Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Лабораторна робота №4

Мережеве програмування в середовищі Unix

Тема: TCP клієнт-сервер з мультиплексуванням введення-виведення

Виконав
студент групи ІП-11:
Панченко С. В.

Перевірив:
Сімоненко А.В.

Київ 2025

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Розробити однопотоковий сервер з використанням мультиплексування введення-виведення для одночасної роботи з кількома клієнтами, застосовуючи системні виклики POSIX select() або poll().

2 ЗАВДАННЯ

Розробити однопотоковий сервер, який виконує наступне:

1. Сервер підтримує аргументи командного рядка, визначені в лабораторній роботі No3. Також сервер підтримує аргумент командного рядка, який визначає максимальну кількість клієнтів, з якими сервер може одночасно працювати. Сервер не приймає нові TCP з’єднання після досягнення цього значення.
2. Сервер працює з клієнтами відповідно до користувальницького протоколу, визначеного в лабораторній роботі No3.
3. Сервер дозволяє одночасно працювати з кількома клієнтами за допомогою мультиплексування введення-виведення. Сервер послуговується системними викликами select() або poll() для мультиплексування введення-виведення.
4. Кількість даних, які сервер зчитує або відправляє одному клієнту під час виконання введення-виведення з ним, треба обмежити. Ця кількість задається в коді сервера константою, яка може мати значення 1 байт та більше. Тобто, якщо сервер отримав інформацію від ядра про можливість виконати введення або виведення для якогось дескриптора файлу сокета, тоді серверу дозволено відправити або отримати даних розміром не більше вказаної константи. Це обмеження дає змогу майже порівну розподіляти час роботи сервера для кожного клієнта, який потребує комунікації. Також невеликі значення цієї константи дозволяють імітувати проблеми з мережею та частково імітувати різну поведінку клієнтів. Сервер не має завершувати своє виконання у випадку виникнення несистемної помилки. Рекомендації для сервера такі самі, які були дані в лабораторній роботі No3.

3 ВИКОНАННЯ

Розглянемо роботу додатку:

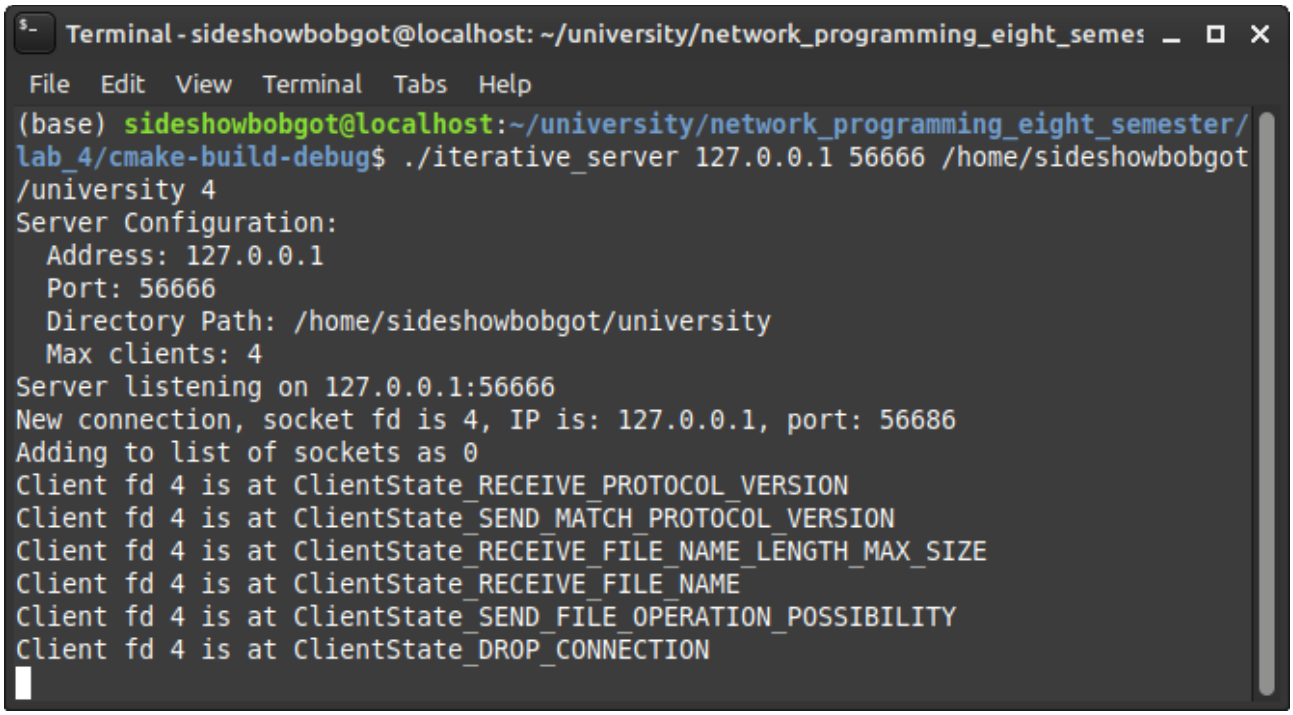

Рисунок 3.1 - Ітеративний сервер

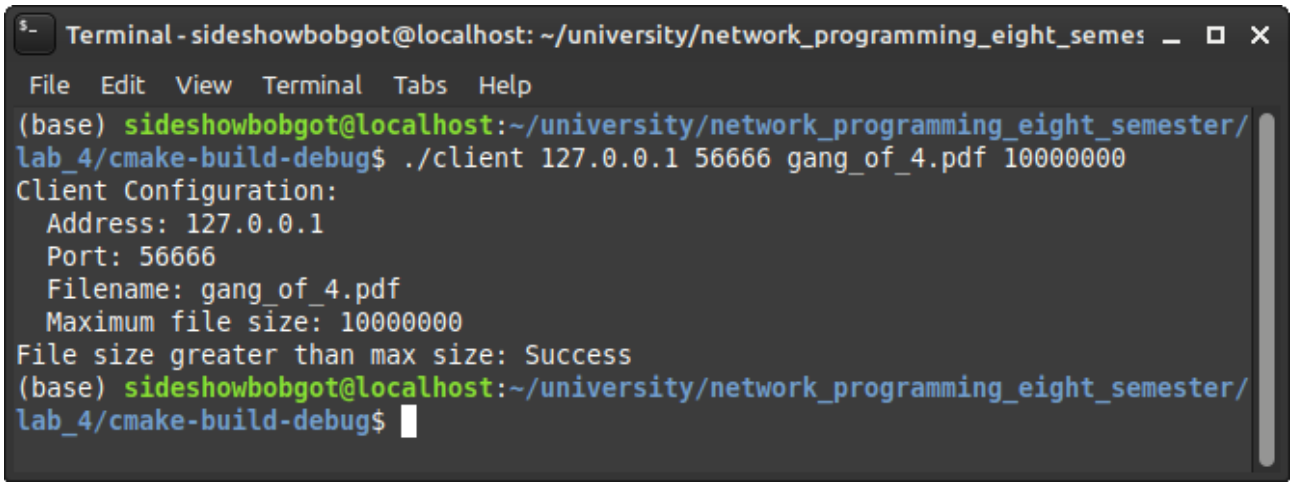

Рисунок 3.2 - Клієнт

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду

(Найменування програми (документа))

Жорсткий диск

(Вид носія даних)

(Обсяг програми (документа), арк.)

Студента групи ІП-11 4 курсу

Панченка С. В

```
// ./lab_4/iterative_server.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <signal.h>
#include <sys/stat.h>
#include <dirent.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <stdbool.h>
#include <errno.h>
#include <string.h>
#include <assert.h>
#include "protocol.h"

typedef struct {
    const char *address;
    int port;
    const char *dir_path;
    uint32_t max_clients;
} IterativeServerConfig;

static void print_config(const IterativeServerConfig *config) {
    printf("Server Configuration:\n");
    printf("  Address: %s\n", config->address);
    printf("  Port: %d\n", config->port);
    printf("  Directory Path: %s\n", config->dir_path);
    printf("  Max clients: %u\n", config->max_clients);
```

```
}

static IterativeServerConfig handle_cmd_args(const int argc, const char * const * const argv) {
    if (argc != 5) {
        fprintf(stderr, "Usage: %s <server_address> <server_port> <directory_path>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const IterativeServerConfig config = {
        .address = argv[1],
        .port = atoi(argv[2]),
        .dir_path = argv[3],
        .max_clients = atoi(argv[4])
    };

    print_config(&config);

    return config;
}

static int create_and_bind_socket(const int port, const char* const address, const int backlog) {
    struct sockaddr_in srv_sin4 = {0};
    srv_sin4.sin_family = AF_INET;
    srv_sin4.sin_port = htons(port);

            const int listenfd = socket(srv_sin4.sin_family, SOCK_STREAM | SOCK_NONBLOCK, IPPROTO_TCP);
    if (listenfd < 0)
```

```
        exit_err("socket()");

    if(inet_pton( srv_sin4.sin_family, address, &srv_sin4.sin_addr) < 0) {
        exit_err("inet_pton()");
    }

    if (bind(listenfd, (const struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
        exit_err("bind()");

    if (listen(listenfd, backlog) < 0)
        exit_err("listen()");

    printf("Server listening on %s:%d\n", address, port);
    return listenfd;
}

volatile sig_atomic_t keep_running = 1;
static void handle_sigint(const int) { keep_running = 0; }

struct ClientState_Invalid {};
struct ClientState_ReceiveProtocolVersion {
    int client_fd;
};
struct ClientState_SendMatchProtocolVersion {
    int client_fd;
    int client_protocol_version;
};
struct ClientState_ReceiveFileNameLengthMaxSize {
    int client_fd;
} ;
struct ClientState_ReceiveFileName {
```

```
    int client_fd;
    FileNameLengthMaxSize file_name_length_max_size;
};
struct ClientState_SendFileOperationPossibility {
    int client_fd;
    enum OperationPossibility operation_possibility;
    FILE* file;
    off_t file_size;
};

struct ClientState_SendFileAndChunkSize {
    int client_fd;
    FILE* file;
    off_t file_size;
};

struct ClientState_SendFileChunk {
    int client_fd;
    FILE* file;
};

struct ClientState_DropConnection {
    int client_fd;
};

typedef enum {
    ClientStateTag_INVALID,
    ClientState_RECEIVE_PROTOCOL_VERSION,
    ClientState_SEND_MATCH_PROTOCOL_VERSION,
    ClientState_RECEIVE_FILE_NAME_LENGTH_MAX_SIZE,
    ClientState_RECEIVE_FILE_NAME,
```

```
    ClientState_SEND_FILE_OPERATION_POSSIBILITY,
    ClientState_SEND_FILE_AND_CHUNK_SIZE,
    ClientState_SEND_FILE_CHUNK,
    ClientState_DROP_CONNECTION
} ClientStateTag;

typedef struct {
    ClientStateTag tag;
    union {
        struct ClientState_Invalid invalid;
        struct ClientState_ReceiveProtocolVersion receive_protocol_version;
        struct ClientState_SendMatchProtocolVersion send_match_protocol_version;
        struct ClientState_ReceiveFileNameLengthMaxSize receive_file_name_length;
        struct ClientState_ReceiveFileName receive_file_name;
        struct ClientState_SendFileOperationPossibility send_file_operation_possibility;
        struct ClientState_SendFileAndChunkSize send_file_and_chunk_size;
        struct ClientState_SendFileChunk send_file_chunk;
        struct ClientState_DropConnection drop_connection;
    } value;
} ClientState;

int ClientState_client_fd(const ClientState* client_state) {
    switch (client_state->tag) {
        case ClientStateTag_INVALID: {
            exit_err("Can not get client_fd from INVALID state");
        }
        case ClientState_RECEIVE_PROTOCOL_VERSION:
            return client_state->value.receive_protocol_version.client_fd;
        case ClientState_SEND_MATCH_PROTOCOL_VERSION:
            return client_state->value.send_match_protocol_version.client_fd;
        case ClientState_RECEIVE_FILE_NAME_LENGTH_MAX_SIZE:
```

```
            return client_state->value.receive_file_name_length.client_fd;
        case ClientState_RECEIVE_FILE_NAME:
            return client_state->value.receive_file_name.client_fd;
        case ClientState_SEND_FILE_OPERATION_POSSIBILITY:
            return client_state->value.send_file_operation_possibility.client_fd;
        case ClientState_SEND_FILE_AND_CHUNK_SIZE:
            return client_state->value.send_file_and_chunk_size.client_fd;
        case ClientState_SEND_FILE_CHUNK:
            return client_state->value.send_file_chunk.client_fd;
        case ClientState_DROP_CONNECTION:
            return client_state->value.drop_connection.client_fd;
        default:
            __builtin_unreachable();
    }
}

#define NAME_MAX_WITHOUT_NULL_TERMINATOR NAME_MAX
#define PATH_MAX_WITHOUT_NULL_TERMINATOR PATH_MAX

static ClientState construct_drop_connection(const int client_fd) {
    ClientState new_state;
    new_state.tag = ClientState_DROP_CONNECTION;
    new_state.value.drop_connection.client_fd = client_fd;
    return new_state;
}

#define CHUNK_SIZE 4096

ClientState ClientState_transition(
    const ClientState* const generic_state,
    const fd_set* const readfds,
```

```
    const fd_set* const writefds,
    const char* const dir_path
) {
    switch (generic_state->tag) {
        case ClientStateTag_INVALID: return *generic_state;
        case ClientState_RECEIVE_PROTOCOL_VERSION: {
                const struct ClientState_ReceiveProtocolVersion* const cur_state =
&generic_state->value.receive_protocol_version;
            if(FD_ISSET(cur_state->client_fd, readfds)) {
                printf("Client fd %d is at ClientState_RECEIVE_PROTOCOL_VERSION\
n", cur_state->client_fd);

                ClientState next_state;
                next_state.tag = ClientState_SEND_MATCH_PROTOCOL_VERSION;
                    next_state.value.send_match_protocol_version.client_fd = cur_state-
>client_fd;
                recv(cur_state->client_fd,
                    &next_state.value.send_match_protocol_version.client_protocol_version
,
                    sizeof(next_state.value.send_match_protocol_version.client_protocol_ve
rsion),
                    0
                );
                return next_state;
            }
            return *generic_state;
        }
        case ClientState_SEND_MATCH_PROTOCOL_VERSION: {
                const struct ClientState_SendMatchProtocolVersion* const cur_state =
&generic_state->value.send_match_protocol_version;
            if(FD_ISSET(cur_state->client_fd, writefds)) {
```

```
                                                printf("Client fd %d is at ClientState_SEND_MATCH_PROTOCOL_VERSION\n", cur_state->client_fd);

            const int server_protocol_version = 1;
               const bool protocol_version_match = cur_state->client_protocol_version == server_protocol_version;
                                    if(send(cur_state->client_fd, &protocol_version_match, sizeof(protocol_version_match), 0) == -1) {
               return construct_drop_connection(cur_state->client_fd);
            }
            if(!protocol_version_match) {
               return construct_drop_connection(cur_state->client_fd);
            }
            ClientState next_state;
                                                                    next_state.tag = ClientState_RECEIVE_FILE_NAME_LENGTH_MAX_SIZE;
            next_state.value.receive_file_name_length.client_fd = cur_state->client_fd;
            return next_state;
         }
         return *generic_state;
      }
      case ClientState_RECEIVE_FILE_NAME_LENGTH_MAX_SIZE: {
          const struct ClientState_ReceiveFileNameLengthMaxSize* const cur_state = &generic_state->value.receive_file_name_length;
         if(FD_ISSET(cur_state->client_fd, readfds)) {
                                                printf("Client fd %d is at ClientState_RECEIVE_FILE_NAME_LENGTH_MAX_SIZE\n", cur_state->client_fd);

            ClientState next_state;
            next_state.tag = ClientState_RECEIVE_FILE_NAME;
```

```
            next_state.value.receive_file_name.client_fd = cur_state->client_fd;
            recv(
                cur_state->client_fd,
                &next_state.value.receive_file_name.file_name_length_max_size,
                sizeof(next_state.value.receive_file_name.file_name_length_max_size),
                0
            );
            return next_state;
        }
        return *generic_state;
    }
    case ClientState_RECEIVE_FILE_NAME: {
        const struct ClientState_ReceiveFileName* const cur_state = &generic_state->value.receive_file_name;
        if(FD_ISSET(cur_state->client_fd, readfds)) {
                printf("Client fd %d is at ClientState_RECEIVE_FILE_NAME\n", cur_state->client_fd);

            char name_buffer[NAME_MAX_WITHOUT_NULL_TERMINATOR + 1] = {0};
            recv(cur_state->client_fd, name_buffer, sizeof(name_buffer), 0);
            name_buffer[NAME_MAX_WITHOUT_NULL_TERMINATOR] = '\0';

             char path_buffer[PATH_MAX_WITHOUT_NULL_TERMINATOR] = {0};
            snprintf(path_buffer, PATH_MAX_WITHOUT_NULL_TERMINATOR, "%s/%s", dir_path, name_buffer);

            ClientState next_state;
            next_state.tag = ClientState_SEND_FILE_OPERATION_POSSIBILITY;
                next_state.value.send_file_operation_possibility.client_fd = cur_state-
```

```
>client_fd;
            next_state.value.send_file_operation_possibility.file = NULL;
                next_state.value.send_file_operation_possibility.operation_possibility =
OPERATION_POSSIBLE;


            struct stat st;
            if(stat(path_buffer, &st) == -1 || !S_ISREG(st.st_mode)) {
                next_state.value.send_file_operation_possibility.operation_possibility =
FILE_NOT_FOUND;
             } else if(cur_state->file_name_length_max_size.file_max_size < st.st_size)
{
                next_state.value.send_file_operation_possibility.operation_possibility =
FILE_SIZE_GREATER_THAN_MAX_SIZE;
            } else {
                                    next_state.value.send_file_operation_possibility.file  =
fopen(path_buffer, "rb");
                if(!next_state.value.send_file_operation_possibility.file) {
                    next_state.value.send_file_operation_possibility.operation_possibility
= FAILED_TO_OPEN_FILE;
                }
            }
            next_state.value.send_file_operation_possibility.file_size = st.st_size;
            return next_state;
          }
          return *generic_state;
       }
       case ClientState_SEND_FILE_OPERATION_POSSIBILITY: {
              const struct ClientState_SendFileOperationPossibility* const cur_state =
&generic_state->value.send_file_operation_possibility;
          if(FD_ISSET(cur_state->client_fd, writefds)) {
                                                printf("Client    fd    %d    is    at
```

```
ClientState_SEND_FILE_OPERATION_POSSIBILITY\n", cur_state->client_fd);

                    if(send(cur_state->client_fd,  &cur_state->operation_possibility,
sizeof(cur_state->operation_possibility), 0) == -1) {
                return construct_drop_connection(cur_state->client_fd);
            }
            if(cur_state->operation_possibility != OPERATION_POSSIBLE) {
                return construct_drop_connection(cur_state->client_fd);
            }
            ClientState next_state;
            next_state.tag = ClientState_SEND_FILE_AND_CHUNK_SIZE;
                    next_state.value.send_file_and_chunk_size.client_fd = cur_state-
>client_fd;
            next_state.value.send_file_and_chunk_size.file = cur_state->file;
            next_state.value.send_file_and_chunk_size.file_size = cur_state->file_size;
            return next_state;
          }
          return *generic_state;
        }
        case ClientState_SEND_FILE_AND_CHUNK_SIZE: {
                const struct ClientState_SendFileAndChunkSize* const cur_state =
&generic_state->value.send_file_and_chunk_size;
          if(FD_ISSET(cur_state->client_fd, writefds)) {
             printf("Client fd %d is at ClientState_SEND_FILE_AND_CHUNK_SIZE\
n", cur_state->client_fd);

                const FileAndChunkSize file_and_chunk_size = {cur_state->file_size,
CHUNK_SIZE};
                                if(send(cur_state->client_fd,   &file_and_chunk_size,
sizeof(file_and_chunk_size), 0) == -1) {
                return construct_drop_connection(cur_state->client_fd);
```

```
            }
            ClientState next_state;
            next_state.tag = ClientState_SEND_FILE_CHUNK;
            next_state.value.send_file_chunk.client_fd = cur_state->client_fd;
            next_state.value.send_file_chunk.file = cur_state->file;
            return next_state;
        }
        return *generic_state;
    }
    case ClientState_SEND_FILE_CHUNK: {
        const struct ClientState_SendFileChunk* const cur_state = &generic_state->value.send_file_chunk;
        if(FD_ISSET(cur_state->client_fd, writefds)) {
            printf("Client fd %d is at ClientState_SEND_FILE_CHUNK\n", cur_state->client_fd);

            char buffer[CHUNK_SIZE];
            size_t bytes_read;
            while ((bytes_read = fread(buffer, 1, sizeof(buffer), cur_state->file)) > 0) {
                // printf("Read bytes: %lu\n", bytes_read);
                if(send(cur_state->client_fd, buffer, bytes_read, 0) == -1) {
                    return construct_drop_connection(cur_state->client_fd);
                }
            }
            fclose(cur_state->file);

            printf("Client fd %d sent file successfuly\n", cur_state->client_fd);
            return construct_drop_connection(cur_state->client_fd);
        }
        return *generic_state;
    }
```

```
        case ClientState_DROP_CONNECTION: {
            const struct ClientState_DropConnection* const cur_state = &generic_state->value.drop_connection;
            printf("Client fd %d is at ClientState_DROP_CONNECTION\n", cur_state->client_fd);

            close(cur_state->client_fd);
            ClientState next_state;
            next_state.tag = ClientStateTag_INVALID;
            return next_state;
        }
        default:
            __builtin_unreachable();
    }
}

int set_read_write_max_fds(
    fd_set *readfds,
    fd_set *writefds,
    const int serverfd,
    const ClientState* client_states,
    const size_t client_sockets_size
) {
    FD_ZERO(readfds);
    FD_ZERO(writefds);
    FD_SET(serverfd, readfds);
    int max_sd = serverfd;
    for (size_t i = 0; i < client_sockets_size; ++i) {
        const ClientState* state = &client_states[i];
        if (state->tag != ClientStateTag_INVALID) {
            const int client_fd = ClientState_client_fd(state);
```

```
            FD_SET(client_fd, readfds);
            FD_SET(client_fd, writefds);
            if (client_fd > max_sd) {
                max_sd = client_fd;
            }
        }
    }
    return max_sd;
}

static void check_accept_connection(
    const fd_set *readfds,
    const int serverfd,
    ClientState* client_states,
    const size_t client_states_size
) {
    if (FD_ISSET(serverfd, readfds)) {
        struct sockaddr_in address;
        socklen_t addr_len = sizeof(address);
        const int client_fd = accept(serverfd, (struct sockaddr *)&address, &addr_len);
        if (client_fd == -1) {
            if(errno != EAGAIN && errno != EWOULDBLOCK) {
                exit_err("accept()");
            }
            return;
        }

        printf("New connection, socket fd is %d, IP is: %s, port: %d\n",
              client_fd, inet_ntoa(address.sin_addr), ntohs(address.sin_port));

        for (size_t i = 0; i < client_states_size; ++i) {
```

```
            if (client_states[i].tag == ClientStateTag_INVALID) {
                client_states[i].tag = ClientState_RECEIVE_PROTOCOL_VERSION;
                client_states[i].value.receive_protocol_version.client_fd = client_fd;
                printf("Adding to list of sockets as %lu\n", i);
                break;
            }
        }
    }
}

static void update_sets(
    fd_set *readfds,
    fd_set *writefds,
    const int serverfd,
    const ClientState* client_sockets,
    const size_t client_sockets_size
) {
        const int max_fd = set_read_write_max_fds(readfds, writefds, serverfd,
client_sockets, client_sockets_size);
    const int activity = select(max_fd + 1, readfds, writefds, NULL, NULL);
    if (activity < 0 && errno != EINTR) {
        printf("select error");
    }
}

int main(const int argc, const char* const argv[]) {
    signal(SIGINT, handle_sigint);
    signal(SIGPIPE, SIG_IGN);
    const IterativeServerConfig config = handle_cmd_args(argc, argv);
        const int server_fd = create_and_bind_socket(config.port, config.address,
config.max_clients);
```

```
    fd_set readfds;
    fd_set writefds;
    ClientState client_sockets[config.max_clients];
    for (size_t i = 0; i < config.max_clients; ++i) {
        client_sockets[i].tag = ClientStateTag_INVALID;
    }

    while (keep_running) {
                        update_sets(&readfds, &writefds, server_fd, client_sockets,
config.max_clients);
                        check_accept_connection(&readfds, server_fd, client_sockets,
config.max_clients);

        for (size_t i = 0; i < config.max_clients; ++i) {
            ClientState* state = &client_sockets[i];
            *state = ClientState_transition(state, &readfds, &writefds, config.dir_path);
        }
    }

    close(server_fd);
    return EXIT_SUCCESS;
}

// ./lab_4/CMakeLists.txt

cmake_minimum_required(VERSION 3.10)
project(lab_4 C)

set(CMAKE_C_STANDARD 11)
```

```
set(strict_compiler_options -Wall -Wextra -Werror)

add_executable(client client.c protocol.h)
target_compile_options(client PRIVATE ${strict_compiler_options})

add_executable(iterative_server iterative_server.c protocol.h)
target_compile_options(iterative_server PRIVATE ${strict_compiler_options})

// ./lab_4/client.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <stdbool.h>
#include "protocol.h"

typedef struct {
    const char *address;
    int port;
    const char *filename;
    off_t max_file_size;
} ClientConfig;

static void print_config(const ClientConfig *config) {
    printf("Client Configuration:\n");
    printf("  Address: %s\n", config->address);
    printf("  Port: %d\n", config->port);
    printf("  Filename: %s\n", config->filename);
```

```
    printf("  Maximum file size: %lu\n", config->max_file_size);
}

static ClientConfig handle_cmd_args(const int argc, char **argv) {
    if (argc != 5) {
            fprintf(stderr, "Usage: %s <server_address> <server_port> <filename> <max_file_size>\n", argv[0]);
        exit(1);
    }

    const ClientConfig config = {
        .address = argv[1],
        .port = atoi(argv[2]),
        .filename = argv[3],
        .max_file_size = atol(argv[4])
    };

    print_config(&config);

    return config;
}

static int create_and_connect_socket(const int port, const char* const address) {
    const int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        exit_err("socket() failed");
    }

    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
```

```
    if (inet_pton(AF_INET, address, &server_addr.sin_addr) <= 0) {
        exit_err("inet_pton() failed");
    }

    if (connect(sock, (const struct sockaddr *)&server_addr, sizeof(server_addr)) == -
1) {
        exit_err("connect() failed");
    }

    return sock;
}

static void send_receive_check_protocol_version(const int sock) {
    const int protocol_version = 1;
    send(sock, &protocol_version, sizeof(protocol_version), 0);

    bool match;
    recv(sock, &match, sizeof(match), 0);

    if(!match) {
        exit_err("Protocol version mismatch");
    }
}

static void send_filename_length_max_size(const int sock, const char* const
filename, const off_t file_max_size) {
    const size_t filename_length = strlen(filename);
     const FileNameLengthMaxSize file_name_length_max_size = {filename_length,
file_max_size};
    send(sock, &file_name_length_max_size, sizeof(file_name_length_max_size), 0);
    send(sock, filename, filename_length, 0);
```

```
}

static void check_operation_possibility(const int sock) {
    enum OperationPossibility operation_possibility;
    recv(sock, &operation_possibility, sizeof(operation_possibility), 0);
    switch (operation_possibility) {
        case FILE_NOT_FOUND: {
            exit_err("File not found");
        }
        case FILE_SIZE_GREATER_THAN_MAX_SIZE: {
            exit_err("File size greater than max size");
        }
        case FAILED_TO_OPEN_FILE: {
            exit_err("Failed to open file");
        }
        case OPERATION_POSSIBLE: break;
    }
}

static FileAndChunkSize receive_file_and_chunk_size(const int sock) {
    FileAndChunkSize file_and_chunk_size;
    recv(sock, &file_and_chunk_size, sizeof(file_and_chunk_size), 0);
    return file_and_chunk_size;
}

static void receive_file(
    const int sock,
    const FileAndChunkSize file_and_chunk_size,
    const char* const filename
) {
    FILE *file = fopen(filename, "wb");
```

```
    if (!file) {
        perror("Failed to open file for writing");
        close(sock);
        exit(EXIT_FAILURE);
    }

    off_t received = 0;
    while (received < file_and_chunk_size.file_size) {
        char buffer[file_and_chunk_size.chunk_size];
        const int bytes = recv(sock, buffer, sizeof(buffer), 0);
        if (bytes <= 0) break;
        fwrite(buffer, 1, bytes, file);
        received += bytes;
    }

    fclose(file);
    close(sock);

    if (received == file_and_chunk_size.file_size) {
        printf("File received successfully.\n");
    } else {
        printf("Error: Incomplete file transfer.\n");
    }
}

int main(const int argc, char *argv[]) {
    const ClientConfig config = handle_cmd_args(argc, argv);
    const int sock = create_and_connect_socket(config.port, config.address);

    send_receive_check_protocol_version(sock);
    send_filename_length_max_size(sock, config.filename, config.max_file_size);
```

```
    check_operation_possibility(sock);
                        const FileAndChunkSize file_and_chunk_size =
receive_file_and_chunk_size(sock);
    receive_file(sock, file_and_chunk_size, config.filename);
    return EXIT_SUCCESS;
}

// ./lab_4/protocol.h

#pragma once

#include <stdint.h>

typedef struct {
    uint8_t file_name_length;
    off_t file_max_size;
} FileNameLengthMaxSize;

typedef struct {
    off_t file_size;
    size_t chunk_size;
} FileAndChunkSize;

enum OperationPossibility {
    OPERATION_POSSIBLE,
    FILE_NOT_FOUND,
    FILE_SIZE_GREATER_THAN_MAX_SIZE,
    FAILED_TO_OPEN_FILE
};

typedef struct {
```

```
    uint64_t size;
} FileSize;

static _Noreturn void exit_err(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}
```