashval % (Total # of Servers)

}
```

This hash function was borrowed from page 144 of The C Programming Language by Kernighan and Ritchie.

Version: 0.3

UDP Reliability

As mentioned earlier, the client will use UDP whenever communicating with the tracker or storing/retrieving state. To ensure reliable delivery of UDP messages, we will attempt to send request messages and call the function *on_udp_attempt* up to 4 times. In between these attempts, we will perform exponential backoff starting from 100ms. If we are unable to successfully send the UDP messages after 4 attempts, the client will call *on_udp_fail* and exit. The following shows the events that would occur during such a failure:

```
on_udp_attempt();
sendto();
wait 100ms – no response
on_udp_attempt();
sendto();
wait 200ms – no response
on_udp_attempt();
sendto();
wait 400ms – no response
on_udp_attempt();
sendto();
wait 800ms – no response
on_udp_fail();
exit();
```

Additionally, the client must effectively handle unexpected responses:

- The *on_invalid_udp_source* function should be called if the response comes from an unexpected source.
- The *on_malformed_udp* function should be called for all other types of irregularities.

Other than calling the above stated functions, these responses should be ignored and the impact to the exponential backoff algorithm minimized.

New logic must also be developed for handling the receipt of requests. In addition to handing malformed packets by calling *on_malformed_udp,* the recipient must be capable of identifying duplicate requests and handling them appropriately. Duplicate requests will be identified by keeping track of the source IP addresses and message IDs of the last 50 messages received using UDP. If the source IP address and message ID of a newly received message matches one of the previous ones, it is considered a duplicate. When a duplicate message is detected, the function *on_udp_duplicate* is to be called and the original response message is to be retransmitted. However, no further action is to be taken.

Finally, we will discuss the generation of the 4B UDP message ID. Upon initialization, the client is to randomly generate the starting ID. It is then to increment it for every

transmitted request (but not for retransmissions). Whenever a response is generated, the response is expected to copy the ID of the corresponding request.

Running Programs

Our tracker program will be run with the following command:

./tracker_program –f <configuration file> -p <port>

We must also change how the client program will be run:

./client_program –s <tracker ip> -p <tracker port>

And how the server program will be run:

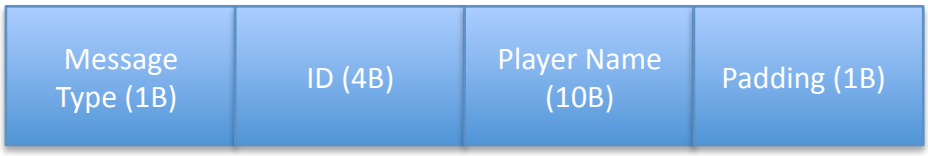
./server_program –t <server tcp port> -u <server udp port>

Protocol Messages

This section shows the messages that must be implemented in order for the server, tracker, and client to properly communicate.

The following set of messages will be transmitted over UDP:

1. STORAGE_LOCATION_REQUEST



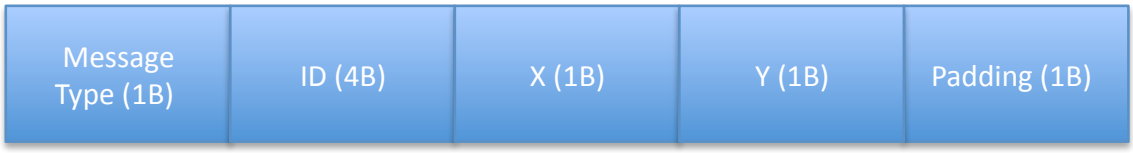
   a. Purpose: Sent from the client to the tracker to request the address of the server responsible for the specified player's state
   b. Message Structure
      i. Message Type (1B) = 0x00
      ii. ID (4B) = Uniquely identifies message
      iii. Player Name (10B) = Name of player whose state we wish to obtain.
      iv. Padding (1B)

2. STORAGE_LOCATION_RESPONSE

   a. Purpose: Sent from the tracker in response to a *STORAGE_LOCATION_REQUEST* message
   b. Message Structure:
      i. Message Type (1B) = 0x01

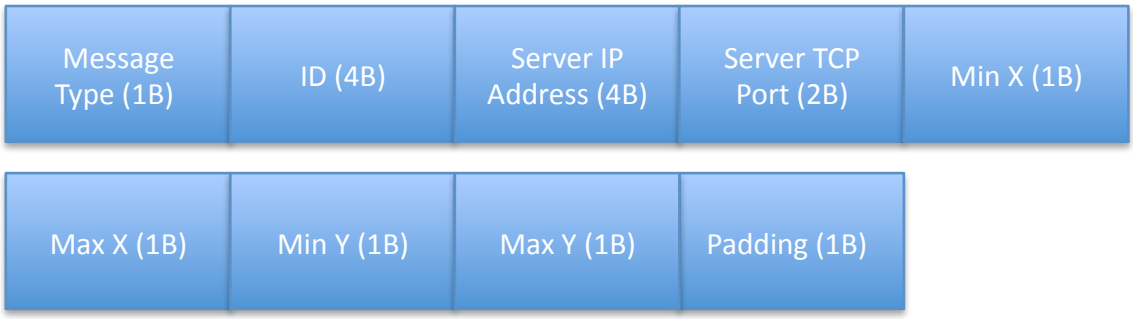
ii. ID (4B) = Copied from corresponding *STORAGE_LOCATION_REQUEST* message
iii. Server IP Address (4B) = IP address of server responsible for storing state.
iv. Server UDP Port (2B) = Port of server responsible for storing state
v. Padding (1B)

3. SERVER_AREA_REQUEST



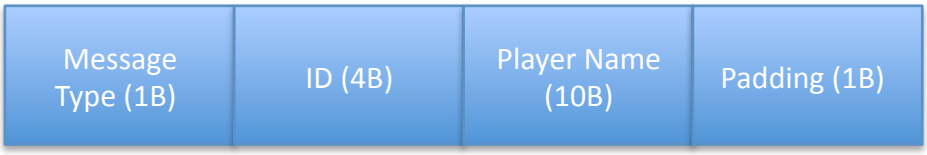
a. Purpose: Sent from client to tracker to request the server responsible for a certain location
b. Message Structure
   i. Message Type (1B) = 0x02
   ii. ID (4B) = Uniquely identifies message
   iii. X (1B) = X coordinate of player's location
   iv. Y (1B) = Y coordinate of player's location
   v. Padding (1B)

4. SERVER_AREA_RESPONSE



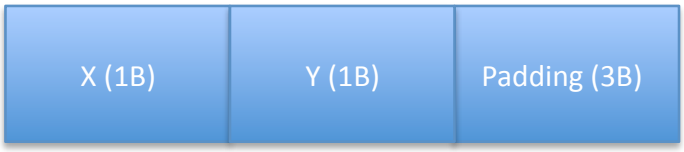
a. Purpose: Sent from tracker in response to a *SERVER_AREA_REQUEST* message
b. Message Structure
   i. Message Type (1B) = 0x03
   ii. ID (4B) = Copied from corresponding *SERVER_AREA_REQUEST* message
   iii. Server IP Address (4B) = IP address of server responsible for requested location
   iv. Server TCP Port (2B) = Port of server responsible for requested location
   v. MinX (1B) = Smallest X value server is responsible for
   vi. MaxX (1B) = Largest X value server is responsible for
   vii. MinY (1B) = Smallest Y value server is responsible for
   viii. MaxY (1B) = Largest Y value server is responsible for
   ix. Padding (1B)

5. PLAYER_STATE_REQUEST



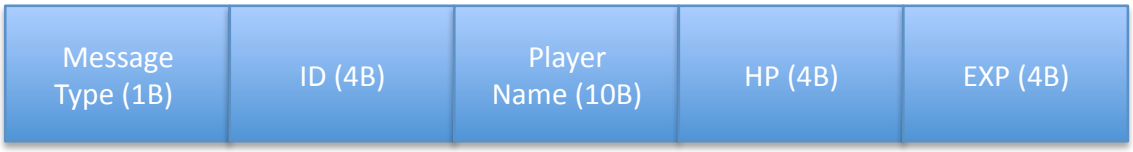
   a. Purpose: Sent from client to the server to obtain the specified player's state
   b. Message structure
      i. Message Type (1B) = 0x04
      ii. ID (4B) = Uniquely identifies message
      iii. Player Name (10B) = Name of player whose state we wish to obtain
      iv. Padding (1B)

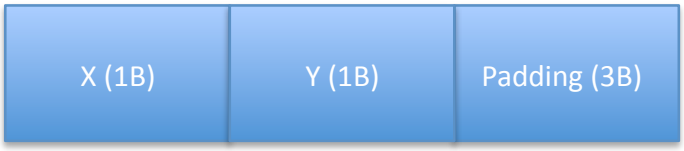
6. PLAYER_STATE_RESPONSE



   a. Purpose: Sent from the server in response to a *PLAYER_STATE_REQUEST* message
   b. Message Structure
      i. Message Type (1B) = 0x05
      ii. ID (4B) = Copied from corresponding *PLAYER_STATE_REQUEST*
      iii. Player Name (10B) = Name of player
      iv. HP (4B) = Health points (as defined in Project 1)
      v. EXP (4B) = Experience (as defined in Project 1)
      vi. X (1B) = X coordinate of player location (as defined in Project 1)
      vii. Y (1B) = Y coordinate of player location (as defined in Project 1)
      viii. Padding (3B)

7. SAVE_STATE_REQUEST





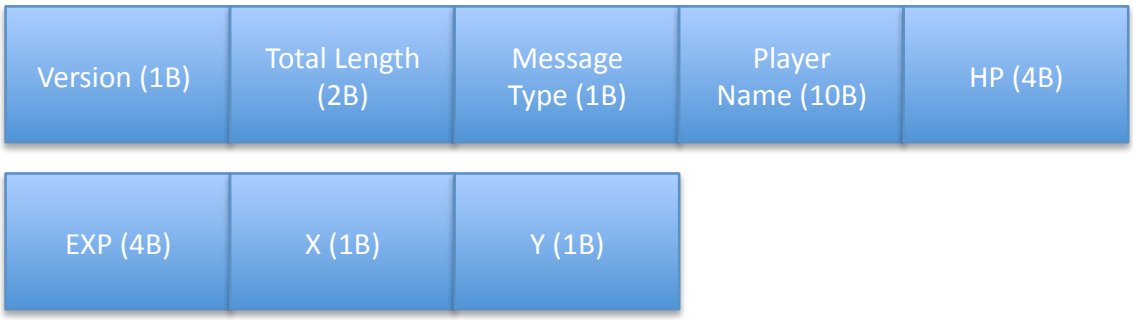
a. Purpose: Sent from the client to the server to save the player's state
b. Message Structure
    i. Message Type (1B) = 0x06
    ii. ID (4B) = Uniquely identifies message
    iii. Player Name (10B) = Name of player
    iv. HP (4B) = Health points (as defined in Project 1)
    v. EXP (4B) = Experience (as defined in Project 1)
    vi. X (1B) = X coordinate of player location (as defined in Project 1)
    vii. Y (1B) = Y coordinate of player location (as defined in Project 1)
    viii. Padding (3B)

8. SAVE_STATE_RESPONSE

a. Purpose: Sent from server in response to a SAVE_STATE_REQUEST
b. Message Structure:
    i. Message Type (1B) = 0x07
    ii. ID (4B) = Copied from corresponding *SERVER_STATE_REQUEST*
    iii. Error Code (1B)
        1. 0x00 = Success
        2. 0x01 = Failure
    iv. Padding (2B)

Lastly, we will modify the structure of the following TCP messages:

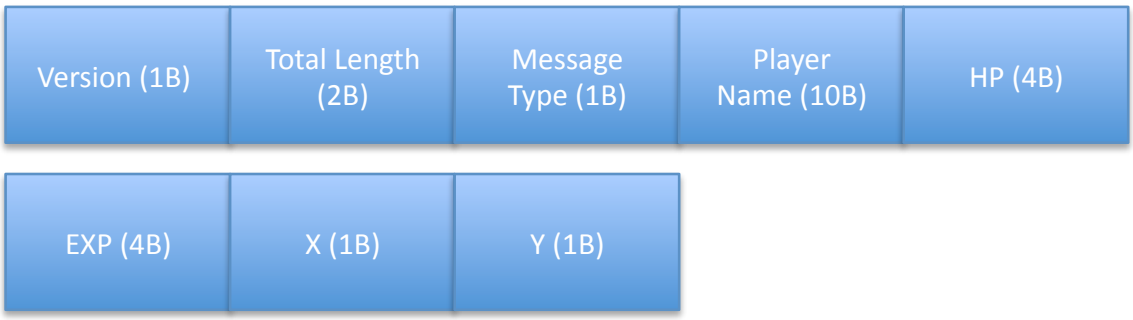
1. LOGIN_REQUEST



a. Message Structure:
    i. Version (1B) = 0x04

ii. Total Length (2B) = 0x18
iii. Message Type (1B) = 0x01
iv. Player Name (10B) = Name of player
v. HP (4B) = Health points (as defined in Project 1)
vi. EXP (4B) = Experience (as defined in Project 1)
vii. X (1B) = X coordinate of player location (as defined in Project 1)
viii. Y (1B) = Y coordinate of player location (as defined in Project 1)

2. LOGOUT_NOTIFY



a. Message Structure:
i. Version (1B) = 0x04
ii. Total Length (2B) = 0x18
iii. Message Type (1B) = 0x0a
iv. Player Name (10B) = Name of player
v. HP (4B) = Health points (as defined in Project 1)
vi. EXP (4B) = Experience (as defined in Project 1)
vii. X (1B) = X coordinate of player location (as defined in Project 1)
viii. Y (1B) = Y coordinate of player location (as defined in Project 1)

Additional Requirements

We will test some additional client logic. Specifically, we will expect that the client will call the function *on_before_login* if the user inputs commands other than login while not logged into a server. Also, we expect that the client will include calls to *on_location_response*, *on_area_response*, *on_state_response*, and *on_save_response* as part of its logic when handling the receipt of appropriate UDP responses. Finally, whenever the client moves close enough to the area boundary that its vision range is impaired (because it cannot see players on other servers), it will call the function *on_close_to_boundary* in addition to *on_move_notify*.

Evaluation Criteria

Version: 0.3

As in Project 1, you are to submit your source code for the client, server, and tracker as well as a brief README. We will compile your source code on x86 Solaris machines and test the resulting binaries for correctness, robustness, and security.

Test scripts as well as reference binaries for the client, server, and tracker will be provided.

Bonus

Only saving state upon logout causes problems when a user plays the game for a long time and the client does not perform a graceful exit (i.e. crashes). Teams who implement logic for auto-saving the player's state every 1 minute will receive extra credit. This new logic should not significantly impact any other client, server, or tracker operations. Please let us know in the README file if you have implemented this functionality.