Национальный исследовательский ядерный университет «МИФИ»

Лабораторная работа по курсу
«Сетевые информационные технологии»

Выполнил: студент группы k8-361,
Рыбников Виталий

Цель работы:

Разработать клиент-серверное приложение, взаимодействующее на основе протокола `TCP/IP`.

Клиент должен быть написан под Windows с использованием C/C++ (MFC, .NET), JAVA или др. и иметь графический интерфейс.

Сервер должен быть написан под UNIX (без графического интерфейса). Работа приложения демонстрируется в компьютерном классе.

Постановка задачи

Крестики-нолики на поле 3x3

Клиент делает ход, ставит крестик, и сообщат об этом серверу.

Сервер делает ход, ставит нолик, и сообщат об этом клиенту.

Выигрывает тот, кто поставит в ряд по вертикали, горизонтали или диагонали три крестика (нолика).

Реализация

Подход

В качестве интерфейса обмена данными выбран интерфейс сокетов. Таким образом, общение между клиентом и сервером осуществляется посредством Inet Stream

сокетов. В качестве языка для реализации выбран — Python, в качестве библиотеки, для создания графического приложения — Gtk.

Алгоритм

Общение между клиентом и сервером решено было осуществлять в формате `json`. Язык Python предоставляет удобные средства для работы с этим форматом, позволяющие в один вызов конвертировать объекты (списки, словари, массивы) в `json`-формат и назад.

Пример передаваемого сообщения:

```
{"step" : [1, 1],
 "winner": 0,
 "error" : false }
```

- `step` — поле содержит координаты хода игрока/сервера в формате *[строка, столбец]*
- `winner` — поле содержит номер победителя:
  - `0` — победитель отсутствует
  - `1` — выиграл клиент
  - `2` — выиграл сервер
  - `3` — ничья
- `error` — логическое поле, сигнализирующее о наличии ошибки (обычно, об ошибке ввода пользователя)

Логика клиенской и серверной частей довольно наглядно представлена на упрощённой схеме 1.

Всё приложение состоит из следующих файлов:

- `tic-tac-server.py` — серверная консольная часть
- `tic-tac-client.py` — клиентская консольная часть

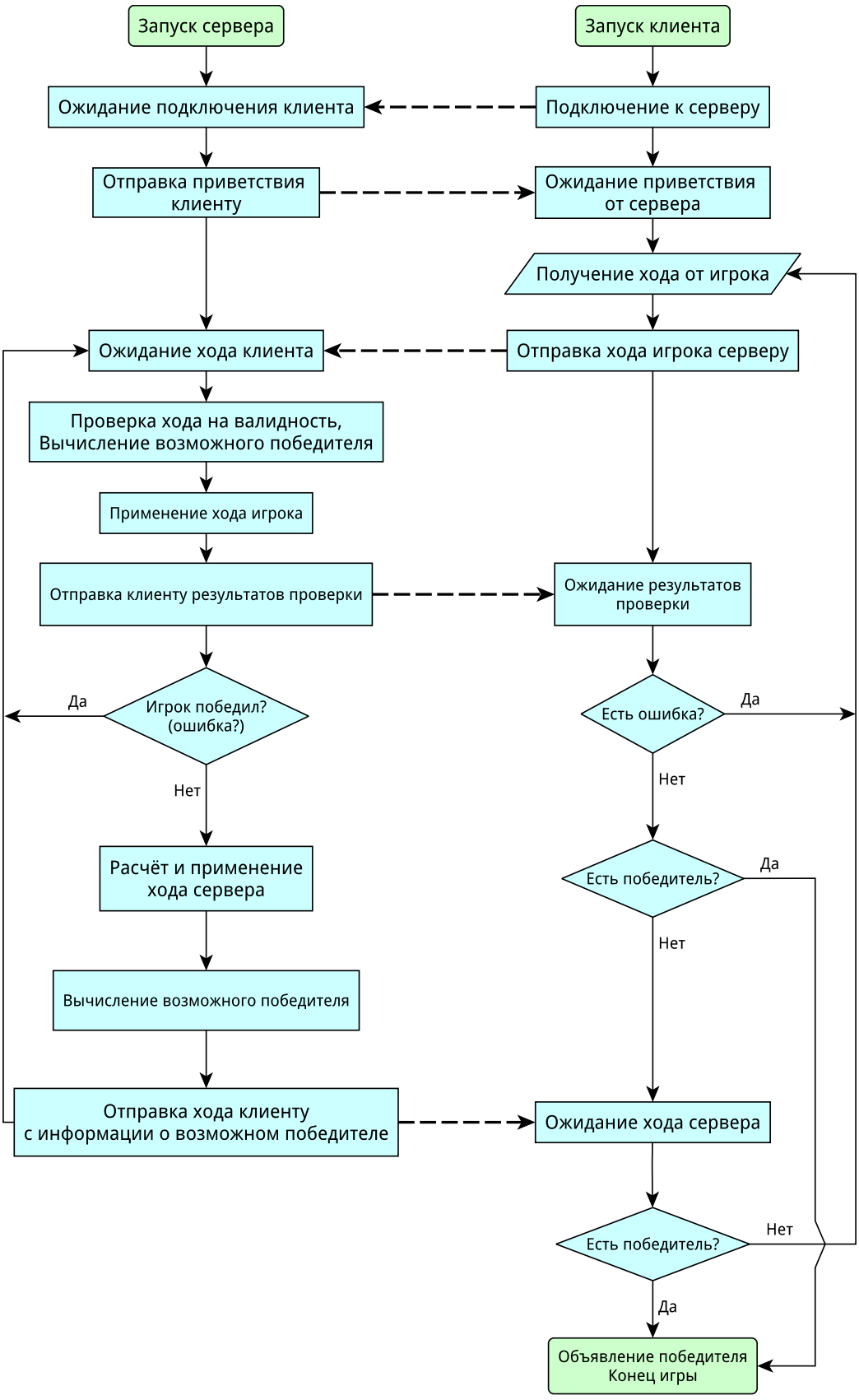

Рис. 1: Алгоритм клиент-серверной игры «Крестики-нолики»

- `tic-tac-client-gui.py` — клиентская графическая часть
- `tic_tac_common.py` — общая логика для всего проекта (используется как клиентской так и серверной частью)

Как видно, в ходе выполнения лабораторной работы, было реализовано две клиенские части — консольная и графическая. Это сделано в учебных целях, для лучшего понимания принципа работы с сетевыми сокетами. После клиентской консольной была написана клиентская графическая часть, которая использует ту же логику, что и консольная (общая логика выделена в отдельный модуль, что исключает дублирование кода).

Внешний вид графического приложения представлен на рисунке 2.

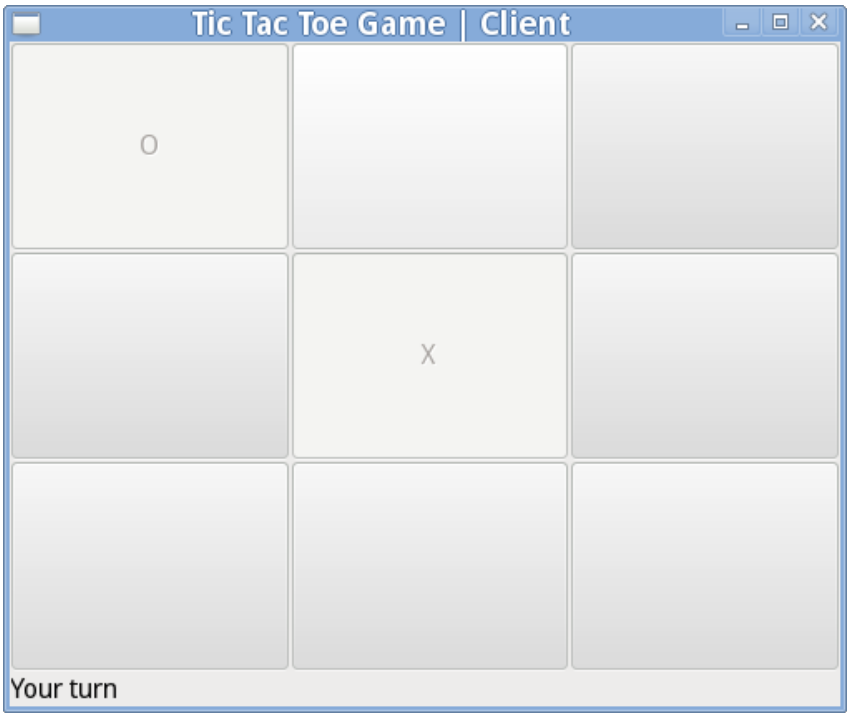

Рис. 2: Графическая клиентская часть

Игрок совершает ход, нажимая на пустые кнопки. Для простоты реализации, игрок всегда ходит «крестиками». В случае завершения игры, появляется сообщение о победе того, или иного игрока, как на рисунке 3.

Полный код лабораторной работы можно скачать по этой ссылке: `https://github.com/Jecomire/tic-tac-toe-game`. Ключевые моменты представлены в приложениях.

Рис. 3: Конец игры

Приложение 1

Основная логика серверной части представлена ниже:

```
def main():

    s = get_server_socket()
    try:
        ### endless loop, for multiple games
        while True:

            print ('Waiting for a player...')
            (clientsocket, address) = s.accept() # blocking line
            print ('New player came from {0}\n'.format(address))
            clientsocket.sendall("Hello from Tic Tac Toe server!")

            gf = copy.deepcopy(ttc.GAME_FIELD)

            ### one game, loop until winner or disconnect
            while True:
```

```
#B get user's turn
try:
    print("Wait␣for␣user's␣turn...")
    user_step = ttc.get_msg_from_socket(clientsocket,
                    exception=True, ex=False)
except Exception as exp:
    ttc.d(exp)
    ttc.d("\n" + 40*"=" + "\n")
    break;

# validate step #
step_check = {}
ttc.d("user␣raw␣turn:␣{}".format(user_step))

# thus, if True -> error = False
step_check["error"] = not ttc.is_step_correct (
        user_step
        , gf)

if not step_check["error"]:
    # i.e. error == False
    ttc.apply_turn (user_step
            , gf
            , ttc.USER_RAW_STEP)
    step_check["winner"] = get_winner(gf)
    ttc.print_game_field(gf)
else:
    step_check["winner"] = 0

#B answer, is step correct #
step_check_str = json.dumps(step_check)
ttc.d("I␣will␣send:␣{0}".format(step_check_str))
clientsocket.sendall(step_check_str)
time.sleep(0.1)

# if an error occured earlier ->
# get new answer from user
if True == step_check["error"] or 0 != step_check["\
   winner"]:
```

```
                continue;

            # do server step #
            ttc.d("proceed␣server␣turn")

            server_step_dict = do_server_step(gf)
            ttc.d("server␣step:␣{}".format(server_step_dict))
            ttc.apply_turn(json.dumps(server_step_dict)
                        , gf
                        , ttc.SERVER_RAW_STEP)

            # check for winners
            server_step_dict["winner"] = get_winner(gf)
            server_step_dict["error"]  = False

            #B send server turn with winner result
            clientsocket.sendall( json.dumps(server_step_dict) )

            ttc.print_game_field(gf)

    except KeyboardInterrupt as exp:
        print ("\nShutting␣down...␣{0}".format(exp))
    except Exception as exp:
        print("Sorry,␣but:␣{0}".format(exp))
    except:
        print("Unexpected␣error:", sys.exc_info()[0])

    try:
        clientsocket.close()
        s.close()
    except Exception as exp:
        # not an error on most cases
        ttc.d("Ooops␣>␣{0}".format(exp))

    sys.exit(0)
```

Приложение 2

Основная логика клиентской части (консольная) представлена ниже:

```
def main():

    s = ttc.get_client_socket()
    try:
        # get hello
        hello_msg = ttc.get_msg_from_socket(s)
        print("\n{0}\n".format(hello_msg))
        print('''
You are a cross (X).
Enter coordinats, where to put next cross.
Suppose, left top corner is (0, 0).
Input in format: <int> <int> <hit Return>
''')
        gf = copy.deepcopy(ttc.GAME_FIELD)
        ttc.print_game_field(gf)

        ### loop for a game, untill winner or ^C
        while True:

            #B get a step from user
            turn_json = ttc.get_turn_from_user(gf)

            #B send step to the server
            s.sendall(turn_json)

            #B get server answer about user step
            res = ttc.get_msg_from_socket(s, exception=False, ex=True)

            # if error - ask step again
            if is_error_in_answer(res):
                print("Ou, bad turn, try again.\n")
                continue;
            else:
                ttc.apply_turn(turn_json, gf, ttc.USER_RAW_STEP)
                ttc.print_game_field(gf)
```

```
            # check for winners in the answer,
            # if exist any - game ends.
            handle_winner_variable(res)

            #B get server step
            print("Wait for server response...")
            server_step = ttc.get_msg_from_socket(s)
            ttc.d("server step: {0}\n".format(server_step))
            ttc.apply_turn(server_step, gf, ttc.SERVER_RAW_STEP)
            handle_winner_variable(server_step)

            ttc.print_game_field(gf)

    except KeyboardInterrupt as k:
        print ("\nShutting down... {0}".format(k))
    except Exception as exp:
        print(": {0}".format(exp))
        ttc.print_game_field(gf)
    except:
        print("Unexpected error:", sys.exc_info()[0])

    s.close()
    sys.exit(0)
```

Приложение 3

Основная логика графической клиентской части представлена ниже:

```
# ------------------------------------------------ #
# main game login is here
# ------------------------------------------------ #
    def on_cell_toggled (self, button, data=None):
        """
        Toogle button with X or O
        """

        # lock UI
```

```
self.TicTacToeWindow.set_sensitive(False)
self.statusbar.push(0
    , "Pressed␣btn␣with␣coords:␣{}".format(data))

# lock cell
button.set_sensitive(False)
button.set_active(True)

# apply user turn
button.set_label(ttc.USER_STEP)

# create correct json-turn
### suppose, developer is True man,
### and all data is correct here =)
user_turn_json = self.convert_str_to_json_dict_step(data)

# send turn to the server
self.s.sendall(user_turn_json)



# get answer
self.statusbar.push(0, "Waiting␣for␣server␣validation...")
res = self._get_msg_from_server_socket()
time.sleep(0.1)

# check for errors and winners in the answer
# if winner - show msg and exit after that
self.handle_server_answer(res)

# get server's turn
self.statusbar.push(0, "Waiting␣for␣server's␣turn...")
server_turn_json = self._get_msg_from_server_socket()

# apply server's turn
self.apply_server_turn(server_turn_json)

# check for winners or TIE
# exit with msg if a winner exists
self.handle_server_answer(server_turn_json)
```

```
        # unlock UI
        self.TicTacToeWindow.set_sensitive(True)
        self.statusbar.push(0, "Your turn")

        # exit handler and wait for user turn
        return;
```