UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANDREI POCHMANN KOENICH

ANDREI ROCHA BERETA

PEDRO COMPANY BECK

TURMA A

RELATÓRIO - TRABALHO PRÁTICO PARTE 1

Disciplina: Sistemas Operacionais II

Professor: Alberto Egon Schaeffer Filho

Porto Alegre, dezembro de 2023.

1 INTRODUÇÃO

O relatório a seguir tem por objetivo fornecer dados a respeito da implementação em C++ de uma aplicação cliente-servidor, utilizada para fornecer um serviço de compartilhamento e sincronização automática de arquivos entre diferentes dispositivos de um mesmo usuário. Nessa aplicação, o servidor deve ser capaz de gerenciar arquivos de diversos usuários remotos. O cliente, por sua vez, corresponde à parte da aplicação presente na máquina dos usuários, que permite ao usuário acessar remotamente seus arquivos mantidos pelo servidor.

A aplicação desenvolvida utiliza o mecanismo de *sockets* para o estabelecimento de comunicação entre o cliente e o servidor, e possui suporte às seguintes funcionalidades:

- Múltiplos usuários: o servidor deve ser capaz de tratar requisições simultâneas de vários usuários.

- Múltiplas sessões: um usuário deve poder utilizar o serviço através de até dois dispositivos distintos simultaneamente.

- Consistência nas estruturas de armazenamento: as estruturas de armazenamento de dados no servidor devem ser mantidas em um estado consistente e protegidas de acessos concorrentes.

- Sincronização: cada vez que um usuário modificar um arquivo contido no diretório local do seu dispositivo, o arquivo deverá ser atualizado no servidor e nos demais diretórios locais sincronizados, dos demais dispositivos daquele usuário. Isso é realizado por meio da API *inotify.*

- Persistência de dados no servidor: diretórios e arquivos de usuários devem ser restabelecidos quando o servidor for reiniciado.

Para fins de simplificação, assume-se que os diretórios dos clientes que fazem uso da aplicação não possuem nenhuma estrutura hierárquica, ou seja, não existem outros diretórios dentro do diretório principal do usuário.

2 DESCRIÇÃO DO AMBIENTE DE TESTES

Abaixo, segue a descrição do ambiente de testes utilizado por cada um dos integrantes do grupo.

2.1 Ambiente de testes do aluno Andrei Pochmann Koenich

Versão do Sistema Operacional: Linux Ubuntu 22.04.3 LTS.

Configurações da máquina: Processador Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, com memória RAM de 16GB.

Compilador utilizado: GCC 12.3.0.

2.2 Ambiente de testes do aluno Andrei Rocha Bereta

Versão do Sistema Operacional: Linux Ubuntu 22.04.3 LTS, sobre o WSL2 no Windows 10.

Configurações da máquina: Processador AMD RYZEN 7 5700X, com memória RAM de 16GB.

Compilador utilizado: GCC 11.4.0.

2.3 Ambiente de testes do aluno Pedro Company Beck

Versão do Sistema Operacional: Linux Ubuntu 20.04.3 LTS, sobre o WSL2 no Windows 10.

Configurações da máquina: Processador Intel Core i5 8350U, com memória RAM de 16GB.

Compilador utilizado: GCC 9.4.0.

3 DESCRIÇÃO GERAL DAS IMPLEMENTAÇÕES

3.1 DESCRIÇÃO GERAL DA IMPLEMENTAÇÃO DA APLICAÇÃO CLIENTE

A aplicação cliente faz uso de três *threads* principais: *thread* principal, *thread* de evento e *thread* de leitura. A *thread* principal espera por comandos digitados pelo usuário. A *thread* de eventos observa o diretório local do usuário e envia notificações dos eventos ocorridos (alteração, renomeação ou deleção de arquivos) nesse diretório para o servidor. A *thread* de leitura é utilizada para o recebimento pacotes do servidor.

Abaixo, seguem as descrições dos comandos utilizados na interface da aplicação cliente, para fazer uso das funcionalidades do sistema.

Tabela 1 - Descrição dos comandos da aplicação cliente

Comando	Descrição
`upload <path/filename.ext>`	Envia o arquivo <filename.ext> para o servidor, colocando-o no repositório remoto do servidor e propagando-o para todos os dispositivos daquele usuário.
`download <path/filename.ext>`	Faz uma cópia não sincronizada do arquivo <filename.ext> do servidor para o diretório local (de onde o servidor foi chamado).
`delete <filename.ext>`	Exclui o arquivo <filename.ext> do diretório local do cliente.
`list_server`	Lista os arquivos salvos no diretório remoto do servidor, associados ao cliente.
`list_client`	Lista os arquivos salvos no repositório local do cliente.
`help`	Exibe um menu de ajuda na interface, mostrando cada comando junto à sua função.
`exit`	Fecha a sessão com o servidor.

3.2 DESCRIÇÃO GERAL DA IMPLEMENTAÇÃO DA APLICAÇÃO SERVIDOR

Assume-se que cada usuário poderá conectar, simultaneamente ao servidor, no máximo dois dispositivos. Cada dispositivo possui um número identificador para diferenciá-los. Eventos gerados por um dispositivo não serão reenviados para esse mesmo dispositivo (ou seja, existe um controle baseado no número identificador do dispositivo).

A leitura e modificação da lista de arquivos em memória é protegida por uma `mutex_lock` (para cada usuário, dispositivos a compartilham).

A leitura e modificação da lista de dispositivos é protegida por uma `mutex_lock` (para cada usuário, dispositivos a compartilham).

A leitura e modificação da estrutura `map` de usuários é protegida por uma `mutex_lock`.

4 CONCORRÊNCIA NO SERVIDOR PARA ATENDIMENTO A MÚLTIPLOS CLIENTES

Para garantir a concorrência no servidor, com atendimento a múltiplos clientes, utilizou-se a primitiva `pthread_create()`, localizada em *Servidor/myServer.cpp:121*. Com essa primitiva, cria-se uma nova *thread* para lidar com cada conexão estabelecida com o servidor por um cliente, logo após o recebimento de uma conexão por meio da primitiva `accept()`, localizada em *Servidor/myServer.cpp:112*.

5 ÁREAS DO CÓDIGO COM GARANTIA DE SINCRONIZAÇÃO PARA ACESSO A DADOS

A garantia de sincronização ocorre com o uso de mutexes, com as primitivas `pthread_mutex_lock()` e `pthread_mutex_unlock()`. No cliente, também foram utilizadas as primitivas `pthread_cond_wait()` e `pthread_cond_signal()`. A seguir, estão descritos os trechos do projeto nos quais houve a necessidade de utilizar a garantia de sincronização com essas primitivas.

5.1 Sincronizações usando mutexes

5.1.1 Sincronizações existentes na aplicação servidor

***Servidor/deviceManager.cpp:203*:** seção crítica utilizada para garantir a propagação, pelo servidor, de um evento de modificação, deleção ou renomeação de arquivo para os diretórios dos demais dispositivos conectados, que correspondam ao usuário proprietário do arquivo modificado. Dessa forma, é possível garantir que a mudança em um diretório seja refletida em todos os diretórios dos dispositivos conectados, do usuário em questão.

***Servidor/deviceManager.cpp:203*:** seção crítica na qual um socket será utilizado para escrita. Com essa seção crítica, é possível evitar que outros pacotes sejam enviados.

***Servidor/deviceManager.cpp:242*:** seção crítica na qual a `hash map` de usuários será acessada e possivelmente alterada.

***Servidor/deviceManager.cpp:267*:** seção crítica para garantir que os dispositivos do usuário não sejam ser alterados durante a verificação de quantos dispositivos existem e durante a possível adição de um novo dispositivo.

***Servidor/deviceManager.cpp:309*:** seção crítica utilizada para acessar informações de um usuário. Com essa seção crítica, podemos evitar alterações durante esse acesso.

***Servidor/deviceManager.cpp:323*:** seção crítica para impedir a alteração da lista de dispositivos enquanto um dispositivo está sendo removido da lista.

***Servidor/deviceManager.cpp:356*:** seção crítica para impedir que o `map` de usuários seja modificado enquanto está desconectando, de forma forçada, todos os dispositivos.

Servidor/serverLoop.cpp:25: seção crítica utilizada para garantir a deleção de um determinado arquivo, do diretório de um usuário. Posteriormente, esse evento de deleção será propagado nos diretórios de todos os demais dispositivos conectados e correspondentes ao usuário em questão (o arquivo será deletado em todos esses diretórios).

Servidor/serverLoop.cpp:46: seção crítica utilizada para garantir a criação de um determinado arquivo, no diretório de um usuário. Posteriormente, esse evento de criação será propagado nos diretórios de todos os demais dispositivos correspondentes ao usuário em questão (todos os diretórios receberão o arquivo criado).

Servidor/serverLoop.cpp:70: seção crítica utilizada para garantir a modificação de um determinado arquivo, no diretório de um usuário. Posteriormente, esse evento de modificação será propagado nos diretórios de todos os demais dispositivos conectados e correspondentes ao usuário em questão (a mudança de arquivo ocorrerá em todos esses diretórios).

Servidor/serverLoop.cpp:86: seção crítica utilizada para garantir a renomeação de um determinado arquivo, do diretório de um usuário. Posteriormente, esse evento de renomeação será propagado nos diretórios de todos os demais dispositivos conectados e correspondentes ao usuário em questão (a renomeação do arquivo em questão será realizada em todos esses diretórios).

Servidor/serverLoop.cpp:115: seção crítica utilizada para propagar um evento de modificação de diretório (deleção, renomeação, movimentação ou criação de arquivo) para todos os demais dispositivos conectados, correspondentes ao usuário proprietário do diretório que sofreu a modificação, nos casos em que isso é necessário.

Servidor/serverLoop.cpp:152: seção crítica para impedir que um arquivo seja alterado, durante o seu envio.

Servidor/serverLoop.cpp:153: seção crítica para impedir que eventos sejam enviados para o dispositivo, enquanto o conteúdo de um arquivo está sendo enviado.

Servidor/serverLoop.cpp:163: seção crítica para impedir que a estrutura contendo as informações de listagem dos arquivos seja alterada enquanto está sendo enviada.

Servidor/serverLoop.cpp:164: seção crítica para garantir que a listagem dos arquivos seja enviada do início ao fim.

Servidor/serverLoop.cpp:176: seção crítica para garantir a leitura de um arquivo enviado pelo usuário por meio da operação de upload, para que esse arquivo seja salvo no diretório do servidor, correspondente ao usuário que realizou o envio.

5.1.2 Sincronizações existentes na aplicação cliente

Cliente/comunicacaoCliente.cpp:174: seção crítica utilizada para enviar um pacote, para pedir ao servidor a listagem de todos os arquivos por ele armazenados, correspondentes ao cliente que fez a solicitação dessa listagem.

Cliente/comunicacaoCliente.cpp:188: seção crítica na qual, após o servidor ter enviado, por meio de um pacote, a listagem de todos os arquivos por ele armazenados, as informações de arquivos existentes no pacote sejam lidas, para serem exibidas.

Cliente/eventThread.cpp:182: seção crítica utilizada para verificar qual foi o último evento ocorrido, para determinar se de fato é necessário notificar o servidor a respeito.

Cliente/eventThread.cpp:202: seção crítica para garantir que não ocorra um loop de notificações de eventos, conforme explicado de forma detalhada na seção 9.1 deste relatório. Isso garante que o usuário não envie, para o servidor, um mesmo evento já lido por ele anteriormente. No caso dessa seção crítica, ignora-se os dados do último evento em questão. A seção crítica garante que não ocorram mudanças simultâneas no pacote de eventos relacionado.

Cliente/eventThread.cpp:209: seção crítica para garantir que o envio das informações relacionadas à ocorrência de um evento, embutidas em um pacote, ocorra do início ao fim, sem interferências.

Cliente/readThread.cpp:71: seção crítica para garantir que não ocorra um loop de notificações de eventos, conforme explicado de forma detalhada na seção 9.1 deste relatório. Isso garante que o usuário não envie, para o servidor, um mesmo evento já lido por ele anteriormente. No caso dessa seção crítica, armazena-se os dados do último evento em questão. A seção crítica garante que não ocorram mudanças simultâneas no pacote de eventos relacionado.

Cliente/readThread.cpp:154: seção crítica na qual o campo `files_list` de `dados_conexao` (que representa a última listagem de arquivos recebida) será alterado. A *thread* atribuirá a lista de arquivos ao campo, e então irá liberar a *lock* e sinalizar `file_list_cond` (mais informações em na seção 5.2).

5.1.3 Sincronizações existentes na aplicação servidor e na aplicação cliente

Common/package_functions.cpp:283: seção crítica utilizada para fins de depuração do sistema implementado, relacionada com a impressão de dados de pacotes no terminal, para verificar se o trânsito de pacotes está ocorrendo de forma adequada. Garante que não exista mais de uma *thread* exibindo informações de pacotes diferentes no terminal, ao mesmo tempo.

5.2 Sincronizações usando signal wait

Na aplicação cliente, a leitura do socket é feita exclusivamente por uma única *thread*, com o ponto de início da leitura sendo a função `readThread()`, presente em *Cliente/readThread.cpp:134*. O cliente pode, a partir da interface, requisitar a listagem de arquivos do servidor. A *thread* responsável por exibir os arquivos irá invocar `pthread_cond_wait()`, e esperar a sinalização que será feita pela *thread* de leitura, ao terminar de ler a listagem de arquivos. O valor dessa sinalização é atribuído à uma variável global e, em seguida, ocorre a chamada da primitiva `pthread_cond_signal()`.

***Cliente/comunicacaoCliente.cpp:196*:** espera pela sinalização de que a leitura da listagem de arquivos foi concluída (irá ficar bloqueado até a sinalização chegar). Essa sinalização será gerada pela *thread* `readThread`, após a leitura da listagem, e irá atribuir o resultado em `file_list`. A variável `is_file_list_readed` é alterada para `true` por `readThread` para indicar que a leitura foi concluída. Essa variável é utilizada como cláusula de teste em uma iteração `while`.

***Cliente/readThread.cpp:162*:** sinaliza para a *thread* da interface (a *thread* será acordada) de que a leitura dos pacotes da listagem de arquivos do servidor foi concluída.

6 PRIMITIVAS DE COMUNICAÇÃO UTILIZADAS

As primitivas de comunicação utilizadas consistem, exclusivamente, na utilização de sockets. Todas as operações de leitura e escrita envolvendo pacotes são feitas pelas funções `read_package_from_socket()` (localizada em *Common/package_functions.cpp:100*) e `write_package_to_socket()` (localizada em *Common/package_functions.cpp:237*).

Antes de serem enviados, o conteúdo dos pacotes é convertido para o formato *big-endian.* Ao serem recebidos por um destinatário (cliente ou servidor), o conteúdo é convertido para a representação local, utilizada no sistema do destinatário. Todos os campos dos pacotes são declarados com utilização da primitiva `alignas()`, dessa forma garantindo um *padding* bem definido e simples de calcular.

Abaixo, seguem as descrições e as localizações das primitivas da API relacionada com sockets, utilizadas nas aplicações cliente e servidor.

Tabela 2 - Primitivas de Sockets Utilizadas na Aplicação Cliente

Primitiva	Localização	Descrição
`socket()`	Cliente/comunicacaoCliente.cpp:37	Usado para criar um novo ponto de extremidade de conexão (socket).
`connect()`	Cliente/comunicacaoCliente.cpp:48	Estabelece uma conexão com o servidor remoto.
`write()`	Common/packageFunctions.cpp:217	Realiza o envio de dados, por meio de um socket conectado.
`read()`	Common/packageFunctions.cpp:34	Realiza a leitura de dados, por meio de um socket conectado.
`close()`	Cliente/MyClient.cpp:50	Encerra uma conexão de socket.

Tabela 3 - Primitivas de Sockets Utilizadas na Aplicação Servidor		
Primitiva	**Localização**	**Descrição**
`socket()`	Servidor/myServer.cpp:84	Usado para criar um novo ponto de extremidade de conexão (socket).
`bind()`	Servidor/myServer.cpp:95	Estabelece uma conexão com o servidor remoto.
`listen()`	Servidor/myServer.cpp:101	Configura o socket do servidor para aceitar conexões de entrada.
`accept()`	Servidor/myServer.cpp:112	Aceita uma conexão de entrada de um cliente.
`read()`	Common/packageFunctions .cpp:34	Realiza a leitura de dados, por meio de um socket conectado.
`write()`	Common/packageFunctions. .cpp:217	Realiza o envio de dados, por meio de um socket conectado.
`close()`	Servidor/myServer .cpp:124 (socket inicial) Servidor/deviceManager .cpp:35 (socket do dispostivo)	Encerra uma conexão de socket.

7 COMUNICAÇÃO ENTRE CLIENTE E SERVIDOR COM PACOTES

A comunicação é iniciada com a identificação do usuário. Um usuário deve informar seu nome, e o servidor irá responder com o ID do dispositivo no caso de sucesso, ou rejeitará a conexão nos casos em que já existem dois dispositivos conectados, correspondentes ao usuário estabelecendo a conexão. As definições das estruturas dos pacotes estão localizadas em *Common/package.hpp*.

A tabela a seguir descreve quais são os diferentes pacotes envolvidos nas diferentes interações possíveis entre o cliente e o servidor.

Tabela 4 - Pacotes Enviados na Aplicação Cliente para o Servidor	
Tipo do Pacote	**Descrição**
PackageUserIndentification	Identificação inicial do usuário, deverá conter seu nome e indicar tipo de conexão como principal.
PackageRequestFileList	Usuário deseja receber a lista de arquivos presentes no servidor.
PackageChangeEvent FILE_DELETED	Usuário removeu um arquivo do seu diretório local.
PackageChangeEvent FILE_CREATED	Usuário criou um arquivo no seu diretório local.
PackageChangeEvent FILE_MODIFIED	Usuário modificou um arquivo no seu diretório local.
PackageFileNotFound	Usuário enviaria o arquivo modificado, mas não foi possível acessá-lo.
PackageUploadFile	Usuário enviará o conteúdo do arquivo modificado.
PackageFileContent	Envio do conteúdo do arquivo modificado.
PackageChangeEvent FILE_RENAME	Usuário renomeu um arquivo no seu diretório local.

Tabela 5 - Pacotes Enviados na Aplicação Servidor para o Cliente	
Tipo do Pacote	**Descrição**
PackageUser IndentificationResponse	Indica se conexão foi aceita ou rejeitada, caso aceita deverá conter o ID do dispositivo.
PackageFileList	Item da lista de arquivos presentes no servidor.
PackageChangeEvent FILE_DELETED	Outro dispositivo removeu um arquivo do seu diretório local.
PackageChangeEvent FILE_CREATED	Outro dispositivo criou um arquivo no seu diretório local.
PackageChangeEvent FILE_MODIFIED	Outro dispositivo modificou um arquivo no seu diretório local.
PackageFileNotFound	O arquivo seria enviado, mas não foi possível acessá-lo.
PackageUploadFile	Conteúdo do arquivo modificado será enviado.
PackageFileContent	Conteúdo do arquivo modificado.
PackageChangeEvent FILE_RENAME	Outro dispositivo renomeou um arquivo no seu diretório local.

8 DESCRIÇÃO DAS PRINCIPAIS ESTRUTURAS E FUNÇÕES

8.1 Principais Estruturas da Aplicação Cliente

struct DadosConexao

Localizada em *Cliente/DadosConexao.hpp: 17.* Estrutura utilizada para armazenar dados importantes a respeito da conexão estabelecida entre um cliente e o servidor.

Tabela 6 – Componentes da Estrutura DadosConexao	
Componente	**Descrição**
`char nome_usuario[DIMENSAO_NOME_USUARIO]`	String utilizada para armazenar o nome do usuário que estabeleceu conexão com servidor.
`char endereco_ip[DIMENSAO_GERAL]`	String utilizada para armazenar o endereço IP do servidor com o qual o usuário está conectado.
`char numero_porta[DIMENSAO_GERAL]`	Armazena o número da porta utilizada para estabelecer a conexão.
`char comando[DIMENSAO_COMANDO]`	Armazena o último comando escolhido pelo usuário.
`int socket`	Identificador do socket a ser utilizado para a comunicação.
`pthread_mutex_t *socket_lock`	*Lock* a ser adquirida antes de enviar pacotes para o servidor.
`pthread_cond_t *file_list_cond`	Condição usada para esperar leitura da lista de arquivos, que deve ser feita pela *thread* de leitura.
`pthread_mutex_t *file_list_lock`	*Lock* utilizado na seção crítica para alterar a lista de arquivos.
`bool is_file_list_readed`	Variável utilizada para indicar se toda a lista de arquivos já foi lida.
`std::vector<File> file_list`	Vetor contendo a última listagem de arquivos recebida do servidor.
`std::optional<pthread_t> sync_thread`	*Thread* de sincronização (lê pacotes do servidor). Caso haja valor, o mesmo será usado para cancelar a *thread* após o comando *exit* ou recebimento do sinal SIGINT.
`std::optional<pthread_t> event_thread`	*Thread* de eventos (monitora o diretório local com *inotify*), caso haja valor o mesmo será usado para cancelar a *thread* após o comando *exit* ou recebimento do sinal SIGINT.
`uint8_t deviceID`	Valor identificador do dispositivo do usuário, que deverá ser desconectado.

struct UserChangeEvent

Localizada em *Cliente/eventThread.hpp: 10.* Estrutura contendo informações a respeito dos eventos gerados pelo *inotify* para o servidor.

Tabela 7 – Componentes da Estrutura UserChangeEvent	
Componente	**Descrição**
`ChangeEvents changeEvent`	Evento de alteração, podendo ser: FILE_DELETED, FILE_CREATED, FILE_MODIFIED, e FILE_RENAMED.
`std::string movedFrom`	Nome do arquivo (para eventos DELETED, CREATED e MODIFIED) ou nome do arquivo original (para evento RENAMED).
`std::string movedTo`	Nome do arquivo final (para evento RENAMED), sendo uma string vazia em outros eventos.
`uint32_t cookie`	Cookie do evento gerado pelo inotify, usado para combinar os eventos IN_MOVED_FROM e IN_MOVED_TO, para então transformá-lo no evento FILE_RENAMED, inicializado com 0 para outros eventos do *inotify*.

8.2 Principais Estruturas da Aplicação Servidor

struct Device

Localizada em *Servidor/deviceManager.hpp: 32.* Contém informações a respeito de um dispositivo conectado, correspondente a um determinado usuário.

Tabela 8 – Componentes da Estrutura Device	
Componente	**Descrição**
`int socket`	Identificador do socket a ser utilizado para a comunicação.
`uint8_t deviceID`	Identificador do dispositivo a ser utilizado para a comunicação.
`pthread_mutex_t *socket_lock`	*Lock* que deve ser adquirida antes de enviar pacotes para o usuário.
`void close_sockets(void)`	Método para encerrar a conexão dos sockets relacionados ao dispositivo.

class DeviceManager

Localizada em *Servidor/deviceManager.hpp: 88.* Essa classe é utilizada para gerenciar o funcionamento de um determinado dispositivo de um usuário, que está realizando uma conexão com o servidor.

Tabela 10 – Componentes da Classe DeviceManager	
Componente	**Descrição**
`std::map<std::string, User *> usuarios;`	Map onde a chave é o nome do usuário e o valor o ponteiro para a estrutura do usuário (será dinamicamente alocada).
`pthread_mutex_t usuarios_lock`	*Lock* para a seção crítica envolvendo usuarios.
`std::optional<DeviceConnectReturn> connect(int socket_id, std::string &user)`	Método utilizado para conectar uma *thread* com um dispositivo de entrada de um uusário.
`void disconnect(std::string &user, uint8_t id)`	Método utilizado para desconectar um determinado dispositivo do usuário.socket_id
`void disconnect_all(void)`	Desconecta forçosamente todos os dispositivos de todos os usuários.

struct User

Localizada em *Servidor/deviceManager.hpp: 49.* Consiste na representação de um usuário, com informações a respeito dos seus arquivos e dispositivos. Todos os métodos assumem que as *locks* já tenham sido obtidas.

Tabela 9 – Componentes da Estrutura User	
Componente	**Descrição**
`pthread_mutex_t *devices_lock`	*Lock* utilizado para alterar os na seção crítica para alterar osdispositivos conectados.
`pthread_mutex_t *files_lock`	*Lock* utilizado na seção crítica para alterar os arquivos do usuário.
`std::vector<File> *files`	Vetor contendo as informações dos arquivos correspondentes ao usuário.
`std::vector<Device *> *devices`	Vetor contendo as informações dos dipositivos correspondentes ao usuário.
`PackageChangeEvent previousPackageChangeEvent`	Corresponde ao pacote referente ao último evento de modificação ocorrido, para evitar um loop de propagação de eventos.
`char username[USER_NAME_MAX_LENGTH];`	String contendo o nome do cliente.
`int init(const char *username)`	Método utilizado para inicializar os campos referentes aos arquivos e ao nome do usuário.
`std::optional<File> get_file(const char filename[NAME_MAX])`	Método utilizado para obter um determinado arquivo do usuário, com base no nome do arquivo.
`void create_file(const char filename[NAME_MAX])`	Método utilizado para criar um novo arquivo correspondente ao usuário.
`void rename_file(const char old_filename[NAME_MAX], const char new_filename[NAME_MAX])`	Método utilizado para renomear um arquivo correspondente ao usuário.
`void remove_file(const char`	Método utilizado para remover um arquivo

`filename[NAME_MAX])`	correspondente ao usuário.
`void add_file_or_replace(File file)`	Método utilizado para adicionar um novo arquivo no conjunto de arquivos do usuário, ou substituir um arquivo já existente.
`void update_file_info(const char *path, const char filename[NAME_MAX])`	Método utilizado para atualizar informações de um arquivo já existente. Esse método chamará o método `add_file_or_replace(File file)`.
`void propagate_event(PackageChangeEvent packageChangeEvent)`	Método utilizado para propagar um determinado evento para os demais dispositivos do usuário, que também estejam conectados.

struct ServerThreadArg

Localizada em *Servidor/ServerThreadArg.hpp: 9.* Esta estrutura é utilizada para armazenar o número identificador do socket que será utilizado pela *thread*, para o estabelecimento da conexão.

Tabela 11 – Componente da Estrutura ServerThreadArg	
Componente	**Descrição**
`int socket_id`	Identificador do socket a ser utilizado para a comunicação.

8.3 Principais Funções da Aplicação Servidor

void *serverThread(void *arg)

Localizada em *Servidor/serverThread.cpp:84*. Esta função é acionada após a criação de uma *thread* para tratar a conexão de um cliente com o servidor, para que o cliente possa fazer uso das funcionalidades do programa.

Tabela 11 – Parâmetro da Função serverThread	
Parâmetro	**Descrição**
`void *arg`	Estrutura utilizada para armazenar o número identificador do socket que será utilizado pela *thread*, para o estabelecimento da conexão.

void serverLoop(int socket_id, int tid, std::string &username, User *&user, Device *&device)

Localizada em *Servidor/serverLoop.cpp:122.* Esta função consiste em um laço utilizado para tratar as requisições e as respostas envolvendo o cliente e o servidor, para um determinado dispositivo do cliente que esteja conectado.

Tabela 12 – Parâmetros da Função serverLoop	
Parâmetro	**Descrição**
`int socket_id`	Identificador do socket a ser utilizado para a comunicação.
`int tid`	Identificador da *thread* sendo utilizada para tratar a comunicação entre o cliente e o servidor.
`string &username`	Nome do usuário (cliente) envolvido na comunicação.
`User *&user,`	Contém informações úteis a respeito do cliente realizando a interação com o servidor.
`Device *&device`	Contém informações úteis a respeito do dispositivo do cliente, utilizado para realizar a interação com o servidor.

void User::propagate_event(PackageChangeEvent packageChangeEvent)

Localizada em *Servidor/deviceManager.cpp:179.* Esta função é utilizada para propagar um determinado evento ocorrido no diretório do cliente, para os diretórios do mesmo cliente (dos seus outros dispositivos conectados com o servidor), no momento da ocorrência do evento.

Tabela 13 – Parâmetros da Função propagateEvent	
Parâmetro	**Descrição**
`PackageChangeEvent packageChangeEvent`	Estrutura contendo informações a respeito de um determinado evento ocorrido no diretório do cliente, que serão enviadas em formato de pacote para notificação.

std::optional<DeviceConnectReturn> DeviceManager::connect(int socket_id, std::string &user)

Localizada em *Servidor/deviceManager.cpp:237.* Esta função é utilizada para realizar uma conexão lógica entre a *thread* responsável por atender as requisições de um dispositivo de um determinado cliente, e o dispositivo em si.

Tabela 14 – Parâmetros da Função connect	
Parâmetro	**Descrição**
`int socket_id`	Identificador do socket a ser utilizado para a comunicação.
`string &user`	String correspondendo ao nome do cliente, proprietário do dispositivo em questão.

void DeviceManager::disconnect(std::string &user, uint8_t id)

Localizada em *Servidor/deviceManager.cpp:306.* Esta função é utilizada para desconectar um determinado dispositivo de um usuário.

Tabela 15 – Parâmetros da Função disconnect	
Parâmetro	**Descrição**
`string &user`	String correspondendo ao nome do cliente, proprietário do dispositivo em questão.
`uint8_t id`	Valor identificador do dispositivo do usuário, que deverá ser desconectado.

8.4 Principais Funções da Aplicação Cliente

void *readThread (void *)

Localizada em *Cliente/readThread.cpp:139.* Esta função é acionada por meio da *thread* utilizada para ler pacote enviados do servidor para o cliente.

std::optional<int> conecta_servidor (DadosConexao &dados_conexao)

Localizada em *Cliente/comunicacaoCliente.cpp:22*. Esta função é utilizada para garantir a comunicação, por parte do cliente, com o servidor. Para isso, utiliza-se a primitiva `connect()`.

Tabela 16 – Parâmetro da Função disconnect	
Parâmetro	**Descrição**
`DadosConexao &dados_conexao`	Estrutura contendo dados relevantes sobre a conexão estabelecida entre o cliente e o servidor, tais como nome do cliente, endereço IP, número da porta usada, dentre outros.

void upload (DadosConexao &dados_conexao)

Localizada em *Cliente/comunicacaoCliente.cpp:104.* Esta função é utilizada pelo cliente para enviar um determinado arquivo ao servidor. Uma vez que esse arquivo é recebido pelo servidor, ele é propagado para os diretórios de todos os demais dispositivos conectados, que correspondem ao cliente que realizou a operação de upload.

Tabela 17 – Parâmetro da Função upload	
Parâmetro	**Descrição**
`DadosConexao &dados_conexao`	Estrutura contendo dados relevantes sobre a conexão estabelecida entre o cliente e o servidor, tais como nome do cliente, endereço IP, número da porta usada, dentre outros.

void download (DadosConexao &dados_conexao)

Localizada em *Cliente/comunicacaoCliente.cpp:127.* Esta função é utilizada pelo cliente para fazer o download (cópia não sincronizada) de um determinado arquivo presente no servidor, para o diretório local do usuário, de onde a aplicação cliente foi chamada.

Tabela 18 – Parâmetro da Função download	
Parâmetro	**Descrição**
`DadosConexao &dados_conexao`	Estrutura contendo dados relevantes sobre a conexão estabelecida entre o cliente e o servidor, tais como nome do cliente, endereço IP, número da porta usada, dentre outros.

void delete_cmd (DadosConexao &dados_conexao)

Localizada em *Cliente/comunicacaoCliente.cpp:148*. Esta função é utilizada pelo cliente para excluir um determinado arquivo do seu diretório local e sincronizado. Após essa deleção, os diretórios dos demais dispositivos do usuário, que estejam conectados, também sofrerão essa mesma deleção.

Tabela 19 – Parâmetro da Função delete_cmd	
Parâmetro	**Descrição**
`DadosConexao &dados_conexao`	Estrutura contendo dados relevantes sobre a conexão estabelecida entre o cliente e o servidor, tais como nome do cliente, endereço IP, número da porta usada, dentre outros.

void list_server (DadosConexao &dados_conexao)

Localizada em *Cliente/comunicacaoCliente.cpp:168.* Esta função é utilizada pelo cliente para solicitar uma listagem de todos os arquivos contidos no repositório remoto que corresponde a ele. Além dos nomes dos arquivos, também ocorre a listagem dos MAC times referentes a cada arquivo: *modification time (mtime), access time (atime) e change or creation time (ctime)*.

Tabela 20 – Parâmetro da Função list_server	
Parâmetro	**Descrição**
`DadosConexao &dados_conexao`	Estrutura contendo dados relevantes sobre a conexão estabelecida entre o cliente e o servidor, tais como nome do cliente, endereço IP, número da porta usada, dentre outros.

void list_client (DadosConexao &dados_conexao)

Localizada *Cliente/comunicacaoCliente.cpp:207.* Esta função é utilizada pelo cliente para solicitar uma listagem de todos os arquivos contidos no repositório local que corresponde a ele. Além dos nomes dos arquivos, também ocorre a listagem dos MAC times referentes a cada arquivo: *modification time (mtime), access time (atime) e change or creation time (ctime)*.

Tabela 21 – Parâmetro da Função list_client	
Parâmetro	**Descrição**
`DadosConexao &dados_conexao`	Estrutura contendo dados relevantes sobre a conexão estabelecida entre o cliente e o servidor, tais como nome do cliente, endereço IP, número da porta usada, dentre outros.

8.5 Principais Funções Presentes nas Aplicações Cliente e Servidor

int read_package_from_socket (int socket, Package &package, std::vector<char> &fileContentBuffer)

Localizada em *Common/package_functions.cpp:100.* Esta função é utilizada, tanto pelo servidor quanto pelo cliente, para realizar a leitura de um pacote que foi recebido (informações do pacote e o arquivo contido nele, caso exista).

Tabela 21 – Parâmetros da Função read_package_from_socket	
Parâmetro	**Descrição**
`int socket`	Identificador do socket a ser utilizado para a comunicação.
`Package &package`	Estrutura contendo todas as informações pacote a ser lido.
`vector<char> &fileContentBuffer`	Buffer utilizado para comportar os bytes de um arquivo, que pode ser transferido com o pacote.

int write_package_to_socket (int socket, Package &package, std::vector<char> &fileContentBuffer)

Localizada em *Common/package_functions.cpp:237.* Esta função é utilizada, tanto pelo servidor quanto pelo cliente, para realizar a escrita das informações de um pacote a ser enviado, utilizando um socket para comunicação (informações do pacote e o arquivo contido nele, caso exista).

Tabela 22 – Parâmetros da Função read_package_from_socket	
Parâmetro	**Descrição**
`int socket`	Identificador do socket a ser utilizado para a comunicação.
`Package &package`	Estrutura contendo todas as informações pacote a ser lido.
`vector<char> &fileContentBuffer`	Buffer utilizado para comportar os bytes de um arquivo, que pode ser transferido com o pacote.

void Package::htobe (void) Localizada em *Common/package.cpp:205.* Função utilizada para converter os bytes do pacote a ser enviado para a notação *big-endian.*

void Package::betoh (void) Localizada em *Common/package.cpp:241.* Função utilizada para converter os bytes de um pacote recebeido, que estão em notação *big-endian,* para a notação utilizada pelo destinatário.

9 PROBLEMAS ENCONTRADOS E SUAS SOLUÇÕES

9.1 Problema do loop de notificações

Nos casos em que existiam dois dispositivos de um mesmo usuário conectados ao servidor, havia a possibilidade de ocorrer um loop de notificações, da forma exemplificada abaixo.

```
[id=0x01] Dispositivo atual onde o arquivo 'teste' foi renomeado para 'exemplo':
1)  WRITE Package(CHANGE_EVENT, 0x01, FILE_RENAME, teste, exemplo)  <- Gerado pelo inotify, será propagado.
4)  READ  Package(CHANGE_EVENT, 0x02, FILE_RENAME, teste, exemplo)  <- Evento recebido pela propagação de 3)
5)  Erro ao renomear arquivo de "teste" para "exemplo".             <- Erro, o arquivo 'teste' não existe mais

[id=0x02] Segundo Dispositivo:
2)  READ  Package(CHANGE_EVENT, 0x01, FILE_RENAME, teste, exemplo)  <- Evento recebido pela propagação de 1)
3)  WRITE Package(CHANGE_EVENT, 0x02, FILE_RENAME, teste, exemplo)  <- Gerado pelo inotify, será propagado.
```

Para garantir que isso seja evitado, o último evento de notificação é armazenado no usuário. Antes do usuário enviar o evento para o servidor, verifica-se se o evento em questão é igual ao que já foi lido anteriormente. Caso seja, o evento é ignorado. O código referente à essa verificação está localizado em *Cliente/eventThread.cpp:187*.

9.2 Problema do IN_CLOSE_WRITE após IN_CREATE

Para a criação de arquivos (ou seja, nos casos em que é recebido um evento `FILE_CREATE` do servidor), utiliza-se `fopen(file, "w")`. A criação do arquivo `file` irá gerar dois eventos em seguida: `IN_CREATE` e `IN_CLOSE_WRITE`. O evento `IN_CLOSE_WRITE` enviará, então, o conteúdo atualizado para o servidor, que o salvará no lugar correto. O problema ocorre quando são utilizados dois dispositivos A e B por um mesmo usuário e, no dispositivo A, um arquivo qualquer é copiado para o diretório sincronizado. Nesse caso, normalmente o *inotify* enviaria os eventos corretamente, e o arquivo seria salvo no servidor. Porém, para aplicar os eventos criados, o dispositivo B criaria o arquivo file, o que geraria os eventos `IN_CREATE` e `IN_CLOSE_WRITE`. Assim, temos um segundo `IN_CLOSE_WRITE`. Com isso, o servidor receberia o conteúdo vazio do arquivo recém-criado, e substituiria o conteúdo que havia sido armazenado do dispositivo A. Como resultado, qualquer arquivo adicionado não teria conteúdo algum.

Para solucionar esse problema, é feita uma verificação para analisar (antes de realizar o envio da notificação do evento `FILE_MODIFIED` para o servidor) se o evento anterior correspondia a um `IN_CREATE` do mesmo arquivo, e se o arquivo modificado está vazio. Essas verificações indicam que o evento `IN_CLOSE_WRITE` foi gerado pela chamada de `fopen(file, "w")`. Nesse caso o evento não será enviado para o servidor. O código referente à essa verificação está localizado em *Client/eventThread.cpp:164*.