

Documentation for Snake Multiplayer solution

Bartłomiej Węgrzyn, Antoni Pietryga, Norbert Ozga

INDEX

Introduction - main goal of the software	2
1. Server - usage and games handling	3
2. Server - communication, types of messages	4
2.1. Successful and non-successful responses	4
2.2. User registration	5
2.3. Listing games	5
2.4. Creating games	5
2.5. Joining games	6
2.6. Leaving games	6
2.7. Game states	6
2.8. Game logic handling	6
3. Client - usage	8
4. Client - communication, types of messages	16
4.1. Every step is checked	16
4.2. User register request	16
4.3. Games listing request	16
4.4. Games creating request	17
4.5. Games joining request	17
4.6. Games leaving request	17
4.7. Sending moves requests	17
4.8. Interpreting game states	17
5. Encryption	19
6. Project highlights	20
7. System requirements	21
8. Problems we had, compromises	22
9. Our feelings about project - what we are proud of	23
10. Project authors	24

Introduction - main goal of the software

The main goal is to create an interactive multiplayer snake game. The assumptions were easy - to create a functional server that will be able to handle multiple games at the same time and a half-GUI client to manage games, create new ones and join them for play with friends.

Both server and client use our own data protocol called *Venom*, which handles exchanged data and compresses it to send as few as possible bytes to avoid too high bandwidth usage or latencies between programs.

Venom protocol is precisely described in its own documentation.

The next chapters shortly describe both client and server.

1. Server - usage and games handling

The server application is a Python code which accepts clients after start. There is no need to provide any parameters to run it - it works automatically. While the server is running, any usable information is logged with the exact date and time to the file *logs.txt* to see what's happening at the moment. These information may be useful for debugging purposes.

The whole game data and users information are exchanged in a dictionary provided by *Venom* protocol. The main fields used in this dictionary for storing information are:

- `nickname` - provides nickname selected during user registration
- `user_id` - provides user identification number given by server
- `game_name` - provides game name selected during game creation
- `game_id` - provides game identification number given by server
- `move` - provides pressed key information for snake manipulation
- `p1_snake` - provides data for player 1's snake
- `p2_snake` - provides data for player 2's snake
- `food` - provides coordinates of generated food
- `pt1` - provides number of points for player 1
- `pt2` - provides number of points for player 2
- `p1_over` - provides information about player 1's game over
- `p2_over` - provides information about player 2's game over

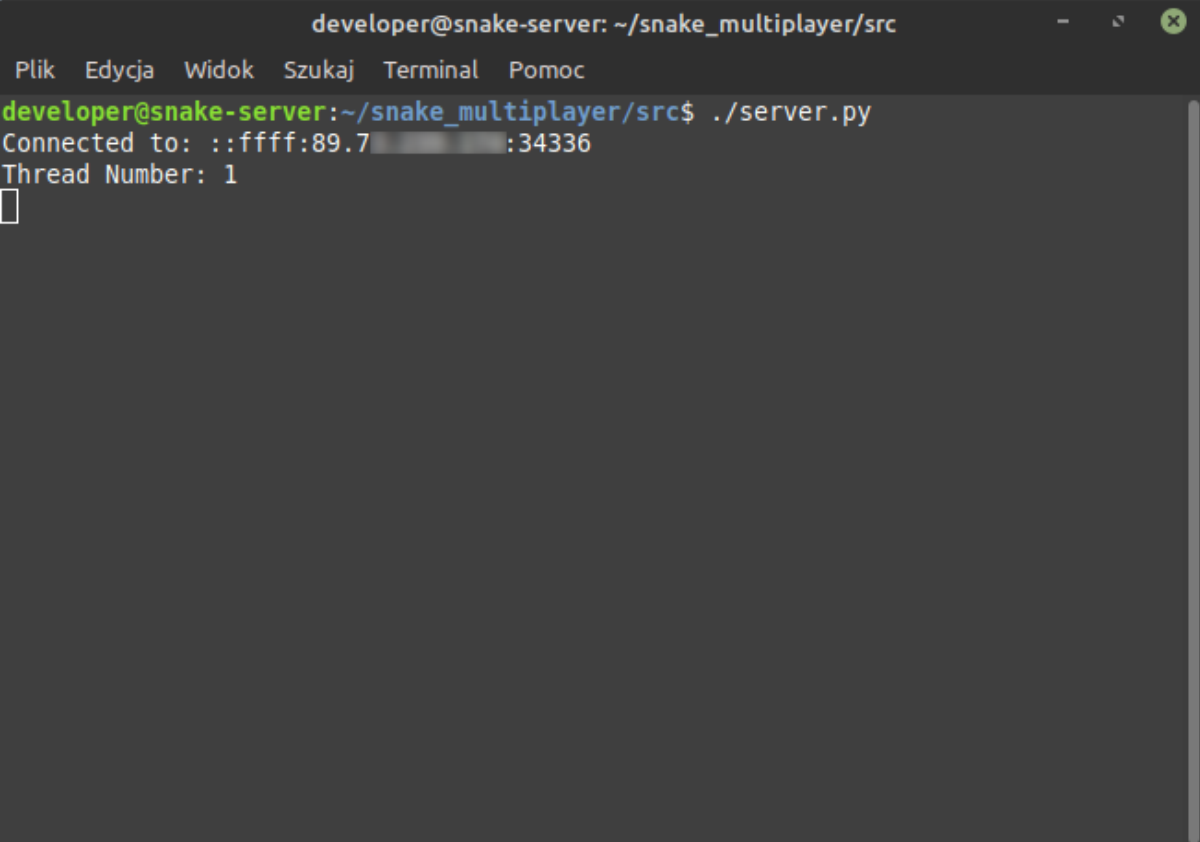
Every game tick generates a new game state that is then sent to the client application, which then draws the game on the screen. Meantime, the server is able to accept new snake direction, that can be sent by client.

As stated before, the whole communication process and events, while the server is running, are saved to *logs.txt* file, which may be very useful for system operator (a.k.a. admin) which maintains the server.

The server can be terminated easily, by hitting CONTROL-C combination, which is a popular and basic keyboard interrupt combination.

2. Server - communication, types of messages

The server communicates using the *Venom* protocol, which, as stated before, is exactly described in its own documentation. However, this protocol is only one layer lower in the whole communication process, which consists of dictionary-like messages with their own types and data stored within. In fact, the very early version of our solution was based on JSON messages handling all necessary dictionaries and *Venom* is a component which tidies the whole communication up and compresses it. However, underneath data exchanging still bases on dictionary messages which may or may not have fields handling necessary information. This chapter shortly describes all the messages and its fields the server may send to the client.

A screenshot of a terminal window titled 'developer@snake-server: ~/snake_multiplayer/src'. The window has a menu bar with 'Plik', 'Edycja', 'Widok', 'Szukaj', 'Terminal', and 'Pomoc'. The terminal shows the command 'developer@snake-server:~/snake_multiplayer/src\$./server.py' being executed. The output is 'Connected to: ::ffff:89.7[REDACTED]:34336' followed by 'Thread Number: 1' and a small cursor icon on the next line.

```
developer@snake-server: ~/snake_multiplayer/src
Plik  Edycja  Widok  Szukaj  Terminal  Pomoc
developer@snake-server:~/snake_multiplayer/src$ ./server.py
Connected to: ::ffff:89.7[REDACTED]:34336
Thread Number: 1
█
```

Picture 1: Server accepting its first connection.

2.1. Successful and non-successful responses

When a message arrives, it may be correct or not. The server tries to properly understand any received data. If it is possible and the desired operation is possible to execute, the server returns a message with the *operation_successful* flag in the dictionary set on. Otherwise, this flag isn't set. Precisely, this flag may be *True* or *False* respectively. Of course it's not the only data sent back to the client. Returned message contains all data that may be

sent - user ID, game ID or snakes' coordinates. It depends on what message actually arrived to the server and if it could be properly understood.

2.2. User registration

User registration is a process in which user sends to the server desired nickname and expects given identification number from the server. Besides the message type that has to be set in header for every message, the only data that is required by server for registration is the nickname. If desired nickname is reserved, the server returns, that operation was not successful, otherwise it was successful and the user ID is returned. Now user gets possibility to list, create and join games. According to *Venom's* documentation, the expected message type for this operation from client is *LOGIN_CLIENT*, while server response's message type is *LOGIN_SERVER*.

2.3. Listing games

Listing available games is a process in which client retrieves from server currently created game rooms and if they are free to join or not. This operation is possible to execute if user has an identification number from the server. If server receives user ID with *LIST_GAMES_CLIENT* message type in upcoming message, it sends back list of available games to client with message type *LIST_GAMES_SERVER*. Every list position contains game id, information about possibility to join it and game name. To join desired game, client has to provide game ID. Like in the registration process, server returns information if the operation was successful or not, which client handles.

2.4. Creating games

Creating a game is a process in which the new game room is created that can handle two players at once. To create a new game, the server expects a message with selected type *CREATE_GAME_CLIENT* and data that contains user ID and new game name. Please note, that two games having the same name may exist at once, because they will have their own IDs. It's a nice server feature that gives user possibility to create any game name they want. This is an ability that is similar to Discord's username which has a hashtag and ID number next to the selected name. After successful game creation, the server returns a message with type *CREATE_GAME_SERVER* and game identification number.

2.5. Joining games

Joining a game is a process in which user can join an existing game that has free place for new player. To join a game, the server expects a `JOIN_GAME_CLIENT` type message, user ID and game ID. If the game can handle yet another player, the server returns that operation was successful and in next steps the game states are being sent to the client.

2.6. Leaving games

Leaving a game is a process in which a client can leave the game and leave free place for another players. To leave a game, server expect message with type `EXIT_GAME_CLIENT`, user ID and game ID. If this process is successful, message with `operation_success` flag set is returned and the game ends.

2.7. Game states

After each game tick on the server, the game state message is sent to the client with type `SEND_STATE`. Each game state consist of both snake coordinates, food position, player 1's and player 2's points and information if the game is over or not at the moment. At this moment the server expects moves information from the players. If any of that kind of message arrives, the server sends if it was successful or not.

2.8. Game logic handling

After creating and joining the game, the server handles the whole game. The whole game logic is running on the server side. The only job that client has is to properly receive server messages, display the game state and send moves in time.

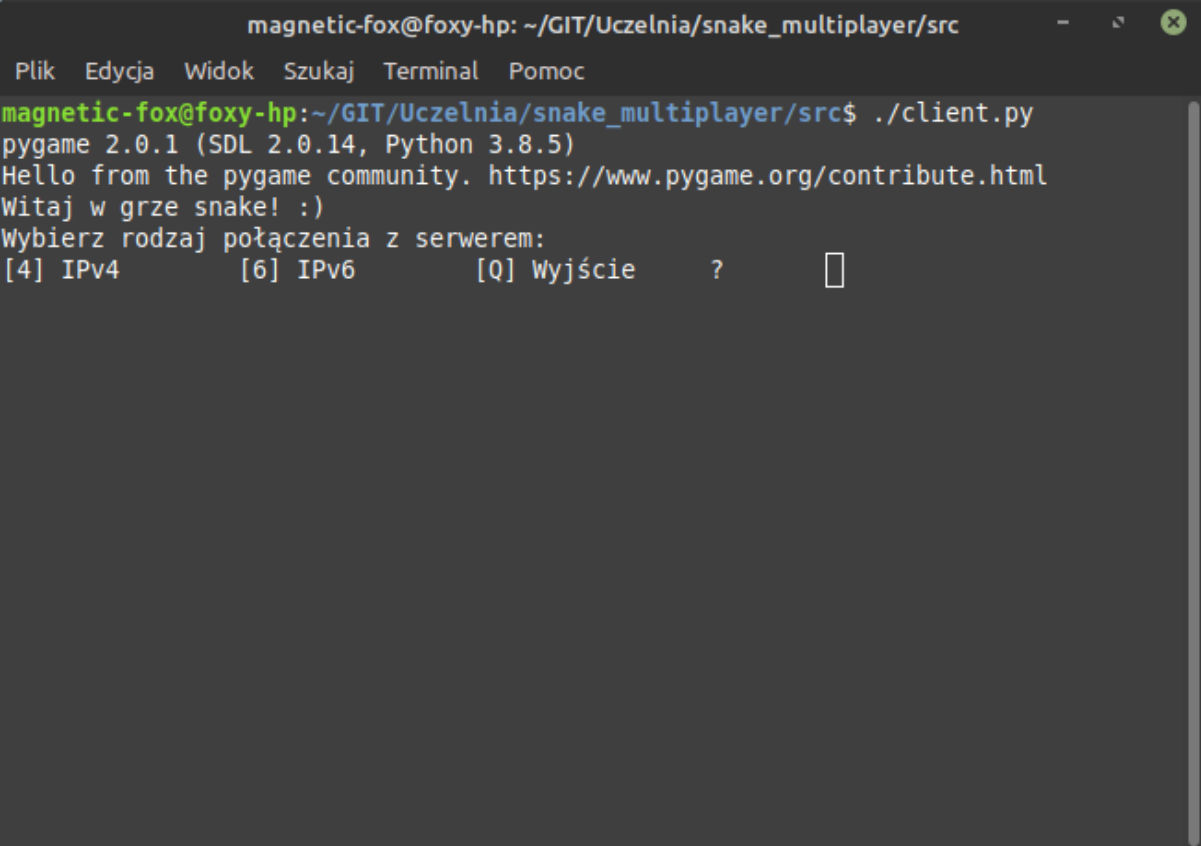
The `process_game` function is responsible for the game logic. First, it is checked whether any of the snakes have eaten the food. If so, the number of points is increased by 10, and a flag is set to increase snake length by 1 element. Then the snakes are moved in the last stored direction. Each snake is actually a list of tuples containing the coordinates of x and y. The first tuple is the head. To move the snake, take its head and move it in the last registered direction and add it to the beginning of the list. If the snake has not eaten the food, the last item in the list is removed. The next step is to check for collisions. First, collisions with walls are checked, then between individual snakes. If the head of one snake, represented by a tuple, is in the list corresponding to the other snake, it means that a

collision has occurred. In the case of collision, an end-of-game flag is set for one of the players - *p1_over/p2_over*. And the message is sent for the last time. The socket is closed.

The *process_game* function is called every time we receive snake movement data from the client, but it is only executed if more than 0.2 seconds have elapsed since the last state of the game.

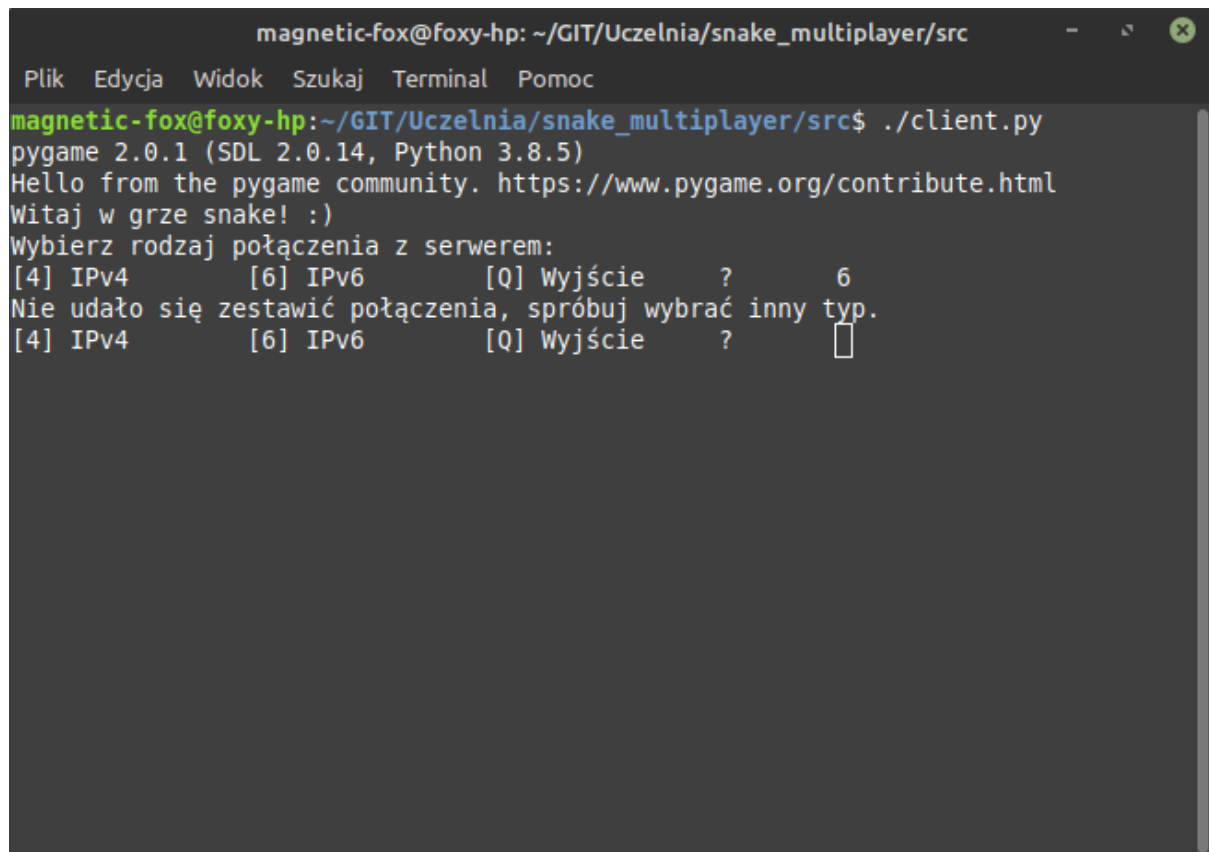
3. Client - usage

The client application is a really simple Python program, which main goal is to receive game states from the server and draw each on the screen on time to provide continuous play, while receiving keyboard events and sending them to the server to change snake's direction. It's a half-GUI program. After start, it prompts the user for the IP protocol version to use for connection. User can select IPv4 or IPv6.

A screenshot of a terminal window titled 'magnetic-fox@foxy-hp: ~/GIT/Uczelnia/snake_multiplayer/src'. The window has a menu bar with 'Plik', 'Edycja', 'Widok', 'Szukaj', 'Terminal', and 'Pomoc'. The terminal output shows the command './client.py' being executed, followed by the pygame version (2.0.1), SDL version (2.0.14), and Python version (3.8.5). It then displays a welcome message in English and Polish, and prompts the user to choose a connection type. The prompt shows '[4] IPv4', '[6] IPv6', and '[Q] Wyjście' with a question mark and a cursor box.

Picture 2: After client application starts.

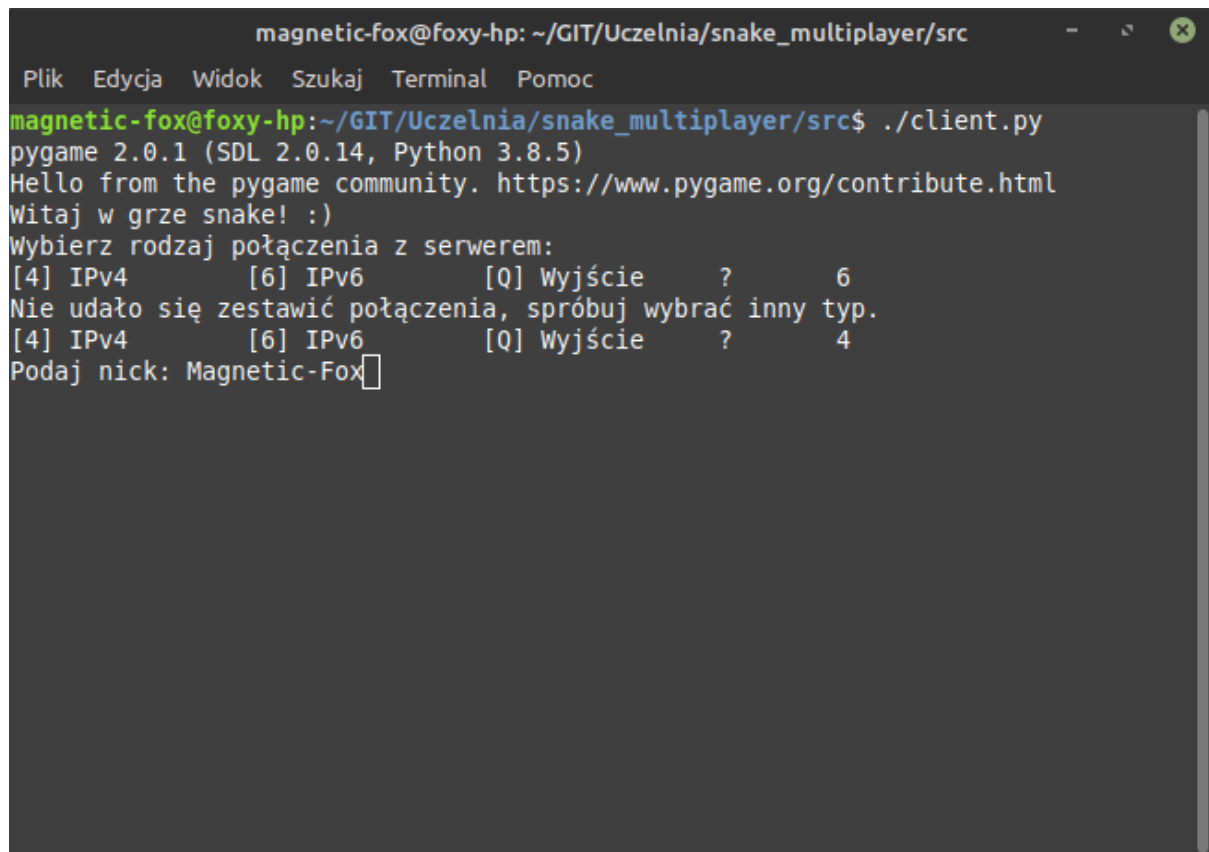
If the chosen IP protocol version is not available on a used operating system, the error message is returned and the user has to choose another one.



```
magnetic-fox@foxy-hp: ~/GIT/Uczelnia/snake_multiplayer/src
Plik  Edycja  Widok  Szukaj  Terminal  Pomoc
magnetic-fox@foxy-hp:~/GIT/Uczelnia/snake_multiplayer/src$ ./client.py
pygame 2.0.1 (SDL 2.0.14, Python 3.8.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
Witaj w grze snake! :)
Wybierz rodzaj połączenia z serwerem:
[4] IPv4      [6] IPv6      [Q] Wyjście    ?      6
Nie udało się zestawić połączenia, spróbuj wybrać inny typ.
[4] IPv4      [6] IPv6      [Q] Wyjście    ?      
```

Picture 3: IPv6 is not available on the user's computer - the message is shown.

After choosing the desired protocol, it asks the user on the console for a nickname and then, after successful registration, gets a list of available games and displays them. Of course, after failed registration, when the selected nickname is already in use, it prompts the user for another one.



```
magnetic-fox@foxy-hp: ~/GIT/Uczelnia/snake_multiplayer/src
Plik Edycja Widok Szukaj Terminal Pomoc
magnetic-fox@foxy-hp:~/GIT/Uczelnia/snake_multiplayer/src$ ./client.py
pygame 2.0.1 (SDL 2.0.14, Python 3.8.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
Witaj w grze snake! :)
Wybierz rodzaj połączenia z serwerem:
[4] IPv4      [6] IPv6      [Q] Wyjście    ?      6
Nie udało się zestawić połączenia, spróbuj wybrać inny typ.
[4] IPv4      [6] IPv6      [Q] Wyjście    ?      4
Podaj nick: Magnetic-Fox
```

Picture 4: Client application prompts for nickname.

After the registration is completed, the program lets users create a new game room or join an existing one if it's possible - the server assumes that a maximum of two players can play at once in the one game room.

```
magnetic-fox@foxy-hp: ~/GIT/Uczelnia/snake_multiplayer/src
Plik Edycja Widok Szukaj Terminal Pomoc
magnetic-fox@foxy-hp:~/GIT/Uczelnia/snake_multiplayer/src$ ./client.py
pygame 2.0.1 (SDL 2.0.14, Python 3.8.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
Witaj w grze snake! :)
Wybierz rodzaj połączenia z serwerem:
[4] IPv4      [6] IPv6      [Q] Wyjście    ?      6
Nie udało się zestawić połączenia, spróbuj wybrać inny typ.
[4] IPv4      [6] IPv6      [Q] Wyjście    ?      4
Podaj nick: Magnetic-Fox
Twoje ID: 10000

ID gry      Wolna?      Nazwa gry
-----
N - nowa gra  D - dołącz  Q - wyjdź    ? n
Nazwa gry: Fajne to jest :) 
```

Picture 5: Creating a new game room.

```
magnetic-fox@foxy-hp: ~/GIT/Uczelnia/snake_multiplayer/src
Plik Edycja Widok Szukaj Terminal Pomoc
magnetic-fox@foxy-hp:~/GIT/Uczelnia/snake_multiplayer/src$ ./client.py
pygame 2.0.1 (SDL 2.0.14, Python 3.8.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
Witaj w grze snake! :)
Wybierz rodzaj połączenia z serwerem:
[4] IPv4      [6] IPv6      [Q] Wyjście    ?      6
Nie udało się zestawić połączenia, spróbuj wybrać inny typ.
[4] IPv4      [6] IPv6      [Q] Wyjście    ?      4
Podaj nick: Magnetic-Fox
Twoje ID: 10000

ID gry      Wolna?      Nazwa gry
-----
N - nowa gra  D - dołącz  Q - wyjdź    ? n
Nazwa gry: Fajne to jest :)
ID gry      Wolna?      Nazwa gry
-----
10000      Tak      Fajne to jest :)
N - nowa gra  D - dołącz  Q - wyjdź    ? 
```

Picture 6: After creating a new game room it is listed to join.

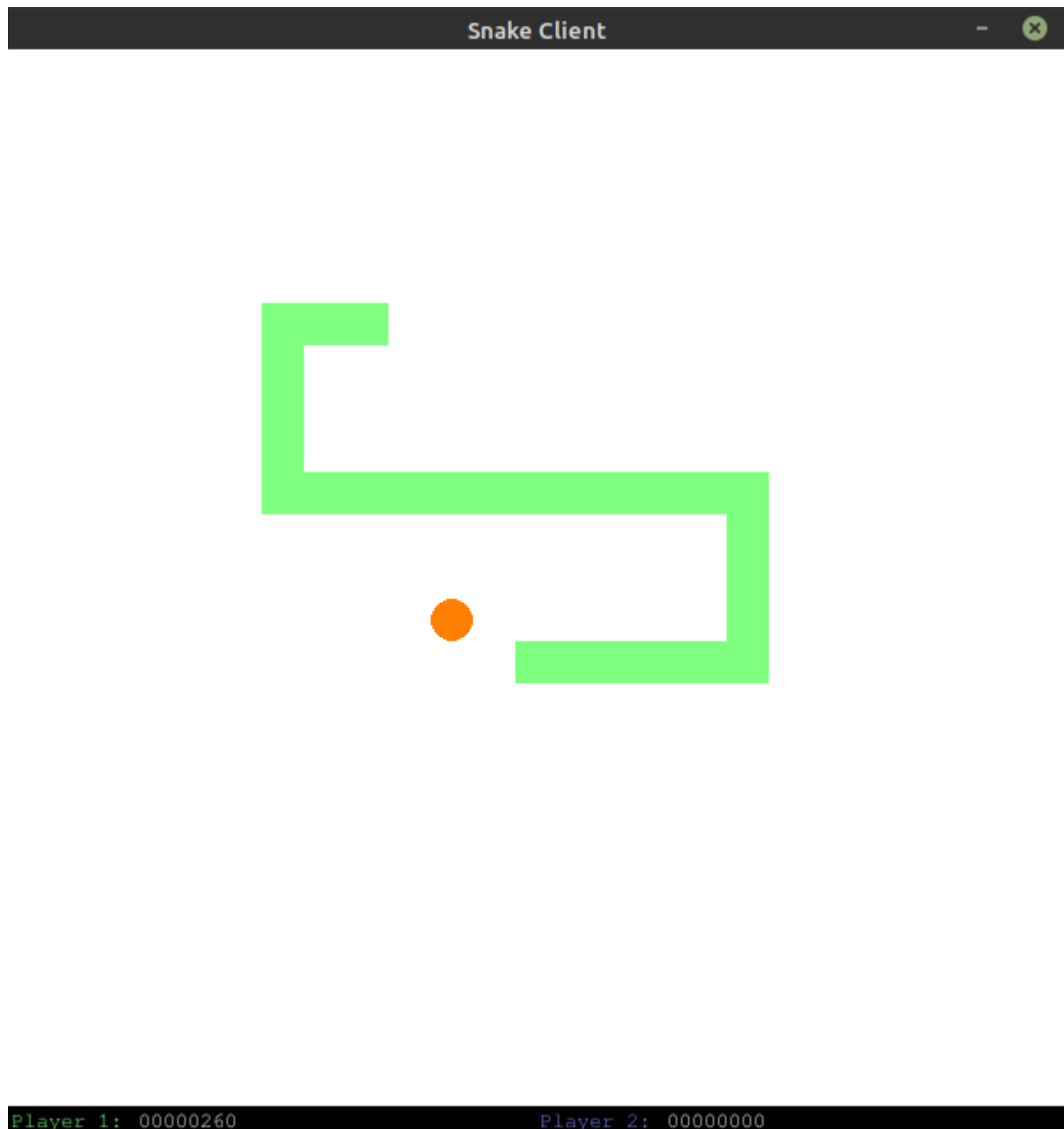
```
magnetic-fox@foxy-hp: ~/GIT/Uczelnia/snake_multiplayer/src
Plik Edycja Widok Szukaj Terminal Pomoc
magnetic-fox@foxy-hp:~/GIT/Uczelnia/snake_multiplayer/src$ ./client.py
pygame 2.0.1 (SDL 2.0.14, Python 3.8.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
Witaj w grze snake! :)
Wybierz rodzaj połączenia z serwerem:
[4] IPv4      [6] IPv6      [Q] Wyjście    ?      6
Nie udało się zestawić połączenia, spróbuj wybrać inny typ.
[4] IPv4      [6] IPv6      [Q] Wyjście    ?      4
Podaj nick: Magnetic-Fox
Twoje ID: 10000

ID gry      Wolna?      Nazwa gry
-----
N - nowa gra  D - dołącz  Q - wyjdź    ? n
Nazwa gry: Fajne to jest :)
ID gry      Wolna?      Nazwa gry
-----
10000      Tak      Fajne to jest :)
-----
N - nowa gra  D - dołącz  Q - wyjdź    ? d
ID gry: 10000
Dołączono do gry # 10000
█
```

Picture 7: Now joining the created game is possible - messages are shown.

After successfully joining a new or existing game, the PyGame window appears which shows both players (first with light green color and second with light blue color) and food (orange ball) on the screen.

Arrow keys are used to manipulate snakes.



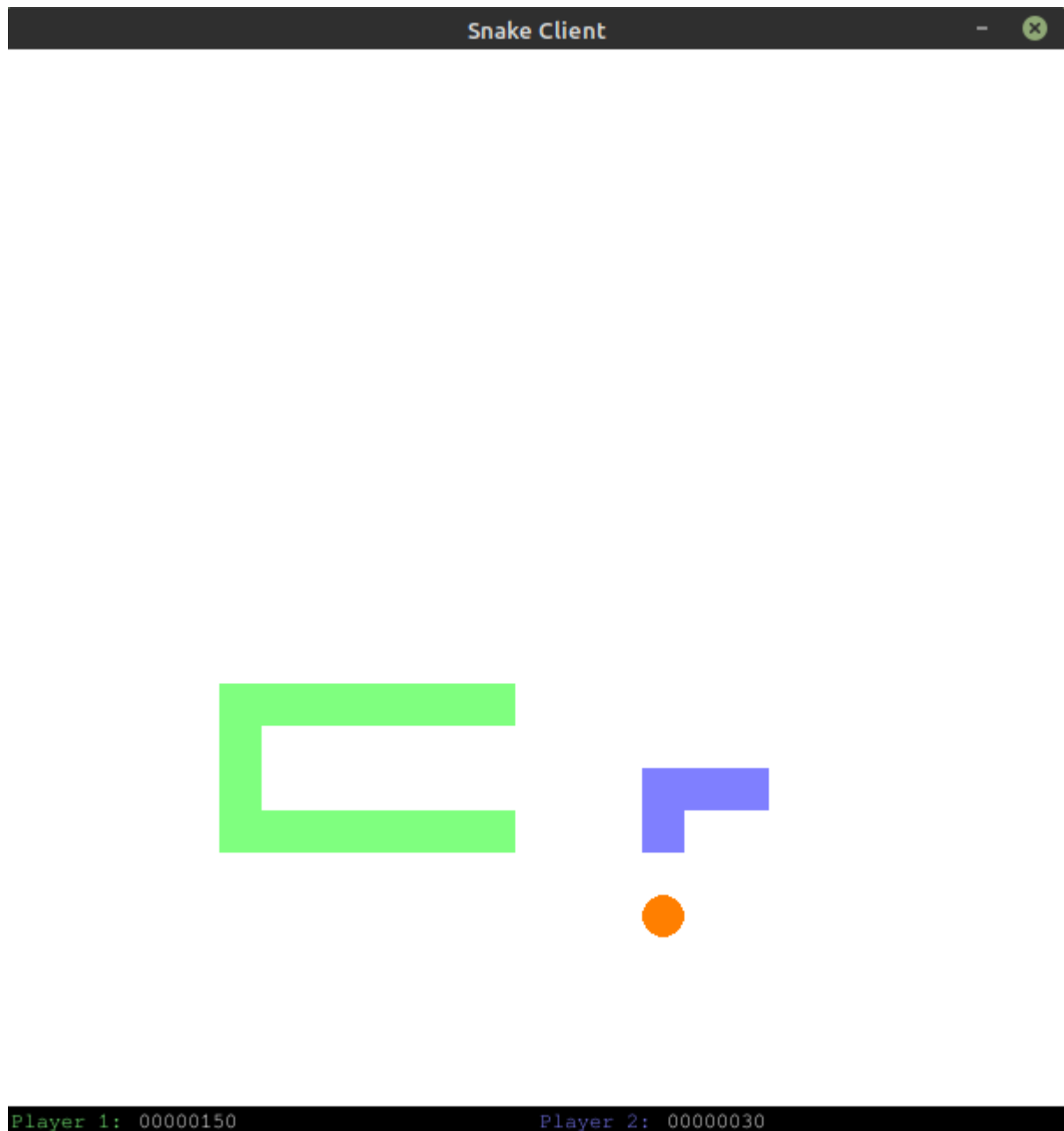
Picture 8: *One player gameplay.*

For a player, the game ends when one snake goes on the another or hits the wall. However, the whole game does not end when there is another player in the room.

```
magnetic-fox@foxy-hp: ~/GIT/Uczelnia/snake_multiplayer/src
Plik Edycja Widok Szukaj Terminal Pomoc
magnetic-fox@foxy-hp:~/GIT/Uczelnia/snake_multiplayer/src$ ./client.py
pygame 2.0.1 (SDL 2.0.14, Python 3.8.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
Witaj w grze snake! :)
Wybierz rodzaj połączenia z serwerem:
[4] IPv4      [6] IPv6      [Q] Wyjście      ?      4
Podaj nick: Magnetic-Fox
Twoje ID: 10000

ID gry      Wolna?      Nazwa gry
-----
N - nowa gra  D - dołącz  Q - wyjdź      ? n
Nazwa gry: Pierwsza
ID gry      Wolna?      Nazwa gry
-----
10000      Tak      Pierwsza
-----
N - nowa gra  D - dołącz  Q - wyjdź      ? d
ID gry: 10000
Dołączono do gry # 10000
Przegrana! Punkty: 420
magnetic-fox@foxy-hp:~/GIT/Uczelnia/snake_multiplayer/src$
```

Picture 9: After game over, the collected points are displayed in the console.



Picture 10: *Multiplayer gameplay example.*

Use of our solution is easy. It works pretty smoothly and reliably. It provides long gameplay without errors.

4. Client - communication, types of messages

The client also communicates using *Venom* protocol. It's main job is to properly receive server messages, draw game states and send moves in time. Of course the client application has to properly communicate with the server to register user, list available games, create a new one and join for play with other players online. This chapter briefly explains each type of message that client may send to the server.

4.1. Every step is checked

Every message sent to the server is composed exactly in the way the server expects. After sending any message, the client expects a response from the server which is checked for correctness. If a returned message doesn't meet expected message type, the client finishes its work - something went really wrong and probably at the server side.

4.2. User register request

At the beginning, the client application prompts user for a nickname. After the user provide desired nickname, the client creates a message (*LOGIN_CLIENT* type) to send to the server and tries to register the user by sending it. If the registration was successful, the client goes to the next step, otherwise prompts the user to select another username, because desired is reserved.

4.3. Games listing request

After successful user registration, the client tries to list game rooms and waits for the user's reaction. After printing game rooms, the client awaits for the user's reaction - for game creation, joining or exit from the program. User can use 'N' key to create a new game, 'D' key to join an existing game or 'Q' to exit a client application. Every other key causes the client to retrieve and print the game list again. Of course, every game list retrieval causes sending *LIST_GAMES_CLIENT* messages.

4.4. Games creating request

If no game exists or none can be joined, user can create a new one. The user is prompted for a new game name. Next, the message for the server is created using the *CREATE_GAME_CLIENT* message type. If the game creation was successful, the client returns to the list of available games. Now the user can join the created game.

4.5. Games joining request

User can join any free game. To join, user is prompted for the desired game identification number. Next, the message for the server is created using the *JOIN_GAME_CLIENT* message type. If joining was possible, the game starts and the client creates a game window using the PyGame module. The game starts after joining.

4.6. Games leaving request

While playing the game, user can leave the joined game by hitting the escape key. Because the game is multiplayer it doesn't ask for confirmation. There is no time for it, because the game is really quick. After hitting the escape key, the client creates an exit message using the *EXIT_GAME_CLIENT* type of message. After sending that message, the client awaits for confirmation from the server.

4.7. Sending moves requests

While in the game, the client receives game states from the server sent as a *SEND_STATE* type message. Of course, the client can send moves using the *SEND_MOVE* type message. Every move is sent as a single character determining the desired direction of the controlled snake. The 'l' character is used to determine the left direction of the snake, 'r' is used for the right direction, 'u' for the up direction and 'd' for the down direction.

4.8. Interpreting game states

Every time the server sends game states, the client's job is to properly interpret it. While the main job is to display snakes and points, there are some things to handle at the time. Every game state message (*SEND_STATE* type) contains much more data than only snakes' and

food's coordinates. It also contains points for player 1 and player 2 and states for both players' games telling if player 1 or 2 has game over or not. Every time the new game state message arrives, the client uses the previous received message to clean the screen to prepare it for drawing a new one. This process avoids blurring the game screen. Of course the players' points are also updated every message. The client application updates the whole game screen every time it is able to properly receive the game state message. This avoids improperly drawing game frames - these may not be drawn in fact causing simple lags that have not much impact on the gameplay. In fact, that kind of problem hardly ever shows up.

5. Encryption

Encryption was a key feature to add in our project. Nowadays, non-encrypted data isn't safe even in so little projects like our snake game. We've managed to add encryption ability to our solution. In our project, communication is encrypted with the *TLS* protocol. At the beginning of the communication between the server and the client, a handshake is performed and the identity and compatibility of the keys are verified. It is a really important part as it checks the validity of keys used for exchanging encrypted data and avoids (basically) using a false key. The keys were generated using the *OpenSSL* library. The functions for generating keys are in the file *generate_keys.py*.

6. Project highlights

Our solution has much more advantages in comparison to others. While other implementations of snake game are basically non-multiplayer, ours is and uses internet connection to provide non-one-device multiplayer gameplay. That assumption forced us to properly implement continuous data handling in multiple server threads and make them also wait for new connections and properly interpret arriving data in time. This caused some problems with synchronization at the first time. The problems we had with implementation were listed in the seventh chapter of this documentation. Another great advantage of our solution is ease of protocol and applications expandability. In fact, every new idea can be easily added to our solution in a couple of minutes. The *Venom* protocol used to exchange messages between client and server is the greatest advantage of this project. Its bit-like nature compresses exchanged messages to the minimum amount of data needed to send which makes connection much more lightweight than others adding ease of use in software. Another great advantage of our solution is encrypted connection between server and client, which provides privacy in gameplay. And one more advantage is the use of cloud technology to run the server for easily testing gameplay and provide live presentation of working software to the public. That way it's ready for production use.

In brief, our solution is something that has never been before.

7. System requirements

Our solution has really simple requirements to run. In fact, the only thing needed to run the server is a working computer with internet connection and Python 3.8 installed. The client is also lightweight, but needs a PyGame module installed to create game window and handle pressed keys.

In brief, our solution needs:

- working computer system supporting networking and Python 3.8
- (additional for client only) PyGame module supported and installed on selected computer system

8. Problems we had, compromises

During development we've had some problems with making all components work correctly. At the first time, we decided to use User Datagram Protocol (UDP) for data transmission which worked fine, but unfortunately not without complications. In this version it was really difficult to add encryption. We didn't want to give it up, so we've managed to convert our working solution to Transmission Control Protocol (TCP) to provide a more stable connection between server and client and ease adding encryption. Before we did it, we had a lot of problems with solution stability. Sometimes our server returned `KeyErrors` while joining games or the client wasn't able to properly receive new messages and returned `NoneType`-like errors. In the early version of server and client implementations we had problems with synchronizing games which usually "lagged" and reacted slowly on keyboard events which made the game unplayable. However, after a lot of patches we've finally made it work. After the game started working fine, another troubles arrived. We've had problems with rejoining existing games or registering again after closing client sessions. However, those problems were patched and the server is now stable and works. We've also managed to add encryption to the communication which now works pretty good. The last part we've had problems with was adding IPv6 support. However, it wasn't so hard to add, because the IPv6 socket accepts IPv4 connections as well as IPv6's ones. So, on the server side it was easy - it was just two modifications in comparison with the IPv4 version. Only the socket type had to be changed and bind address from '0.0.0.0' to '::'. More code to add was on the client side. We have to add a choice possibility - if the user wishes to use IPv4 or IPv6 and check if using IPv6 is possible. It needed adding some lines of settings. One part is for establishing an IPv4 connection and the second part is for establishing an IPv6 one. Not many lines, but they had to be in client code instead of the server's one. Fortunately, we've managed to correct the whole code to work properly and tested IPv6 functionality, which now also works fine.

9. Our feelings about project - what we are proud of

Our solution consists of components that were created with love to programming, snake game and the will to play this nice game with friends via the internet connection. The component we are the most proud of is the great *Venom* protocol which is probably the greatest advantage of this version of snake game. Its ease of use and great compression possibilities makes data lightweight while both server and client can exchange every data needed to provide gameplay and game rooms management. The next component we are really proud of is the multithreading server that can handle a lot of games at the same time with encrypting the whole connection and is just doing it pretty well - it always works in time without any errors. I (Bartłomiej Węgrzyn), as the main creator of this documentation, am really proud of my little client code. Maybe it's not perfectly nor cleanly prepared Python code, but I'm really proud of it, as it just works and is able to handle arriving game state messages in time to provide continuous gameplay and reacts on keyboard events sending them back to the server to manipulate controlled snake. In fact, I'm really proud of our team that made all of this possible - it was an unforgettable feeling when all components finally worked together and we were able to play together. I think that our teamwork was also a great experience for all of us. It was some really great weeks of teamwork that made us all learn a lot of new things especially from networking and Python language programming. It was a really nice time.

10. Project authors

- Norbert Ozga** - *Venom* protocol design and code, fixes in server and client code, help with adapting protocol
- Antoni Pietryga** - Main server code and fixes in client code, development
- Bartłomiej Węgrzyn** - Main client code and simple fixes in server code, this documentation

The project was developed especially for Programming Networking Applications studies in May/June 2021.