Lab #5

Submit Assignment

Due Dec 9 by 11:59pm **Points** 100 **Submitting** a website url

Networking Lab - Game of War

Pre-requisites

Review 11/30 lecture and read through Microsoft's Socket documentation. **Sockets | Microsoft Docs** (https://docs.microsoft.com/en-us/dotnet/framework/network-programming/sockets) , including the Socket examples. Here's also a **Python Networking Reference for any one interested** (https://realpython.com/python-sockets/) . This assignment will require synchronous sockets, though you're welcome to do it asynchronously if you desire.

This assignment will require you to write an implementation of a server for a very simple stateful network protocol in C# or Python. You will implement the server and game logic. You will be provided with two different clients for testing (Python and C#). Your server will be expected to speak the protocol correctly to each client and technically should work with any client I were to test it with, using any programming language.

WarGame-Sync-Starter.zip

WAR: A Card game

For this assignment, we will be programming a cross-platform implementation of the “war” card game server.

WAR: The simplified rules

The simplified rules for our version of war are as follows: the dealer (server) deals half of the deck, at random, to the two players (clients). Each player “turns over” (sends) one of their cards to the server, and the server responds to each player “win” “lose” or “draw.” Unlike normal war (as this assignment is about network programming not video game programming), in the event of a tie neither player receives a “point” and play simply moves on to the next round. After all of the cards have been used, play stops and each client knows (based on the number of points they received) whether they won or they lost. Once each player has received the results of 26 rounds, they send the QuitGame command to the server and disconnect.

WAR: The message format

All WAR game messages follow the WAR message format. Each type of message has a fixed size, but not all messages are the same size. Each message consist: a one byte "message size", followed by a one byte “command”, followed by either a one byte payload or a 26 byte payload. The command values map as such:

command	value
want game	0
game start	1
play card	2
play result	3
quit game	4

For want game, play card, and play result, the payload is one byte long. For the “want game” and "quite game" messages, the “payload” should always be the value 0.

For the “game start” message (where the payload is a set of 26 cards), the payload is 26 bytes representing 26 cards. The byte representation of cards are a mapping between each of the 52 cards in a standard deck to the integers [0..51]. Mapping cards follows the suit order clubs, diamonds, hearts, spades, and within each suit, the rank order by value (i.e. 2, 3, 4, … , 10, Jack, Queen, King, Ace). Thus, 0, 1, and 2 map onto 2 of Clubs, 3 of Clubs, 4 of Clubs; 11, 12, 13, and 14 map onto the King of Clubs, the Ace of Clubs, the 2 of Diamonds, and the 3 of Diamonds; and finally 49, 50, and 51 map onto the Queen, King, and Ace of Spades. Note that you cannot compare card values directly to determine the winner of a “war” - you’ll need to write a custom comparison function which maps two different card values onto win, lose, or draw.

When sending a “game start” message, the server sends half of the deck, at random, to each player by sending 26 bytes with the values [0..51] to one player and the remaining 26 to the other player.

When sending a “play card” message, the client sends one byte which represents one of their cards. Which card to send is undefined, but you cannot send the same

card twice within the same game.

Within a “play result” message, the one byte payload values map as such:

result	value
win	0
draw	1
lose	2

WAR: the network protocol

Parallel to the simplified rules, the WAR protocol is as follows. A war server listens for new TCP connections on a given port. It waits until two clients have connected to that port. Once both have connected, the clients send a message containing the “want game” command. If both clients do this correctly, the server responds with a message containing the “game start” command, with a payload containing the list of cards dealt to that client.

After each client has received their half of the deck, the clients send messages including the “play card” message, and set the payload to their chosen card. After the server has received a “play card” message from both clients, it will respond with a “play result” message to both clients, telling each one whether they won or lost. This process continues until each player has sent all of their cards and received all of their play results; after receiving the last play result, the clients disconnect; after sending the last play result, the server also disconnects.

For War v.1, your server and client only need to play one game, and then exit.

Template

For this lab, I wrote a ClientSocket and ServerSocket solution using Microsoft's documentation code for synchronous sockets. Your job is to complete an implementation of the game server program using the ServerSocket code.

Provided (Python) Client

The provided python program also implements a client, using the built in python event loop. This client can be run one at a time, so that you could e.g. run two terminals that each run `python war.py client 127.0.0.1 11000`, and as long as you have a server listening on `127.0.0.1:11000`, each client will connect and play a game of war. Upon correct operation, the provided client outputs literally nothing. It does, however, have several log messages in the source.

The built in event loop is a new feature introduced in Python 3.5, which enables event driven coding without having to write callbacks or use any extra libraries. This allows writing code in a synchronous style that all runs within one thread. Throwing debug statements in that code is a great idea.

The provided client will run one client with the argument `client`. It can also run an arbitrary number of clients with the first argument `clients`, and an additional argument that is the number of clients to run. You can use this to stress test your implementation, for instance by launching 1000 clients with the command `python war.py clients 127.0.0.1 11000 1000`.

Handling multiple clients

Most servers are only useful if they can serve multiple requests simultaneously. You will get basic credit for handling two clients playing one game, but the remainder of the credit will come from serving multiple games at the same time. It is up to you to choose how to implement such a server.

Grading

If at any time a client sends an *incorrect* message, your code should close the connection for BOTH clients, but otherwise continue operating.

Your server should not exit when anything unexpected happens, but rather close client connections and continue accepting new connections and supporting current games.

Except in exceptional cases, all games should complete in well under **one second**. Any games taking more than one second to complete will be treated as broken / non-functioning.

Scoring

task	points
Successfully run a game of war between two correctly functioning clients	60
Successfully handle an incorrect functioning client, with one game running	15
Run multiple games simultaneously	25