Relatório - Trabalho 1

Arthur Alves Ferreira Melo - 00333985
Pedro Henrique Boniatti Colle - 00333916
Sofia Maciel D'avila - 00323829
Vithor Barros Pileco - 00326674

Ambiente de Testes

Sistema Operacional (versão e distribuição): Ubuntu 11.4.0-1ubuntu1~22.04
Configuração da máquina (processador(es) e memória): 4 cores Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz e 12Gb de memória RAM com 2133 MT/s
Compiladores (versões): g++ 11.4.0

Implementação

Implementação da concorrência no servidor.

O servidor possui diversas threads, sendo elas:

- Uma thread "listener" que tem como função aguardar novas conexões de usuários e assim que conectados, criar as threads que se comunicam com eles.
- A partir de cada conexão de sessão de usuário são criadas duas novas threads:
    - Uma thread "commands" que tem como função aguardar novas operações advindas do cliente e executá-las.
    - Uma thread "data" que tem como função sincronizar arquivos entre diferentes sessões.

```
/* Cria nova thread de comandos e começa o server loop nela */
/* A thread atual se mantém a espera de conexão */
if (s_commands != nullptr){
    s_commands->print_address();
    pthread_t t_commands;
    pthread_create(&t_commands, NULL, server_loop_commands, s_commands);
}
/* Cria nova thread de dados e começa o server loop nela */
if (s_data != nullptr){
    s_data->print_address();
    pthread_t t_data;
    pthread_create(&t_data, NULL, server_loop_data, s_data);
}
```

Implementação da concorrência no cliente.

Cada sessão de usuário possui duas threads associadas a ela, sendo elas:

- Uma thread “commands” que recebe operações do terminal, processa as partes *client-side* dos comandos e envia as demais partes da operação para a thread “commands” do server.
- Uma thread “data” que recebe arquivos da thread “data” do servidor que são advindos das operações realizadas em uma outra sessão do mesmo usuário.

```
/* Início da thread para leitura de dados */
pthread_t t_data;
pthread_create(&t_data, NULL, client_loop_data, socket_data.get());

/* Loop de espera de comandos */
client_loop_commands(socket_commands, serde, argv[1]);
```

Áreas do código com sincronização no acesso de dados

Nós temos no servidor uma classe compartilhada (*UserServer*) entre diferentes sessões que faz o uso de diversos mutex para sincronizar os diferentes atributos entre as 2 sessões de um usuário, permite também a troca de pacotes entre as sessões para sincronização dos diretórios:

```
int UserServer::get_session_connections_num() {
    pthread_mutex_lock(&(this->mutex_session_connection_num));
    int session_connections_ = this->session_connections;
    pthread_mutex_unlock(&(this->mutex_session_connection_num));
    return session_connections_;
}

bool UserServer::is_logged(int id) {
    pthread_mutex_lock(&(this->mutex_session_connection_num));
    for (int i = 0; i < session_ids.size(); i++){
        if (session_ids[i] == id) {
            pthread_mutex_unlock(&(this->mutex_session_connection_num));
            return true;
        }
    }
    pthread_mutex_unlock(&(this->mutex_session_connection_num));
    return false;
}

void UserServer::add_session(int id) {
    pthread_mutex_lock(&(this->mutex_session_connection_num));
    for (int i = 0; i < session_ids.size(); i++){
        if (session_ids[i] == id) {
            pthread_mutex_unlock(&(this->mutex_session_connection_num));
            return;
        }
    }
    session_connections++;
    session_ids.push_back(id);
    pthread_mutex_lock(&(this->mutex_synched_files_at_start));
    synched_files_at_start[id] = false;
    pthread_mutex_unlock(&(this->mutex_synched_files_at_start));
    pthread_mutex_unlock(&(this->mutex_session_connection_num));
}

void UserServer::remove_session(int id){
    pthread_mutex_lock(&(this->mutex_session_connection_num));
    if (session_ids.empty()){
        for (auto it = session_ids.begin(); it != session_ids.end(); it++){
            if (*it == id) session_ids.erase(it);
        }
    }
    this->session_connections--;
    data_packets_map.erase(id);
    pthread_mutex_unlock(&(this->mutex_session_connection_num));
}
```

Temos também uma sincronização por condição, também no UserServer, que serve para sincronizar as 2 threads de uma mesma sessão no servidor, só permitindo que a thread de comandos prossiga após o usuário receber os dados persistidos no servidor.

```
84
85  bool UserServer::is_ready(int id){
86      pthread_mutex_lock(&(this->mutex_synched_files_at_start));
87      bool ret = synched_files_at_start[id];
88      pthread_mutex_unlock(&(this->mutex_synched_files_at_start));
89      return ret;
90  }
91
92  void UserServer::set_ready(int id){
93      pthread_mutex_lock(&(this->mutex_synched_files_at_start));
94      synched_files_at_start[id] = true;
95      pthread_mutex_unlock(&(this->mutex_synched_files_at_start));
96  }
```

Por fim temos uma sincronização entre as 2 threads de uma mesma sessão no servidor para a criação do diretório do usuário no sistema de arquivos do servidor.

```
45
46      pthread_mutex_lock(&mutex_check_directory_exists);
47      // Create directory if it doesn't exist
48      if (!std::filesystem::exists(userfolder)) {
49          std::filesystem::create_directory(userfolder);
50      }
51      pthread_mutex_unlock(&mutex_check_directory_exists);
52
```

Principais estruturas e funções

Além da classe do *UserServer* que serve para compartilhamento de pacotes e informações entre 2 sessões e da classe payload que serve para a comunicação entre o cliente e servidor, e a classe *Socket*, responsável pela comunicação e gerenciamento do sistema de arquivo, temos também:

- *Handle_events*: Que configura a biblioteca *iNotify* para notificar mudanças no diretório sincronizar.
- *Server_loop_commands* e *client_loop_commands*: Onde as 2 threads de comandos executam a versão do servidor responsável por responder os payload emitido pela thread do *client*.
- *Server_loop_data* e *client_loop_data:* Onde as 2 threads de dados executam, a versão do servidor é responsável por enviar mudanças que tenham ocorrido no diretório sincronizado, e a versão do cliente é responsável por receber as mudanças e aplicá-las no diretório sincronizado.

Uso das primitivas de comunicação

O trabalho se utiliza da API de sockets do Linux, com uma abstração de classe sobre ela. Fazendo classes de socket de servidor e cliente que convertem para uma socket real, usando o build pattern.

```
class Socket{
    public:

        /*...*/

        void send_checked(const void *buf, const int len);
        void send_checked(flatbuffers::FlatBufferBuilder *buff);
        uint8_t* read_full_pckt();
```

Os pacotes de comunicação são feitos com um sistema simples de polimorfismo. Cada pacote enviado é definido partindo de uma classe Payload base. Ela tem 3 métodos, send, reply, await_response. As funções send e reply são virtuais puras, e await_response é apenas virtual, tendo um comportamento base. Nelas o pacote é definido como é enviado, como é respondido e o que deve fazer quando ele receber a resposta.

```
class Payload {
    public:
        Payload(Net::Operation op): operation_type(op){}
        inline Net::Operation get_type(){ return operation_type; }

        virtual void send(Serializer& serde, std::shared_ptr<Socket> socket) = 0;
        virtual void reply(Serializer& serde, std::shared_ptr<Socket> socket) = 0;
        //comportamento default de await_response é esperar um ok, se n da err
        virtual void await_response(Serializer& serde, std::shared_ptr<Socket> socket);
    private:
        const Net::Operation operation_type;
};
```

Por exemplo, o pacote de download:

- Send: envia o pacote com o nome do arquivo
- Reply: envia o arquivo via pacotes
- Await_response: recebe os pacotes e finaliza o arquivo.

```cpp
class Download : public Payload {
    public:
        //dir_name is the username or 'sync_dir'
        Download(const char* filename);

        //sends the name of the file to be downloaded
        void send(Serializer& serde, std::shared_ptr<Socket> socket);
        //opens the file (if it has), and sends the meta + chunks of data
        void reply(Serializer& serde, std::shared_ptr<Socket> socket);
        //awaits for the file and saves it
        void await_response(Serializer& serde, std::shared_ptr<Socket> socket) override;
    private:
        const std::string filename;
        SyncFile file;
        uint64_t size;
};
```

Para fazer a serialização, está se utilizando da biblioteca [flatbuffers](). Ela é uma biblioteca bem simples em que se define a estrutura dos pacotes em um arquivo .fbs e ele o compila em um arquivo c++.

```
table Packet {
    //the size of the packet is set with with prefix size and retrived with such
    //that is to see if it sent it whole packet, that it did not fragment
    op: Operation;
}
table Ping {}

enum ChannelType: ubyte { Main, FileWatch, Relay }

table Connect {
    id: ulong;
    type: ChannelType;
    username: string;
    //TODO: add type:
    // - main -> for client cmd req to server reply
    // - side -> for client file watch req to server reply
    // - sync -> for server relay file update to clinet reply
}
```

Problemas de Implementação

Definir como fazer a parte de sincronização de diretório foi uma das tarefas mais desafiadoras, inicialmente pensamos em fazer 3 threads por sessão e então percebemos que apenas 2 bastavam. Então fazê-las funcionarem de forma

sincronizada tornou-se uma tarefa um pouco trabalhosa. Tivemos alguns deadlocks e data racings, que foram resolvidos com auxílio do GDB e muita paciência.