EE122 Project 1: *tiny World of Warcraft*

Instructor: Prof. Ion Stoica (istoica@eecs.berkeley.edu)
Junda Liu (liujd@eecs.berkeley.edu)
TAs: DK Moon (dkmoon@eecs.berkeley.edu)
David Zats(dzats@eecs.berkeley.edu)

Due: 11:50pm on October 7, 2009 through bspace

1 Overview

As a lead engineer at a company "*Snowstorm*", you have been assigned a task to implement the prototype of its new on-line role playing game. Since you learned from your experience that it is usually a good idea to gradually evolve a project, you decided to first implement its skeleton version "*tiny World of Warcraft (TWW)*". From the skeleton implementation, you are expecting you can prove the client-server framework and its protocol.

2 Basic Game Rules

Details about how the client and the server must behave are given in the next section. This section summaries basic gaming rules of *TWW*.

1. Dungeon size, movement distance, and vision range
   (a) The game has only one 2-dimensional dungeon. It's sized 100x100 and wraps around.
   (b) Each player keeps track of (x, y) location information. The location information is 0-based. So $0 \leq x, y < 100$.
   (c) Each player can move 3 distances toward one of North, South, East, and West. Please recall that the dungeon wraps around. For example, moving toward East from (99, 50) results in a new location (2, 50).
   (d) The vision range of each player is a square whose diagonal points are $(x - 5,\ y - 5)$ and $(x + 5,\ y + 5)$. The range is inclusive. The player can attack any other player in this vision range. For the sake of simplicity, vision range does *not* wrap around.
2. Each player maintains health points(HP), experience points(EXP), and a location in the dungeon.
   (a) The HP ranges from 1 to $2^{31} - 1$. If the HP reaches 0, the player dies.
   (b) The HP decreases when attacked by enemies.
   (c) The player automatically heals at the rate of 1 HP every 5 seconds.
   (d) Each time the player attacks an enemy, the player gains EXP.

(e) EXP ranges from 0 to $2^{31} - 1$.

3. Handling a new player and a continuing player

   (a) New player is given a random location in the dungeon and random HP from 100 to 120.

   (b) Continuing player must be able to start from its previous HP, EXP, and location. This means each player data must be kept in a file.

4. Handling attack and EXP

   (a) When being attacked, the player gets random HP damage from 10 to 20. And the attacker earns as much EXP as the damage.

   (b) If HP becomes 0 or below 0, the player dies. The dead victim resurrects with random HP between 30 and 50 at a random location.

5. User commands

   (a) TWW is a text-based game. It supports the commands in Table 1.

3 Programming Model

After intensive brainstorming with your talented colleague engineers, you highlighted that such a role playing game can tolerate latency variance and thus you drew a conclusion to adopt the client-server model over TCP to benefit from TCP's reliable delivery. (In reality, many games also use UDP. Can you guess why? Hint: try to think why VoIP or streaming video use UDP. Then you can answer what kind of games would prefer UDP to TCP.)

Although every important decision including visibility test should be made by the server for security reasons (Gamers can tweak their clients to cheat), you've decided to put the visibility test at the client side to simplify the server and to more focus on the networking model:

1. The server simply *broadcasts move/attack* messages.

2. The client keeps track of its current location by *LOGIN_REPLY* message and *MOVE_NOTIFY* message with its name.

3. The client also keeps track of the locations of other players from incoming *MOVE_NOTIFY* messages.

4. Based on the location information, it filters out *move/attack* events invisible to the player and denies commands that the player cannot issue. For example, the client does not print out an attack event happened far away, and it does not allow to attack a player not in its vision.

Thus, the server's role is to keep track of client connections and to broadcast network messages, while the client visualizes the network messages from the server and converts client commands into network messages if they are valid within the vision. Please note that such a broadcasting-based[1] approach generally does not scale since the amount of network traffic rapidly increases with

[1]When this term is used throughout the document, it does *not* refer to using IP address 255.255.255.255. Instead, it indicates that the server will send the message to all clients.

the number of players. But it would be OK with this skeleton implementation. The following subsections describe how the client and the server must behave.

3.1 Client program

Client command	Description	Generated message
login <player_name>	Logs in with the player account. If no account, create one and logs in.	LOGIN_REQUEST
move <north|south|east|west>	Moves distance 3 toward the direction.	MOVE
attack <player_name>	Attacks the player. The player must be in sight.	ATTACK
speak <msg>	Broadcasts a speaking message.	SPEAK
logout	Exits the game. The player's state will be saved.	LOGOUT

Table 1: Client commands and matching client packets. Argument in < > is required, while argument in [ ] is optional. Player name does not contain either spaces or tabs, but message for the speak command can do.

1. The client program runs with the following program arguments.
   $ ./client -s <server IP address> -p <server port>
2. If the server is not available and thus the connection cannot be made, the client must exit with the error message **"The gate to the tiny world of warcraft is not ready."**. (TAs will provide *C* functions to generate standardized error messages. You must properly call one of the functions.)
3. After connected to the server, the client shows a prompt **"command>"** and waits for one of the commands in Table 1.
4. If not yet logged in, the gamer can use the command *login*. Player name must be alphanumeric and up to 9 characters. The other commands will be denied by the server with the *INVALID_STATE* message with the error code of 0. (Please refer to the structure of *INVALID_STATE* message in Section 6.)
5. After logged in, the gamer can use *move*/*attack*/*speak*/*logout* commands. Each command sends a matching network message, as specified in Table 1, to the server. If the gamer tries to use the command *login* after logged in, the server will deny the command with *INVALID_STATE* with the error code 1.
6. The client keeps track of the player's location from the *LOGIN_REPLY* message and the *MOVE_NOTIFY* message with its name. In addition, it bookkeeps other players in the game from *MOVE_NOTIFY* messages from the server.
7. The client must filter out the *attack* command with an error message **"The target is not visible."** if the gamer tries to attack a victim not in its vision.
8. The client must not print out a human readable message for *ATTACK_NOTIFY* and *MOVE_NOTIFY* from the server if those network messages involve in a player not in its vision. (Human readable messages to print out are described in Section 6, and TAs will provide *C* functions to print out the messages.)

9. The client must be prepared asynchronous messages from the server. For example, when other player initiates a *speak* command, the client will receive *SPEAK_NOTIFY* message from the server even though the client has not asked for the message. This means the client must not block on user input. Messages from the server are described in Section 6.

10. If the client receives a malformed message from the server, it must exit with an error message **"Meteor is striking the world."**

11. If the connection with the server has terminated for somehow (*e.g.*, the server crashed, the network went down, ...), the client must exit with the error message **"The gate to the tiny world of warcraft has disappeared."**

3.2 Server program

1. The server program runs with the following program argument.
   $ ./server -p <server port>

2. The server must be able to bind the port and must not be affected by the TCP's 2MSL[2] constraint.

3. After binding the port, the server waits for client connections.

4. Each client can log in using the *login* command. Other client commands must be denied with the packet *INVALID_STATE* with the error code 0.

5. Once logged in, the player cannot issue *login*. The server must deny the commands with *INVALID_STATE* with the error code 1.

6. The server must handle malformed messages from clients. Please remember that it is highly likely that gamers tweak the client program for cheating. Thus, sanity check at the server is a very important practice in this environment. In our game, we simply disconnect the client sent a malformed message.

7. The server must handle disconnection from clients. Clients can arbitrarily join and leave the game, and the server must not crash due to one player leaving the game. If a player left the game, it broadcasts *LOGOUT_NOTIFY* to the remaining players.

8. The server must not experience deadlocks. Similarly, it must not block on a single client.

9. The server must save the player data into a file no matter if the player abnormally disconnected or disconnected by the *logout* command.

4 Project Organization

The first part of the project is to implement a client program that can communicate with a reference server provided by your TAs. You will later implement your own server working with your client as the second part of the project.

[2]This specifies how long TCP must wait before re-using a connection. You can overcome this by using the function *setsockopt()* and specifying that addresses can be reused.

The client is relatively simple. It 1) translates a client command into a network message to the server, 2) converts a network message from the server to a human readable text, and 3) performs vision range check to filter out client commands and server messages which involve players not in the vision. Despite its simplicity, this client programming will be a good exercise to learn how to use a TCP socket. To test your client, the TAs will provide a reference server binary to which you can connect, and a bot client binary which controls bot players.

The second part of the project is to implement the server. As opposed to the client, the server is generally more complex because it must handle multiple clients without blocking and resource leak. Further information on the server will be provided when the second part is announced.

5 Evaluation Criteria

This section describes only the client evaluation criteria. The server's evaluation criteria will be announced later.

TAs will evaluate only your binary executables on the following criteria.

1. You have to deliver source code (no executables) and a brief README explaining what you have done. The submitted project needs to compile and run on the Unix instructional machines (x86 Solaris). You can find a list of available machines from this link:
   `http://inst.eecs.berkeley.edu/cgi-bin/pub.cgi?file=ee122.help`

2. *Correctness*: your client must behave as specified in Section 6. This means your client must generate a correct network message for each command and must not send malformed network messages to the server. In addition, you must correctly prints out network messages from the server if it conforms the visibility constraint.

3. *Robustness*: your client must be able to handle socket errors without crash. Please recall that the client can fail *connect()*, *recv()*, and *send()*.

4. *Security*: you client must avoid buffer overflow and memory leak. TAs will intentionally inject a malformed message to exploit them.

You will be provided a server binary and a test script to test your client. You can start testing by manually running your client and server, and type in commands to see if your client works well. Then you can run the test script for automatic evaluation. The test script will be updated to include more test cases, so you should always use the latest version when TAs announce. To use test script, put both client and server binary executables and test script in the same folder, and run

$ python test1a.py <client binary>.

You need python version 2.4 or higher installed. <client binary> is the file name of your client binary. If it's not provided, the test script will use "client" as default.

In addition, TAs will provide *C* functions, in a *C* header file "message.h", to generate error messages. This is to standardize error message format and to prevent you from getting penalty due to simple message format difference. So, you must call an appropriate function from them.

6 Protocol Details

Since *tiny world of warcraft (TWW)* is implemented on top of TCP/IP, the overall protocol layout is like Table 2. Subsection 6.1 describes the TWW header and the remaining subsections details payload for each message type.

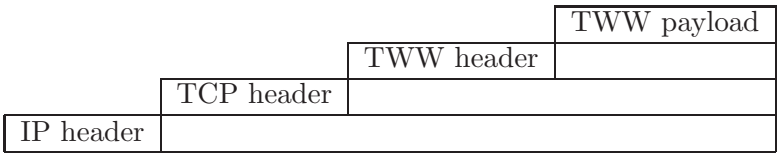

Table 2: Protocol layout. Sending a message over TCP/IP socket automatically prepends a TCP header and an IP header. Thus, you will implement the TWW header and TWW payload.

6.1 Header format

As we studied from the socket programming lecture, TCP treats all messages as a single bytes stream. Thus, we need to prepend a message header to specify the length of each message. Thus, *TWW* has the following message header format. Please note that it has a length field to retrieve exactly one message from the bytes stream. Numbers in parenthesis specify field widths in bits.

⟸ 32 bits ⟹		
Version (8)	Total length (16)	Msg type (8)
Per-msg payload		
...		

Table 3: TWW header structure. You can figure out the length of each TWW message from the length field in the header.

1. Version: 8 bits. Specifies the protocol version. This is currently fixed to 4.
2. Total length: 16 bits in network-byte endian. Specifies the total length of the message in bytes, including the header.
3. Msg type: 8 bits. Specifies the message type. Message types are listed in the subsequent sections.
4. Per-msg payload: variable length. It depends on the message type.

6.2 LOGIN_REQUEST message

Player name (80)	
Player name (cont'd)	
Player name (cont'd)	Padding (16)

Table 4: The payload structure of *LOGIN_REQUEST*

1. Message type number = 0x01
2. From the client to the server.
3. Player name: 80 bits. The name must be null-terminated. Thus, the effective maximum name length is 9 bytes. The name must contain only alphanumeric characters.
4. Padding: 16 bits. It's to make the message 32-bit aligned. This is a common practice in network programming. IP/TCP/UDP headers are all 32-bit aligned.
5. The server's reaction:
   (a) If the client sent this packet after the login phase, the server sends *INVALID_STATE* message with the error code of 1.
   (b) If there's a player already logged in with the same name, the server sends back *LOGIN_REPLY* with the error code of 1.
   (c) If there's no player data file for the given name, the server creates a player data file, containing the player type, random HP from 100 to 120, 0 EXP, and a random location.
   (d) The server loads the player data from the file, and sends back *LOGIN_REPLY* with the error code of 0.
   (e) The server sends each player in the dungeon *MOVE_NOTIFY* with the new player's name.
   (f) The server sends the new player *MOVE_NOTIFY* with each of already existing players. This is to have the new player learn existing players' information.

Example: *LOGIN_REQUEST* message to login as a player "ee122":

Version (8) = 0x04	Total length (16) = 0x0010		Msg type (8) = 0x01
e	e	1	2
2	0x00	0x00	0x00
0x00	0x00	Padding = 0x00	Padding = 0x00

6.3 LOGIN_REPLY message

Error code (8)	HP (32)		
# HP (cont'd)	EXP (32)		
# EXP (cont'd)	X (8)	Y (8)	Padding (8)

Table 5: The payload structure of *LOGIN_REPLY*

1. Message type number = 0x02
2. From the server to the client.
3. Error code: 8 bits.
   error code = 0: **"Welcome to the tiny world of warcraft."**
   error code = 1: **"A player with the same name is already in the game."**
4. The client's reaction
   (a) Prints out a matching text message above.
   (b) Initializes the player information from the message only when the error code is 0.
   (c) Treats it as a malformed message if it receives another *LOGIN_REPLY* after succeeding the login procedure.

Example: when there's already a player with the same name in the game, *LOGIN_REPLY* must looks like:

Version (8) = 0x04	Total length (16) = 0x0010		Msg type (8) = 0x02
Error code (8) = 0x01	0x00	0x00	0x00
0x00	0x00	0x00	0x00
0x00	0x00	0x00	Padding = 0x00

6.4 MOVE message

Direction (8)	Padding (24)

Table 6: The payload structure of *MOVE*

1. Message type number = 0x03
2. From the client to the server.
3. Direction: 8 bits. 0 = North, 1 = South, 2 = East, 3 = West.
4. The client must keep track of the player's location before sending this message.
5. The server's reaction:
   (a) Updates the player's location. (Note multiple players can be at the same location.)
   (b) Broadcasts *MOVE_NOTIFY* with the player name.

Example: *MOVE* toward West must looks like:

Version (8) = 0x04	Total length (16) = 0x0008		Msg type (8) = 0x03
Direction (8) = 0x03	Padding = 0x00	Padding = 0x00	Padding = 0x00

6.5 MOVE_NOTIFY message

Player name (80)		
Player name (cont'd)		
Player name (cont'd)	X location (8)	Y location (8)
HP (32)		
EXP (32)		

Table 7: The payload structure of *MOVE_NOTIFY*

1. Protocol type number = 0x04
2. From the server to the client.
3. Player name: 80 bits. The name must be null-termianted. So the maximum name length is 9 bytes.
4. The client's reaction
   (a) The client keeps track of players in the game.
   (b) If the moving player was not is sight and the new location is also invisible, simply ignores the message.
   (c) Otherwise, prints out a text message
   "<*player name*>: location=(x, y)**, HP=<*HP*>, EXP=<*EXP*>."**

Example: if the player "ee122", with 16 HP and 32 EXP, moved to $(8, 9)$, the server will send a message like this:

Version (8) = 0x04	Total length (16) = 0x0018		Msg type (8) = 0x04
e	e	1	2
2	0x00	0x00	0x00
0x00	0x00	X location (8) = 0x08	Y location (8) = 0x09
HP (32) = 0x00000010			
EXP (32) = 0x00000020			

Say, my location is $(80, 90)$. Then, my vision range is a square whose diagonal points are $(75, 85)$ and $(85, 95)$. Because "ee122" is out of my sight, the client must ignore the message.

If my location is $(13, 14)$, then my vision range is from $(8, 9)$ to $(18, 19)$ and the client must prints out a message **ee122: location=**$(8, 9)$**, HP=32, EXP=16.**

6.6 ATTACK message

Victim name (80)	
Victim name (cont'd)	
Victim name (cont'd)	Padding (16)

Table 8: The payload structure of *ATTACK*

1. Message type number = 0x05
2. From the client to the server.
3. Victim name: 80 bits. The name must be null-termianted. Thus, the maximum length of the name is 9 bytes.
4. The client must not send this message if the victim is out of sight. Instead it prints out **"The target is not visible."**
5. The client must not send this message if the victim is itself. It simply ignores such a case.
6. The server's reaction:
   (a) If the victim is the same as the attacker, ignores the message.
   (b) If there's no player with the given victim name, ignores the message.
   (c) Computes a random damage from 10 to 20. If the damage is larger than the victim's HP, sets the damage to the victim's HP.
   (d) Subtracts the damage from the victim's HP, and adds the amount to the attacker's EXP.
   (e) Broadcasts ATTACK_NOTIFY with the attacker's name, the victim's name, the damage, and the victim's updated HP.
   (f) If the victim is dead (i.e., the HP became 0),
      i. Sets the victim's HP to random value between $30 - 50$.
      ii. Sets the victim's location randomly. (This will issue a following *MOVE_NOTIFY* message because we are updating the player's location, too.)

6.7 ATTACK_NOTIFY message

Attacker name (80)	
Attacker name (cont'd)	
Attacker name (cont'd)	Victim name (80)
Victim name (cont'd)	
Victim name (cont'd)	
Damage (8)	HP (32)
HP (cont'd)	Padding (24)

Table 9: The payload structure of *ATTACK_NOTIFY*

1. Message type number = 0x06
2. From the server to the client.
3. Attacker name: 80 bits. The name must be null-terminated.
4. Victim name: 80 bits. The name must be null-termianted.
5. Damage: 8 bits. Specifies the amount of HP the victim lost.
6. HP: 32 bits. Specifies the updated HP of the victim in network-byte-order. Cannot be negative.
7. The client's reaction
   (a) If both the attacker and the victim are in sight, prints out a text message.
      i. If the updated HP is not zero, prints out a message in this form **"<*attacker name*> damaged <*victim name*> by <*damage*>. <*victim name*>'s HP is now <*HP*>."**
      ii. If the updated HP is zero and the victim is dead, prints out a message in this form **"<*attacker name*> killed <*victim name*>."**

Example: if the player "cal" damaged "stanfurd" by 20 HP, and the stanfurd's HP became 1, *ATTACK_NOTIFY* from the server must look like:

Version (8) = 0x04	Total length (16) = 0x0020		Msg type (8) = 0x06
c	a	l	0x00
0x00	0x00	0x00	0x00
0x00	0x00	s	t
a	n	f	u
r	d	0x00	0x00
Damage (8) = 0x14	HP (32) = 0x00000001		
HP (cont'd)	Padding = 0x00	Padding = 0x00	Padding = 0x00

6.8 SPEAK message

Msg (variable length)	
Msg (continued)	Padding (variable)

Table 10: The payload structure of *SPEAK*

1. Message type number = 0x07
2. From the client to the server.
3. Msg: variable length. Specifies the message to send. It must be null-terminated, printable characters and cannot longer than 255 bytes.
4. The server's reaction:
   (a) Broadcasts *SPEAK_NOTIFY* with the broadcaster name and the message.

Example: if said "Go, bears!", *SPEAK* must look like:

Version (8) = 0x04	Total length (16) = 0x0010		Msg type (8) = 0x07
G	o	,	
b	e	a	r
s	!	0x00	Padding = 0x00

6.9 SPEAK_NOTIFY message

Broadcaster name (80)		
Broadcaster name (cont'd)		
Broadcaster name (cont'd)	Msg (variable length)	
Msg (cont'd)		Padding (variable)

Table 11: The payload structure of *SPEAK_NOTIFY*

1. Message type number = 0x08
2. From the server to the client.
3. Broadcaster name: 80 bits. It must be null-terminated.
4. Msg: variable length. It must be null terminated and cannot be longer than 255 bytes.
5. The client's reaction
   (a) The client must show a text message in this format:
   "<*name*>: <*msg*>"

Example: if the player "cal" said "Go, bears!", *SPEAK_NOTIFY* must look like:

Version (8) = 0x04	Total length (16) = 0x001c		Msg type (8) = 0x08
c	a	l	0x00
0x00	0x00	0x00	0x00
0x00	0x00	G	o
,		b	e
a	r	s	!
0x00	Padding = 0x00	Padding = 0x00	Padding = 0x00

6.10 LOGOUT message

1. Message type number = 0x09. No payload.
2. From the client to the server.
3. The server's reaction:
   (a) Stores the player data in its file.
   (b) Disconnects the player.
   (c) Broadcasts *LOGOUT_NOTIFY*.

Example:

Version (8) = 0x04	Total length (16) = 0x0004	Msg type (8) = 0x09

6.11 LOGOUT_NOTIFY message

Player name (80)	
Player name (cont'd)	
Player name (cont'd)	Padding (16)

Table 12: The payload structure of *SPEAK_NOTIFY*

1. Message type number = 0x0a.
2. From the server to the client.
3. Player name: the exiting player's name. It must be null-terminated.
4. The client's reaction:
   (a) Prints out **"Player <*player name*> has left the tiny world of warcraft."**

Example:

Version (8) = 0x04	Total length (16) = 0x0008		Msg type (8) = 0x0a
c	a	l	0x00
0x00	0x00	0x00	0x00
0x00	0x00	Padding = 0x00	Padding = 0x00

6.12 INVALID_STATE message

Error code (8)	Padding (24)

Table 13: The payload structure of *INVALID_STATE*

1. Message type number = 0x0b
2. From the server to the client.
3. Error code: 8 bits.
   error code = 0: **"You must log in first."**
   error code = 1: **"You already logged in."**
4. The client's reaction
   (a) the client prints out a matching text message above.