--Python version 3.10.5

Generating the Genesis block:

- In the Assignment3/ directory, open a terminal and run the program ‘genesis_block.py’ to generate the genesis block, 0.json.

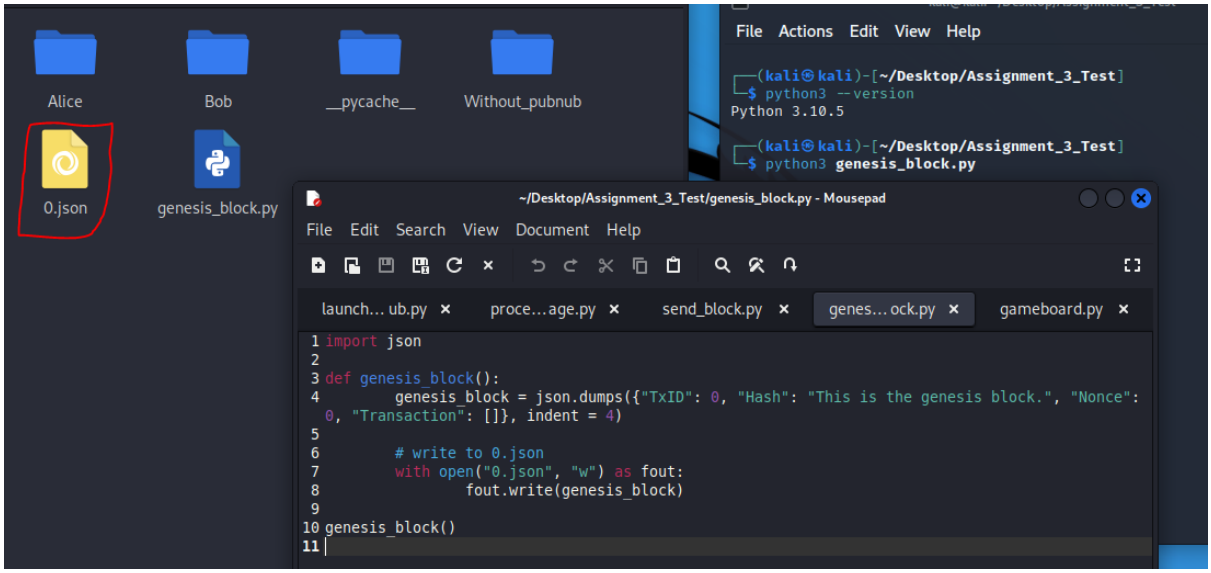

Figure 1: Generating the Genesis Block

- This is the genesis block in which both players will build upon. I assume that both players will be able to get a hold of this file somehow in the real world, but in this case, they will be able to grab this file and store it in their local storage through the relative path.

```
# get the genesis block
with open("../0.json", "r") as fin:
        genesis_block = json.loads(fin.read())

with open("0.json", "w") as fout:
        fout.write(json.dumps(genesis_block, indent = 4))
```

Figure 2: Storing 0.json in the player's local storage

Connecting both players to the pubnub channel:

- PLEASE NOTE: Because of the way I implemented my code, and since Alice is the player who will always make the first move, it requires that Alice joins the pubnub channel before Bob.

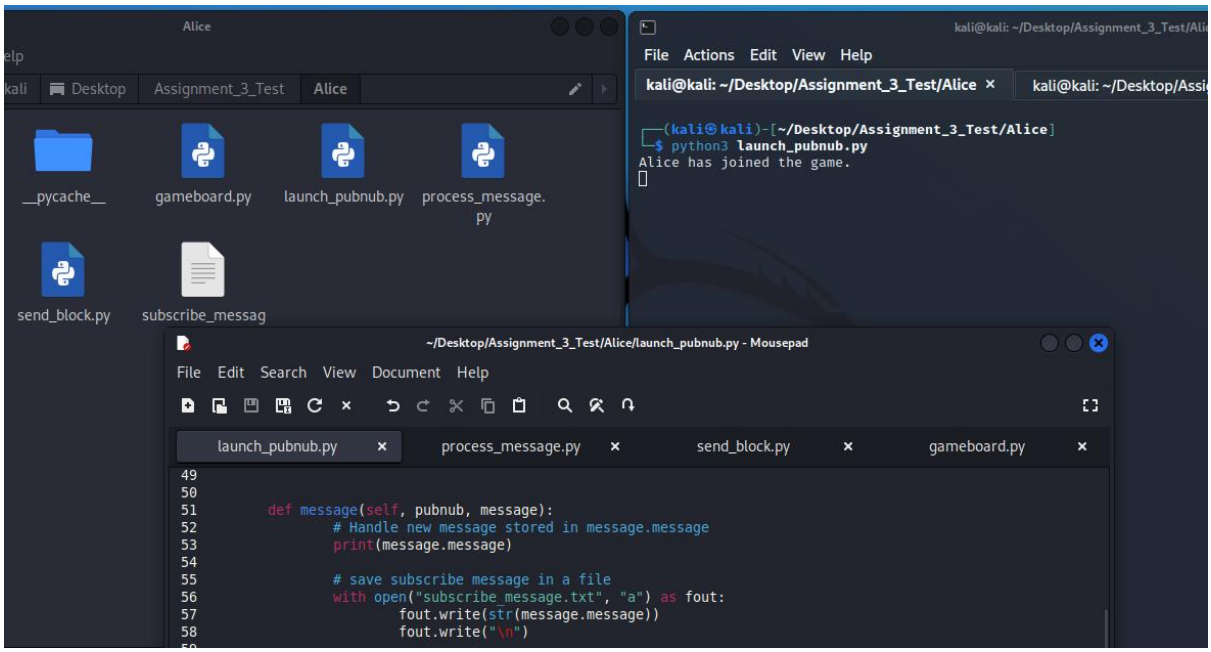

Figure 3: Alice connects to Pubnub channel

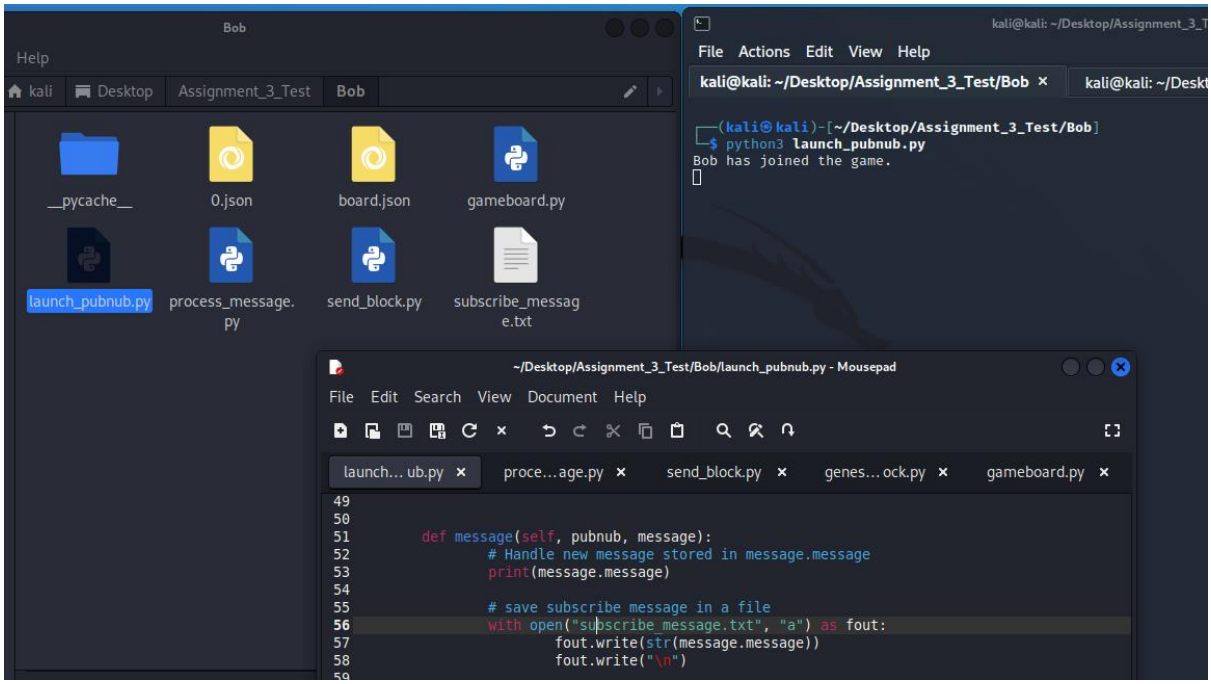

Figure 4: Bob connects to Pubnub channel

- When the players connect to the pubnub channel, every message they receive on the channel will be written to a text file “subscribe_message.txt”. It can serve as a message log, or a history of the messages(blocks) exchanged between both players.

After connecting to the channel:

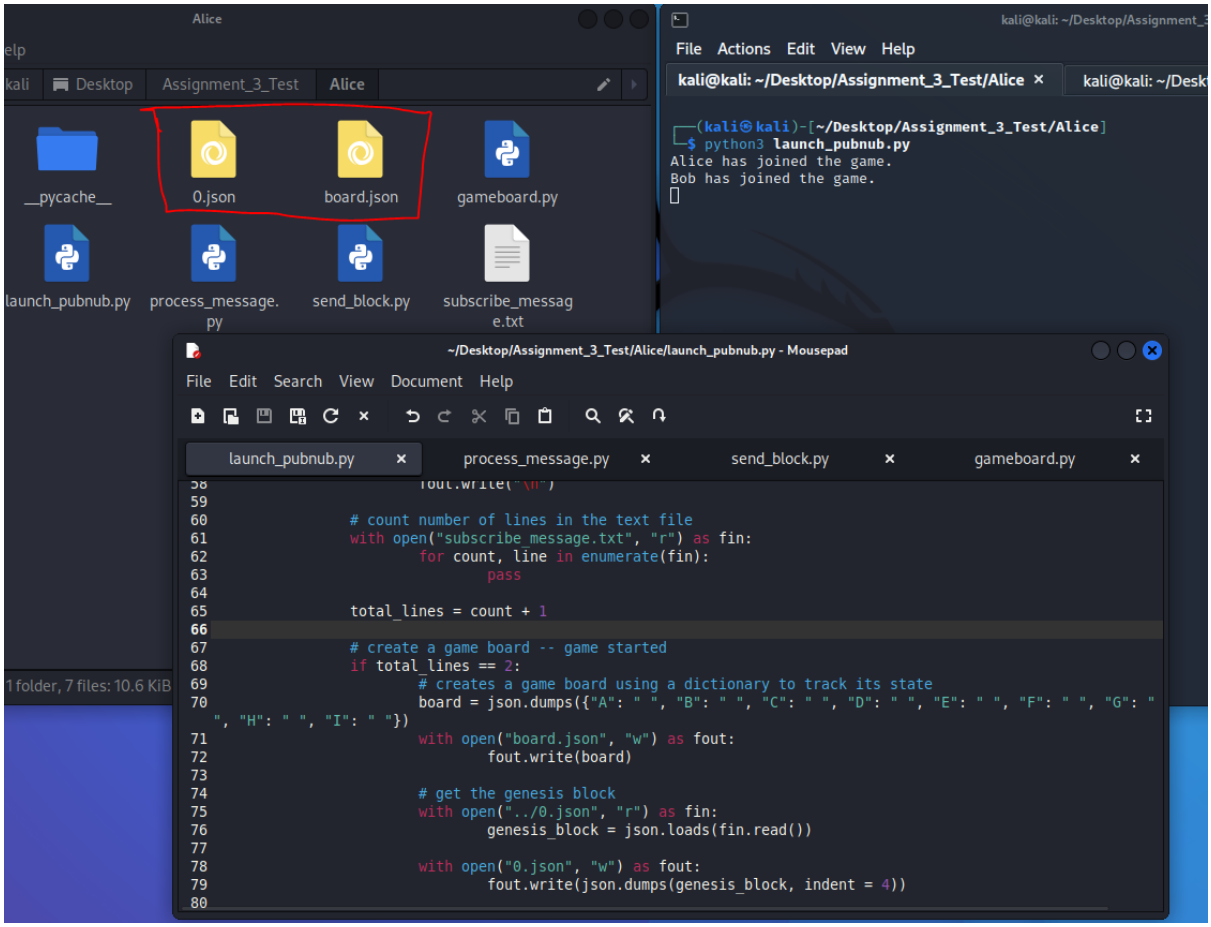

Figure 5: Necessary files

- Once both player joins the channel, it is assumed that the game is started.

- With the way I designed the program, I created two directories named “Alice” and “Bob”, which represents each of the players’ local storage, which means that ‘0.json’ can be found in the parent directory.

- I created a ‘board.json’ file to represent the state of the tic-tac-toe game board. It contains a dictionary where its keys are the positions of the game board, and the values represent the player which occupies the position. By default, every value in the board is a space character(“ “). This file will be updated after every move executed by the players.

Begin the game:

- Run the send_block.py program in the Assignment3/Alice directory to generate a random move and send it over to Bob through the pubnub channel.

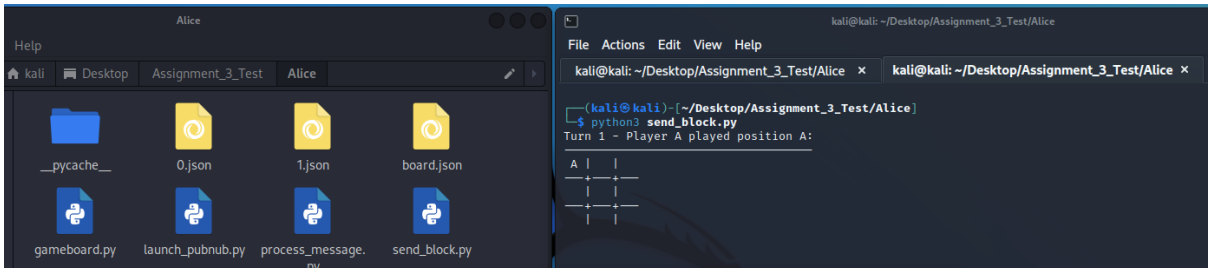

Figure 6: Alice starting the game

- The messages sent to the pubnub channel can be viewed from each of the player's main terminal.

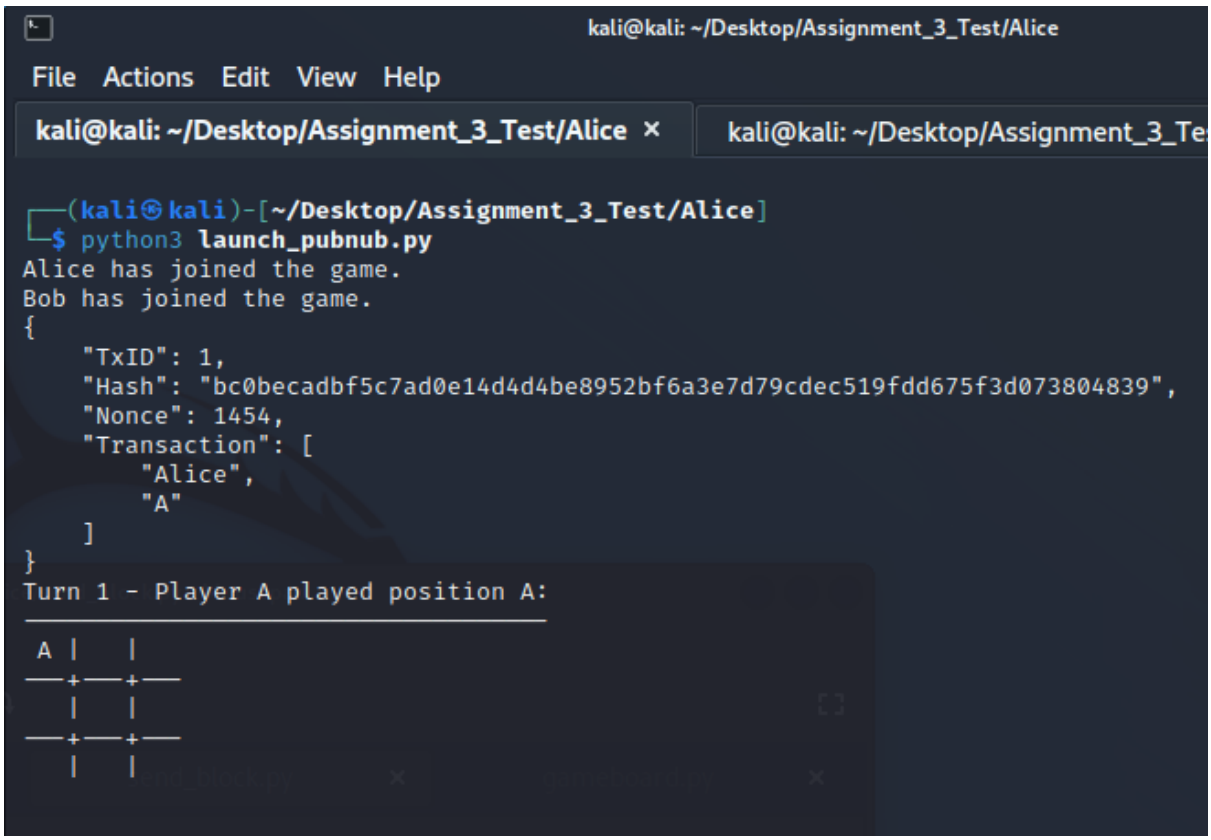

Figure 7: Messages received in the Pubnub channel

- Note that Bob can make a move in a similar fashion, just that the program must be executed in the Assignment3/Bob directory.

Writing the block as a json file:

- Once Alice makes the first move, the program will count the total number of lines in "subscribe_message.txt". The first two lines in Alice's "subscribe_message.txt" is "Alice has joined the game" and "Bob has joined the game". Anything afterwards represents a block. Because of how the block contents are stored, each block takes up nine lines in the file.

- To count the number of blocks generated, I subtract the first two lines from the total number of lines in the file and divide the result by nine.

- I do not want to process the json files that already exist, so I skip them and proceed on to write the newest block and store it in the player's local storage.

```
# count number of lines in the text file
with open("subscribe_message.txt", "r") as fin:
        for count, line in enumerate(fin):
                pass

total_lines = count + 1
```

Figure 8: Total lines in subscribe_message.txt

```
# if there are moves played (blocks)
if total_lines > 2:
        number_of_blocks_written = int((total_lines - 2) / 9)

        # stores the new block in a json file
        with open("subscribe_message.txt", "r") as fin:
                next(fin) # skips the first line of message - "Alice has joined the game"
                next(fin) # skips the second line of message - "Bob has joined the game"

                # skip irrelevant lines
                for i in range((number_of_blocks_written - 1) * 9):
                        next(fin)

                # store new block
                with open(str(number_of_blocks_written) + ".json", "w") as fout:
                        temp = json.loads(fin.read()) # store content in json format
                        fout.write(json.dumps(temp, indent = 4)) # write in json format
```

Figure 9: Write the new json block

Sending a block:

[Some basic checks]

- Some checks will be done before a player is able to make a move and send a block over to the other player. For Alice's case, she will not be able to send a block if 1. Bob has not joined the channel; 2. It is not Alice's turn to make a move.

- If Alice is allowed to make a move, another check will be conducted to see whether she is making the first move of the game. If she is, she will have to grab the genesis block from the parent directory and write (download) into her local storage.

```
def write_block():
        with open("subscribe_message.txt", "r") as fin:
                for count, line in enumerate(fin):
                        pass
        total_lines = count + 1

        if total_lines == 1:
                exit("Bob has not joined yet!")

        # no blocks yet - except genesis block
        if total_lines - 2 == 0:
                with open("../0.json", "r") as genesis_block:
                        # hash of genesis block
                        hash_of_previous_block = hashlib.sha256(genesis_block.read().encode()).hexdigest()

                        genesis_block.seek(0) # move cursor back to start of file

                        # TxID of genesis block
                        txid = json.loads(genesis_block.read())["TxID"] + 1
```

Figure 10: Storing 0.json in local storage

```
# check turn order
if txid % 2 == 0:
        print(f"It is not your turn yet {pnconfig.user_id}")
        return
```

Figure 11: Checking turn order

[Generate a new block – Make a move]

- Next, the player will open the latest block sent and generate its hash value as well as the TxID.

```
else:
        number_of_blocks = int((total_lines - 2) / 9)

        # open latest block
        with open(str(number_of_blocks) + ".json", "r") as fin:
                # hash of previous block
                hash_of_previous_block = hashlib.sha256(fin.read().encode()).hexdigest()

                fin.seek(0) # move cursor back to start of file

                # TxID of genesis block
                txid = json.loads(fin.read())["TxID"] + 1
```

Figure 12: Get hash value and TxID

- The function 'make_a_move' will randomly generate an available move for the player.

```
        # generate new block
        nonce = 0
        flag = True
        move = make_a_move(pnconfig.user_id, txid)

        while(flag):
                block = json.dumps({"TxID": txid, "Hash": hash_of_previous_block, "Nonce": nonce,
"Transaction": [pnconfig.user_id, move]}, indent = 4)
                hash_block = hashlib.sha256(block.encode()).hexdigest()

                # require the hash to be < 2**244/2**256:
                # (256 - 244) / 4 == 3
                if int(hash_block, 16) <
int("000fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff", 16):
                        flag = False
                        send_message(block)

                nonce += 1
```

Figure 13: Generate a new legitimate block

```
# player makes a move
def make_a_move(player, txid):
        player = player[0]

        position_is_not_filled = True #  make a move on an empty space

        with open("board.json", "r") as fin:
                board = json.loads(fin.read())

        # check whether space is empty
        while(position_is_not_filled):
                # generate random letter for move to be played
                move = chr(random.randint(ord('A'), ord('I')))

                if board[move] == " ": # position to be played is empty
                        print(f"Turn {txid} - Player {player} played position {move}:")
                        print("-----------------------------------")
                        board[move] = player.upper()

                        position_is_not_filled = False # exit while loop

                        print_board(board) # print state of board after move

                        return move
                else: # if space is not empty, pick another spot randomly again
                        continue
```

Figure 14: Generate a random available move

Process the message received:

[Verifying the new blocks]

- After receiving the latest block, the player needs to verify that the block is valid. The verification process is simple – compare the hash value of the previously written block to the hash value written in the newest block. If the hash values are the same, the block is valid. Else, the block in invalid.

```
# verify new block
valid = verify_new_block(number_of_blocks_written)
```

Figure 15: Execute the verification function

```
# verify new block
def verify_new_block(block_number):
        with open(str(block_number - 1) + ".json", "r") as fin:
                hash_of_previous_block = hashlib.sha256(fin.read().encode()).hexdigest()

        with open(str(block_number) + ".json", "r") as fin:
                hash_of_new_block = json.loads(fin.read())["Hash"]

        if hash_of_previous_block == hash_of_new_block:
                return True
        else:
                return False
```

Figure 16: The verification function

[Updating the game board]

- Once the latest board is verified, the game board will be updated.

- The function below will return the game board as a dictionary, the player who created the latest block, and the move the player made. This information will be used for further processing.

```
# get the move played by the opponent
board, player, move = update_board(str(number_of_blocks_written) + ".json")
```

Figure 17: Update board.json

```
# read a file and return player and move played
def read_file(filename):
        with open(filename, "r") as fin:
                transaction = json.loads(fin.read())["Transaction"]
                player = transaction[0]
                move = transaction[1]

        return player, move

# update the state of the board
def update_board(filename):
        player, move = read_file(filename)
        with open("board.json", "r") as fin:
                board = json.loads(fin.read())

        with open("board.json", "w") as fout:
                board[move] = player[0]
                fout.write(json.dumps(board))

        with open("board.json", "r") as fin:
                return json.loads(fin.read()), player, move
```

Figure 18: Function to update board.json

Concluding the game:

- If the number of blocks generated is >= 5, it means that there is a possibility that a winner can be found. I created a function, 'winning_combination', which takes in a list of moves the latest player has played. This list will then be processed against the list of moves, win_con, that satisfies the winning condition. If the player has played the moves in win_con, it means that the player has won.

- Before calling the 'winning_combination' function, I created a separate function 'win_or_draw'. The turn number will determine which player will be checked for a chance to win. The dictionary item will be processed to generate a list of moves each player has played. This list will then be passed to the 'winning_combination' function to determine whether a winner can be found.

```
# verify win_con
if number_of_blocks_written >= 5:
        winner_found = win_or_draw(number_of_blocks_written, board)

        if winner_found:
                print(f"{player} has won the match!")
                pubnub.unsubscribe().channels("Channel-mzonikory").execute()

        if not winner_found and number_of_blocks_written == 9: # Draw
                print("Game board is full. No more playable moves!\nResult: Draw")
                pubnub.unsubscribe().channels("Channel-mzonikory").execute()
```

Figure 19: Check for conclusion of game

```
# check for winning condition
def win_or_draw(turn_number, board):
        # AFTER player makes a move, 2 conditions have to be checked
        # FIRST CONDITION --> Winner is found
        # list of moves each player has made
        alice_played = [i for i in board if board[i] == "A"]
        bob_played = [i for i in board if board[i] == "B"]
        # check for Alice winning condition after her move
        if turn_number % 2 != 0:
                win = winning_combination(alice_played)
                if win:
                        return True
        else: # check for Bob winning condition after his move
                win = winning_combination(bob_played)
                if win:
                        return True
        # SECOND CONDITION --> Board is full
        # if no winner is found after board has been filled, end game in draw
        if turn_number == 9:
                return False

        return False
```

Figure 20: Function to check for a possible winner

```
# check for winning combinations
def winning_combination(player_played):
        # winning combinations
        win_con = [["A", "B", "C"], ["D", "E", "F"], ["G", "H", "I"], ["A", "D", "G"], ["B", "E", "H"], ["C",
"F", "I"], ["A", "E", "I"], ["C", "E", "G"]]

        for i in win_con:
                # check whether player has played the winning move
                check = all(item in player_played for item in i)

                # if winning combination is found, return True
                if check:
                        return True

        return False
```

Figure 21: Function with the winning combinations

Some Screenshots of the game:

- To illustrate the messages sent through the channel, the program prints out the contents of the block as well as the state of the game board after every message the channel receives.

- Below is a screenshot of Alice winning the game.

Figure 22: Alice wins a game

- Below is a screenshot of a game ending in a draw.

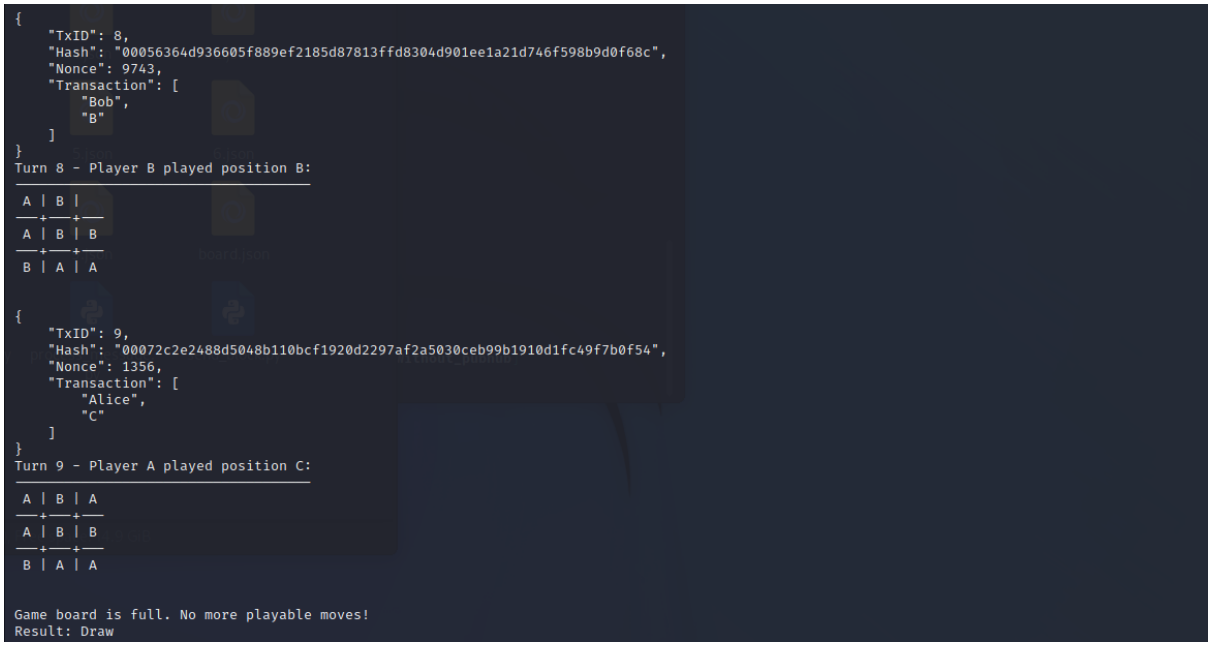

Figure 23: Draw