ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФГОБУ ВПО “СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ”

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту по дисциплине

“Отказоустойчивые вычислительные системы ”

на тему

РЕАЛИЗАЦИЯ ОТКАЗОУСТОЙЧИВОГО СЕТЕВОГО ПРИЛОЖЕНИЯ

Выполнили студенты Марков В.А. Самойлов Д.И.

Ф.И.О.

Группы _____

Работу принял _____ доцент Ю.С. Майданов

подпись

Защищена _____ Оценка _____

Новосибирск – 2016

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Большинство задач, которые приходится решать программисту в повседневной жизни, связано с выполнением локальных действий в системе. Обычно они не выходят за рамки управления мышью, клавиатурой, экраном и файловой системой. Гораздо труднее заставить взаимодействовать несколько программ, которые работают на разных компьютерах, подключенных к сети. Сетевое программиро вание, несомненно, полезно для опыта, так как приходится координировать работу множества компонентов приложения и тщательно распределять обязанности между ними. Фундаментальной единицей всего сетевого программирования в Linux (и большинстве других операционных систем) является сокет. Так же, как функции файлового ввода вывода определяют интерфейс взаимодействия с файловой системой, сокет соединяет программу с сетью. С его помощью она посылает и принимает сообщения. Создавать сетевые приложения труднее, чем даже заниматься многозадачным программированием, так как круг возникающих проблем здесь гораздо шире. Необходимо обеспечивать максимальную производительность, координировать обмен данными и управлять вводом выводом. В рамках данного курсового проекта реализовано клиент-серверное приложение - игра «Крестики-нолики».

1. Асинхронные сокеты Linux. Epoll().

Epoll (extended poll) — API мультиплексированного ввода-вывода, предоставляемого Linux для приложений. API позволяет приложениям осуществлять мониторинг нескольких открытых файловых дескрипторов (которые могут быть файлами, устройствами или сокетами, в том числе сетевыми), для того, чтобы узнать, готово ли устройство для продолжения ввода (вывода). Epoll планировался как более эффективная замена вызовам select() и poll(), определёнными в POSIX. Epoll может предоставить более эффективный механизм для приложений, обрабатывающих большое количество одновременно открытых соединений — со сложностью O(1) в отличие от стандартного механизма, обладающего сложностью O(n). Epoll аналогичен системе Kqueue из FreeBSD и также представляет собой объект ядра, предоставляемый для работы в пространстве пользователя в виде файлового дескриптора.

2. Отказоустойчивость

Отказоустойчивая архитектура с точки зрения инженерии — это метод проектирования отказоустойчивых систем, которые способны продолжать выполнение запланированных операций (возможно, с понижением эффективности) при отказе их компонентов. Термин часто используется для описания компьютерных систем, спроектированных продолжать работу в той или иной степени, с возможным уменьшением пропускной способности или увеличением времени отклика, в случае отказа части системы. Это означает, что система в целом не прекратит свою работу при возникновении проблем с аппаратной или программной частью

Если каждый компонент системы может продолжать работать при отказе одной из его составляющих, то вся система, в свою очередь, также продолжает работать.

Впервые идея включения избыточных частей для увеличения надежности системы была высказана Джоном фон Нейманом в 1950-х годах.

Существует два типа избыточности: пространственная и временная.

- Избыточность пространства реализуется путем введения дополнительных компонентов, функций или данных, которые не нужны при безотказном функционировании. Дополнительные (избыточные) компоненты могут быть аппаратными, программными и информационными.

- Временная избыточность реализуется путем повторных вычислений или отправки данных, после чего результат сравнивается с сохранённой копией предыдущего.

3. Постановка задачи

Разработать клиентскую и серверную часть приложения, которые должны позволить пользователю играть посредством локальной сети или сети «Интернет» с другими пользователями в игру «Крестики-Нолики». При возникновение отказов с серверной или клиентской стороны они должны быть распознаны и при возможности исправлены:

- при отказе клиента, сервер должен продолжить функционирование, сообщив второму игроку об уходе первого;
- при отказе всех серверов клиенты это должны распознать, вывести сообщение и завершить свою работу;
- необходимо реализовать альтернативный сервер управления игрой с протоколом синхронизации между основным и альтернативными серверами;
- при отказе основного сервера, клиенты должны переподключиться на альтернативный сервер и продолжить игру;

4. Протокол сетевого взаимодействия

```
/*
 * Перечисление типов игровых сообщений
 * AUTH - авторизация
 * GAME - игра
 * CHAT - чат
 * RESR – резервный сервер
 * EVN  - события
*/
enum MSG { AUTH, GAME, CHAT, RESR, EVN };

/*
 * Перечисление действий
 * YES - да
 * NO  – нет
*/
enum ACT { YES, NO };

/*
 * Перечисление меток
 * ZERO  - игрок-ноль
 * CROSS – игрок-крест
 * EMPTY – пустая игровая комната
 * FULL  - полная игровая комната
 */
enum MRK { ZERO, CROSS, EMPTY, FULL };

/*
 * Перечисление событий
 * TECH_WIN – техническая победа
 * UPD      - обновление
*/
enum EVN { TECH_WIN, UPD };

typedef struct {
    int room_state;
    int room_number;
```

```
    int room_players[2];
    int room_field[3][3];
} room_info; // описание игровой комнаты

typedef struct {
    int request;
    room_info game_rooms[4];
} resr_msg; // резервное сообщение

typedef struct {
    int new_game;
      int number;
    int player;
    int start
} auth_msg; // сообщение авторизации

typedef struct {
    int number;
    int player;
    int row;
    int col;
} game_msg; // сообщение игровой сессии

typedef struct {
    int number;
    int player;
    char string[256];
} chat_msg; // сообщение чата

typedef struct {
    int event;
} evnt_msg; // сообщение события

typedef struct {
    int type
    union
    {
            evnt_msg evnt;
            auth_msg auth;
            game_msg game;
            chat_msg chat;
            resr_msg resr;
    };
} common_msg; // основное сообщение
```

5. Представление игровых ситуаций

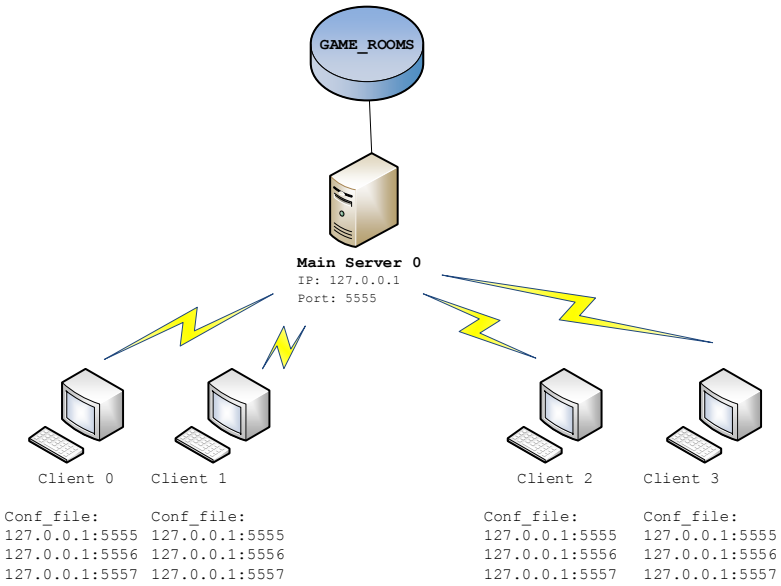

Рис. 1 – Стандартная игровая ситуация

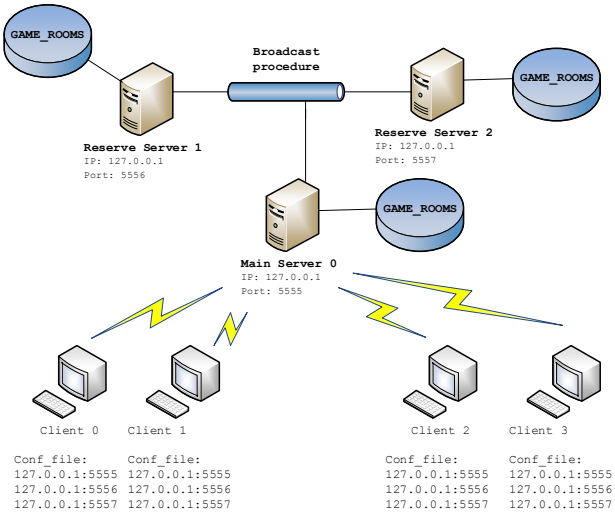

Рис. 2 – Игровая ситуация с подключенными резервными серверами

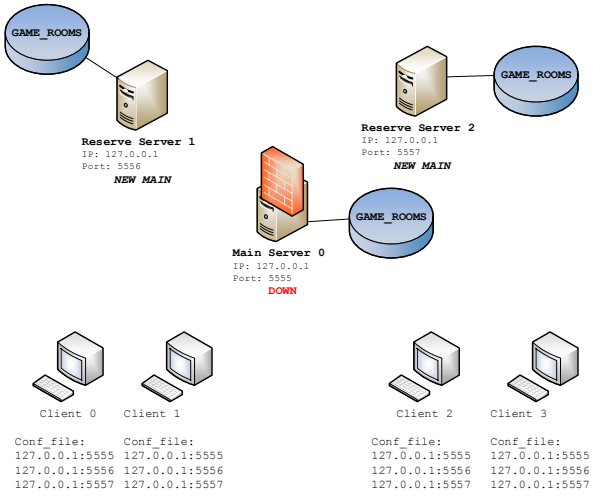

Рис. 3 – Игровая ситуация с основным сервером потеряно соединение

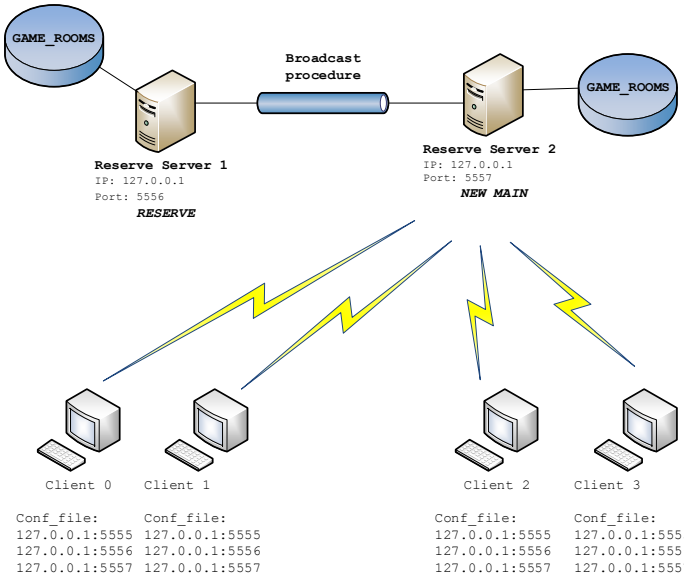

Рис. 4 – Игровая ситуация резервные сервер 2 принял подключения, стал новым главным, резервный сервер 1 не принял подключения, стал резервным.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы разработана и реализована отказоустойчивая система сетевого взаимодействия на примере игры «Крестики-нолики».

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ru.wikipedia.org/wiki/Отказоустойчивость
2. ru.wikipedia.org/wiki/Epoll
3. www.opennet.ru/man
4. Уолтон Ш. Создание сетевых приложений в среде Linux.: Пер. с англ.— М.: Вильямс, 2001. — 464 с.
5. Иванов Н.Н. Программирование в Linux. Самоучитель. –СПб.: БХВ-Петербург, 2007. – 416 с.

Message.h

```
#ifndef MESSAGE_H
#define MESSAGE_H

enum MSG { AUTH, GAME, CHAT, RESR, EVN };       // enumeration of messages
enum ACT { YES, NO };                           // enumeration of actions
enum MRK { ZERO, CROSS, EMPTY, FULL };          // enumeration of marks:
                                                //   - valid of players and game_filed
enum SST { MAIN, RESERVE };                                     // enumeration of server state
enum EVN { TECH_WIN, LOST_CONN, UPD };                  // enumeration of events

typedef struct {
         int room_state;
    int room_number;         // number of game room
    int room_players[2];     // store players fd:
                             //   - indexation [0] equal [ZERO] - players;
                             //   - indexation [1] equal [CROSS] - players.
    int room_field[3][3];    // game field
    int room_last;
} room_info;

typedef struct {
    int request;
    /*
     * case NO  - nothing
     * case YES - main server send actual information about all game rooms
     */
    room_info game_rooms[4];
} resr_msg;

typedef struct {
         // to server
         int new_game;               // YES - new, NO - continue

         /*
          * case: new_game = NO -> 'client' fill game_room and player field
          */

         // from server
         int number;                     // number of game room
         int player;                 // type of player ZERO or CROSS
         int start;                // game state YES or NO
} auth_msg;

typedef struct {
         int number;                     // number of game room
         int player;                     // type of players ZERO or CROSS
         int row;                    // y
         int col;                    // x
} game_msg;

typedef struct {
         int number;                         // room game number
         int player;                     // type of players ZERO or CROSS
         char string[256];  // chat message
} chat_msg;

typedef struct {
         int event;
} evnt_msg;

/*
 * Encapsulation of all type messages, between client and server and reserve
 */

typedef struct {
         int type;           // type of message
         /*
          * AUTHOR - for authorization
          * GAME   - for game process
          * CHAT   - for chat communication
          * RESR   - for main-reserve communication
```

```
          */
         union
         {
                  evnt_msg evnt;
                  auth_msg auth;  // authorization message
                  game_msg game;  // game message
                  chat_msg chat;  // chat message
                  resr_msg resr;  // reserve message
         };
} common_msg;

/*
 * Regular situation: Each message will be broadcasted to all reserver servers.
 * If main server is down: Use resr_msg with field request = 1.
 */

#endif
```

Client.c

```
#include "header.h"
int GAME_ROOM; // gome room number
int PLAYER;    // player X or O
int SERVER_FD; // server
// print network message
// initialize socket
static struct sockaddr_in configure_socket(struct sockaddr_in server, int port)
{
    bzero(&server, sizeof(server));

    server.sin_family = AF_INET;
    // переводим в сетевой порядок
    server.sin_port = htons(port);
    // преобразует строку с адресом в двоичную форму с сетевым порядком
    inet_aton(SERVER_IP, &(server.sin_addr));

    return server;
}

// try to connect to server
int try_to_connect(int server_fd, int mode)
{
    int i, fail_conn;
    struct sockaddr_in server_addr;
    common_msg msg;

    server_fd = socket(AF_INET, SOCK_STREAM, 0); // initialize socket
    if (server_fd < 0 ) {
        fprintf(stderr, "error: can't create socket\n");
        return -1;
    }

    // Loop for the try to connect (main or reserve0 or reserve1)
    for (i = 0, fail_conn = 0; i < NUM_SERVER; i++) {
        // Configure socket by differnt port
        server_addr = configure_socket(server_addr, CONF_LIST[i]);
        // Try to connect
        if (connect(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
            // Fail, reconnect to another
            //fprintf(stderr, "error: can't connect to %s:%d. try to another...\n", SERVER_IP,
CONF_LIST[i]);
            fail_conn++;
        } else {
            // Success
            switch(mode)
            {
                case 0: // branch new connection
                    printf("connected to %s:%d\n",SERVER_IP,CONF_LIST[i]);
                    msg.type = AUTH;
                    msg.auth.new_game = YES;
                    write(server_fd, &msg, sizeof(msg));
                    bzero(&msg, sizeof(msg));
                    read(server_fd, &msg, sizeof(msg));

                    // set field game room and player X or O
                    GAME_ROOM = msg.auth.number;
```

```
                    PLAYER = msg.auth.player;
                    print_msg(msg);
                    break;

                case 1: // branch restore connection
                    displayMsgChat("connection restore...\n");

                    msg.type = AUTH;
                    msg.auth.new_game = NO;
                    msg.auth.number = GAME_ROOM;
                    msg.auth.player = PLAYER;
                    write(server_fd, &msg, sizeof(msg));
                    break;

                default:
                 break;
            }
            break;
        }

        if (fail_conn == NUM_SERVER) {
            // All server are down
            fprintf(stderr, "error: all servers are down, shutting down...\n");
            close(server_fd);
            return -1;
        }
    }
    return server_fd;
}

int main(int argc, char const *argv[])
{
    struct pollfd fdt[1]; // poll
    struct sockaddr_in server_addr; // server structure

    SERVER_FD = try_to_connect(SERVER_FD, 0); // try to connect

    if (SERVER_FD == -1) {
        return -1;
    }

    game_session(SERVER_FD, 1);

    close(SERVER_FD);

    return 0;
}
```

Server/implementation.c

```
#include "header.h"

int CONF_LIST[] = { 2345,    // Main server port
                    2346,    // First reserve server port
                    2347 };  // Second reserve server port

extern room_info GAME_ROOMS[NUM_GAMES];

// Global variable (0-main/1-reserve)
int SERVER_STATE;
int reserve_list[100];
int counter = 0;

int create_socket(int fd)
{
    return socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, 0);
}

int close_socket(int fd)
{
    return close(fd);
}

int bind_socket(int fd, struct sockaddr_in server_addr, int port)
{
```

```
    int rs; // return status

    int option = 1;

    // Clean up structure
    bzero(&server_addr, sizeof(server_addr));

    // Fill the structure
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);                 // specify port
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);    // 127.0.0.1

    //Bind the server socket to the required ip-address and port.
    rs = bind(fd, (struct sockaddr*)&server_addr, sizeof(server_addr));

    if (setsockopt(fd, SOL_SOCKET,(SO_REUSEPORT | SO_REUSEADDR),(char*)&option,sizeof(option)) < 0)
    {
        fprintf(stderr, "error: setsockopt failed!\n");
        close(fd);
        exit(1);
    }

    return rs;
}

int listen_socket(int fd)
{
    return listen(fd, MAXCONN);
}

int create_epoll(int epoll_fd)
{
    return epoll_create(MAXCONN);
}

int modify_epoll_context(int epoll_fd, int operation, int server_fd, uint32_t events, void* data)
{
    int rs;
    struct epoll_event server_listen_event;

    server_listen_event.events = events;
    server_listen_event.data.ptr = data;

    rs = epoll_ctl(epoll_fd, operation, server_fd, &server_listen_event);

    return rs;
}

int make_socket_non_blocking(int fd)
{
    int flag;

    flag = fcntl(fd, F_GETFL, NULL);

    if (flag == -1) {
        err_msg("[FCNTL] F_GETFL failed.", strerror(errno));
        return -1;
    }

    flag |= O_NONBLOCK;

    if (fcntl(fd, F_SETFL, flag) == -1) {
        err_msg("[FCNTL] F_SETFL failed.%s", strerror(errno));
        return -1;
    }

    return 0;
}

void broadcast_to_reserve(common_msg data, int lenght, void *ptr)
{
    printf("\nbroadcast\n");
    int i;
    for (i = 0; i < counter; i++) {
        printf("=>%d ",reserve_list[i]);
        write(reserve_list[i], &data, lenght);
    }
```

```
    printf("\n");
}

// clear up one room
void free_room(int index)
{
    int j, k;

    GAME_ROOMS[index].room_state = EMPTY;

    for (j = 0; j < 2; j++) {
        GAME_ROOMS[index].room_players[j] = -1;
    }

    for (j = 0; j < 3; j++) {
        for (k = 0; k < 3; k++) {
            GAME_ROOMS[index].room_field[j][k] = -1;
        }
    }

    print_game_rooms(GAME_ROOMS, NUM_GAMES);
}

int get_another_client(int fd, int *index)
{
    int i, j;

    for (i = 0; i < NUM_GAMES; i++) {

        if (GAME_ROOMS[i].room_state == EMPTY) {
            return 0;
        }

        for (j = 0; j < 2; j++) {
            if (GAME_ROOMS[i].room_players[j] == fd) {
                *index = i;
                if (j == 0) {
                    return GAME_ROOMS[i].room_players[1];
                } else {
                    return GAME_ROOMS[i].room_players[0];
                }
            }
        }
    }
    return -1;
}

void game_handler(int fd, int event)
{
    int index; // game_room[index]
    common_msg msg;
    msg.type = EVN;
    msg.evnt.event = TECH_WIN;

    if (event == 1) {
        int client = get_another_client(fd, &index);
        if (client == -1) {
            fprintf(stderr, "Error: can't get player\n");
            return;
        } else if (client == 0) {
            fprintf(stderr, "NOTE: player left\n");
        } else {
            free_room(index);
            write(client, &msg, sizeof(common_msg));
        }
    }
}

// main network epoll handler
void *handle(void *ptr, int epoll_fd)
{
    struct EchoEvent *echoEvent = ptr;
    common_msg broadcast_msg;

    if (EPOLLIN == echoEvent->event) {
        int n = read(echoEvent->fd, &echoEvent->msg, sizeof(common_msg));
        if (n == 0) {
```

```
            // Client closed connection
            if (echoEvent->type == 0) {
                printf("\nPlayer on fd = %d closed connection!\n", echoEvent->fd);
                game_handler(echoEvent->fd, 1);
                printf("broadcast player left\n");
                broadcast_msg = send_update(broadcast_msg, 0);
                broadcast_to_reserve(broadcast_msg, sizeof(common_msg), NULL);
            } else {
                printf("\nServer on fd = %d closed connection!\n", echoEvent->fd);
            }
            close(echoEvent->fd);
            free(echoEvent);
        } else if(n == -1) {
            printf("\nClient error - closed connection.\n");
            close(echoEvent->fd);
            free(echoEvent);
        } else {
            echoEvent->length = n;
            printf("\nRead data form fd: %d\n", echoEvent->fd);
            switch (echoEvent->msg.type)
            {
                case GAME:
                    game_routine(echoEvent->msg);
                    broadcast_to_reserve(echoEvent->msg, sizeof(common_msg), ptr);
                    break;

                case CHAT:
                    chat_routine(echoEvent->msg);
                    break;
                default:
                    break;
            }
            modify_epoll_context(epoll_fd, EPOLL_CTL_ADD, echoEvent->fd, EPOLLOUT, echoEvent);
        }

    } else if (EPOLLOUT == echoEvent->event) {
        modify_epoll_context(epoll_fd, EPOLL_CTL_ADD, echoEvent->fd, EPOLLIN, echoEvent);
    }
}

struct sockaddr_in configure_socket(struct sockaddr_in server, int port)
{
    bzero(&server, sizeof(server));

    server.sin_family = AF_INET;
    // переводим в сетевой порядок
    server.sin_port = htons(port);
    // преобразует строку с адресом в двоичную форму с сетевым порядком
    inet_aton(SERVER_IP, &(server.sin_addr));

    return server;
}

// additional main server function
int create_main_new_server(int server_port, char *argc)
{
    char status[256];

    int option = 1;

    int epoll_fd;
    int server_fd;

    struct sockaddr_in server_addr;
    struct sockaddr_in client_addr;

    struct epoll_event *events;

    socklen_t client_len = sizeof(client_addr);

    // Create the server socket
    server_fd = create_socket(server_fd);
    if (server_fd == -1) {
        err_msg("[ERROR] Failed to create socket.", strerror(errno));
        return -1;
    }
```

```
    if (setsockopt(server_fd, SOL_SOCKET,(SO_REUSEPORT | SO_REUSEADDR),(char*)&option,sizeof(op-
tion)) < 0) {
        fprintf(stderr, "error: setsockopt failed!\n");
        close(server_fd);
        exit(1);
    }
    log_msg("[LOG] CREATE SOCKET       - OK", 1);
    //Bind the server socket
    if (bind_socket(server_fd, server_addr, server_port) == -1) {
    //if (bind_socket(server_fd, server_addr, SERVERPORT) == -1) {
        err_msg("[ERROR] Failed to bind.", strerror(errno));
        return -1;
    }
    log_msg("[LOG] BIND SOCKET         - OK", 1);
    // Listen the server socket
    if (listen_socket(server_fd) == -1) {
        err_msg("[ERROR] Failed to listen.", strerror(errno));
        return -1;
    }
    log_msg("[LOG] LISTEN SOCKET       - OK", 1);
    //Create the epoll context
    epoll_fd = create_epoll(epoll_fd);
    if (epoll_fd == -1) {
        err_msg("[ERROR] Failed to create epoll context", strerror(errno));
        return -1;
    }
    // Create the read event for server socket
    if (modify_epoll_context(epoll_fd, EPOLL_CTL_ADD, server_fd,\
                                                EPOLLIN, &server_fd) == -1)
    {
        err_msg("[ERROR]Failed to add an event for socket", strerror(errno));
        return -1;
    }
    log_msg("[LOG] EPOLL CREATE SOCKET - OK", 1);

    events = calloc(MAXEVENTS, sizeof(struct epoll_event));
    log_msg("[LOG] WAITING CONNECTION  - ...", 1);
    bool first_cycle = true;
    // Main loop
    while(1) {
        int events_count;

        if (first_cycle) {
            events_count = epoll_wait(epoll_fd, events, MAXEVENTS, 6000);
            if (events_count == 0) {
                printf("No connection accepted, change state to reserve\n");
                sleep(1);
                create_reserve_server(server_port, argc);
            }
            first_cycle = false;
        } else {
            events_count = epoll_wait(epoll_fd, events, MAXEVENTS, -1);
        }

        if (events_count ==-1) {
            err_msg("[ERROR] Failed to wait.", strerror(errno));
            return -1;
        }
        int i;

        for (i = 0; i < events_count; i++) {

            if (events[i].data.ptr == &server_fd) {
                if (events[i].events & EPOLLHUP || events[i].events & EPOLLERR) {
                    close(server_fd);
                    return 1;
                }

                int conn_fd = accept(server_fd, (struct sockaddr*)&client_addr, &client_len);
                if (conn_fd == -1) {
                    printf("Accept failed. %s", strerror(errno));
                    return 1;
                } else {
                    printf("[NET] Accepted connection...");
                    int type;
                    int conn_type;
                    common_msg msg;
```

```
                    read(conn_fd, &msg, sizeof(common_msg));
                    switch (msg.type)
                    {
                        case AUTH:
                            printf("client %d\n", conn_fd);
                            accept_new_client(msg, conn_fd);
                            conn_type = 0;
                            break;
                        case RESR:
                            printf("server %d\n",conn_fd);
                            accept_new_rserver(msg, conn_fd);
                            conn_type = 1;
                            break;

                        default:
                            fprintf(stderr, "Error: unknow connection %d - closing connection!\n",
conn_fd);
                            close(conn_fd);
                            break;
                    }

                    struct EchoEvent *echoEvent = calloc(1, sizeof(struct EchoEvent));
                    echoEvent->fd = conn_fd;
                    echoEvent->type = conn_type;
                    //Add a read event
                    modify_epoll_context(epoll_fd, EPOLL_CTL_ADD, echoEvent->fd, EPOLLIN, ech-
oEvent);

                }

            } else {
                if (events[i].events & EPOLLHUP || events[i].events & EPOLLERR) {
                    struct EchoEvent* echoEvent = (struct EchoEvent*) events[i].data.ptr;
                    printf("\nClosing connection socket\n");
                    close(echoEvent->fd);
                    free(echoEvent);
                } else if (EPOLLIN == events[i].events) {
                    struct EchoEvent* echoEvent = (struct EchoEvent*) events[i].data.ptr;
                    echoEvent->event = EPOLLIN;
                    // Delete the read event

                    printf("epoll 1 \n");

                    modify_epoll_context(epoll_fd, EPOLL_CTL_DEL, echoEvent->fd, 0, 0);
                    handle(echoEvent, epoll_fd);
                } else if (EPOLLOUT == events[i].events) {
                    struct EchoEvent* echoEvent = (struct EchoEvent*) events[i].data.ptr;
                    echoEvent->event = EPOLLOUT;
                    // Delete the write event.

                    printf("epoll 2 \n");

                    modify_epoll_context(epoll_fd, EPOLL_CTL_DEL, echoEvent->fd, 0, 0);
                    handle(echoEvent, epoll_fd);
                }
            }
        }
    }
    free(events);
    if (close_socket(server_fd) == 1) {
        err_msg("[ERROR] Failed to close.", strerror(errno));
        return -1;
    }
    return 0;
}
// reserver server main function
int create_reserve_server(int server_port, char *argc)
{
    int i, fail_conn;
    int server_fd;

    struct sockaddr_in server_addr;

    struct pollfd fdt[1];

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0 ) {
```

```
        fprintf(stderr, "error: can't create socket\n");
    }

    // Loop for the try to connect (main or reserve0 or reserve1)
    for (i = 0, fail_conn = 0; i < NUM_SERVER; i++) {
        // Configure socket by differnt port
        server_addr = configure_socket(server_addr, CONF_LIST[i]);
        // Try to connect
        if (connect(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
            // Fail, reconnect to another
            fprintf(stderr, "error: can't connect to %s:%d. try to another\n", SERVER_IP,
CONF_LIST[i]);
            fail_conn++;
        } else {
            // Success
            common_msg msg;
            msg.type = RESR;
            msg.resr.request = YES;

            write(server_fd, &msg, sizeof(common_msg));
            printf("connected to %s:%d\n",SERVER_IP,CONF_LIST[i]);
            read(server_fd, &msg, sizeof(common_msg));
            update_rooms(msg);
            break;
        }
        if (fail_conn == NUM_SERVER) {
            // All server are down
            fprintf(stderr, "error: all servers are down, shutting down...\n");
            return -1;
        }
    }

    // Настраиваем что проверять функции poll
    fdt[0].fd = server_fd;
    fdt[0].events = POLLIN;

    // Main loop
    while(1) {

        common_msg msg;
        // Ожидаем приход пакета от сервера
        int len = poll(fdt, 1, 5000);
        len = read(server_fd, &msg, sizeof(msg));// ожидание
        if (len == 0 ) {
            printf("\nMain server is down. Change state from reserve to main\n");

            close(server_fd);

            create_main_new_server(server_port, argc);

        } else {
            printf("[BROADCAST]\n");
            switch(msg.type)
            {
                case GAME:
                    printf("GAME MSG\n");
                    game_routine(msg);
                    break;

                case EVN:

                    break;

                case RESR:
                    printf("UPDATE MSG\n");
                    update_rooms(msg);
                    break;

                default:
                    break;
            }
        }
    }

    close(server_fd);

    return 0;}
```

```
// accept new player
void accept_new_client(common_msg msg, int fd)
{
    int type_of_player;
    int action;

    common_msg broadcast_msg;

    switch(msg.auth.new_game)
    {
        case YES:
            msg.auth.number = set_new_player_in_room(fd, &type_of_player, &action);
            msg.auth.player = type_of_player;
            msg.auth.start = action;
            printf("client %d, assing room %d player %d\n", fd, msg.auth.number, type_of_player);

            broadcast_msg = send_update(broadcast_msg, 0);
            broadcast_to_reserve(broadcast_msg, sizeof(common_msg), NULL);

            if (action == YES) {
                write(fd, &msg, sizeof(msg));
                msg.auth.player = ZERO;
                write(GAME_ROOMS[msg.auth.number].room_players[0], &msg, sizeof(msg));
                printf("game room %d start game\n", msg.auth.number);
            }
            break;

        case NO:
            restore_game(msg.auth.number, msg.auth.player, fd);
            printf("client %d, restore to room %d player %d\n", fd, msg.auth.number,
msg.auth.player);
        default:
            break;
    }
}

// accept new reserver server
void accept_new_rserver(common_msg msg, int fd)
{
    reserve_list[counter] = fd;
    counter++;
    msg = send_update(msg, fd);
    write(fd, &msg, sizeof(msg));
}

// send upadate message to reserve servers
common_msg send_update(common_msg msg, int fd)
{
    printf("SEND: upadate rooms to reserve server\n");
    msg.type = RESR;
    msg.resr.request = NO;
    int i;
    for (i = 0; i < NUM_GAMES; i++) {
        int j, k;
        msg.resr.game_rooms[i].room_state = GAME_ROOMS[i].room_state;
        msg.resr.game_rooms[i].room_number = GAME_ROOMS[i].room_number;
        msg.resr.game_rooms[i].room_last = GAME_ROOMS[i].room_last;
        for (j = 0; j < 2; j++) {
            msg.resr.game_rooms[i].room_players[j] = GAME_ROOMS[i].room_players[j];
        }
        for (j = 0; j < 3; j++) {
            for (k = 0; k < 3; k++) {
                msg.resr.game_rooms[i].room_field[j][k] = GAME_ROOMS[i].room_field[j][k];
            }
        }
    }
    return msg;
}

// assing new player in the game room
int set_new_player_in_room(int fd, int *type_of_player, int *action)
{
    int index = get_free_room();
    int i;
    // search empty place for player
    for (i = 0; i < 2; i++) {
        if (GAME_ROOMS[index].room_players[i] == -1) {
```

```
            GAME_ROOMS[index].room_players[i] = fd;
            if (i == 0) {
                *action = NO;
                *type_of_player = ZERO;
                return index;
            } else {
                GAME_ROOMS[index].room_state = FULL;
                *action = YES;
                *type_of_player = CROSS;
                return index;
            }
            break;
        }
    }

    GAME_ROOMS[index];
}

// reconnect client fd with game room structure
int restore_game(int number, int player, int fd)
{
    GAME_ROOMS[number].room_players[player] = fd;
}

// get empty game room
int get_free_room()
{
    int i;
    for (i = 0; i < NUM_GAMES; i++) {
        if (GAME_ROOMS[i].room_state == EMPTY) {
            return i;
        }
    }
}

// chat routine
void chat_routine(common_msg msg)
{
    int index = msg.game.number;
    int player = msg.game.player;

    if (player == ZERO) {
        printf("chat message from: ZERO\n");
        printf("chat msg: from %d to %d \n", GAME_ROOMS[index].room_players[player], GAME_ROOMS[in-
dex].room_players[CROSS]);
        write(GAME_ROOMS[index].room_players[CROSS], &msg, sizeof(common_msg));
    } else {
        printf("chat message from: CROSS\n");
        printf("chat msg: from %d to %d \n", GAME_ROOMS[index].room_players[player], GAME_ROOMS[in-
dex].room_players[ZERO]);
        write(GAME_ROOMS[index].room_players[ZERO], &msg, sizeof(common_msg));
    }
}

// game routine
void game_routine(common_msg msg)
{
    int index = msg.game.number;
    int row = msg.game.row;
    int col = msg.game.col;
    int player = msg.game.player;

    if (player == ZERO) {
        printf("move: ZERO\n");
    } else {
        printf("move: CROSS\n");
    }

    // make move
    if (GAME_ROOMS[index].room_field[row][col] == -1) {
        GAME_ROOMS[index].room_field[row][col] = player;
        GAME_ROOMS[index].room_last = player;
        if (player == ZERO) {
            printf("from %d to %d \n", GAME_ROOMS[index].room_players[player], GAME_ROOMS[in-
dex].room_players[CROSS]);
            write(GAME_ROOMS[index].room_players[CROSS], &msg, sizeof(common_msg));
        } else {
```

```
            printf("from %d to %d \n", GAME_ROOMS[index].room_players[player], GAME_ROOMS[in-
dex].room_players[ZERO]);
            write(GAME_ROOMS[index].room_players[ZERO], &msg, sizeof(common_msg));
        }
    } else {
        // send ERR MSG to fd
        fprintf(stderr, "Error: cell is taken!\n");
        return;
    }
    print_game_room(GAME_ROOMS[index]);
}
```