Kacper Kurz, Sudoku, gra dla dwóch osób

Mechanizm komunikacji międzyprocesowej lub synchronizacji wątków:

W projekcie został wybrany mechanizm socketów datagramowych. Umożliwia on grę na wielu komputerach w różnych miejscach i prostą komunikację między nimi.

Opis użytkowania programu z uwzględnieniem sytuacji błędnych obsługiwanych przez program:

Żeby wystartować grę potrzebny jest serwer i dwóch klientów. Przy uruchamianiu serwera podajemy ip i port jako argument. Podobnie jest z klientami tylko zamiast ip i portu hosta podajemy ip i port serwera. Klient czeka aż dołączy przeciwnik po czym zostaje mu przyznany identyfikator. Gracz 1 startuje a po nim gracz 2.
Gdy jest tura gracza to musi on podać pozycje wiersza i kolumny i liczbę którą chce zapisać. Program sprawdzi czy użytkownik podał dane w poprawnym formacie, czy podane liczby mają sens (np. czy nie są ujemne, czy nie są większe niż 9), jeśli użytkownik podał złe dane to program zapyta go ponownie o ich wprowadzenie. Jeśli dane były wprowadzone poprawnie to gra sprawdza ruch i ocenia go.
Gracz może dodatkowo wyjść z gry wpisując "quit". Tury graczy toczą się aż do wypełnienia sudoku, po czym wyświetla się wynik (wygrałeś, przegrałeś, remis). Klienci kończą pracę a serwer zaczyna nową grę.

Kod źródłowy

Klient:

```
import random
import math
import socket
import sys

IP = sys.argv[1]
PORT = int(sys.argv[2])
BUFF_SIZE = 168


class Sudoku:
    def __init__(self, N, K):
        self.N = N
```

```
        self.K = K

        # Compute square root of N
        SRNd = math.sqrt(N)
        self.SRN = int(SRNd)
        self.mat = [[0 for _ in range(N)] for _ in range(N)]

    def fillValues(self):
        # Fill the diagonal of SRN x SRN matrices
        self.fillDiagonal()

        # Fill remaining blocks
        self.fillRemaining(0, self.SRN)

        # Remove Randomly K digits to make game
        self.removeKDigits()

        for i in range(self.N):
            for j in range(self.N):
                if self.mat[i][j] == 0:
                    self.mat[i][j] = ' '

    def fillDiagonal(self):
        for i in range(0, self.N, self.SRN):
            self.fillBox(i, i)

    def unUsedInBox(self, rowStart, colStart, num):
        for i in range(self.SRN):
            for j in range(self.SRN):
                if self.mat[rowStart + i][colStart + j] == num:
                    return False
        return True

    def fillBox(self, row, col):
        num = 0
        for i in range(self.SRN):
            for j in range(self.SRN):
                while True:
                    num = self.randomGenerator(self.N)
                    if self.unUsedInBox(row, col, num):
                        break
                self.mat[row + i][col + j] = num
```

```
    def randomGenerator(self, num):
        return math.floor(random.random() * num + 1)

    def checkIfSafe(self, i, j, num):
        return (self.unUsedInRow(i, num) and self.unUsedInCol(j, num) and
self.unUsedInBox(i - i % self.SRN,

j - j % self.SRN, num))

    def unUsedInRow(self, i, num):
        for j in range(self.N):
            if self.mat[i][j] == num:
                return False
        return True

    def unUsedInCol(self, j, num):
        for i in range(self.N):
            if self.mat[i][j] == num:
                return False
        return True

    def fillRemaining(self, i, j):
        # Check if we have reached the end of the matrix
        if i == self.N - 1 and j == self.N:
            return True

        # Move to the next row if we have reached the end of the current row
        if j == self.N:
            i += 1
            j = 0

        # Skip cells that are already filled
        if self.mat[i][j] != 0:
            return self.fillRemaining(i, j + 1)

        # Try filling the current cell with a valid value
        for num in range(1, self.N + 1):
            if self.checkIfSafe(i, j, num):
                self.mat[i][j] = num
                if self.fillRemaining(i, j + 1):
                    return True
                self.mat[i][j] = 0
```

```
        # No valid value was found, so backtrack
        return False

    def removeKDigits(self):
        count = self.K

        while (count != 0):
            i = self.randomGenerator(self.N) - 1
            j = self.randomGenerator(self.N) - 1
            if (self.mat[i][j] != 0):
                count -= 1
                self.mat[i][j] = 0

        return

    def checkMove(self, i, j, num):
        if self.mat[i][j] == ' ':
            if self.checkIfSafe(i, j, num):
                return True
        return False

    def makeMove(self, i, j, num):
        if self.checkMove(i, j, num):
            self.mat[i][j] = num
            return True
        return False

    def checkWin(self):
        for i in range(self.N):
            for j in range(self.N):
                if self.mat[i][j] == ' ':
                    return False
        return True

    def __str__(self):
        return '╔═══╦═══╦═══╗\n' + \
               '║' + str(self.mat[0][0]) + ' ' + str(self.mat[0][1]) + ' ' +
str(self.mat[0][2]) + '║' + str(
            self.mat[0][3]) + ' ' + \
               str(self.mat[0][4]) + ' ' + str(self.mat[0][5]) + '║' +
str(self.mat[0][6]) + ' ' + str(
            self.mat[0][7]) + ' ' + \
               str(self.mat[0][8]) + '║\n' + \
```

```
            '╟' + str(self.mat[1][0]) + ' ' + str(self.mat[1][1]) + ' ' +
str(self.mat[1][2]) + '╟' + str(
            self.mat[1][3]) + ' ' + \
               str(self.mat[1][4]) + ' ' + str(self.mat[1][5]) + '╟' +
str(self.mat[1][6]) + ' ' + str(
            self.mat[1][7]) + ' ' + \
               str(self.mat[1][8]) + '╟\n' + \
               '╟' + str(self.mat[2][0]) + ' ' + str(self.mat[2][1]) + ' ' +
str(self.mat[2][2]) + '╟' + str(
            self.mat[2][3]) + ' ' + \
               str(self.mat[2][4]) + ' ' + str(self.mat[2][5]) + '╟' +
str(self.mat[2][6]) + ' ' + str(
            self.mat[2][7]) + ' ' + \
               str(self.mat[2][8]) + '╟\n' + \
               '╠═══════╬═══════╬═══════╣\n' + \
               '╟' + str(self.mat[3][0]) + ' ' + str(self.mat[3][1]) + ' ' +
str(self.mat[3][2]) + '╟' + str(
            self.mat[3][3]) + ' ' + \
               str(self.mat[3][4]) + ' ' + str(self.mat[3][5]) + '╟' +
str(self.mat[3][6]) + ' ' + str(
            self.mat[3][7]) + ' ' + \
               str(self.mat[3][8]) + '╟\n' + \
               '╟' + str(self.mat[4][0]) + ' ' + str(self.mat[4][1]) + ' ' +
str(self.mat[4][2]) + '╟' + str(
            self.mat[4][3]) + ' ' + \
               str(self.mat[4][4]) + ' ' + str(self.mat[4][5]) + '╟' +
str(self.mat[4][6]) + ' ' + str(
            self.mat[4][7]) + ' ' + \
               str(self.mat[4][8]) + '╟\n' + \
               '╟' + str(self.mat[5][0]) + ' ' + str(self.mat[5][1]) + ' ' +
str(self.mat[5][2]) + '╟' + str(
            self.mat[5][3]) + ' ' + \
               str(self.mat[5][4]) + ' ' + str(self.mat[5][5]) + '╟' +
str(self.mat[5][6]) + ' ' + str(
            self.mat[5][7]) + ' ' + \
               str(self.mat[5][8]) + '╟\n' + \
               '╠═══════╬═══════╬═══════╣\n' + \
               '╟' + str(self.mat[6][0]) + ' ' + str(self.mat[6][1]) + ' ' +
str(self.mat[6][2]) + '╟' + str(
            self.mat[6][3]) + ' ' + \
               str(self.mat[6][4]) + ' ' + str(self.mat[6][5]) + '╟' +
str(self.mat[6][6]) + ' ' + str(
            self.mat[6][7]) + ' ' + \
```

```
                str(self.mat[6][8]) + '\n' + \
                '' + str(self.mat[7][0]) + ' ' + str(self.mat[7][1]) + ' ' +
str(self.mat[7][2]) + '' + str(
            self.mat[7][3]) + ' ' + \
                str(self.mat[7][4]) + ' ' + str(self.mat[7][5]) + '' +
str(self.mat[7][6]) + ' ' + str(
            self.mat[7][7]) + ' ' + \
                str(self.mat[7][8]) + '\n' + \
                '' + str(self.mat[8][0]) + ' ' + str(self.mat[8][1]) + ' ' +
str(self.mat[8][2]) + '' + str(
            self.mat[8][3]) + ' ' + \
                str(self.mat[8][4]) + ' ' + str(self.mat[8][5]) + '' +
str(self.mat[8][6]) + ' ' + str(
            self.mat[8][7]) + ' ' + \
                str(self.mat[8][8]) + '\n' + \
                ''


def init(sock):
    print("Czekam na drugiego gracza...")
    sock.sendto("Hello".encode(), (IP, PORT))
    data, _ = sock.recvfrom(BUFF_SIZE)
    return data.decode()


def dataEncode(sudoku, playerScores, N):
    data = ""
    for i in range(N):
        for j in range(N):
            data += str(sudoku.mat[i][j]) + ","
    data += str(playerScores[0]) + (3 - len(str(playerScores[0]))) * " " +
str(playerScores[1]) + (
            3 - len(str(playerScores[1]))) * " "
    return data.encode()


def dataDecode(data):
    data = data.decode()
    data = data.split(',')
    matrix = data[:-1]
    board = []
    for i in range(0, 9):
        board.append([])
```

```
        for j in range(0, 9):
            toAppend = ' '
            if matrix[i * 9 + j] != ' ':
                toAppend = int(matrix[i * 9 + j])
            board[i].append(toAppend)
    scores = data[-1]
    score1 = int(scores[:3])
    score2 = int(scores[3:])
    return board, score1, score2


def main():
    N = 9
    K = 10
    sudoku = Sudoku(N, K)
    sudoku.fillValues()
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    player = init(sock)
    playerScores = [0, 0]
    print("Jesteś graczem: ", player)
    if player == "2":
        print(sudoku)
        print("Wynik: ", playerScores[0], ":", playerScores[1])
        print("Oczekiwanie na ruch przeciwnika")
        data, _ = sock.recvfrom(BUFF_SIZE)
        sudoku.mat, playerScores[0], playerScores[1] = dataDecode(data)
    while True:
        print("Wynik: ", playerScores[0], ":", playerScores[1])
        print(sudoku)
        print("Wpisz wiersz, kolumnę i liczbę oddzielone spacją")
        print("Wpisz 'quit' aby zakończyć grę")
        data = input()
        if data == "quit":
            break
        choice = data.split()
        if len(choice) != 3:
            print("Błędne dane")
            continue
        try:
            choice = [int(x) for x in choice]
        except ValueError:
            print("Błędne dane")
            continue
```

```
        choice[0] -= 1
        choice[1] -= 1
        if choice[0] < 0 or choice[0] >= N or choice[1] < 0 or choice[1] >=
N or choice[2] < 0 or choice[2] > N:
            print("Błędne dane")
            continue
        if sudoku.mat[choice[0]][choice[1]] != ' ':
            print("Pole zajęte")
            continue
        if not sudoku.makeMove(choice[0], choice[1], choice[2]):
            print("Błędny ruch")
            playerScores[int(player) - 1] -= 1
        else:
            print("Poprawny ruch")
            playerScores[int(player) - 1] += 1

        if sudoku.checkWin():
            print("Wygrałeś!" if playerScores[int(player) - 1] >
playerScores[int(player) % 2] else "Przegrałeś!") \
                if playerScores[int(player) - 1] != playerScores[int(player)
% 2] else "Remis!"
            break

        print(sudoku)
        print("Wynik: ", playerScores[0], ":", playerScores[1])
        print("Oczekiwanie na ruch przeciwnika")
        data = dataEncode(sudoku, playerScores, N)
        sock.sendto(data, (IP, PORT))
        data, _ = sock.recvfrom(BUFF_SIZE)
        sudoku.mat, playerScores[0], playerScores[1] = dataDecode(data)
        if sudoku.checkWin():
            print("Wygrałeś!" if playerScores[int(player) - 1] >
playerScores[int(player) % 2] else "Przegrałeś!") \
                if playerScores[int(player) - 1] != playerScores[int(player)
% 2] else "Remis!"
            break


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

Serwer:

```python
import socket
import random
import sys

IP = sys.argv[1]
PORT = int(sys.argv[2])
BUFF_SIZE = 168


def init(sock):
    players = []
    _, addr1 = sock.recvfrom(BUFF_SIZE)
    _, addr2 = sock.recvfrom(BUFF_SIZE)
    players.append(addr1)
    players.append(addr2)
    player1 = random.choice(players)
    players.remove(player1)
    player2 = players[0]
    sock.sendto("1".encode(), player1)
    sock.sendto("2".encode(), player2)
    return player1, player2


def checkEnd(data):
    data = data.decode()
    data = data.split(',')
    matrix = data[:-1]
    for row in matrix:
        if ' ' in row:
            return False
    return True


def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((IP, PORT))
    player1, player2 = init(sock)
    while True:
        player1_data, _ = sock.recvfrom(BUFF_SIZE)
        sock.sendto(player1_data, player2)
        if checkEnd(player1_data):
            break
        player2_data, _ = sock.recvfrom(BUFF_SIZE)
```

```
        sock.sendto(player2_data, player1)
        if checkEnd(player2_data):
            break
    sock.close()


if __name__ == "__main__":
    while True:
        main()
```