Universidade Federal do Rio Grande do Sul
Instituto de Informática
INF01151 – Sistemas Operacionais II N
Professor Alberto Egon Schaeffer Filho

Relatório

Eduarda Waechter – 00304585

Giulia Stefainski – 00289108

Maiara Trevisol – 00275946

Matheus Henrique Sabadin – 00228729

1. **Ambiente**
    a. Eduarda
        i. Sistema operacional: VDI Ubuntu 22.04.5 LTS 32GB
        ii. Processador: AMD EPYC 7763 @ 3.50GHz (64C/128T)
        iii. RAM: 32 GB
        iv. Compilador: g++ 11.4.0
    b. Giulia
        i. Sistema operacional: Ubuntu 24.04.2 LTS
        ii. Processador: Intel Core i5-7200U @ 2.50GHz (2C/4T)
        iii. RAM: 3.7 GB
        iv. Compilador: g++ 11.4.0
    c. Maiara
        i. Sistema operacional: Ubuntu 22.04.5 LTS
        ii. Processador: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz
        iii. RAM: 8GB
        iv. Compilador: g++ 11.4.0
    d. Matheus
        i. Sistema operacional: Windows 11 Pro 10.0.261 Build 26100 64-bit usando WSL2
        ii. Processador: Intel Core i7-14700K @ 3.40GHz (20C/28T)
        iii. RAM: 64 GB
        iv. Compilador: g++ 11.4.0

2. **Implementação**

(A) Como foi implementada a concorrência no servidor para atender múltiplos clientes;

A concorrência do servidor foi estruturada segundo o modelo *thread-per-task*, explorando as *std::thread* da biblioteca-padrão C++17 sobre sockets TCP.

1. **Escuta em um único socket e thread de handshake**
   O processo principal abre um socket fixo (porta 4000) e entra num laço accept(). Cada nova conexão recebida é imediatamente despachada para uma std::thread destacada dentro da função handle_new_connection — o lambda passado a std::thread processa apenas o handshake inicial do cliente.

2. **Controle de sessões simultâneas**
   Na thread de handshake, o servidor consulta o SessionManager para saber quantos dispositivos daquele usuário já estão conectados. A estrutura mantém contadores e tabelas protegidos por sessions_mtx; se o usuário já possui dois dispositivos ativos, a conexão é negada.

3. **Criação de três canais dedicados**
   Se o handshake é aceito, o servidor abre três sockets dinâmicos (função create_dynamic_socket) para:

   a. command – envio de comandos textuais;

   b. watcher – notificações de alterações;

   c. file – transferências de arquivos.
      Os números dessas portas são enviados ao cliente, e o descritor do socket 4000 é fechado. O listener da porta 4000 continua ativo.

4. **Delegação por função**

   a. Após aceitar as conexões nos sockets command e watcher, o servidor cria uma thread dedicada para cada um (handle_command_client e handle_watcher_client).

   b. O socket file fica em um laço que executa accept(); cada pedido de upload/download gera uma nova thread que roda handle_file_client, permitindo várias transferências paralelas para o mesmo usuário ou entre usuários distintos session_managerserver_tcp.

5. **Sincronização de recursos compartilhados**
   Regiões críticas — por exemplo, escrita/remoção de arquivos no disco — são

protegidas por file_mutex. As estruturas de sessão já citadas usam std::mutex (e, em versões alternativas do código, std::condition_variable) para garantir consistência dos contadores e acordar handshakes bloqueados quando um dispositivo se desconecta server_tcpserver_tcp.

6. **Benefícios da arquitetura**

    a. Isolamento de canais evita que um upload extenso bloqueie comandos ou notificações.

    b. Paralelismo real: em máquinas multicore, cada cliente (e cada transferência) executa em seu próprio núcleo quando disponível.

    c. Back-pressure explícita: o limite de dois dispositivos por usuário é aplicado sem bloquear o listener, preservando disponibilidade para outros usuários.

    d. Baixo overhead: threads de usuário do Linux/WSL2 têm custo de troca pequeno.

(B) Em quais áreas do código foi necessário garantir sincronização no acesso a dados;

Para que várias threads do servidor trabalhassem em paralelo sem corromper estado compartilhado, introduzimos pontos de sincronização em quatro frentes principais:

1. **Tabelas globais de sessão de usuários**
   Estruturas: sessions_by_cmd_fd, username_by_cmd_fd, device_count_by_user e o mapa auxiliar user_controls.
   Proteção: um std::mutex sessions_mtx envolve toda operação de inserção, busca ou remoção nesses mapas, garantindo atomicidade quando várias threads de handshake, comando ou limpeza acessam esses contêineres ao mesmo tempo .
   Além disso, cada usuário possui um mutex próprio (session->mtx) para serializar atualizações no contador connected_devices e no vetor sockets, evitando disputas entre as duas sessões permitidas por usuário .
   Por fim, o objeto user_controls[username] contém outro mtx + condition_variable para bloquear o handshake quando o limite de dois dispositivos é alcançado e acordá-lo quando uma sessão se encerra.

2. **Sistema de arquivos do servidor**
   Todas as operações que criam, sobrescrevem ou removem arquivos dentro de sync_dir são envolvidas por um mutex global file_mutex .
   Esse lock é usado tanto na rotina de upload/delete, onde a thread grava blocos recebidos no disco session_manager, quanto nos comandos de leitura (por exemplo, list_server) para que a listagem não colida com um upload em andamento no server_tcp.

3. **Recursos de rede críticos**
   Durante a criação de sockets dinâmicos para o canal de arquivos empregamos socket_creation_mutex. Ele evita que duas threads chamem create_dynamic_socket() simultaneamente e tentem reutilizar a mesma porta local antes que o bind() seja concluído server_tcpserver_tcp.

4. **Contadores de sessões ativas por usuário**
   O campo active_sessions em UserSessionControl é incrementado na chegada de um dispositivo e decrementado quando a respectiva thread de comando termina. Ambos os acessos ocorrem dentro de um std::lock_guard e cada término de sessão executa cv.notify_one() para liberar exatamente um handshake bloqueado, garantindo efeito de back-pressure sem busy-waiting server_tcpserver_tcp.

Empregamos mutexes (e uma condition variable) apenas onde dados realmente são partilhados entre threads:

- metadados de sessão,
- coleção de descritores de socket,
- diretórios/arquivos no disco,
- e a lógica de admissão de novos dispositivos.

(C) Descrição das principais estruturas e funções que você implementou;

O núcleo do sistema é um protocolo binário cujas mensagens seguem a estrutura Packet.
Ela contém o tipo do pacote, número de sequência, tamanho total, comprimento do payload e um buffer fixo de 1 KiB, permitindo fragmentação e remontagem transparentes nos pares send_packet/recv_packet .

Para manter a árvore de arquivos de cada usuário foi criado o módulo common, com duas rotinas essenciais:

- ensure_sync_dir(base, username) – cria (ou confirma) o diretório sync_dir_<user> no lado cliente e servidor, devolvendo o caminho absoluto;
- list_files_with_mac(dir) – percorre o diretório e devolve uma string formatada com atime/mtime/ctime, cumprindo o requisito dos MAC times.
  Ambas aparecem em common.cpp.

No servidor, o controle de sessões ficou concentrado em três objetos:

- UserSession (mutex próprio, contador connected_devices, vetores de sockets);
- SessionManager (mapas globais sessions_by_cmd_fd, username_by_cmd_fd, device_count_by_user, protegidos por sessions_mtx). Métodos-chave: try_connect, register_session, close_session_by_cmd_fd ;
- UserSessionControl (mutex + condition_variable + contador) guardado no mapa user_controls, usado para bloquear a terceira tentativa de login do mesmo usuário.

Essas estruturas asseguram o limite de dois dispositivos simultâneos por usuário e a limpeza automática das tabelas quando uma sessão termina.

A aceitação de clientes passa por create_dynamic_socket, que pede ao SO uma porta livre, faz listen() e devolve o fd e o número da porta .
No handshake (handle_new_connection) o servidor:

1. recebe o nome de usuário;
2. chama SessionManager::try_connect;
3. cria três sockets dinâmicos (command, watcher, file);
4. devolve as portas ao cliente;
5. regista a sessão e lança três threads dedicadas.

- handle_command_client processa comandos de texto (list_server, exit, ...) sob o mutex global file_mutex para evitar colisão com uploads packetcommon.
- handle_watcher_client usa inotify para vigiar o diretório do usuário e envia pacotes PACKET_TYPE_NOTIFY sempre que um arquivo é criado, alterado ou removido session_managerserver_tcp.

- handle_file_client recebe cabeçalhos putfile|... ou delfile|... e depois os blocos DATA, gravando ou apagando o arquivo sob proteção de file_mutex server_tcpserver_tcp.

Este arranjo isola comandos, notificações e transferências em canais independentes, evitando que um upload longo bloqueie o restante do protocolo.

No lado cliente destacam-se:

- connect_to_port (reúso de lógica de socket para todas as portas efêmeras);
- watch_sync_dir_inotify, que vigia o sync_dir local e dispara send_file ou comandos delfile|... conforme os eventos server_tcpserver_tcp;
- o mini-shell em command_interface.cpp, cujo vetor command_map despacha as palavras-chave upload, download, list_server, etc., para funções especializadas client_tcpcommand_interface.

Rotinas utilitárias como move_file_to_sync_dir, download_from_sync_dir e delete_from_sync_dir encapsulam operações de cópia e remoção com tratamento de erros e mensagens ao usuário.

Em conjunto, essas estruturas e funções implementam:

- um protocolo de pacotes unificado (Packet);
- gerência de sessões com limite de dispositivos, mutexes e condition variables;
- concorrência segura via threads dedicadas por canal;
- sincronização automática baseada em inotify;
- uma CLI enxuta, mas completa, para interação com o serviço.

(D) Explicar o uso das diferentes primitivas de comunicação;

Para garantir troca de dados confiável entre processos, detecção imediata de mudanças no disco e coerência interna entre *threads*, o projeto recorre a três grupos de primitivas.

1. **Comunicação entre processos – TCP sockets**

O servidor mantém um listener fixo na porta 4000; a cada handshake bem-sucedido ele gera três sockets dinâmicos, um para command, um para watcher e outro para file, usando as syscalls clássicas socket → bind → listen na função create_dynamic_socket . Do lado do cliente, connect_to_port encapsula o connect() para qualquer porta recebida do servidor, reutilizando a mesma lógica sempre que é preciso abrir um canal efêmero .

2. **Protocolos send_packet / recv_packet**

Sobre os sockets corre um protocolo binário unificado cujo cabeçalho está na estrutura Packet (tipo, sequência, tamanho, comprimento e payload). As rotinas send_packet e recv_packet serializam o cabeçalho, usam MSG_WAITALL para garantir entrega integral e tratam fragmentação de até 1 KiB por pacote .
Essas funções são chamadas por todo o código de alto nível: handshake, comandos de texto, blocos DATA de upload/download, ACKs e pacotes NOTIFY.

3. **Comunicação de eventos usando inotify**

Para tornar a sincronização imediata, tanto cliente quanto servidor usam inotify_init1 + inotify_add_watch para vigiar seus respectivos diretórios de trabalho. No servidor, handle_watcher_client lê os eventos e converte cada um num PACKET_TYPE_NOTIFY enviado pelo socket watcher ; no cliente, watch_sync_dir_inotify recebe eventos locais e dispara putfile|... ou delfile|... conforme necessário server_tcpclient_tcp. Assim evitamos polling e propagamos modificações quase em tempo real.

4. **Sincronização intra-processo – `std::mutex` (e `std::condition_variable`)**

Embora não troquem dados entre processos, os mutexes são essenciais para que as mensagens trafeguem em estado consistente:

- file_mutex serializa qualquer alteração real no sistema de arquivos durante uploads, deletes ou list_server server_tcpclient_tcp.

- sessions_mtx protege as tabelas globais do SessionManager, impedindo races entre threads de handshake, comando e limpeza de sessão server_tcpsession_manager.

Em versões alternativas do código, um condition_variable desperta handshakes bloqueados quando um dos dois dispositivos permitidos por usuário se desconecta.

Encadeamento das primitivas

1. O usuário digita list_server; o cliente empacota o texto em Packet e envia pelo socket command.

2. O servidor recebe via recv_packet, processa sob file_mutex, responde com ACK e devolve a listagem.

3. Se um upload acontece, o cabeçalho putfile|user|nome e os blocos DATA trafegam exclusivamente pelo socket file, garantindo que um envio longo não bloqueie comandos.

4. Quando um arquivo muda no servidor, o inotify dispara; o watcher thread prepara um NOTIFY que o cliente consome, chamando sync_with_server.

Desse modo, sockets cuidam da entrega ponto-a-ponto, inotify transforma alterações de disco em eventos de rede, e mutexes/condvars mantêm a memória compartilhada íntegra, formando um pipeline robusto que sustenta concorrência e consistência em múltiplos dispositivos.

(D) Problemas encontrados e soluções adotadas;

- Versões iniciais não usavam a abertura dinâmica de sockets e usavam apenas as portas 4000, 4001 e 4002. A Transição para portas dinâmicas gerou muitos bugs difíceis de detectar. A solução foi transicionar porta a porta e ir testando o programa.

- Quando três instâncias do mesmo usuário tentavam entrar, a terceira ficava em busy-wait. Substituímos o polling por condition_variable, acordando exatamente um handshake quando uma sessão encerra server_tcp.

- Uploads repetidos travavam porque a porta do lado cliente ficava presa em TIME_WAIT. A estratégia foi criar um novo socket a cada upload (connect_to_port é chamado de novo e depois fecha com shutdown+close) server_tcp e client_tcp.

- O debug consistiu majoritariamente em uma série de logs, pois debugar step by step usando a IDE geralmente se tornava um desafio devido às múltiplas threads e aos loops contínuos.

- Várias implementações mexiam em uma porção muito grande da base de código, gerando conflitos constantes de merge na ferramenta de versionamento e fazendo com que o trabalho fosse difícil de paralelizar.