DATA2410 - Networking and cloud computing
Portfolio 2: Showtime
Task 2: A multiplayer game

Group 16
Andreas Torres Hansen (s338851)
Anders Hagen Ottersland (s341883)
Uy Quoc Nguyen (s341864)

Deadline:
May 23th 2021

Abstract

This document describes the details of our implementation on this project. This project is one of two projects given on the course DATA2410 - Networking and cloud computing that counts towards our final grade. In this project, we were supposed to implement a solution for one of two tasks, a webshop using a REST API or a multiplayer game with gRPC. Our team chose the latter due to our enthusiasm with games. Both tasks had requirements in terms of a list of user stories and some stretch goals. This document will go through the given tasks and how our implementation solves these goals in details. Before, we start off with the details we will go through the most exciting part first, namely how to run the implementation, our snake game, in the first section. Furthermore, this project can also be found on Github.

Contents

1 How to run

In this section, we will give a detailed step for step on how to start up our snake game. This demonstration is performed on WSL 2 with Windows powershell distro on a Windows 10 computer due to the project has been developed using Windows 10 with PyCharm. Thus, since WSL 2 with Ubuntu does not have a GUI we will be using Windows PowerShell distro on WSL 2 instead. We suspect that if the program is ran on a UNIX system it should be possible to run the game with similar commands, by just replacing `py` with `python3`. Moreover, the python version installed on this computer is 3.9.4 with pip 2.1. In addition, it would be required to have Docker and Docker Compose are installed on the computer following this demonstration.

1.1 The game

Firstly, we need to start the server! Per required from the task, there is a Dockerfile included in the project folder. Start off by building a Docker image from the Dockerfile with

```
> docker build -t name:tag .
```

the `name:tag` could be any name you want to give the image. In our case, we chose to simply call it "snake-service". Hence, our command is therefore

```
> docker build -t snake-service .
```

Now that the image has been built:

```
> docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
snake-service latest 3f3c839e8012 2 minutes ago 938MB
```

We run our game-server container with

```
> docker run --rm -it -p 50051:50051 --name snake-service snake-
    ↪ service
Server is listening...
```

The server is blocking; Which means that you cannot execute additional commands on this terminal. Therefore, open up a new terminal and start the game from the project folder with

```
> py client.py
```

After executing the command above you should have the following window open.

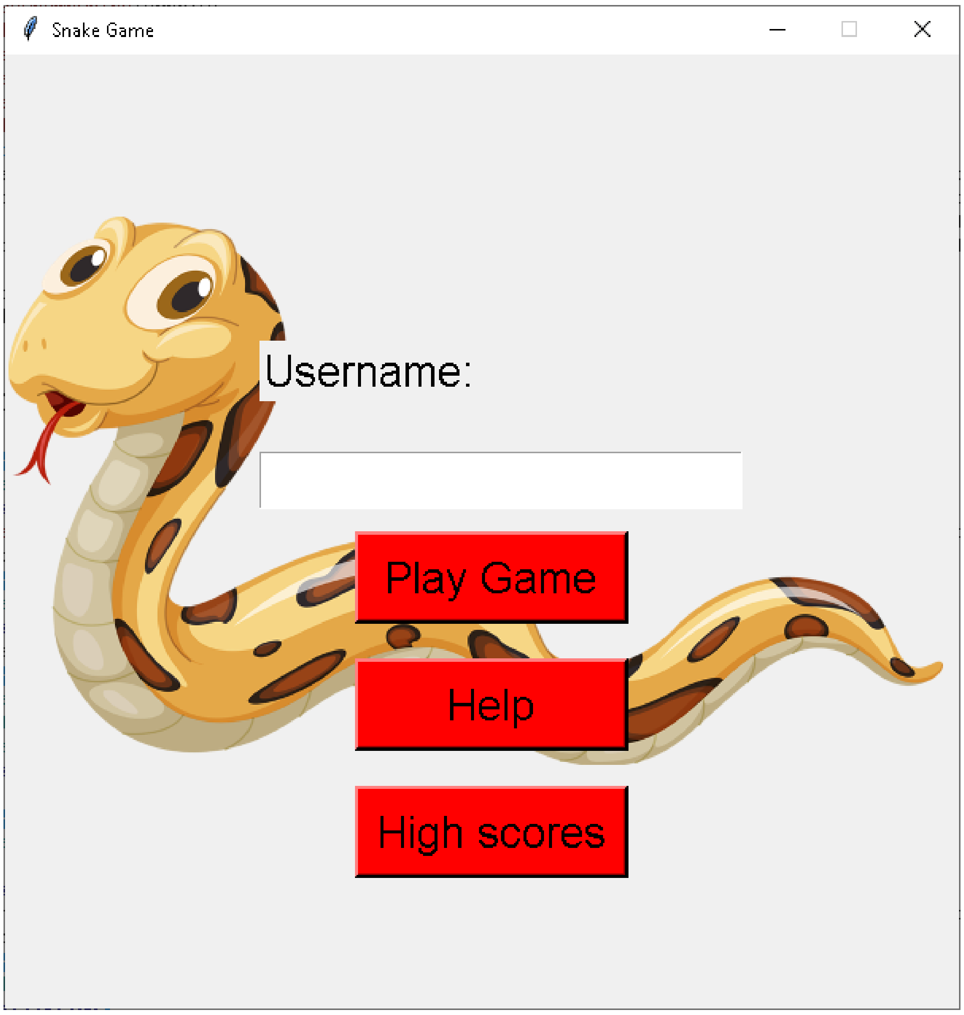


Figure 1: Index page of the snake-game developed on this project.

Because we have implemented a backend to store high scores it would be necessary to provide a username you want to play as. After you have provided the GUI with a username you can either hit Enter or click on "Play

Game" button to start playing our snake game.

There are additional buttons presents on the window above as well. The "Help" button will give you a basic overview on what the game is about and provide you with instructions on how to play the game. Additionally, the "High scores" button will give you a list of high scores of different players that has played this game. High scores are stored on a SQL server running on google cloud. Hence, the list contains some of our high scores.

1.2 The game with monitoring

Now, the project also includes a docker-compose.yml file that also starts up containers pulled from the web as well as building the.

```
> docker-compose up
```

You will build and run Prometheus, cAdvisor and Grafana containers as well as the game-server. The three formerly mentioned containers are used for monitoring the resource usage as specified by one of the Stretch goals of the task. Prometheus is used for scraping metrics from cAdvisor, wheras cAdvisor is getting resource and network traffics from all the running containers. Grafana is used for visualisation. All of these containers can be accessed with `http://localhost:9090/`, `http://localhost:8080/` and `http://localhost:3000/` on a web browser respectively, after the containers are up and running.

We tend to use Grafana to monitor our resource usage. Head to http://localhost:3000/ on your favorite browser and you will be met with the page shown below:

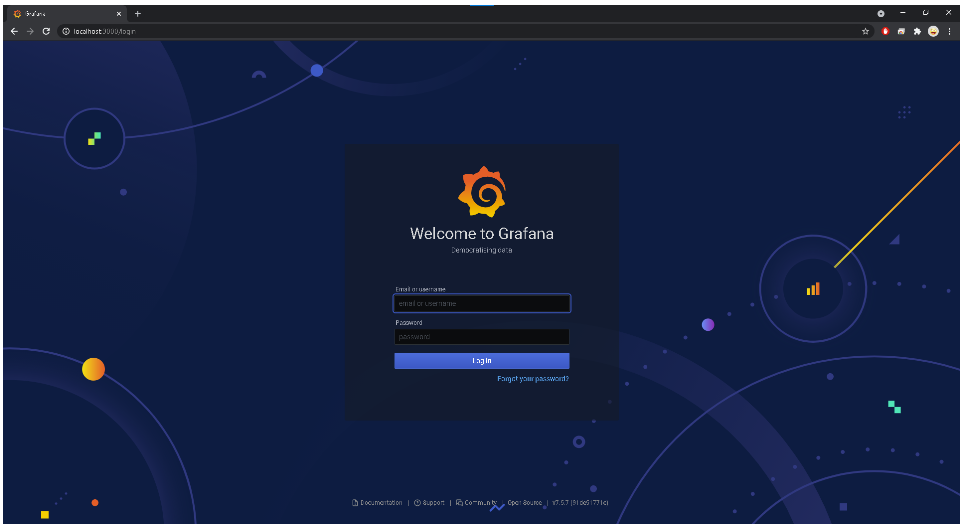
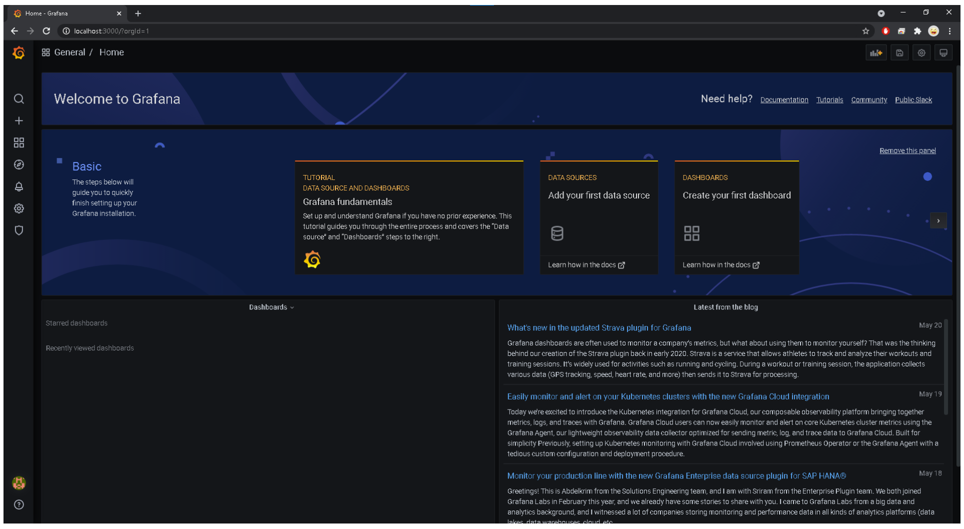


Figure 2: User interface of Grafana after running its container.

Sign in with `username:` *admin* and `password:` *admin* then skip creating a new password. You should be seeing the following



The steps below will guide you to quickly finish setting up your Grafana installation.

Set up and understand Grafana if you have no prior experience. This tutorial guides you through the entire process and covers the "Data source" and "Dashboards" steps to the right.

What's new in the updated Strava plugin for Grafana — May 20

Grafana dashboards are often used to monitor a company's metrics, but what about using them to monitor yourself? That was the thinking behind our creation of the Strava plugin back in early 2020. Strava is a service that allows athletes to track and analyze their workouts and training sessions. It's widely used for activities such as running and cycling. During a workout or training session, the application collects various data (GPS tracking, speed, heart rate, and more) then sends it to Strava for processing.

Easily monitor and alert on your Kubernetes clusters with the new Grafana Cloud integration — May 19

Today we're excited to introduce the Kubernetes integration for Grafana Cloud, our composable observability platform bringing together metrics, logs, and traces with Grafana. Grafana Cloud users can now easily monitor and alert on core Kubernetes cluster metrics using the Grafana Agent, our lightweight observability data collector optimized for sending metric, log, and trace data to Grafana Cloud. Built for simplicity Previously, setting up Kubernetes monitoring with Grafana Cloud involved using Prometheus Operator or the Grafana Agent with a tedious custom configuration and deployment procedure.

Monitor your production line with the new Grafana Enterprise data source plugin for SAP HANA® — May 18

Greetings! This is Abdelkrim from the Solutions Engineering team, and I am with Sriram from the Enterprise Plugin team. We both joined Grafana Labs in February this year, and we already have some stories to share with you. I came to Grafana Labs from a big data and analytics background, and I witnessed a lot of companies storing monitoring and performance data in all kinds of analytics platforms (data lakes, data warehouses, cloud, etc

Figure 3: Grafana after signing in.

From here click on on the veritcal navigation bar on the right and choose Data Sources, then Add data source. Choose the Time series database prometheus and change the "URL" to "prometheus:9090" then go ahead and click "Save & Test" at the bottom of that page. If everything is going well, this should pop up:

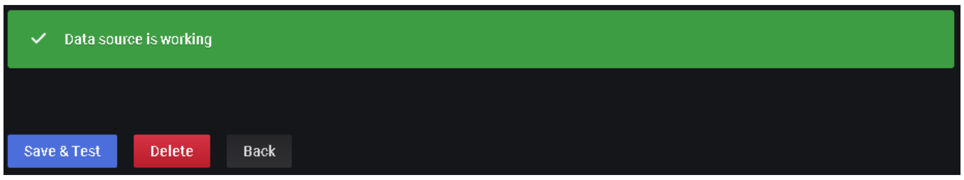
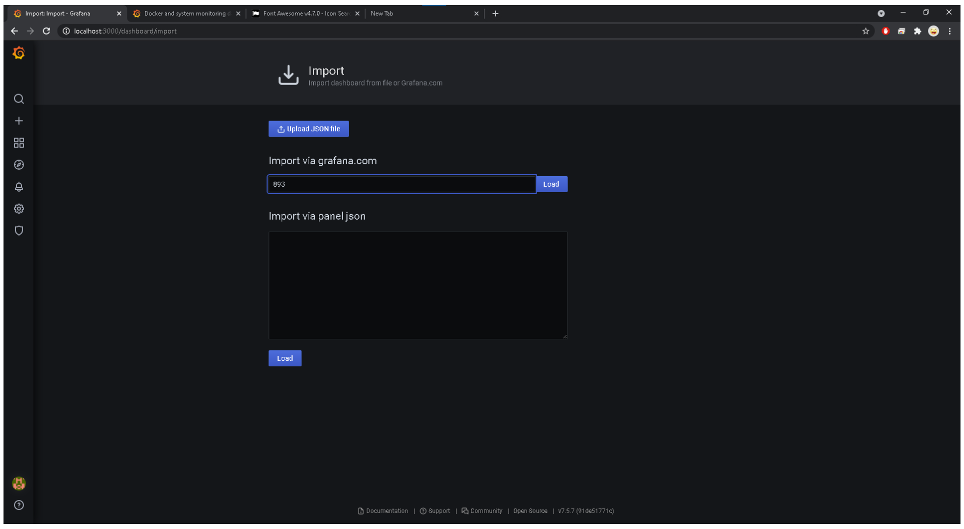

Data source is working

Figure 4: After following the steps above you should get this message after clicking "Save & Test".

Then, click on the icon on the vertical navigation bar and choose import. From here you can choose "Import via Grafana.com" or "Import via panel json". The easiest is to head over to `https://grafana.com/grafana/dashboards/` and find a dashboard you like. We used `https://grafana.com/grafana/dashboards/893` when tracking our resource usage. So we enter "893" into the "Import via Grafana.com" field and hit "load"
.



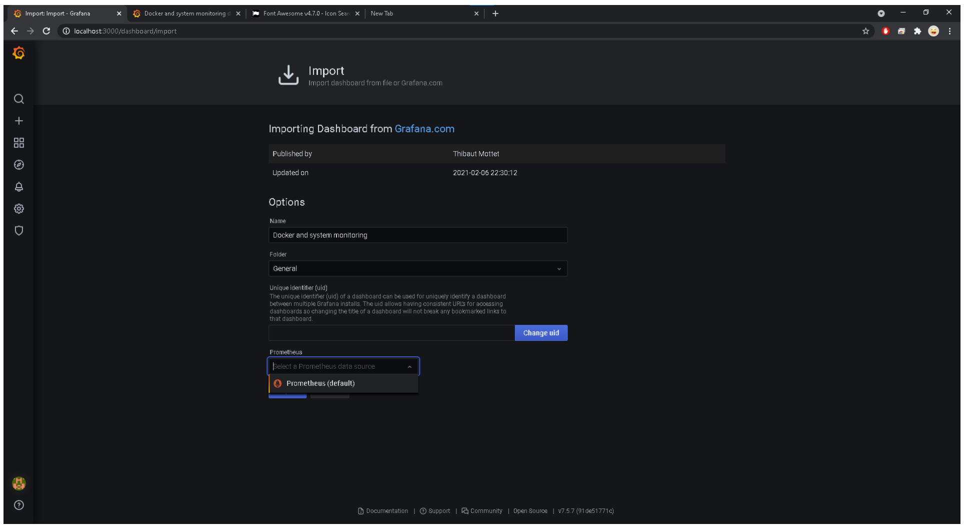
Import via grafana.com

893

Import via panel json

Figure 5: Loading dashboard https://grafana.com/grafana/dashboards/893 in Grafana

After hitting “load” you should have this page on your browser:



Importing Dashboard from Grafana.com

Published by	Thibaut Mottet
Updated on	2021-02-06 22:30:12

Options

Name: Docker and system monitoring

Folder: General

Unique identifier (uid)
The unique identifier (uid) of a dashboard can be used for uniquely identify a dashboard between multiple Grafana installs. The uid allows having consistent URL's for accessing dashboards so changing the title of a dashboard will not break any bookmarked links to that dashboard.

Prometheus: Select a Prometheus data source

Prometheus (default)

Figure 6: Page for importing dashboard in Grafana.

Select “Prometheus” then click on “import”. The result should look like this:

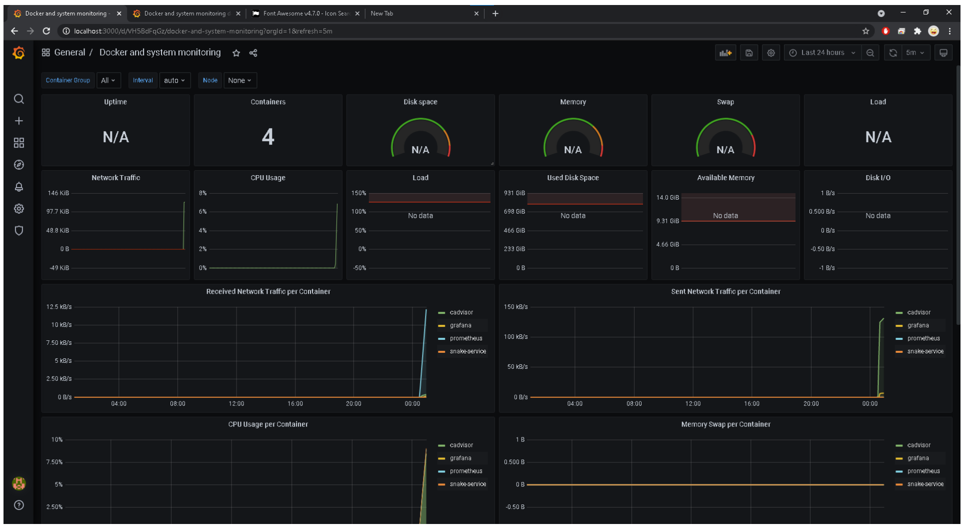


Figure 7: Monitoring dashboard in Grafana.

Set the appropriate time range and interval, then it should be easy to monitor the resource usage, network traffic of the game-server container. To make it easier to monitor the traffic, we implemented bots to play the game. To start a bot execute the command

```
> py client.py --bot
```

This will skip the page given in Figure 1 and you will see the bot is playing by itself. There is also a script included in the project folder that can concurrently start n amount of bots playing. Execute

```
> ./start_bots.cmd n
```

in Windows PowerShell to start n (n is an integer number) bots to play our snake game. This concludes this section. In the next section we will give a rundown on the implemented code on both the client and server side of this game.

2 Implementation

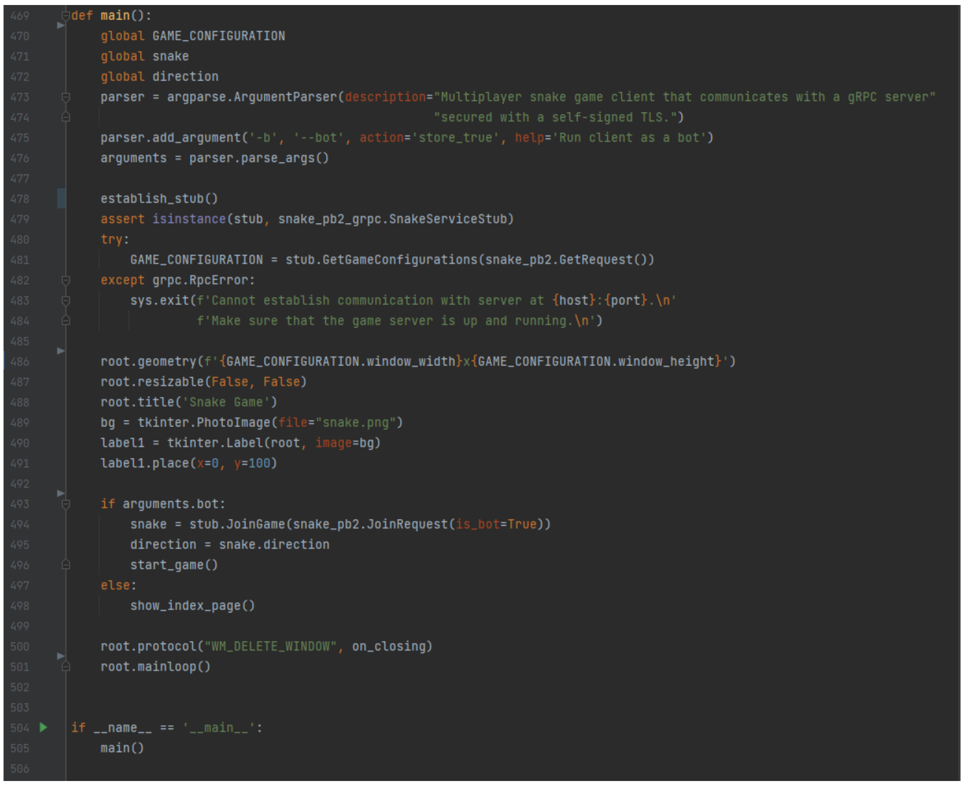
In this section, we will walk through the code we have written to make all this magic possible. Before we dive in let's take a look at the project folder:

```
Data2410-snake
|
+-- service
| |
| +-- protobufs
| | |
| | +-- snake.proto
| +-- snake-server
| |
| +-- bot-names.json
| +-- crt.pem
| +-- key.pem
| +-- requirement.txt
| +-- server.py
| +-- snake_pb2.py
| +-- snake_pb2_grpc.py
| +-- tkinter-colors.json
+-- client.py
+-- crt.pem
+-- docker-compose.yml
+-- Dockerfile
+-- prometheus.yml
+-- README.md
+-- snake.png
+-- snake_pb2.py
+-- snake_pb2_grpc.py
+-- start_bots.cmd
```

As we can see there are some redundancy of files. For instance snake_pb2.py and snake_pb2_grpc.py is both present in root and ./service/snake-server folders. They are both output when compiling snake.proto. The same goes for crt.pem. The reason for this is that it makes it easier for us to run the program from PyCharm without getting file- and module not found errors. The README.md is there because we wanted to have a nice readme file for our Github.

2.1 Entrypoint main() function of client.py.

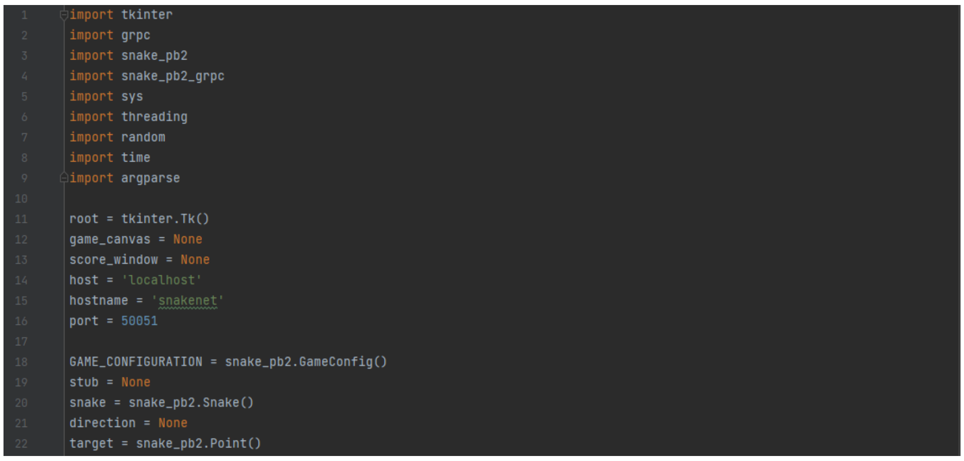
Assuming that the server is up and running. When starting client.pyit will call on the function `main()` shown below: As seen above, we have defined



```python
def main():
    global GAME_CONFIGURATION
    global snake
    global direction
    parser = argparse.ArgumentParser(description="Multiplayer snake game client that communicates with a gRPC server"
                                                 "secured with a self-signed TLS.")
    parser.add_argument('-b', '--bot', action='store_true', help='Run client as a bot')
    arguments = parser.parse_args()

    establish_stub()
    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    try:
        GAME_CONFIGURATION = stub.GetGameConfigurations(snake_pb2.GetRequest())
    except grpc.RpcError:
        sys.exit(f'Cannot establish communication with server at {host}:{port}.\n'
                 f'Make sure that the game server is up and running.\n')

    root.geometry(f'{GAME_CONFIGURATION.window_width}x{GAME_CONFIGURATION.window_height}')
    root.resizable(False, False)
    root.title('Snake Game')
    bg = tkinter.PhotoImage(file="snake.png")
    label1 = tkinter.Label(root, image=bg)
    label1.place(x=0, y=100)

    if arguments.bot:
        snake = stub.JoinGame(snake_pb2.JoinRequest(is_bot=True))
        direction = snake.direction
        start_game()
    else:
        show_index_page()

    root.protocol("WM_DELETE_WINDOW", on_closing)
    root.mainloop()


if __name__ == '__main__':
    main()
```

Figure 8: Entrypoint of client.py.

some global variables which all the later discussed functions is going to access.

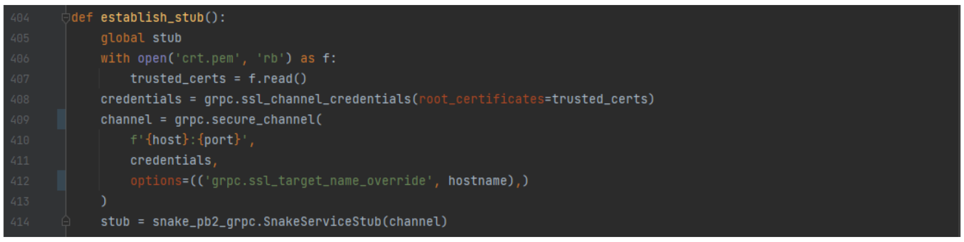


```python
import tkinter
import grpc
import snake_pb2
import snake_pb2_grpc
import sys
import threading
import random
import time
import argparse

root = tkinter.Tk()
game_canvas = None
score_window = None
host = 'localhost'
hostname = 'snakenet'
port = 50051

GAME_CONFIGURATION = snake_pb2.GameConfig()
stub = None
snake = snake_pb2.Snake()
direction = None
target = snake_pb2.Point()
```

Figure 9: Imported modules and global variables in client.py.

As we can see from main(), that it starts with the **argparse** module. This was added in the later stages so that bots can bypass the **show_index_page()** as we simply wanted the bots to start playing the game instead of having to enter a username and click on start playing on the page shown in Figure 1. This made it easier to create the script start_bots.cmd to make n number of bots play the game from PowerShell, which we used for monitoring later.

First, the code will execute is the establish_stub() function The estab-

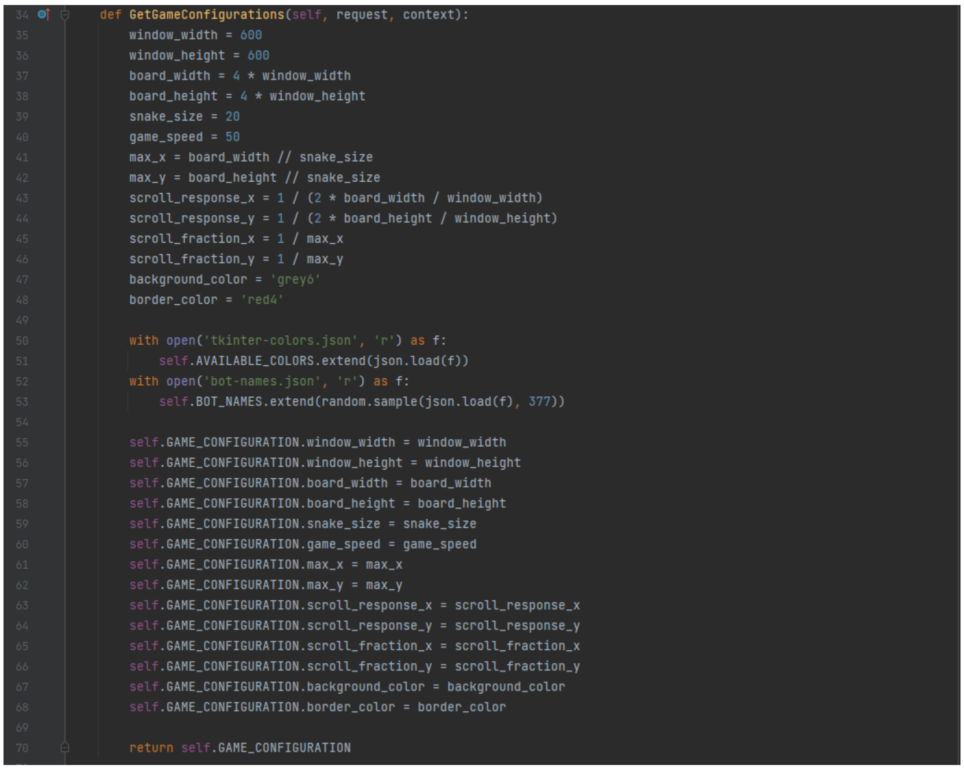


```python
def establish_stub():
    global stub
    with open('crt.pem', 'rb') as f:
        trusted_certs = f.read()
    credentials = grpc.ssl_channel_credentials(root_certificates=trusted_certs)
    channel = grpc.secure_channel(
        f'{host}:{port}',
        credentials,
        options=(('grpc.ssl_target_name_override', hostname),)
    )
    stub = snake_pb2_grpc.SnakeServiceStub(channel)
```

Figure 10: establish_stub() function in client.py.

lish_stub() function opens the SSL/TLS certificate and reads it. Then, pass it to credentials so we can use it to make a secure channel. We also have options which then tells the server that we are connecting with the common name of the certificate instead of the host. This was necessary to pass the SSL/TLS handshake. Lastly, the stub variable is assigned with the SnakeServiceStub specified by snake.proto.

The game configurations such as window height- and width are constants determined inside server.py. We chose to have it their in case of maintainance by the developers.

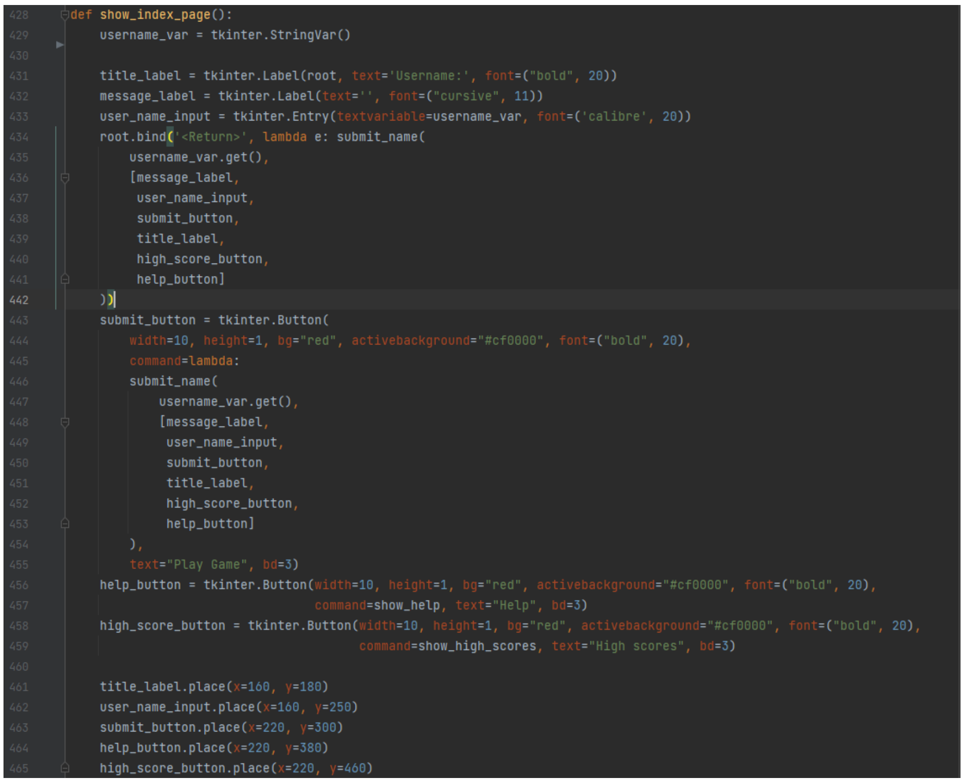


```python
    def GetGameConfigurations(self, request, context):
        window_width = 600
        window_height = 600
        board_width = 4 * window_width
        board_height = 4 * window_height
        snake_size = 20
        game_speed = 50
        max_x = board_width // snake_size
        max_y = board_height // snake_size
        scroll_response_x = 1 / (2 * board_width / window_width)
        scroll_response_y = 1 / (2 * board_height / window_height)
        scroll_fraction_x = 1 / max_x
        scroll_fraction_y = 1 / max_y
        background_color = 'grey6'
        border_color = 'red4'

        with open('tkinter-colors.json', 'r') as f:
            self.AVAILABLE_COLORS.extend(json.load(f))
        with open('bot-names.json', 'r') as f:
            self.BOT_NAMES.extend(random.sample(json.load(f), 377))

        self.GAME_CONFIGURATION.window_width = window_width
        self.GAME_CONFIGURATION.window_height = window_height
        self.GAME_CONFIGURATION.board_width = board_width
        self.GAME_CONFIGURATION.board_height = board_height
        self.GAME_CONFIGURATION.snake_size = snake_size
        self.GAME_CONFIGURATION.game_speed = game_speed
        self.GAME_CONFIGURATION.max_x = max_x
        self.GAME_CONFIGURATION.max_y = max_y
        self.GAME_CONFIGURATION.scroll_response_x = scroll_response_x
        self.GAME_CONFIGURATION.scroll_response_y = scroll_response_y
        self.GAME_CONFIGURATION.scroll_fraction_x = scroll_fraction_x
        self.GAME_CONFIGURATION.scroll_fraction_y = scroll_fraction_y
        self.GAME_CONFIGURATION.background_color = background_color
        self.GAME_CONFIGURATION.border_color = border_color

        return self.GAME_CONFIGURATION
```

Figure 11: GetGameConfigurations in server.py.

Thus, main() is asking the server to give it all the configurations needed to start creating the root window and its contents.

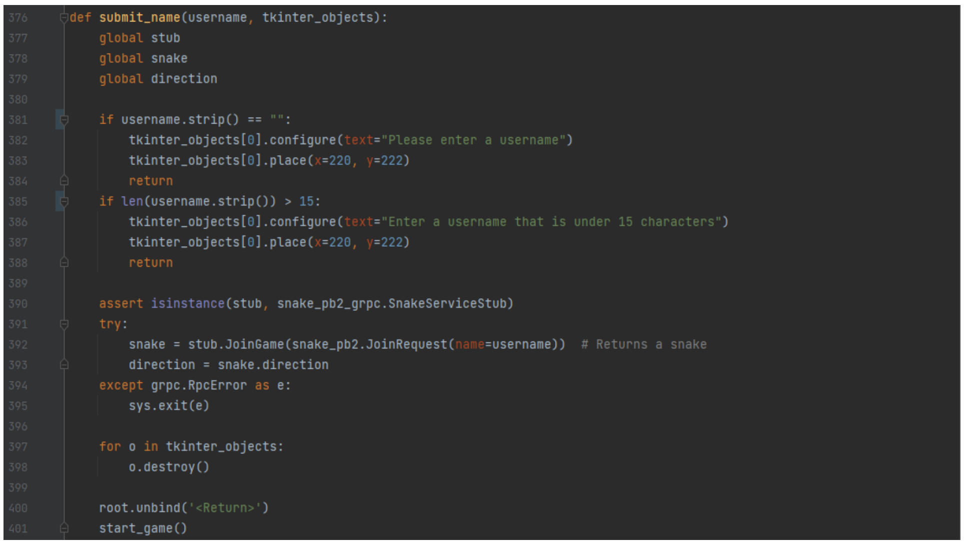
The root window is then created using the global `GAME_CONFIGURATION`. Then, the index page is drawn with its buttons and labels with `show_index_page()`:



```python
def show_index_page():
    username_var = tkinter.StringVar()

    title_label = tkinter.Label(root, text='Username:', font=("bold", 20))
    message_label = tkinter.Label(text='', font=("cursive", 11))
    user_name_input = tkinter.Entry(textvariable=username_var, font=('calibre', 20))
    root.bind('<Return>', lambda e: submit_name(
        username_var.get(),
        [message_label,
         user_name_input,
         submit_button,
         title_label,
         high_score_button,
         help_button]
    ))
    submit_button = tkinter.Button(
        width=10, height=1, bg="red", activebackground="#cf0000", font=("bold", 20),
        command=lambda:
        submit_name(
            username_var.get(),
            [message_label,
             user_name_input,
             submit_button,
             title_label,
             high_score_button,
             help_button]
        ),
        text="Play Game", bd=3)
    help_button = tkinter.Button(width=10, height=1, bg="red", activebackground="#cf0000", font=("bold", 20),
                                 command=show_help, text="Help", bd=3)
    high_score_button = tkinter.Button(width=10, height=1, bg="red", activebackground="#cf0000", font=("bold", 20),
                                       command=show_high_scores, text="High scores", bd=3)

    title_label.place(x=160, y=180)
    user_name_input.place(x=160, y=250)
    submit_button.place(x=220, y=300)
    help_button.place(x=220, y=380)
    high_score_button.place(x=220, y=460)
```

Figure 12: Code responsible for producing the window shown in Figure 1.

After inputting the username and clicking “Play Game” the `submit_name()` function gets executed:

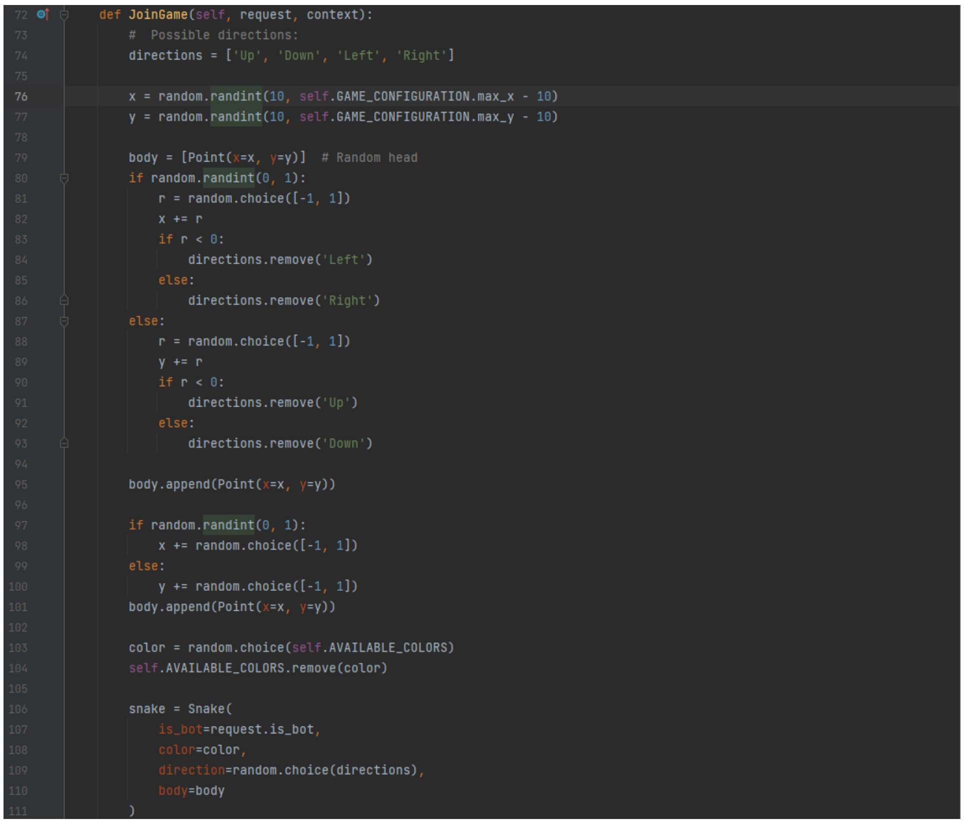


```python
def submit_name(username, tkinter_objects):
    global stub
    global snake
    global direction

    if username.strip() == "":
        tkinter_objects[0].configure(text="Please enter a username")
        tkinter_objects[0].place(x=220, y=222)
        return
    if len(username.strip()) > 15:
        tkinter_objects[0].configure(text="Enter a username that is under 15 characters")
        tkinter_objects[0].place(x=220, y=222)
        return

    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    try:
        snake = stub.JoinGame(snake_pb2.JoinRequest(name=username))  # Returns a snake
        direction = snake.direction
    except grpc.RpcError as e:
        sys.exit(e)

    for o in tkinter_objects:
        o.destroy()

    root.unbind('<Return>')
    start_game()
```

Figure 13: Code block of submit_name function in client.py.

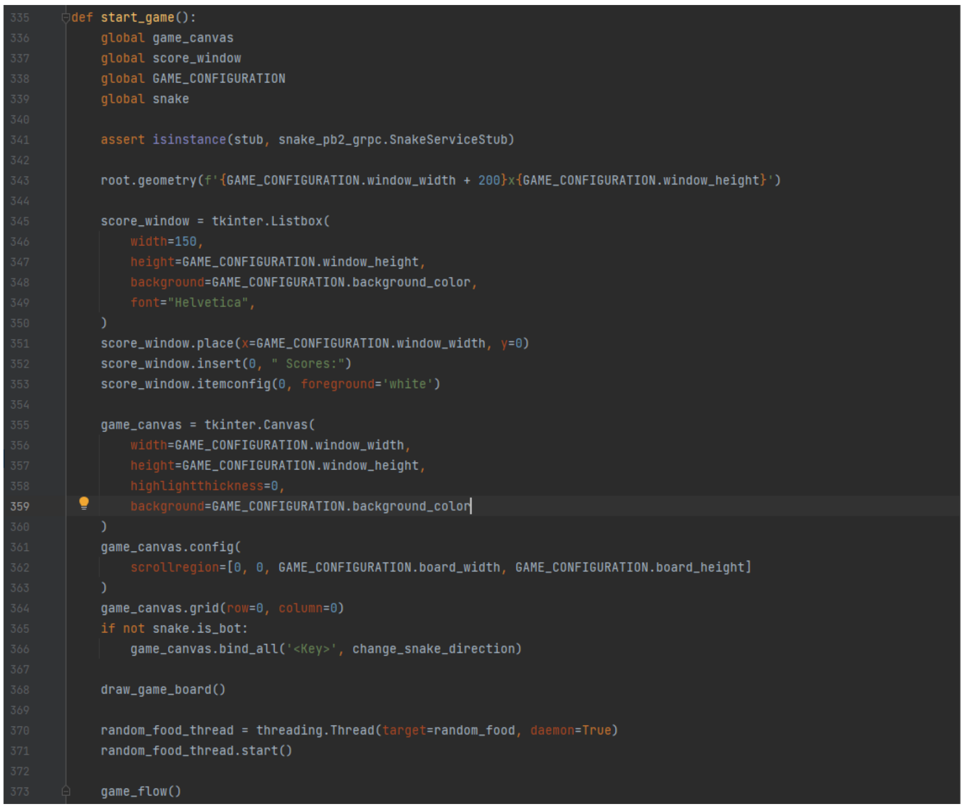
The first part of the code just ensures that the username is not empty, blank or too long. Given that the username has been accepted, the stub asks server to create a snake with random configuration:

The created snake is sent back to the client. At the end, the `start_game()` function is called:



```python
    def JoinGame(self, request, context):
        #  Possible directions:
        directions = ['Up', 'Down', 'Left', 'Right']

        x = random.randint(10, self.GAME_CONFIGURATION.max_x - 10)
        y = random.randint(10, self.GAME_CONFIGURATION.max_y - 10)

        body = [Point(x=x, y=y)]  # Random head
        if random.randint(0, 1):
            r = random.choice([-1, 1])
            x += r
            if r < 0:
                directions.remove('Left')
            else:
                directions.remove('Right')
        else:
            r = random.choice([-1, 1])
            y += r
            if r < 0:
                directions.remove('Up')
            else:
                directions.remove('Down')

        body.append(Point(x=x, y=y))

        if random.randint(0, 1):
            x += random.choice([-1, 1])
        else:
            y += random.choice([-1, 1])
        body.append(Point(x=x, y=y))

        color = random.choice(self.AVAILABLE_COLORS)
        self.AVAILABLE_COLORS.remove(color)

        snake = Snake(
            is_bot=request.is_bot,
            color=color,
            direction=random.choice(directions),
            body=body
        )
```

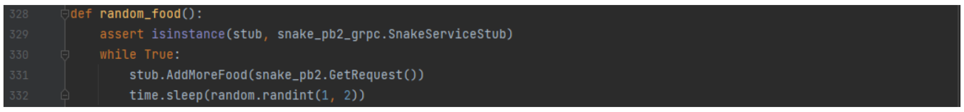
Figure 14: How the server creates a random configured snake in server.py.



```python
def start_game():
    global game_canvas
    global score_window
    global GAME_CONFIGURATION
    global snake

    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)

    root.geometry(f'{GAME_CONFIGURATION.window_width + 200}x{GAME_CONFIGURATION.window_height}')

    score_window = tkinter.Listbox(
        width=150,
        height=GAME_CONFIGURATION.window_height,
        background=GAME_CONFIGURATION.background_color,
        font="Helvetica",
    )
    score_window.place(x=GAME_CONFIGURATION.window_width, y=0)
    score_window.insert(0, " Scores:")
    score_window.itemconfig(0, foreground='white')

    game_canvas = tkinter.Canvas(
        width=GAME_CONFIGURATION.window_width,
        height=GAME_CONFIGURATION.window_height,
        highlightthickness=0,
        background=GAME_CONFIGURATION.background_color
    )
    game_canvas.config(
        scrollregion=[0, 0, GAME_CONFIGURATION.board_width, GAME_CONFIGURATION.board_height]
    )
    game_canvas.grid(row=0, column=0)
    if not snake.is_bot:
        game_canvas.bind_all('<Key>', change_snake_direction)

    draw_game_board()

    random_food_thread = threading.Thread(target=random_food, daemon=True)
    random_food_thread.start()

    game_flow()
```

Figure 15: Code block of start_game function in client.py.

This block of code starts off by removing everything from the root window and defines a canvas where our game is living. Additionally it creates a list to the right of the window with the purpose of giving the player details on which players are currently connected and the points the players have accummulated. A thread is also being created. Since, the board is very big the game board could look like a barren wasteland. Hence, the threads main function is to ask the server to add new food to the game after each 1 or 2 seconds:

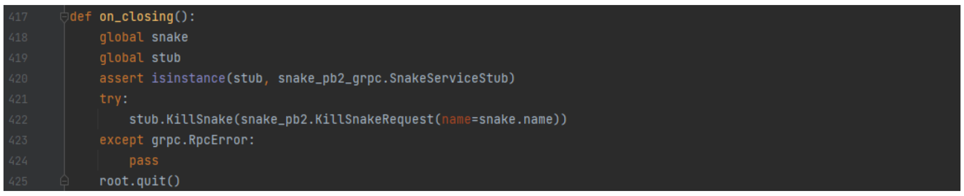


```python
def random_food():
    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    while True:
        stub.AddMoreFood(snake_pb2.GetRequest())
        time.sleep(random.randint(1, 2))
```

Figure 16: Function responsible for adding food to the game each 1 or 2 seconds in client.py

The `game_flow()` function called at the end is determining the main flow of the game, i.e. what is to be happened for each frame the player sees. This function deserves its own section, which we will discuss next.

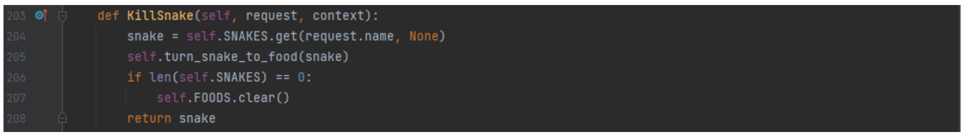
A comment before going off to the next section of the document: In the `main()` code block given in main(), the root window of the game has a protocol for what is going to happen when closing the window. This is determined by the `on_closing()` function:



```python
def on_closing():
    global snake
    global stub
    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    try:
        stub.KillSnake(snake_pb2.KillSnakeRequest(name=snake.name))
    except grpc.RpcError:
        pass
    root.quit()
```

Figure 17: Protocol for closing the game

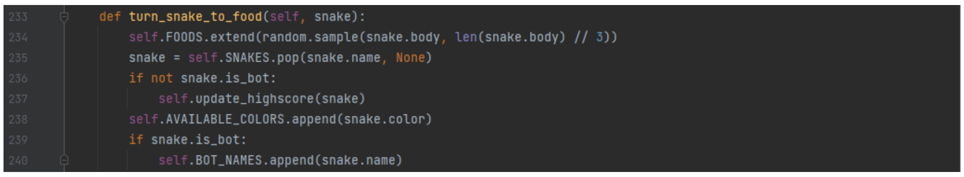
Which, basically tells the game to remove the snake from the server by killing it if it exists:



```python
    def KillSnake(self, request, context):
        snake = self.SNAKES.get(request.name, None)
        self.turn_snake_to_food(snake)
        if len(self.SNAKES) == 0:
            self.FOODS.clear()
        return snake
```

Figure 18: How the server kills the snake in server.py

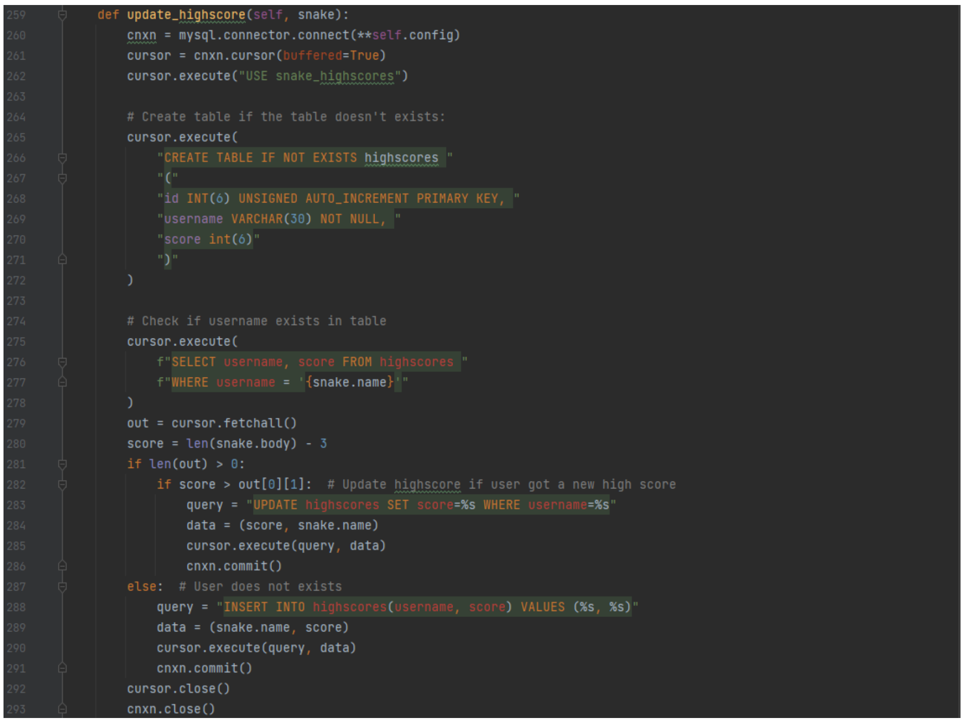
The server will turn the snake to 1 food per 3 length:



```python
    def turn_snake_to_food(self, snake):
        self.FOODS.extend(random.sample(snake.body, len(snake.body) // 3))
        snake = self.SNAKES.pop(snake.name, None)
        if not snake.is_bot:
            self.update_highscore(snake)
        self.AVAILABLE_COLORS.append(snake.color)
        if snake.is_bot:
            self.BOT_NAMES.append(snake.name)
```

Figure 19: The code executed when the snake dies in server.py.

and update the high scores with `update_highscore(snake)` in our database:



```python
    def update_highscore(self, snake):
        cnxn = mysql.connector.connect(**self.config)
        cursor = cnxn.cursor(buffered=True)
        cursor.execute("USE snake_highscores")

        # Create table if the table doesn't exists:
        cursor.execute(
            "CREATE TABLE IF NOT EXISTS highscores "
            "("
            "id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY, "
            "username VARCHAR(30) NOT NULL, "
            "score int(6)"
            ")"
        )

        # Check if username exists in table
        cursor.execute(
            f"SELECT username, score FROM highscores "
            f"WHERE username = '{snake.name}'"
        )
        out = cursor.fetchall()
        score = len(snake.body) - 3
        if len(out) > 0:
            if score > out[0][1]:  # Update highscore if user got a new high score
                query = "UPDATE highscores SET score=%s WHERE username=%s"
                data = (score, snake.name)
                cursor.execute(query, data)
                cnxn.commit()
        else:  # User does not exists
            query = "INSERT INTO highscores(username, score) VALUES (%s, %s)"
            data = (snake.name, score)
            cursor.execute(query, data)
            cnxn.commit()
        cursor.close()
        cnxn.close()
```

Figure 20: Updating high score with mysql.connector in server.py

Then, the game window is allowed to close. If the game hasn't started it will simply just close the game window.

2.2 The game_flow() of the game

When the code starts to execute the start_game function provided in figure 15, the player should be able to see this window:

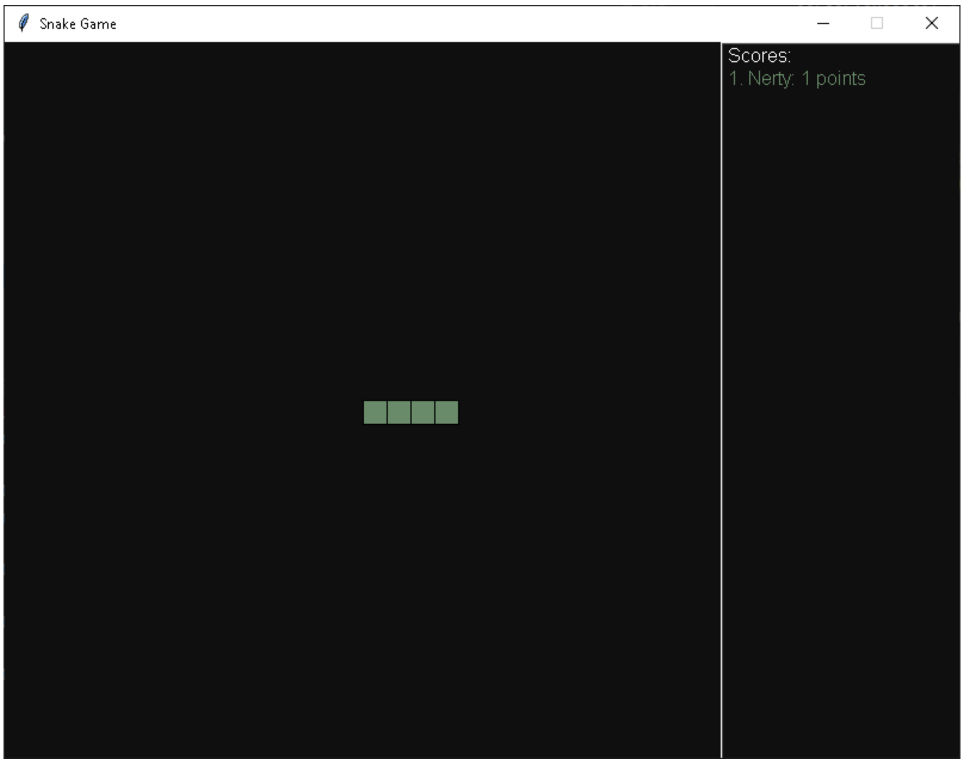
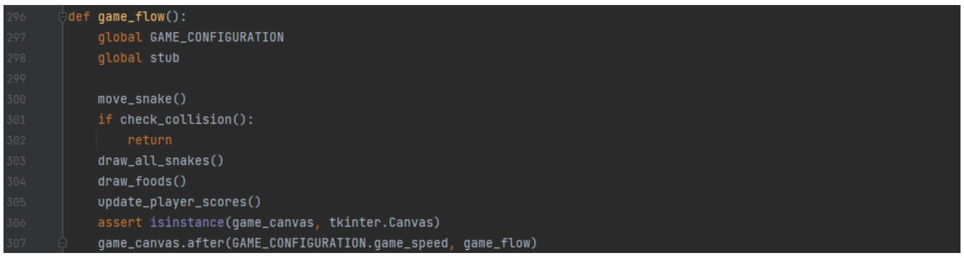


Figure 21: A lonely bot with random name "Nerty" playing the game

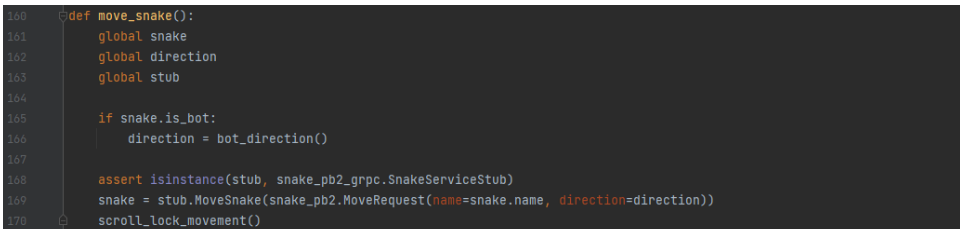
At the end of start_game the function `game_flow()` is called:



```python
def game_flow():
    global GAME_CONFIGURATION
    global stub

    move_snake()
    if check_collision():
        return
    draw_all_snakes()
    draw_foods()
    update_player_scores()
    assert isinstance(game_canvas, tkinter.Canvas)
    game_canvas.after(GAME_CONFIGURATION.game_speed, game_flow)
```

Figure 22: The game_flow() function in client.py.

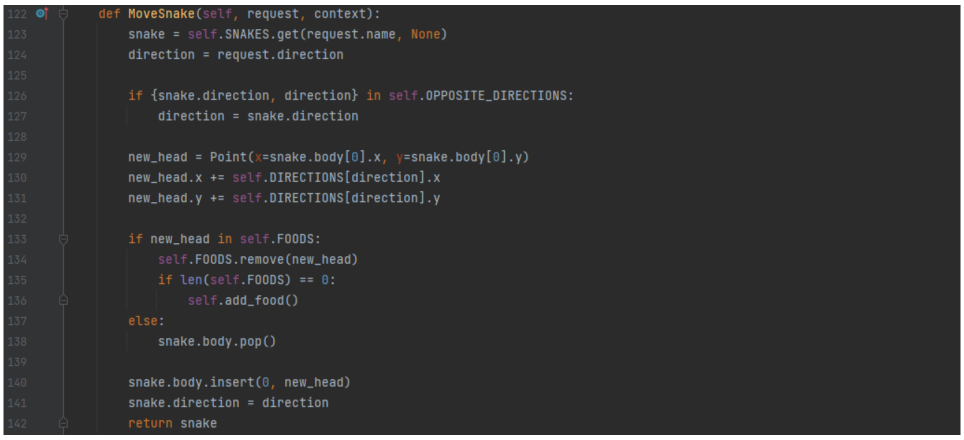
As we can see `game_flow()` is just a series of function calls. However, this defines what is happening for each frame of the game. Firstly, the `move_snake()` is responsible to move the snake:



```python
def move_snake():
    global snake
    global direction
    global stub

    if snake.is_bot:
        direction = bot_direction()

    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    snake = stub.MoveSnake(snake_pb2.MoveRequest(name=snake.name, direction=direction))
    scroll_lock_movement()
```

Figure 23: How to snake moves on the client.py

There is not much to be seen here. The only thing this function do is to request the server to move their snake:



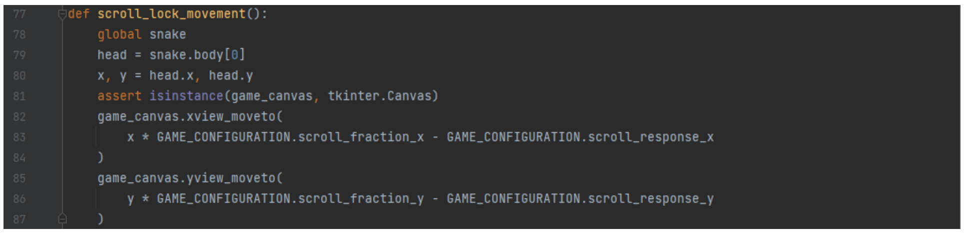
```python
    def MoveSnake(self, request, context):
        snake = self.SNAKES.get(request.name, None)
        direction = request.direction

        if {snake.direction, direction} in self.OPPOSITE_DIRECTIONS:
            direction = snake.direction

        new_head = Point(x=snake.body[0].x, y=snake.body[0].y)
        new_head.x += self.DIRECTIONS[direction].x
        new_head.y += self.DIRECTIONS[direction].y

        if new_head in self.FOODS:
            self.FOODS.remove(new_head)
            if len(self.FOODS) == 0:
                self.add_food()
        else:
            snake.body.pop()

        snake.body.insert(0, new_head)
        snake.direction = direction
        return snake
```

Figure 24: Server moving the snake in server.py.

This is where the magic happens. First, the server checks if the requested direction is the opposite direction to the snake's course. The server then, creates a new head for the snake. It also checks if the new head it has created is located at the same coordinates as one of the food on the game board. This is how the snake grows. If the new head and food coincides, it will remove this food from the game board. If this is not the case it will remove the tail of the snake instead. At the end it will insert the new head into the snake's body at the beginning of its list, and update direction accordingly.

The server then returns the updated snake to the client, where the it assigns this new snake to its global variable.

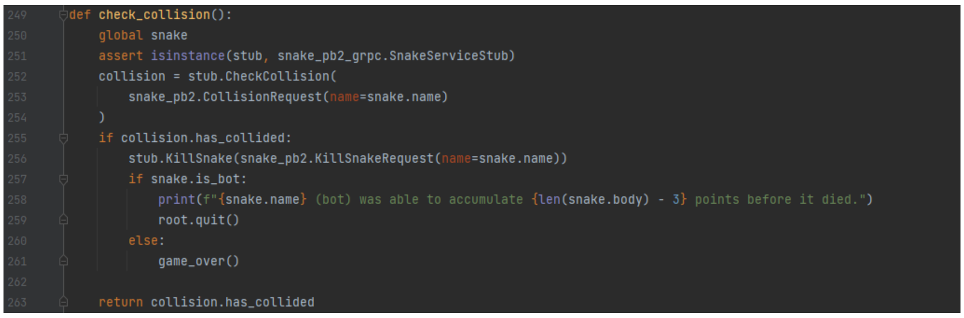
From here, `scroll_lock_movement()` ensures that the window view is following our snake relative to where its head is:



```python
def scroll_lock_movement():
    global snake
    head = snake.body[0]
    x, y = head.x, head.y
    assert isinstance(game_canvas, tkinter.Canvas)
    game_canvas.xview_moveto(
        x * GAME_CONFIGURATION.scroll_fraction_x - GAME_CONFIGURATION.scroll_response_x
    )
    game_canvas.yview_moveto(
        y * GAME_CONFIGURATION.scroll_fraction_y - GAME_CONFIGURATION.scroll_response_y
    )
```

Figure 25: The code responsible to scroll the view with respect to the snake's movement in client.py.

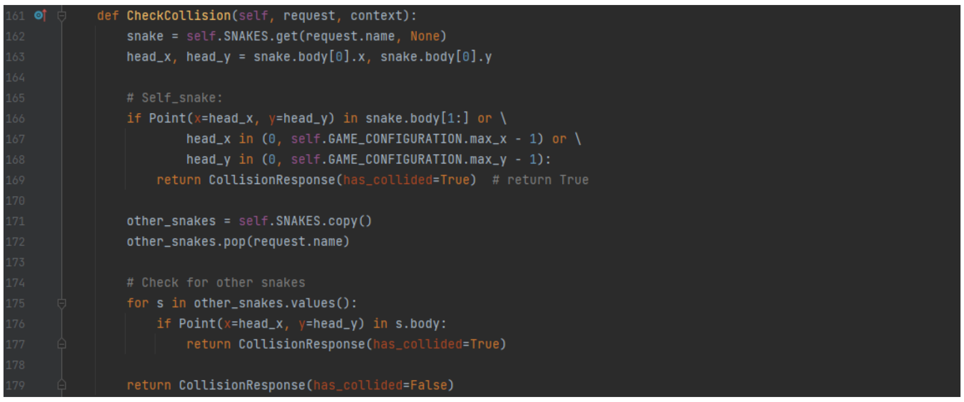
After the movement has been established, `game_flow()` executes the next function which will check if the snake has collided either with itself, the border or other snakes:



```python
def check_collision():
    global snake
    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    collision = stub.CheckCollision(
        snake_pb2.CollisionRequest(name=snake.name)
    )
    if collision.has_collided:
        stub.KillSnake(snake_pb2.KillSnakeRequest(name=snake.name))
        if snake.is_bot:
            print(f"{snake.name} (bot) was able to accumulate {len(snake.body) - 3} points before it died.")
            root.quit()
        else:
            game_over()

    return collision.has_collided
```

Figure 26: Code that determines if the snake has collided or not in client.py

Since, the client is only responsible to draw what the server is sending it, therefore cannot determine if the snake has collided or not. Hence, `check_collision()` asks the server to figure out if the snake has collided:

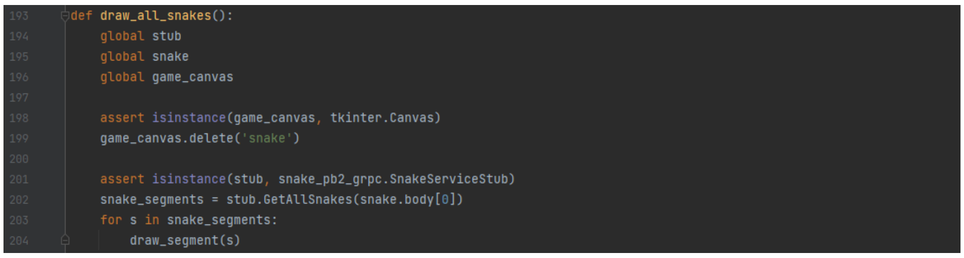


```python
    def CheckCollision(self, request, context):
        snake = self.SNAKES.get(request.name, None)
        head_x, head_y = snake.body[0].x, snake.body[0].y

        # Self_snake:
        if Point(x=head_x, y=head_y) in snake.body[1:] or \
                head_x in (0, self.GAME_CONFIGURATION.max_x - 1) or \
                head_y in (0, self.GAME_CONFIGURATION.max_y - 1):
            return CollisionResponse(has_collided=True)  # return True

        other_snakes = self.SNAKES.copy()
        other_snakes.pop(request.name)

        # Check for other snakes
        for s in other_snakes.values():
            if Point(x=head_x, y=head_y) in s.body:
                return CollisionResponse(has_collided=True)

        return CollisionResponse(has_collided=False)
```

Figure 27: How the server determines if the requested snake has collided in server.py.

The server, first check if the head has the same coordinates as one of the points in its body excluding its head. Or, if the head has the same coordinates as the borders. If this is not the case it will go ahead and check wether the head has the same coordinates as on of the other snake's body and return accordingly.

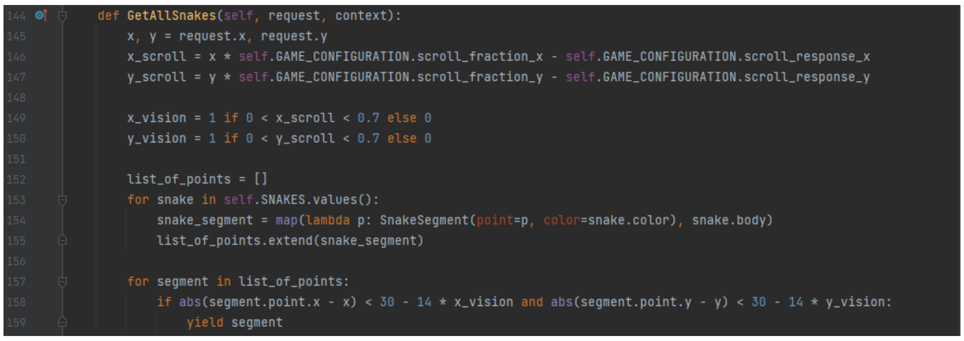
The client will then ask the server to kill the snake if it has collided and send the player to the "Game Over" page by calling the function `game_over()`. If the snake has not collided it will return `False` and `game_flow()` executes the next call:



```python
def draw_all_snakes():
    global stub
    global snake
    global game_canvas

    assert isinstance(game_canvas, tkinter.Canvas)
    game_canvas.delete('snake')

    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    snake_segments = stub.GetAllSnakes(snake.body[0])
    for s in snake_segments:
        draw_segment(s)
```

Figure 28: Function that draws all the snakes within view of the player's snake in client.py.

This deletes everything on the canvas tagged as "snake". Then, the client sends a request to the server to retrieve all the snake segments within the view of its head:

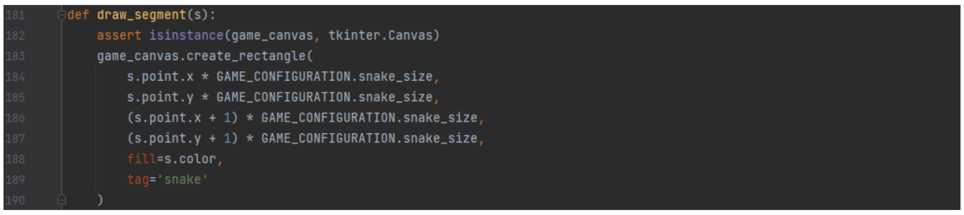


```python
def GetAllSnakes(self, request, context):
    x, y = request.x, request.y
    x_scroll = x * self.GAME_CONFIGURATION.scroll_fraction_x - self.GAME_CONFIGURATION.scroll_response_x
    y_scroll = y * self.GAME_CONFIGURATION.scroll_fraction_y - self.GAME_CONFIGURATION.scroll_response_y

    x_vision = 1 if 0 < x_scroll < 0.7 else 0
    y_vision = 1 if 0 < y_scroll < 0.7 else 0

    list_of_points = []
    for snake in self.SNAKES.values():
        snake_segment = map(lambda p: SnakeSegment(point=p, color=snake.color), snake.body)
        list_of_points.extend(snake_segment)

    for segment in list_of_points:
        if abs(segment.point.x - x) < 30 - 14 * x_vision and abs(segment.point.y - y) < 30 - 14 * y_vision:
            yield segment
```

Figure 29: Server responds to the client request to fetch all snakes in server.py

When the server receives point from the client, it will find all the snakes that is currently playing the game and stream back every snake segments within the view of that player to the client. Before we implemented this, the server just sent back every snake. This means that the client is actually drawing every snake and food on the whole board. The game started to lag with the vast amount of food and snake it had to draw before refreshing the frame. Hence, we improved the game by just sending back whatever the player is seeing.

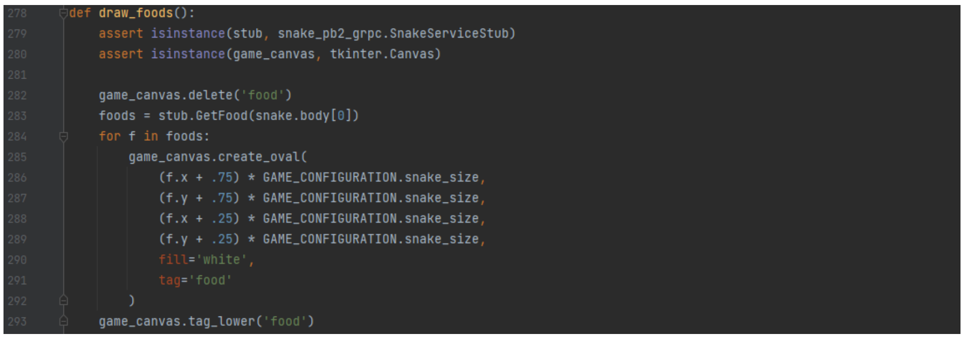
When the client reveives the snake segments it loops through the iterator and draw each segment to the canvas with as a square and tags it as “snake” :



```python
def draw_segment(s):
    assert isinstance(game_canvas, tkinter.Canvas)
    game_canvas.create_rectangle(
        s.point.x * GAME_CONFIGURATION.snake_size,
        s.point.y * GAME_CONFIGURATION.snake_size,
        (s.point.x + 1) * GAME_CONFIGURATION.snake_size,
        (s.point.y + 1) * GAME_CONFIGURATION.snake_size,
        fill=s.color,
        tag='snake'
    )
```

Figure 30: Function that produces 1 square at x, y on the canvas in client.py .

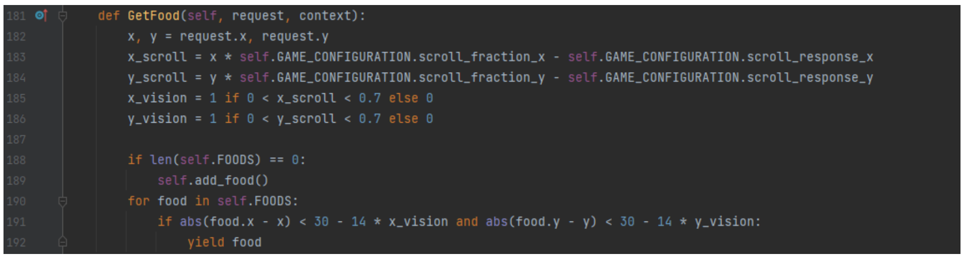
The draw food follows the same principle as the previous two figures:



```python
def draw_foods():
    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    assert isinstance(game_canvas, tkinter.Canvas)

    game_canvas.delete('food')
    foods = stub.GetFood(snake.body[0])
    for f in foods:
        game_canvas.create_oval(
            (f.x + .75) * GAME_CONFIGURATION.snake_size,
            (f.y + .75) * GAME_CONFIGURATION.snake_size,
            (f.x + .25) * GAME_CONFIGURATION.snake_size,
            (f.y + .25) * GAME_CONFIGURATION.snake_size,
            fill='white',
            tag='food'
        )
    game_canvas.tag_lower('food')
```

Figure 31: Client request all the food within view of player in client.py.

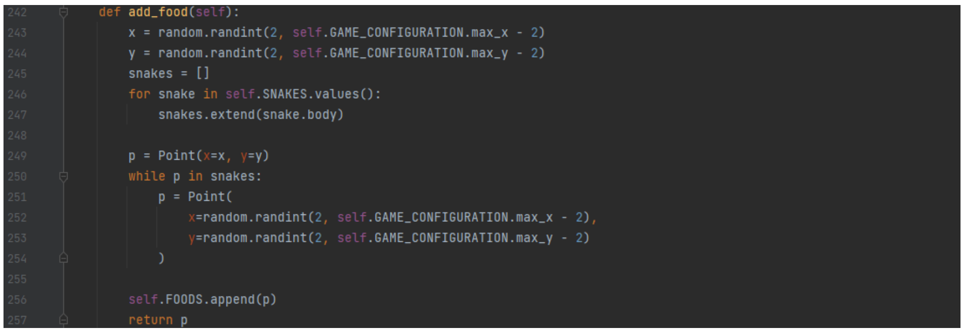
The server will respond with a stream of food within view of the player:



```python
    def GetFood(self, request, context):
        x, y = request.x, request.y
        x_scroll = x * self.GAME_CONFIGURATION.scroll_fraction_x - self.GAME_CONFIGURATION.scroll_response_x
        y_scroll = y * self.GAME_CONFIGURATION.scroll_fraction_y - self.GAME_CONFIGURATION.scroll_response_y
        x_vision = 1 if 0 < x_scroll < 0.7 else 0
        y_vision = 1 if 0 < y_scroll < 0.7 else 0

        if len(self.FOODS) == 0:
            self.add_food()
        for food in self.FOODS:
            if abs(food.x - x) < 30 - 14 * x_vision and abs(food.y - y) < 30 - 14 * y_vision:
                yield food
```

Figure 32: Server responds back with a stream of food in server.py

However, the server will also always ensure that there are at least 1 food in the game by first checking its list of foods still to be eaten and call `add_food()` accordingly:

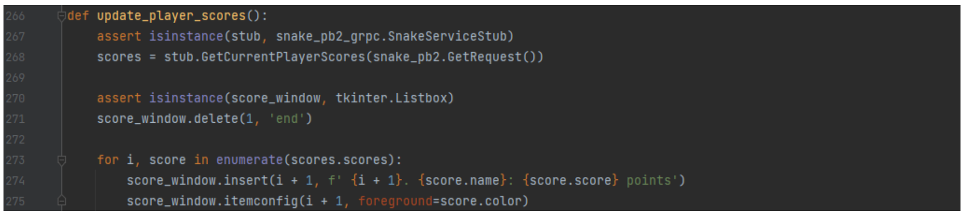


```python
    def add_food(self):
        x = random.randint(2, self.GAME_CONFIGURATION.max_x - 2)
        y = random.randint(2, self.GAME_CONFIGURATION.max_y - 2)
        snakes = []
        for snake in self.SNAKES.values():
            snakes.extend(snake.body)

        p = Point(x=x, y=y)
        while p in snakes:
            p = Point(
                x=random.randint(2, self.GAME_CONFIGURATION.max_x - 2),
                y=random.randint(2, self.GAME_CONFIGURATION.max_y - 2)
            )

        self.FOODS.append(p)
        return p
```

Figure 33: How the server add foods into the game in server.py

The `add_food()` function creates food at a random coordinate that is not under one of the snakes or on the border.

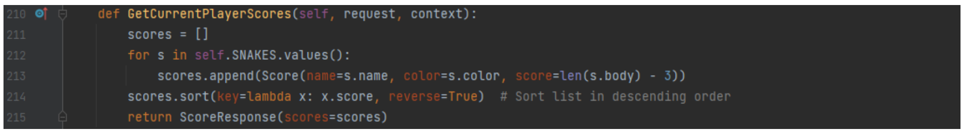
Lastly, `game_flow()` updates all the player scores:



```python
def update_player_scores():
    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    scores = stub.GetCurrentPlayerScores(snake_pb2.GetRequest())

    assert isinstance(score_window, tkinter.Listbox)
    score_window.delete(1, 'end')

    for i, score in enumerate(scores.scores):
        score_window.insert(i + 1, f' {i + 1}. {score.name}: {score.score} points')
        score_window.itemconfig(i + 1, foreground=score.color)
```

Figure 34: Updating the score list to the right of the game canvas in client.py

The function fetch all the scores from the server by sending it a request:



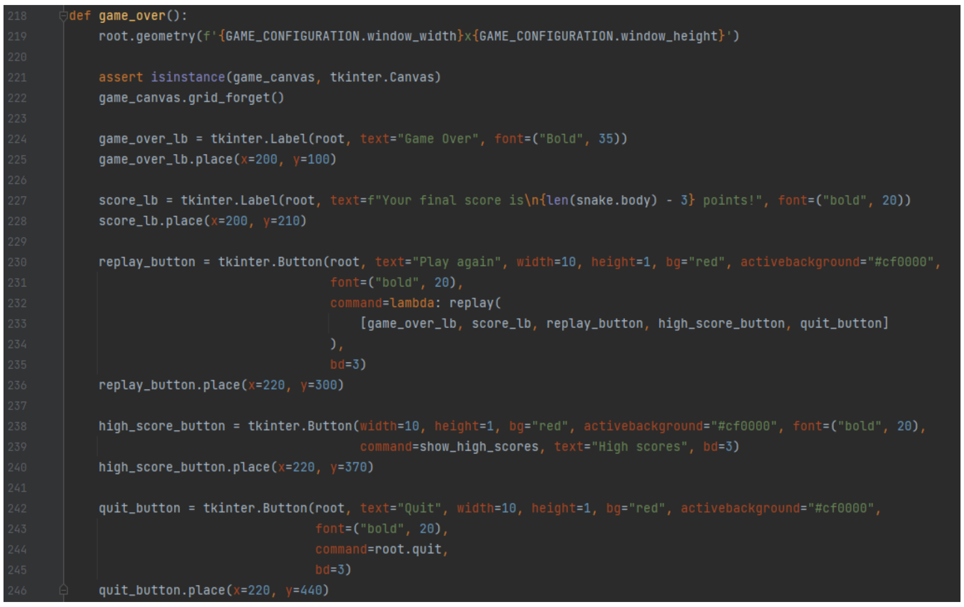
```python
    def GetCurrentPlayerScores(self, request, context):
        scores = []
        for s in self.SNAKES.values():
            scores.append(Score(name=s.name, color=s.color, score=len(s.body) - 3))
        scores.sort(key=lambda x: x.score, reverse=True)  # Sort list in descending order
        return ScoreResponse(scores=scores)
```

Figure 35: The server responds back with a sorted list of scores of current players in server.py

The client refreshes the score window by deleting every entries and print out an updated list of scores of each player in the game ranked by their accumulative score so far. Then, the client refreshes itself and performs the same function calls again in the same order.

2.3 The end of the game

The game will just continue in the same fashion as described in the previous section unless the snake that the player controls collides. This will happen during the function call `check_collision()` provided in figure 26. In this case, the client sends a requests to the server to kill the snake executed by the code in figure 18. From here `check_collisiion` will call the function `game_over()`:



```python
def game_over():
    root.geometry(f'{GAME_CONFIGURATION.window_width}x{GAME_CONFIGURATION.window_height}')

    assert isinstance(game_canvas, tkinter.Canvas)
    game_canvas.grid_forget()

    game_over_lb = tkinter.Label(root, text="Game Over", font=("Bold", 35))
    game_over_lb.place(x=200, y=100)

    score_lb = tkinter.Label(root, text=f"Your final score is\n{len(snake.body) - 3} points!", font=("bold", 20))
    score_lb.place(x=200, y=210)

    replay_button = tkinter.Button(root, text="Play again", width=10, height=1, bg="red", activebackground="#cf0000",
                                   font=("bold", 20),
                                   command=lambda: replay(
                                       [game_over_lb, score_lb, replay_button, high_score_button, quit_button]
                                   ),
                                   bd=3)
    replay_button.place(x=220, y=300)

    high_score_button = tkinter.Button(width=10, height=1, bg="red", activebackground="#cf0000", font=("bold", 20),
                                       command=show_high_scores, text="High scores", bd=3)
    high_score_button.place(x=220, y=370)

    quit_button = tkinter.Button(root, text="Quit", width=10, height=1, bg="red", activebackground="#cf0000",
                                 font=("bold", 20),
                                 command=root.quit,
                                 bd=3)
    quit_button.place(x=220, y=440)
```

Figure 36: The code that produces the game over page in client.py

With the code above, the player should see this following page:

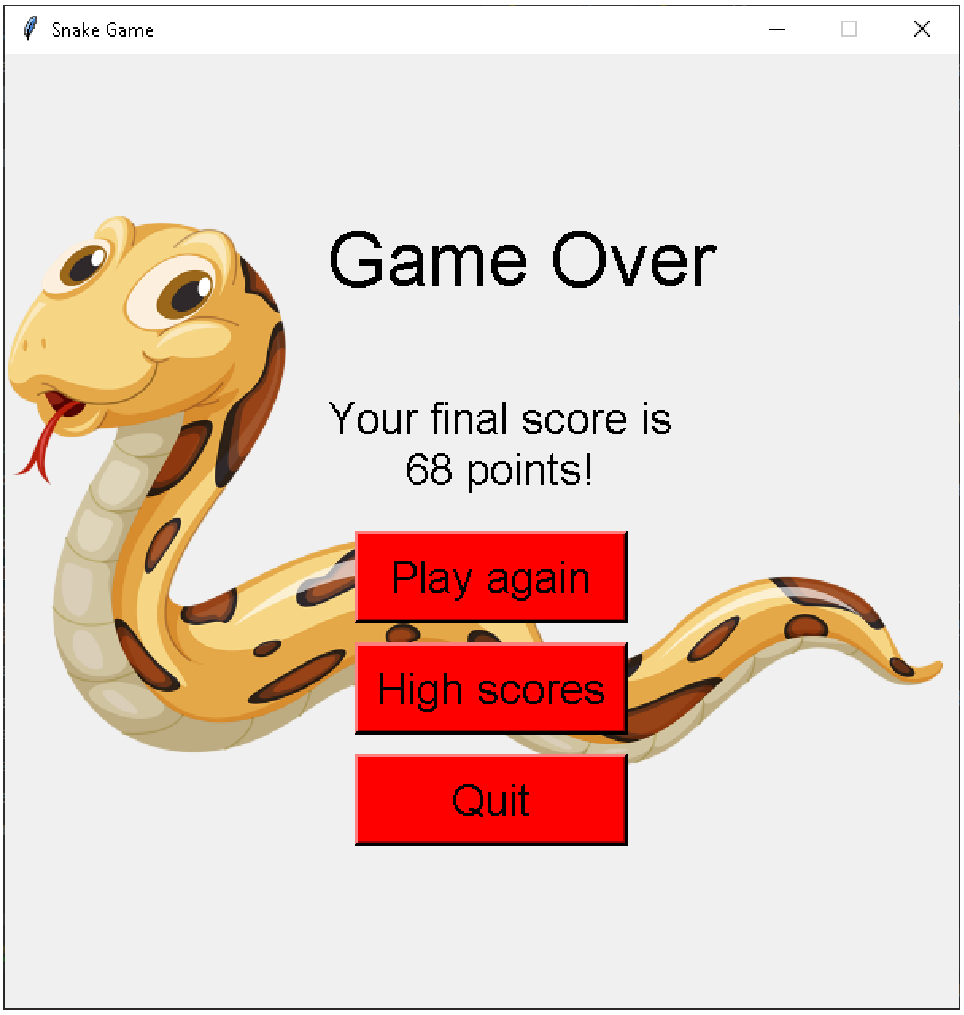
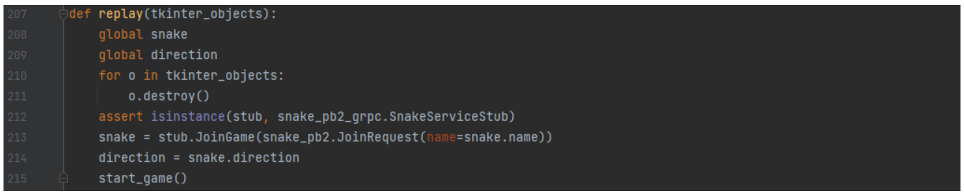


Figure 37: The game over page when your snake has collided

If the player wants to rejoin the game and play again by clicking on the "Play again" button, the following code will be exectud:

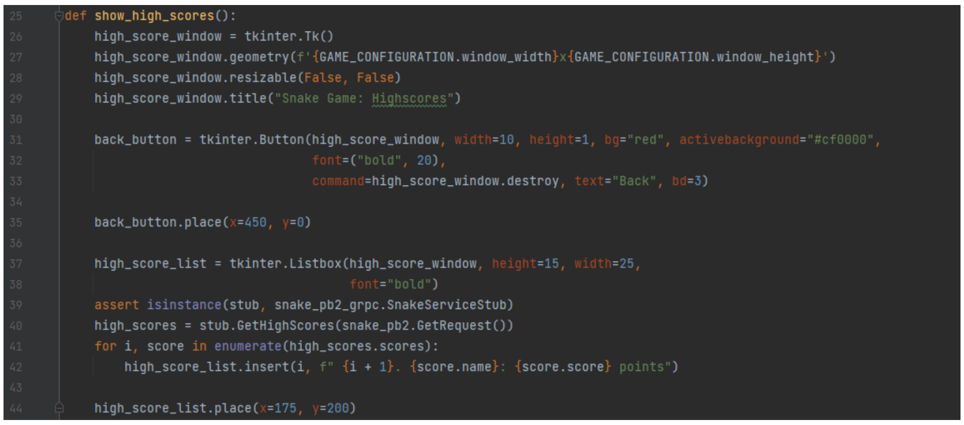


```python
def replay(tkinter_objects):
    global snake
    global direction
    for o in tkinter_objects:
        o.destroy()
    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    snake = stub.JoinGame(snake_pb2.JoinRequest(name=snake.name))
    direction = snake.direction
    start_game()
```

Figure 38: The code executed when player clicks "Play again" in the game over window shown in figure 37

The code simply requests the server to create a new snake and send that snake to the player. The code that the server will use to respond to the request is provided in figure 14. Then it executes `start_game` provided in figure 15.

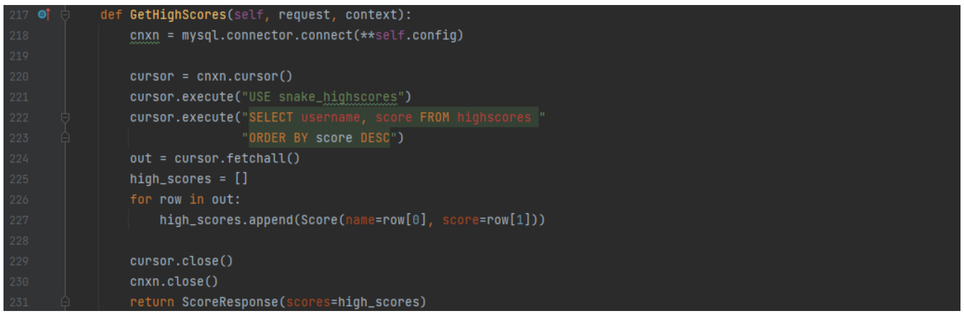
If the player wants to see the list of high score holders, it can be accessed by clicking on the "High scores" button during the index page and game over page as seen in figure 1 and figure 37 respectively. This will call the function `show_high_scores()`:



```python
def show_high_scores():
    high_score_window = tkinter.Tk()
    high_score_window.geometry(f'{GAME_CONFIGURATION.window_width}x{GAME_CONFIGURATION.window_height}')
    high_score_window.resizable(False, False)
    high_score_window.title("Snake Game: Highscores")

    back_button = tkinter.Button(high_score_window, width=10, height=1, bg="red", activebackground="#cf0000",
                                 font=("bold", 20),
                                 command=high_score_window.destroy, text="Back", bd=3)

    back_button.place(x=450, y=0)

    high_score_list = tkinter.Listbox(high_score_window, height=15, width=25,
                                      font="bold")
    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    high_scores = stub.GetHighScores(snake_pb2.GetRequest())
    for i, score in enumerate(high_scores.scores):
        high_score_list.insert(i, f" {i + 1}. {score.name}: {score.score} points")

    high_score_list.place(x=175, y=200)
```

Figure 39: Code executed when clicking on "High scores" button shown in figure 1 and figure 37.

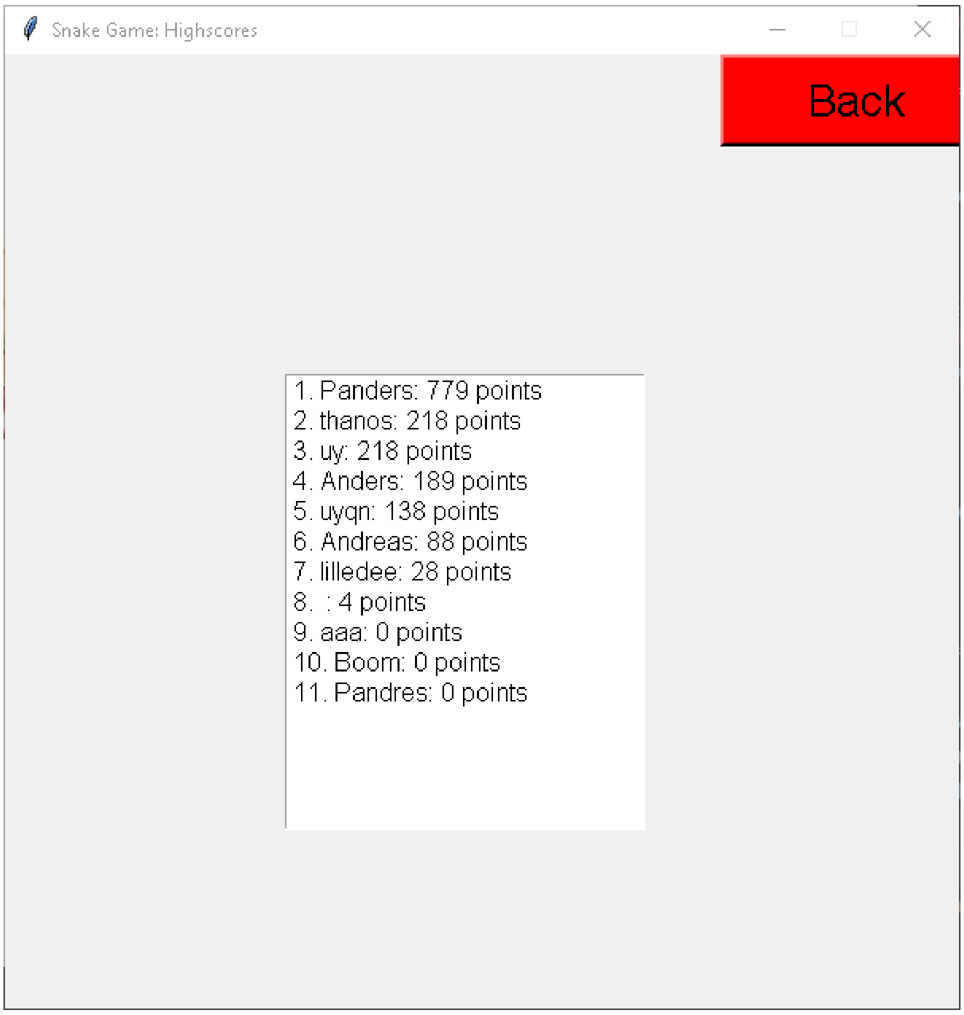
The client will then send a request to the server:



```
    def GetHighScores(self, request, context):
        cnxn = mysql.connector.connect(**self.config)

        cursor = cnxn.cursor()
        cursor.execute("USE snake_highscores")
        cursor.execute("SELECT username, score FROM highscores "
                       "ORDER BY score DESC")
        out = cursor.fetchall()
        high_scores = []
        for row in out:
            high_scores.append(Score(name=row[0], score=row[1]))

        cursor.close()
        cnxn.close()
        return ScoreResponse(scores=high_scores)
```

Figure 40: High scores response from the server in server.py

where the server will query the SQL database for stored high scores and send the list to the client sorted in descending order. Consequently, the client prints out a list of high scores in a seperate window and displays it to the player:



```
1. Panders: 779 points
2. thanos: 218 points
3. uy: 218 points
4. Anders: 189 points
5. uyqn: 138 points
6. Andreas: 88 points
7. lilledee: 28 points
8.  : 4 points
9. aaa: 0 points
10. Boom: 0 points
11. Pandres: 0 points
```

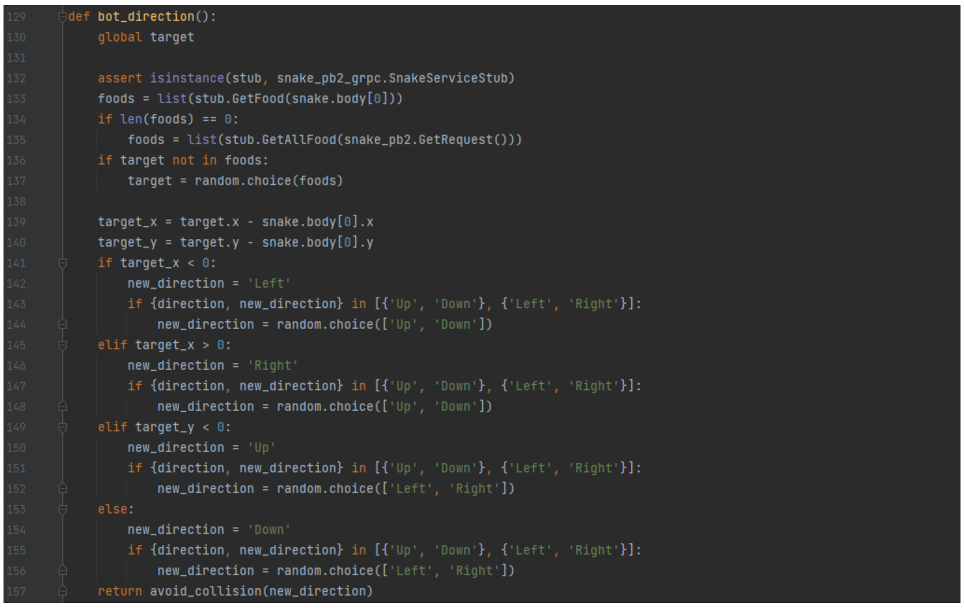
Figure 41: A list of current high score holders

2.4 Our simple bots that dies too fast

Since we were aiming to complete one of the stretch goals to monitor network traffic and resource usage we added bots into our game. To start a bots one can simply execute the following command:

```
> py client.py --bot
```

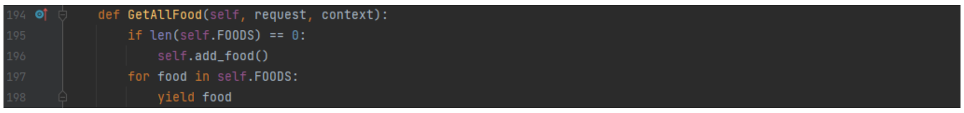
This will start executing the `main()` function in figure 8 in which it will skip the `show_index_page()` in figure 12 and just start the game like in figure 21. The bot will also execute `game_flow()`. However, the `move_snake()` (figure 23) is a little bit different. As we can see from the code, the direction that the bot will send to the server is determined by the following code:



```python
def bot_direction():
    global target

    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    foods = list(stub.GetFood(snake.body[0]))
    if len(foods) == 0:
        foods = list(stub.GetAllFood(snake_pb2.GetRequest()))
    if target not in foods:
        target = random.choice(foods)

    target_x = target.x - snake.body[0].x
    target_y = target.y - snake.body[0].y
    if target_x < 0:
        new_direction = 'Left'
        if {direction, new_direction} in [{'Up', 'Down'}, {'Left', 'Right'}]:
            new_direction = random.choice(['Up', 'Down'])
    elif target_x > 0:
        new_direction = 'Right'
        if {direction, new_direction} in [{'Up', 'Down'}, {'Left', 'Right'}]:
            new_direction = random.choice(['Up', 'Down'])
    elif target_y < 0:
        new_direction = 'Up'
        if {direction, new_direction} in [{'Up', 'Down'}, {'Left', 'Right'}]:
            new_direction = random.choice(['Left', 'Right'])
    else:
        new_direction = 'Down'
        if {direction, new_direction} in [{'Up', 'Down'}, {'Left', 'Right'}]:
            new_direction = random.choice(['Left', 'Right'])
    return avoid_collision(new_direction)
```

Figure 42: How the bot determines its movement in client.py

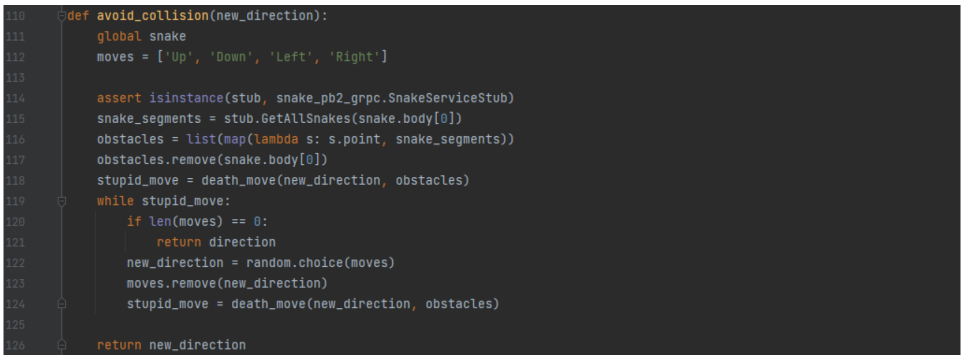
We have instructed the bot to request the foods within view from the server and lock onto one of them randomly. However, if there are no foods within view we tell the bot to choose a random food outside of view by requesting all the food present in the game from the server:



```python
    def GetAllFood(self, request, context):
        if len(self.FOODS) == 0:
            self.add_food()
        for food in self.FOODS:
            yield food
```

Figure 43: Server responds with a stream of food when client request all the food

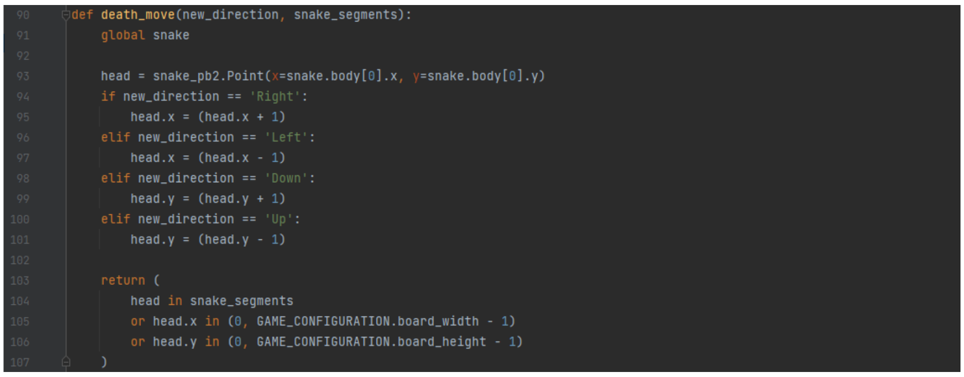
Then, we tell the bot to figure out where the food is and ask them to move towards the food, first in the horizontal direction then in the vertical direction. Also, we tell the bot to move randomly if the intended direction is an opposite direction of its course. At the end, we tell it return the direction that will prevent the bot to collide with `avoid_collision()`:



```python
def avoid_collision(new_direction):
    global snake
    moves = ['Up', 'Down', 'Left', 'Right']

    assert isinstance(stub, snake_pb2_grpc.SnakeServiceStub)
    snake_segments = stub.GetAllSnakes(snake.body[0])
    obstacles = list(map(lambda s: s.point, snake_segments))
    obstacles.remove(snake.body[0])
    stupid_move = death_move(new_direction, obstacles)
    while stupid_move:
        if len(moves) == 0:
            return direction
        new_direction = random.choice(moves)
        moves.remove(new_direction)
        stupid_move = death_move(new_direction, obstacles)

    return new_direction
```

Figure 44: The code that attempts to predict a direction that will prevent the bot from colliding in client.py.

With the code shown above, the bot starts with a list of direction that it can choose from. It then request the server to give it all the snakes within its view (figure 29) and remove its own head from that list. Then it checks if the direction given to it is a move that will determine its fate with `death_move()`:



```python
def death_move(new_direction, snake_segments):
    global snake

    head = snake_pb2.Point(x=snake.body[0].x, y=snake.body[0].y)
    if new_direction == 'Right':
        head.x = (head.x + 1)
    elif new_direction == 'Left':
        head.x = (head.x - 1)
    elif new_direction == 'Down':
        head.y = (head.y + 1)
    elif new_direction == 'Up':
        head.y = (head.y - 1)

    return (
        head in snake_segments
        or head.x in (0, GAME_CONFIGURATION.board_width - 1)
        or head.y in (0, GAME_CONFIGURATION.board_height - 1)
    )
```

Figure 45: The code that determines if the bot will collide when moving in a certain direction in client.py.

If the code determines that the bot will collide, it will remove that direction from the list of moves and try again with one of the remaining possible directions. When the list is exhausted, it means that the bot has to accept its fate and just collide. Whatever the result is, the bot will assign its direction and request the server to move the snake (figure 24) regardless. Other than that, the `game_flow()` (figure 22) is the same as the one we documented in the previous respective section. However, the server will not save the score of the bot (figure 20) nor will the game over page be displayed (figure 37) when the bot dies. The game will simply quit and display the score in the console/terminal.

The bot could have been better implemented to play more optimally. However, this was enough for the purpose of monitoring the resource usage. However, machine learning could have been implemented to make bots a more worthy opponent for the players. Since, this course is not a course about artificial intelligence, thus we ceased the opportunity to create an intelligent bot. Maybe, we could implement a smarter bot if we enroll in a course about AI using this code base.

Before we end this section, we would like to emphasize that we have a script start_bots.cmd script that can start multiple bots concurrently with Windows PowerShell, e.g:

```
> ./start_bots 4
```

will yield the following result:

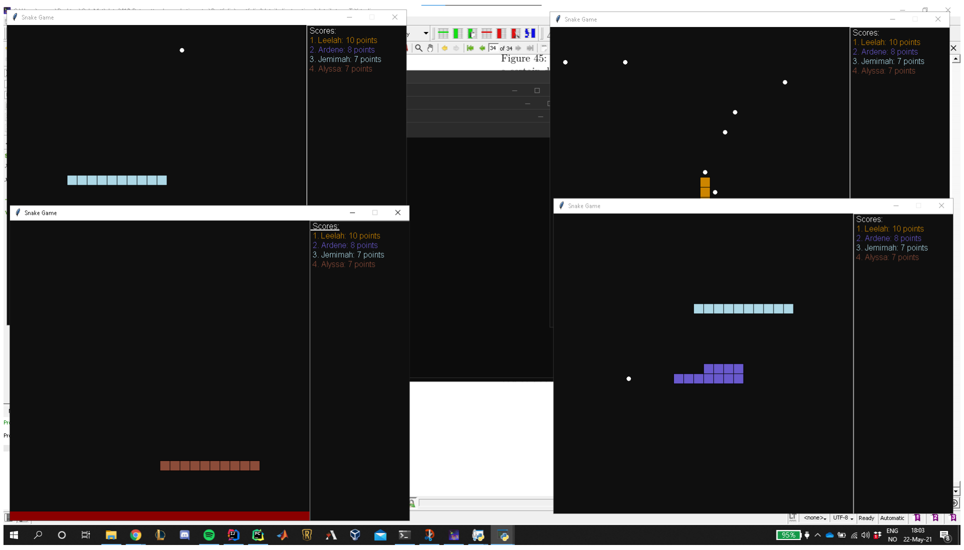


Figure 46: 4 bots playing against each other

3 Task 2: A multiplayer game, our solution

In this section we will be discussing and answering all the required and non-mandatory tasks provided to be fulfilled. This section of the document is a more detailed version of the README.md that is included in this project. Moreover, it will also serve as an insight on our thoughts, answers and reflection on the given tasks.

Before, discussing further, we would like to provide our reason for choosing this task in contrast to “Task 1: A web shop with a REST API”. Since we all the members of this group was gaming enthusiast it was natural that developing a game was more appealing. Furthermore, developing a REST API was already given as a mandatory assignment in this course. Hence, we wanted to challenge ourselves by doing something else. Thus, developing a gRPC server will provide us with new experiences and equip us with more technologies in our reportaire.

3.1 Technology choices

From the given list of suggested technologies we were recommended to choose from. It felt natural for us to go with “Python 3 with tkinter for graphics and gRPC for networking”. The degree that we all have enrolled in has taught us mostly Java. Moreover, this course has in many occassions demonstrated networking concepts using Python 3. We took it as our opportunity to learn Python 3 and felt that, given the popularity of the language, this will serve as good practice to further develop projects with the language.

3.2 User stories

The required goals to fulfill was stated as user stories. In this section, we shall attempt to give our answers to these stories

3.2.1 Playing alone

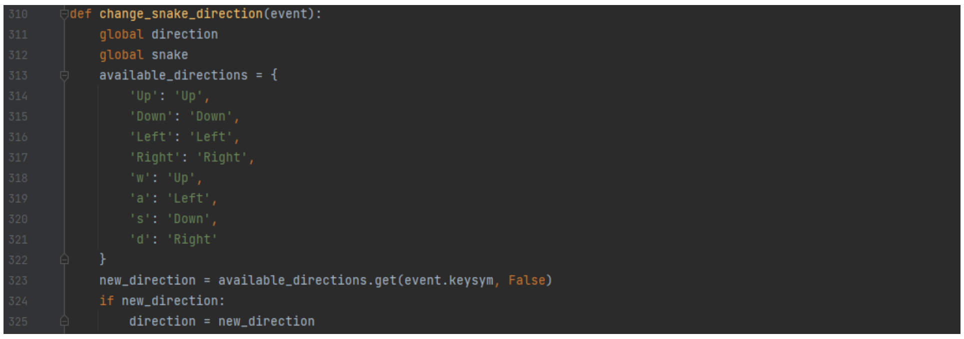
“I can start the program and I easily get to start playing alone even if no other players are connected.”

By following the instructions provided in 1 how to run the game it is possible to start the game even when no other players are connected. This has also been demonstrated in figure 21 under the previous section 2.2 The game_flow() of the game.

3.2.2 Moving the snake

“ can move the snake around the board using the keyboard to avoid obstacles (walls or players) or steer towards rewards (e.g. food - optionally dead players can turn into food).”

It might not have explicitly stated, but the code provided in figure 15 line 366 of client.pybinds all the keys to an event that calls the function `change_snake_direction()`:



```python
def change_snake_direction(event):
    global direction
    global snake
    available_directions = {
        'Up': 'Up',
        'Down': 'Down',
        'Left': 'Left',
        'Right': 'Right',
        'w': 'Up',
        'a': 'Left',
        's': 'Down',
        'd': 'Right'
    }
    new_direction = available_directions.get(event.keysym, False)
    if new_direction:
        direction = new_direction
```

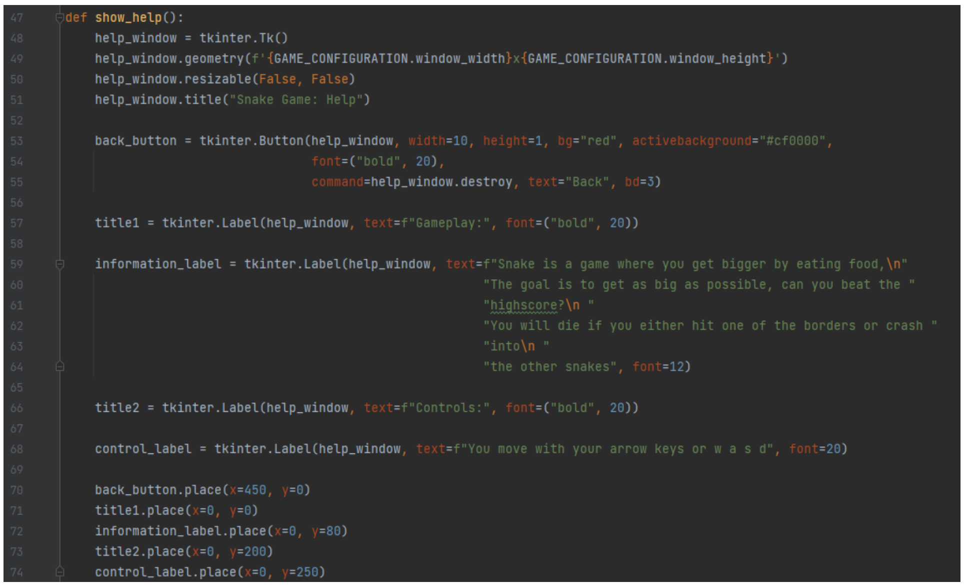
Figure 47: This code allows the player to move their snake with arrow keys and W, A, S, D on their keyboard in client.py

With the code shown above it is possible to move your snake with the arrow keys and W, A, S, D with your keyboard. The code above also disable other keys such that nothing happens if you press on other keys by accident. Furthermore, the server also ensures that dead players turn into food with the code provided in figure 19.

3.2.3 How to move the snake

"The game tells me which keys to use so that I don't have to refer to the documentation."

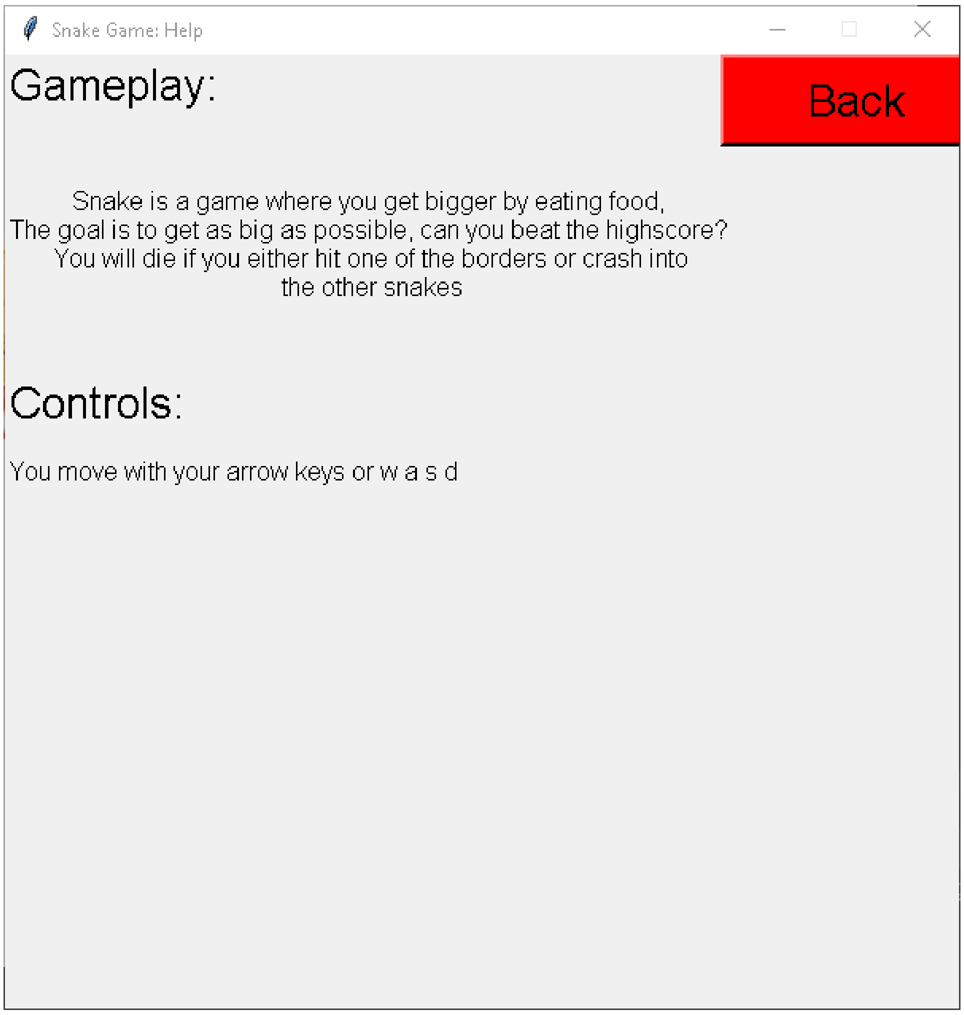
As mentioned in 1.1 How to run the game players can click on the "Help" button which calls the `show_help()` function:



```python
def show_help():
    help_window = tkinter.Tk()
    help_window.geometry(f'{GAME_CONFIGURATION.window_width}x{GAME_CONFIGURATION.window_height}')
    help_window.resizable(False, False)
    help_window.title("Snake Game: Help")

    back_button = tkinter.Button(help_window, width=10, height=1, bg="red", activebackground="#cf0000",
                                 font=("bold", 20),
                                 command=help_window.destroy, text="Back", bd=3)

    title1 = tkinter.Label(help_window, text=f"Gameplay:", font=("bold", 20))

    information_label = tkinter.Label(help_window, text=f"Snake is a game where you get bigger by eating food,\n"
                                                        "The goal is to get as big as possible, can you beat the "
                                                        "highscore?\n "
                                                        "You will die if you either hit one of the borders or crash "
                                                        "into\n "
                                                        "the other snakes", font=12)

    title2 = tkinter.Label(help_window, text=f"Controls:", font=("bold", 20))

    control_label = tkinter.Label(help_window, text=f"You move with your arrow keys or w a s d", font=20)

    back_button.place(x=450, y=0)
    title1.place(x=0, y=0)
    information_label.place(x=0, y=80)
    title2.place(x=0, y=200)
    control_label.place(x=0, y=250)
```

Figure 48: Code to show the help page in client.py.

The code above should produce the following window:



Gameplay:

Snake is a game where you get bigger by eating food,
The goal is to get as big as possible, can you beat the highscore?
You will die if you either hit one of the borders or crash into
the other snakes

Controls:

You move with your arrow keys or w a s d

Figure 49: A window with information about the game produced by the code in figure 48.

However, the case could be made that the controller information could be advantagous during the game also.

3.2.4 Ways to kill a snake

"If I run into walls (if there are any) or other players it will kill me. If other players run into me it will kill them."

This part is covered by `check_collision()` (figure 26) and `CheckCollision()`

(figure 27) in conjunction with `KillSnake()` (figure 18) during `game_flow()` in 2.2 The game_flow() of the game.

3.2.5 Game over

"I clearly see an indication that I've died if I run into another player or a wall "

It is indicated by the game over page (figure 36) that the snake has collided with another snake, itself or another wall.

3.2.6 Scroll with snake direction

"If there are no outer walls in the game (e.g. like slither.io) will scroll with the direction of my snake when I move it."

The code implemented in `scroll_lock_movement()` (figure 25) ensures that the view of the player scrolls with the snake.

3.2.7 List of connected users

"I can clearly see if other players connect (including bots if any), in a list of connected users. A minimum of 2 players must be supported."

As per demonstration in figure 46, each of the 4 windows where the bots are playing, there is a list to the right of each window that shows which players are connected together with their score and respective ranks of the current game. However, it might be difficult to distinguish any of the names in the list from bots, but we purposely did not want players to know which connected players are bots. The reason being, that this project might be reused in other project e.g. in AI courses as discussed earlier in 2.4 Our simple bots that dies too fast.

3.2.8 Return of the bots

"There may be other players operated by the software (e.g. return of the bots - as snakes!) but this is not a requirement. If there are bots they behave like any other player and I won't be able to tell the difference (except maybe if they play really badly or superhumanly well)".

As discussed in 2.4 Our simple bots that dies too fast we have indeed implemented bots that can be initiated in our project. However, during the subsection we also discussed the issues that arises from our implementation of the bots, i.e. they are not very smart. However, it could be that they are indistinguishable from bad players.

3.2.9 Growing the snake

“I can get the snake to grow by running into ”food”. The food can be just colored squares but I can see that they’re different from walls if there are any.”

The food is represented by white dots on the game screen just like the ones visible in figure 46. It might be hard to distinguish the length in figure 46 but comparing that figure with figure 21 it is indeed comparable in terms of the snake size. The code that is responsible for growing the snake is howevered covered by `MoveSnake()` (figure 24).

3.2.10 Snake length and score proportionality

“I get points for getting a bigger snake and I can see how many points I have.”

Each time a snake is eating food it gets one segment longer and the player adds 1 point to their total score. This is again demonstrated in figure 46 that there is a difference in score in the list to the right of the game. Also, it is clear by comparing the demonstrations in figure 21 and figure 46 that their length and scores are proportional.

3.3 Requirements and Implementation options

3.3.1 Minimum of 2 players

“A minimum of 2 players must be supported”

We have demonstrated in figure 46 that at least 4 players can play the game at the same time. We have also tested with more players running the game concurrently. We will therefore consider this requirement fulfilled.

3.3.2 Implement more games

“You can implement more games than one if you want. You can also choose another game, but make sure to ask the teacher for advice if you do.”

Due to time constraints and that some of us has enrolled into extra courses this year, consequently had to focus on exams for the additional courses we chose to just focus on the one suggested game (snake).

3.4 Deployment with Docker

“There must be a dockerfile for the server that allows you to start a game server using docker build / docker run commands. It should then be possible to deploy your game server in a public cloud making it available for players across the internet.”

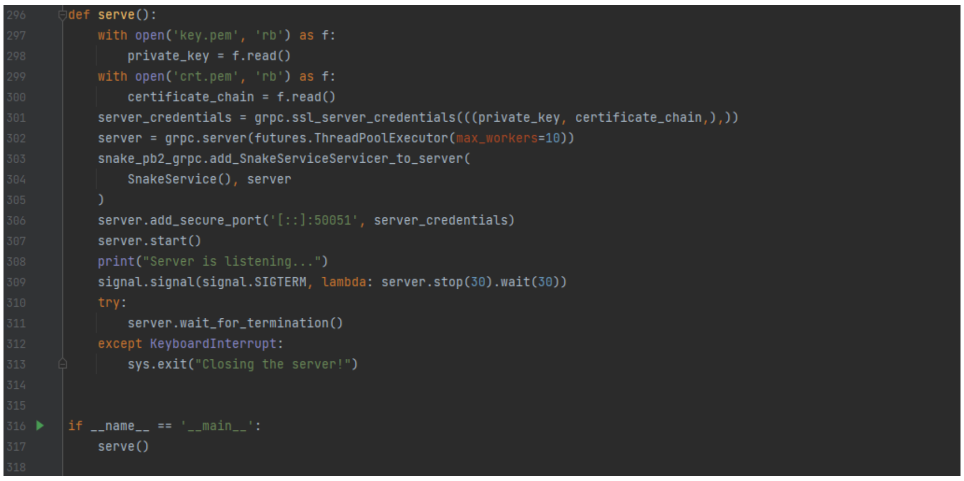
The project folder includes a Dockerfile. Which can be built and run according to the instructions provided in 1.1 How to run the game. We also mentioned in the next section 1.2 How to run the game with monitoring that we also have included a docker-compose.ymland how to run it with several pre-specified containers that serves as monitoring.

3.5 Stretch goals

3.5.1 Securing communications with TLS

"Secure all communication with TLS. Look at `https://www.grpc.io/docs/guides/auth/#examples` for examples in many languages."

By following the link above we have successfully secured all the communications with TLS. This can be seen under `establish_stub()` (figure 10) and the code written under the `serve()` function:



```python
def serve():
    with open('key.pem', 'rb') as f:
        private_key = f.read()
    with open('crt.pem', 'rb') as f:
        certificate_chain = f.read()
    server_credentials = grpc.ssl_server_credentials(((private_key, certificate_chain,),))
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    snake_pb2_grpc.add_SnakeServiceServicer_to_server(
        SnakeService(), server
    )
    server.add_secure_port('[::]:50051', server_credentials)
    server.start()
    print("Server is listening...")
    signal.signal(signal.SIGTERM, lambda: server.stop(30).wait(30))
    try:
        server.wait_for_termination()
    except KeyboardInterrupt:
        sys.exit("Closing the server!")


if __name__ == '__main__':
    serve()
```

Figure 50: Function that runs the server when the python script is called in server.py.

As we can see from the code above that the server requires two files key.pem and crt.pem which are a private key and a certificate respectively. These were produced with the openssl using the command in Windows PowerShell

```
> openssl req -newkey rsa:2048 -nodes -keyout key.pem -x509 -days
    ↪ 365 -out crt.pem -subj /CN=snakenet
```

From here we see that we specified the common name as "snakenet" thus the global variable "hostname" in client.py(figure 9) was assigned "snakenet". Furthermore, line 412 in `establish_stub()` (figure 10) became the missing piece to successfully pass the TLS handshake when we ran the game server in a docker container.

3.5.2 Monitoring with Prometheus

"Add monitoring using Prometheus to track the resource usage of your game server. Document how the resource usage changes when many players are connected."

As mentioned in 1.2 How to run the game with monitoring, docker-compose.yml made it easy to start all of the containers necessary to fulfill this goal. As required, we had Prometheus scrape the container that ran with cAdvisor, as configured in prometheus.yml. Furthermore, cAdvisor was collecting metrics of all the running containers including our game server. Then, we used Grafana, that uses the data scraped by Prometheus, for its beautiful dashboard and easy to read graphs to monitor our game server.

Furthermore, we tried to monitor the resources by starting up 12 bots with

```
> ./start_bots 12
```

What we noticed is that it started to lag alot. We will assume that this is due to the amount of food being calculated to return back to the client by the server. We know this is because every client is actually starting a thread to randomly add extra food to the game board. Consequently, the server must loop through a pretty long and perform many calculations to return the food within view to each client and the `game_flow()` does not get to refresh itself before executing all the functions discussed earlier. After a time, two bots were remaining. We can see the amount of food present in the client for these two bots in the figure below:

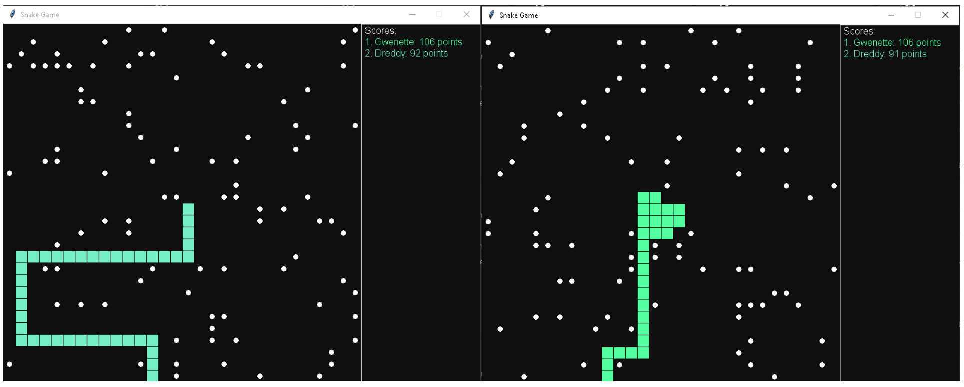


Figure 51: The 2 bots that survived the 12 bot battle royale.

We also managed to capture these graphs from Grafana of the game:

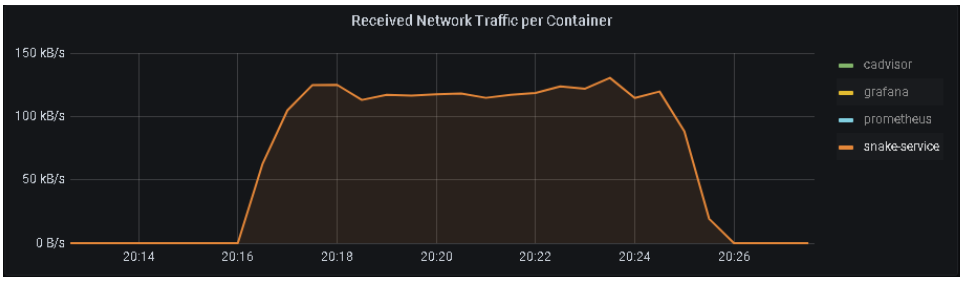


Figure 52: Graph from Grafana: Received network traffic of snake service container

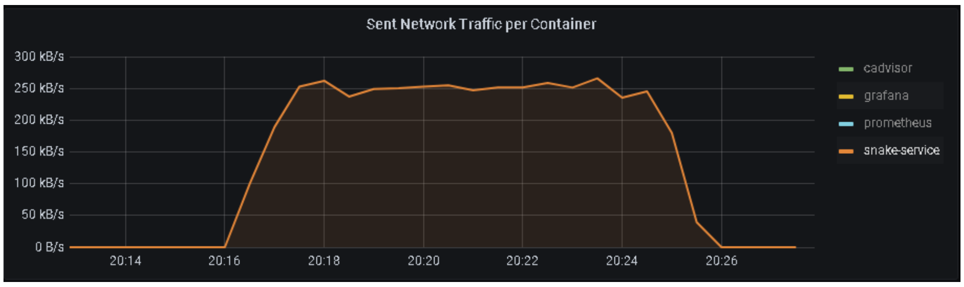


Figure 53: Graph from Grafana: sent network traffic of snake service container

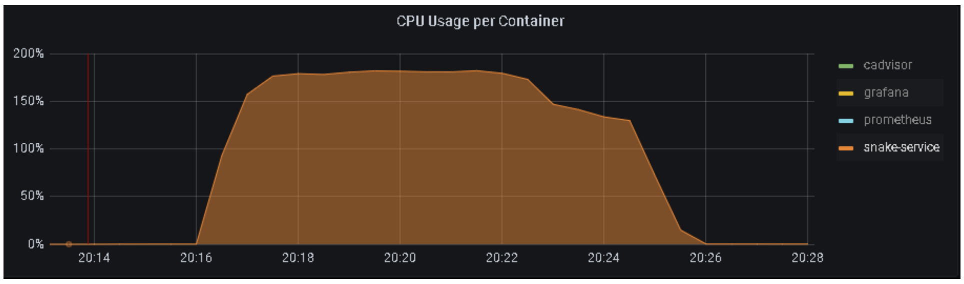


Figure 54: Graph from Grafana: CPU usage of snake service container

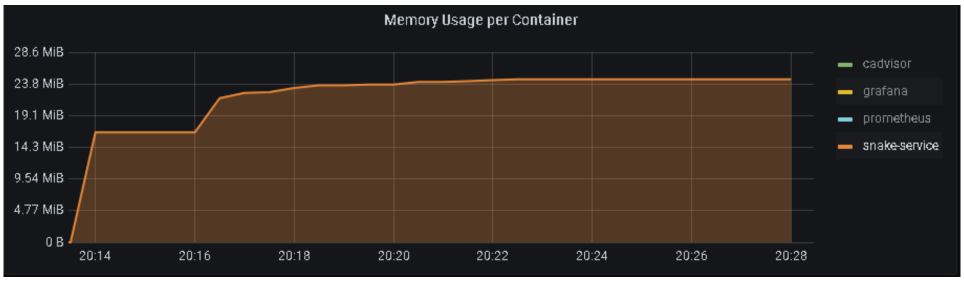


Figure 55: Graph from Grafana: Memory usage of snake service container

From the 4 graphs from grafana we can see that the network traffic, both received and sent is actually pretty good. The issue is definitely not the underlying gRPC implementation but rather the implementation that we ourselves coded, reflected upon the cpu usage. We can deduce that the issue stems from the sheer amount of computations required before responding back to the client is the issue. In addition, the lag that we see, we assume, is due to the sequence of function calls in `game_flow()` becomes very slow when it becomes alot of objects to draw on the canvas.

3.5.3 Many simultaneous players

"Allow for many (> 10) or unlimited players. This will require you to manage a large grid and you probably have to make the game board itself scroll around the snake than to make the snakes move around the game board."

During the previous section 3.5.2 Monitoring with Prometheus we initiated a game of 12 players. Given the code in written in figure 11 we can see that it opens two json files tkinter-colors.json and bot-names.json these two files provide colors and names to the bots respectively. We have not tried but we assume given the code we have written that the amount of players that can concurrently run the game is limited by the number of colors in tkinter-colors.json. Also, as discussed earlier, the function `scroll_lock_movement()` (figure 25) makes the view scroll with respect to the head of the snake. In fact, we actually made the game board 16 times bigger than the window view to account for many players. However, with the lag present, in might not be a good experience to play with that many players.

3.5.4 Adding bots

“Add bots. This is a good way to test the scalability of your service by pushing it to the limit.”

We did indeed add bots as mentioned before. However, this implementation of the game is not at all scalable. Given more time, we believe our group could have managed to make the game scalable by refactoring the code with better algorithms.

3.5.5 Database for storing high scores

“Add a persistent high-score list with a database backend. Note that high-score lists can be fairly hard to keep from being hacked (I’ve tried!). Some intelligent notes on why that is, and a possible solution would be an interesting read.”

A database has been added to the server side of the code to keep track of the scores. The database was created in Google Cloud Platform. Thus, we chose to not run a docker container for the database. The game has implementation to query the database when a snake dies. One can also view a list of high scores of players by clicking on the ”High scores” button at the start of the game or at the ”game over” page as demonstrated in figure 41. Since, the database is in Google Cloud Platform, it will not persist after their given deadline, which should be mid August. A solution to this could be to create a docker container that runs MySQL with configurations of our choosing.

With respect to the database security, by connecting to the database in the server.py-file, some security issues arise. Most obviously, the login-information to the database becomes visible to anyone who possesses access to the file. We solved this by creating an extra user in the database (”app-user”) with very specific rights, instead of using the root-user. Although there is no crucial information in the database, we wanted to limit the risk of hackers accessing it and deleting information. However, this does not prevent the “app-user” user to inserting false data to the database. Another way to solve this is maybe encrypting access with an authentication protocol. However, due to time constraints, we did not explore further security measures for our service.