HW1 – Computer Networks

Yael Lerech – 308327972 - yaellerech@mail.tau.ac.il

Offek Gil – 308315092 - offekgil@mail.tau.ac.il

The Program Structure

Our program implements a Nim game between a single server and a single client (at a time).

The python files are:

1. **nim-server.py** - The server side, that manages the game against clients. The server needs to be run first, before running any client.
   To run the server, you need to give it 3 int parameters as the 3 nim heap sizes. You may also give a 4th argument as the wanted port number (default is 6444).
2. **nim.py** – The client side, that connects to an already running server. The client side manages contact with a user that plays the nim game against the server.
   Running the client with no arguments , will by default connect the client to localhost as server, with port 6444. You may run the Client with a hostname/IP address of your choosing (as server side) and you may also run with both a host argument and a port number argument**.**
3. **shared_global.py** – A shared file used both by the client and server. Contains shared enum values for different messages sent between the two sides, as well as the commonly used send_all and recv_all methods. This file does not need to be run on it's own, but needs to be in the same directory as where you run the client and server sides.

The Protocol

1. To run the game, a nim-server.py file must be run, with 3 integers between 1 and 1000 as the 3 heap sizes, and an optional port number. The server will open a socket, bind the given port, and listen and wait for an incoming connection.
2. Once the server is up, you may run the client nim.py from the same device, or a different device – in which case a hostname of the server must be given, and optionally a shared port number, is the server's port is not the default 6444.
3. **Gameplay:**
   a. Once both sides are running, the server will immediately send the client the 3 heap values, together with the current game status, and will continue doing so each iteration of the game, until it ends. These values are sent as 4 integers in a byte object made with struct.pack().
   b. The first 3 integers, represent the values of heaps A, B, C, by order. Once receiving the message, the client will print the heap values to the user.

c. The game status that is sent, can be one of 3 options - PLAYERS_TURN, SERVER_WINS, PLAYER_WINS. Each having a relevant integer value, shared by the client and server in shared_global.py, of 10, 11, 12, respectively.
d. The game will proceed for each side, according to the status that was sent –
    i. **PLAYERS_TURN** –
        1. The client will print "Your turn:" to the user and wait for him to enter a game move.
        2. A legal game move consists of a large letter A, B or C, followed by any number of spaces, and then a positive integer of value smaller then the chosen heaps current value (that was printed to the user). Alternatively, the user may enter a large Q, to quit the game and immediately close the connection on both sides, once the move is sent to the server.
        3. The client will then check that the entered move is of legal format (and A,B,C or Q, followed by any integer), and send a formatted byte object to the server, containing said move.
        4. What is sent, is two numbers. The first, is the relevant value set in shared_global.py, of the enums - HEAP_A(0), HEAP_B(1), HEAP_C(2), ILLEGAL_HEAP_INPUT(4), QUIT(5).
           A legal move will send the relevant heap enum value, together with the number the user asked to subtract from that heap.
           Any identified illegal move, will result in ILLEGAL_HEAP_INPUT being sent, with the integer 0.
           Q, will result in the QUIT enum value to be sent, with number 0.
        5. Once the server fully receives the client's message, it will check to see if the move is legal by the game rules, and enact it on the heaps. Then it will check if the client has won (all heaps are 0), and change it's inner game status accordingly.
        6. The server will the send back to the client one of two coded messages accordingly, with values set in shared_global.py. If the move is illegal, PLAYER_ILLEAGL_MOVE(20) will be sent. If it is legal, then the wanted value will be subtracted from wanted heap, and PLAYER_MOVE_ACCEPTED(21) will be sent.
        7. The client prints the received message to the user, or an error message if an unidentified value was received. Afterwards the client' s game loop starts over.
        8. After sending the message, the server will play it's own move on the heaps, and check if it has won (all heaps are 0), and change it's inner game status accordingly. Afterwards the server's game loop starts over.
    ii. **SERVER_WINS, PLAYER_WINS** – On the client side, a relevant message announcing the winner will be printed to the user ("Server win!" or "You win!"). Afterwards, the client will close its socket connection and finish running. On the server side, the connection will close, and the server will go back to waiting for a new connection.
e. Throughout the program, each side checks to see if the other side closed its connection, while sending and receiving messages. In case this happens, the client will print a relevant message.

Program Methods

1. **nim-server.py**
    a. **start_server():**
    This function runs when the server starts, and has no return value. It uses the arguments for the server program and sends them to the nim_game_server() function.
    Also wraps the nim_game_server() function in case of an error and prints "connection failed" in case of OSError-ECONNREFUSED error.
    b. **nim_game_server(my_port, n_a, n_b, n_c):**
    This function is responsible for the server socket connection and the server game logic. The function has no return value.
    It opens a socket, with socket(), bind() and listen() commands – continually waiting for new connections when one is over, in an infinite loop, or until the process is manually shut down.
    When a client tries to connect, it accepts a single connection and the game begins by inserting the given heap values, into the program variable heaps (The same values will be inserted on each new client connection). When the connection is closed, the method returns to listening until a new client connect. In case of an error, the server closes the connection and the function returns.
    c. **def server_move():**
    This method implements the server side nim game logic, for when it is the server's turn to play. It removes value of 1 from the largest heap (a global variable in the program). If more then one largest heap, it removes from the lowest indexed heap.
    d. **is_legal_move(move):**
    A boolean function, given a tuple with two integers in it that represents the player on the client sides wanted move in the game (chosen heap on amount to detract from it).
    The function returns True iff the move is legal by game logic.
    e. **is_win():**
    A boolean function that checks at given moment, if the game has been won – if the all heaps are empty. Return True if so, False otherwise.

2. **nim.py**
    a. **start_client():**
    This function runs when the client starts. It gets the arguments for the client's program and sends them to the nim_game_client() function.
    It also wraps the nim_game_server() function in case of an error, and specifficaly prints to the client "connection refused by server", in the case of a connection error of OSError-ECONNREFUSED error.
    b. **nim_game_client(my_host, my_port):**
    Method in charge of establishing the socket connection with the server my_host, using port my_port. After connection, the method oversees the game loop and all communication with the server side.
    In an ongoing game loop,the method maintains communication with the user that is playing. It print's the heaps status and game status that it receives froms

from the serve, it gets the player's input, checks it's in the correct format and sends it to the server.
In case of an error, the connection is closed and the function returns. Specifically in case of the server side connection being closed, the method prints "Disconnected from server".
When current game is finished (either side wins, or user asked to quit), the method returns.

c. **print_heaps(n_a, n_b, n_c):**
This function print's the heaps status in the HW format.

d. **create_turn_to_send(play):**
This function checks that the user input, received in the parameter play as a list of strings, is in the correct format of a letter (A,B,C) and an integer, or just the letter Q.
It creates a length 2 tuple with the coded data to be sent to the server, accordingly.

3. **shared_global.py**

a. **send_all(soc, data):**
This function gets a socket and data parsed as a bytes object. It continuesly envokes the soc.send() method, until the wanted data is completely sent to the other side (checked by size of data), or an error occurred.
The method returns 1 on success, 2 if connection closed on the other side (via EPIPE or ECONNRESET) and 0 for any other OSError.

b. **recv_all(soc, st):**
This function receives messeges from the other side of the given socket, and makes sure it is received in full, unless an error occurs.
It gets a socket and a string st, signifying the struct format of the expected message to be received.
The method returns the received message (as byte object) on success, 2 if connection is closed (via OSError ECONNREFUSED or recv invocation returned False) and 0 for any other OSError.