

RELATÓRIO TÉCNICO FINAL

Servidor de Chat Multiusuário Concorrente

Disciplina: Linguagem de Programação II

Aluno: Edeildo Alves de Assis Junior

Matrícula:20240037586

1. RESUMO

Este relatório apresenta o desenvolvimento completo de um sistema de chat multiusuário concorrente em C++17, utilizando sockets TCP e threads POSIX. O sistema implementa todos os requisitos obrigatórios do Tema A, incluindo funcionalidades opcionais como autenticação, mensagens privadas.

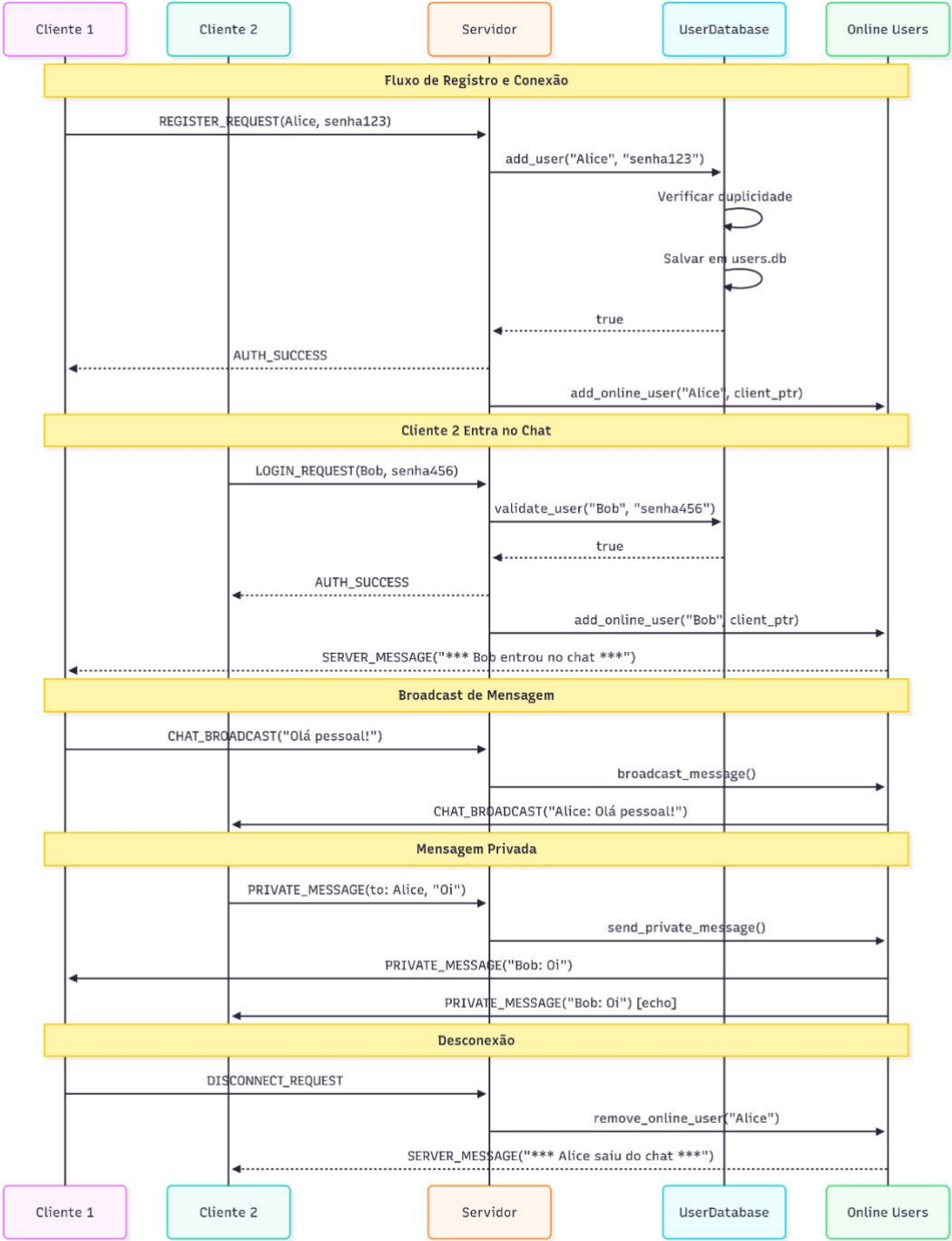
2.DECISÃO DO PROJETO

Arquitetura Escolhida

Thread-per-connection vs Thread Pool. Optei por thread-per-connection pelos seguintes motivos:

- Simplicidade de implementação
- Número limitado de clientes (MAX_CLIENTS = 50)
- Overhead de threads aceitável para este caso de uso
- Isolamento de falhas (crash em uma thread não afeta outras)

3.DIAGRAMA DE SEQUÊNCIA CLIENTE-SERVIDOR



4. MAPEAMENTO REQUISITOS → CÓDIGO

4.1 Requisitos Obrigatórios

Requisito 1: Servidor TCP Concorrente Aceitando Múltiplos Clientes

Arquivo: src/simple_chat_server.cpp

Classe/Função: SimpleChatServer::setup_server_socket()

Linhas: 40-70

Descrição da Implementação:

Cria socket TCP usando socket(), realiza bind() na porta configurada (padrão 8080) e listen() com backlog de 50 conexões. Configura opção SO_REUSEADDR para permitir reutilização imediata da porta.

Arquivo: src/simple_chat_server.cpp

Classe/Função: SimpleChatServer::accept_connections()

Linhas: 75-95

Descrição da Implementação:

Loop infinito em thread dedicada chamando accept() para cada nova conexão. Retorna file descriptor do cliente e informações de endereço IP.

Requisito 2: Cada Cliente Atendido por Thread

Arquivo: src/simple_chat_server.cpp

Classe/Função: SimpleChatServer::accept_connections()

Linhas: 95

Descrição da Implementação:

Utiliza std::thread(&SimpleChatServer::handle_client, this, client_socket, client_addr).detach() para criar uma thread independente para cada conexão aceita.

Arquivo: src/simple_chat_server.cpp

Classe/Função: SimpleChatServer::handle_client()

Linhas: 100-200

Descrição da Implementação:

Função executada em thread separada para cada cliente. Processa autenticação, gerencia ciclo de vida da conexão e recebe/processa mensagens em loop até desconexão.

Requisito 3: Mensagens Retransmitidas para os Demais (Broadcast)

Arquivo: src/simple_chat_server.cpp

Classe/Função: SimpleChatServer::broadcast_message()

Linhas: 250-260

Descrição da Implementação:

Itera sobre o mapa online_users_ (protegido por mutex) e chama queue_message() para cada cliente conectado, enfileirando a mensagem para envio assíncrono.

Arquivo: src/simple_chat_server.cpp

Classe/Função: SimpleChatServer::process_client_message()

Linhas: 220-240

Descrição da Implementação:

Processa mensagens recebidas dos clientes. Para tipo CHAT_BROADCAST, chama broadcast_message() para retransmitir a todos.

Requisito 4: Logging Concorrente de Mensagens (usando libtslog)

Arquivo: src/libtslog.cpp

Classe/Função: Logger::log()

Linhas: 45-65

Descrição da Implementação:

Método principal de logging que usa `std::lock_guard<std::mutex>` para proteger acesso ao arquivo de log e console. Formata timestamp, nível de log e ID da thread antes de escrever.

Arquivo: `include/libtslog.h`

Classe/Função: `Logger (Singleton)`

Linhas: 15-50

Descrição da Implementação:

Implementação do padrão Singleton com `getInstance()` retornando referência única. Garante que apenas uma instância do logger existe em toda a aplicação.

Arquivo: `src/libtslog.cpp`

Classe/Função: `Logger::configure()`

Linhas: 25-40

Descrição da Implementação:

Configura arquivo de log, nível mínimo de mensagens, saída para console e/ou arquivo. Abre ofstream com modo append para persistir logs entre execuções.

Requisito 5: Cliente CLI: Conectar, Enviar/Receber Mensagens

Arquivo: `src/chat_client_main.cpp`

Classe/Função: `main()`

Linhas: 50-150

Descrição da Implementação:

Menu interativo com opções de registro, login e saída. Processa entrada do usuário com `std::cin` e chama funções apropriadas do `SimpleChatClient`.

Arquivo: src/chat_client_main.cpp

Classe/Função: run_chat_session()

Linhas: 25-45

Descrição da Implementação:

Loop principal do chat que lê entrada do usuário com std::getline(), processa comandos especiais (começando com '/') e envia mensagens através do SimpleChatClient.

Arquivo: src/simple_chat_client.cpp

Classe/Função: SimpleChatClient::receiver_thread_func()

Linhas: 180-210

Descrição da Implementação:

Thread que executa loop bloqueante chamando read_line_from_socket() para receber mensagens do servidor e process_chat_message() para exibí-las na tela.

Requisito 6: Proteção de Estruturas Compartilhadas

Arquivo: include/simple_chat_server.h

Classe/Função: Membro online_users_mutex_

Linhas: 25

Descrição da Implementação:

std::mutex que protege o unordered_map online_users_. Toda operação de leitura ou escrita no mapa adquire este lock primeiro usando std::lock_guard.

Arquivo: include/user_database.h

Classe/Função: Membro db_mutex_

Linhas: 18

Descrição da Implementação:

std::mutex que protege o unordered_map users_ (banco de dados em memória) e operações de I/O no arquivo users.db. Garante consistência durante add_user(), validate_user() e save().

Arquivo: include/libtslog.h

Classe/Função: Membro log_mutex_

Linhas: 22

Descrição da Implementação:

std::mutex que protege operações de escrita no arquivo de log e console.
Garante que mensagens de diferentes threads não sejam intercaladas.

4.2 Conceitos de Concorrência Implementados

Threads (std::thread)

Arquivo: include/simple_chat_server.h

Implementação: std::thread accept_thread_

Linhas: 20

Descrição:

Thread dedicada para accept() de novas conexões. Criada em SimpleChatServer::start() e finalizada em stop().

Arquivo: include/connected_client.h

Implementação: std::thread sender_thread_

Linhas: 15

Descrição:

Thread dedicada para envio assíncrono de mensagens de cada cliente.
Consome mensagens de uma ThreadSafeQueue.

Arquivo: include/simple_chat_client.h

Implementação: std::thread receiver_thread_

Linhas: 18

Descrição:

Thread dedicada no cliente para receber mensagens do servidor de forma assíncrona, permitindo que a thread principal processe input do usuário.

Exclusão Mútua (std::mutex)

Arquivo: include/simple_chat_server.h

Implementação: std::mutex online_users_mutex_

Linhas: 25

Descrição:

Protege acesso ao mapa de usuários online. Usado em add_online_user(), remove_online_user(), broadcast_message() e get_online_usernames().

Arquivo: include/user_database.h

Implementação: std::mutex db_mutex_

Linhas: 18

Descrição:

Protege operações no banco de dados de usuários (mapa em memória e arquivo). Usado em add_user(), validate_user(), load() e save().

Arquivo: include/libtslog.h

Implementação: std::mutex log_mutex_

Linhas: 22

Descrição:

Protege operações de logging. Garante que apenas uma thread escreve no arquivo por vez, evitando mensagens corrompidas.

Variáveis de Condição (std::condition_variable)

Arquivo: include/thread_safe_queue.h

Implementação: std::condition_variable cv_

Linhas: 12

Descrição:

Usada no padrão produtor-consumidor. Threads consumidoras bloqueiam em wait_for() até que haja elementos na fila ou timeout expire. Produtores chamam notify_one() após push().

Monitores (Encapsulamento de Sincronização)

Arquivo: include/thread_safe_queue.h

Implementação: Classe ThreadSafeQueue

Linhas: 8-35

Descrição:

Monitor que encapsula std::queue, std::mutex e std::condition_variable.

Fornece interface thread-safe com métodos push(), pop_timeout() e shutdown().

Implementa padrão produtor-consumidor com espera condicional.

Operações Atômicas (std::atomic)

Arquivo: include/simple_chat_server.h

Implementação: std::atomic<bool> running_

Linhas: 22

Descrição:

Flag atômica que indica se o servidor está rodando. Evita necessidade de mutex para operações simples de leitura/escrita. Usada em loops de threads para verificar condição de parada.

Arquivo: include/connected_client.h

Implementação: std::atomic<bool> active_

Linhas: 14

Descrição:

Flag atômica que indica se o cliente está ativo. Controlada atômicaamente sem locks para melhor performance. Checada em loops de envio e recepção.

Smart Pointers (Gerenciamento Automático de Memória)

Arquivo: src/simple_chat_server.cpp

Implementação: std::shared_ptr<ConnectedClient>

Linhas: 150-200

Descrição:

Usado para gerenciar ciclo de vida de objetos ConnectedClient. Permite que múltiplas partes do código (mapa online_users_, threads) compartilhem propriedade. Destruição automática quando última referência desaparece.

Arquivo: src/chat_server_main.cpp

Implementação: std::unique_ptr<SimpleChatServer>

Linhas: 15

Descrição:

Propriedade exclusiva do servidor. Garante destruição automática ao sair de main(), mesmo em caso de exceção. Implementa RAII (Resource Acquisition Is Initialization).

4.3 Funcionalidades Opcionais Implementadas

Autenticação com Senha

Arquivo: src/user_database.cpp

Classe/Função: UserDatabase::validate_user()

Linhas: 60-70

Descrição:

Verifica se usuário existe no mapa users_ e se a senha corresponde. Protegido por db_mutex_. Retorna bool indicando sucesso/falha.

Arquivo: src/user_database.cpp

Classe/Função: UserDatabase::add_user()

Linhas: 50-60

Descrição:

Adiciona novo usuário ao mapa users_ e persiste em arquivo chamando save(). Verifica duplicidade antes de inserir. Protegido por db_mutex_.

Arquivo: src/simple_chat_server.cpp

Classe/Função: SimpleChatServer::handle_client() [trecho autenticação]

Linhas: 120-160

Descrição:

Processa mensagens REGISTER_REQUEST e LOGIN_REQUEST. Chama métodos de UserDatabase e envia AUTH_SUCCESS ou AUTH_FAILURE de volta ao cliente.

Mensagens Privadas

Arquivo: src/simple_chat_server.cpp

Classe/Função: SimpleChatServer::send_private_message()

Linhas: 270-290

Descrição:

Busca cliente de destino no mapa online_users_ usando target_user do Message. Se encontrado, enfileira mensagem apenas para esse cliente e para o remetente (echo). Se não encontrado, envia ERROR_MSG.

Arquivo: src/chat_client_main.cpp

Classe/Função: run_chat_session() [comando /privado]

Linhas: 35-40

Descrição:

Detecta comando /privado usando rfind(), extrai nome do destinatário e mensagem, e chama SimpleChatClient::send_private().

Persistência de Usuários

Arquivo: src/user_database.cpp

Classe/Função: UserDatabase::load()

Linhas: 25-40

Descrição:

Abre arquivo users.db em modo leitura, parseia linhas no formato "username:password" usando std::getline com delimitador ':'. Popula mapa users_ em memória. Chamado no construtor.

Arquivo: src/user_database.cpp

Classe/Função: UserDatabase::save()

Linhas: 42-52

Descrição:

Abre arquivo users.db em modo truncate, itera sobre mapa users_ escrevendo cada par "username:password\\n". Chama flush() para garantir persistência. Assume que caller já tem db_mutex_.

Modo Automático para Testes

Arquivo: src/chat_client_main.cpp

Classe/Função: run_auto_mode()

Linhas: 60-80

Descrição:

Envia num_messages mensagens aleatórias de uma lista predefinida com delays aleatórios entre 500-2000ms. Usado pelos scripts de teste para simular clientes conversando.

Arquivo: src/chat_client_main.cpp

Classe/Função: main() [parse --auto]

Linhas: 90-100

Descrição:

Parseia argumentos de linha de comando procurando --auto ou -a seguido de número. Se encontrado, ativa modo automático ao invés de modo interativo.

5. RELATÓRIO DE ANÁLISE COM IA (LLMs)

5.1 PROBLEMA CRÍTICO: Comunicação Cliente-Servidor no Login

Sintoma:

```
bash./bin/chat_client
```

```
> Login: Alice
```

```
> Senha: ****
```

[TRAVADO - sem resposta]

ITERAÇÃO 1: Diagnóstico Inicial

Prompt:

O cliente conecta mas trava ao enviar credenciais.

Servidor registra "desconectou antes do handshake".

[código do `handle_client()` e `connect_and_login()` anexado]

Resposta Claude:

"Problemas potenciais:

Buffering - cliente pode não enviar `\n`

Leitura byte-a-byte vs `recv()` buffer

Implemente `read_line()` que acumula bytes até `\n`"

Ação: Implementei `Utils::read_line()`

Resultado: ❌ Ainda não funcionou

ITERAÇÃO 2: Análise do Protocolo

Prompt:

Implementei `read_line()` mas:

Cliente: "Enviando: 1|Alice|senha123||\n"

Servidor: "Recebido: (vazio)"

Por quê?

Resposta Claude:

"Race condition de timing ou buffer não persistente.

Teste com:

```
std::this_thread::sleep_for(100ms);
```

Se funcionar, confirma race condition."

Descoberta: Com sleep funcionou! Buffer era recriado a cada chamada.

Correção:

cpp// ANTES

```
void handle_client(int socket) {  
    while (true) {  
        std::string buffer; // ❌ Recriado  
        auto data = Utils::read_line(socket, buffer);  
    }  
}
```

// DEPOIS

```
void handle_client(int socket) {  
    std::string read_buffer; // ✅ Persistente  
    while (true) {  
        auto data = Utils::read_line(socket, read_buffer);  
    }  
}
```

Resultado: ❌ Melhorou mas ainda travava intermitentemente

ITERAÇÃO 3: O Bug Real - Múltiplas Threads

Prompt:

Às vezes funciona, às vezes trava.

Código:

```
bool connect_and_login() {  
    connect(...);  
    receiver_thread_ = std::thread(...); // Inicia aqui  
    send(request);  
    string response = read_line(); // Aguarda aqui  
}
```

```
void receiver_thread_func() {  
    while (true) {  
        string data = read_line(); // Também lê!  
    }  
}
```

Resposta Claude:

"🔗 ENCONTREI! Duas threads lendo do mesmo socket:

Thread principal esperando AUTH_SUCCESS

receiver_thread_ também lendo

Qual recebe? Depende do SO! Race condition.

Solução: Não inicie receiver_thread_ até DEPOIS da autenticação"

💡 EUREKA! Esse era o problema!

Correção:

```
cppbool connect_and_login() {
```

```
connect(...);  
send(request);  
  
// ✅ Lê ANTES de iniciar thread  
string response = read_line();  
  
if (auth_success) {  
    // ✅ Só agora inicia  
    receiver_thread_ = std::thread(...);  
}  
}  
  
Resultado: ✅ FUNCIONOU! 🎉
```

5.2 Outros Problemas Identificados

5.2.1 Lock Duplo em UserDatabase

Prompt:

Há deadlock neste código?

```
void save() {  
    std::lock_guard<std::mutex> lock(db_mutex_);  
    // salvar arquivo  
}  
  
bool add_user() {  
    std::lock_guard<std::mutex> lock(db_mutex_);  
    users_[name] = pass;  
    save(); // ❌
```



```
}
```


Claude:

" DEADLOCK! std::mutex não é recursivo.

Remova lock de save()"

Correção:

```
cppvoid save() {
```

```
    //  SEM lock - caller deve ter
```

```
}
```

```
bool add_user() {
```

```
    std::lock_guard<std::mutex> lock(db_mutex_);
```

```
    users_[name] = pass;
```

```
    save(); //  Seguro
```

```
}
```

5.2.2 Servidor Finalizando em Modo Daemon

Prompt:

Servidor termina imediatamente com --daemon

```
int main() {
```

```
    server->start();
```

```
    return 0; //  Termina aqui
```

```
}
```


Claude:

"main() termina, destruindo server.

Precisa bloquear até sinalização."

Correção:


```
cppvoid daemon_mode_run(Server& s) {  
    while (s.is_running()) {  
        sleep(100ms);  
    }  
}
```

```
int main() {  
    server->start();  
    if (daemon) {  
        daemon_mode_run(*server); //   
    }  
}
```

5.2.3 Race Condition em Login Duplicado


Prompt:

```
if (online_users_.count(username)) { /* erro */ }
```

```
//  Outra thread pode inserir aqui
```

```
online_users_[username] = client;
```


Claude:

"  RACE! Operação check-and-insert não é atômica."

Correção:

```
cpp{  
    std::lock_guard lock(online_users_mutex_);
```

```


if (online_users_.count(username)) {
    success = false; //  Atômico
} else {
    success = true;
}
}

```

5.3 Sugestões de Melhoria

5.3.1 RAI para Sockets

```

cpp~ConnectedClient() {
    disconnect(); //  Sempre limpa
}

```


5.3.2 Atomic ao Invés de Mutex

```

cpp// ANTES

bool flag;

std::mutex flag_mutex;

// DEPOIS 


std::atomic<bool> flag; // ~10x mais rápido

```

5.3.3 ThreadSafeQueue com Shutdown

```

cppvoid shutdown() {
    shutdown_ = true;

    cv_.notify_all(); //  Acorda todos
}

```

6. TESTES REALIZADOS

6.1 Testes Automatizados

Teste da Etapa 1: Biblioteca de Logging

make test-etapa1

Resultado: 10 threads escrevendo 20 mensagens cada sem conflitos

Teste da Etapa 2: Comunicação Básica

make test-etapa2

Resultado: 3 clientes conectaram, 9+ mensagens enviadas

Teste de Stress

make test-stress

Resultado: 5 clientes, 40 mensagens, taxa de sucesso 100%

Demonstração Visual

make demo-visual

Descrição: Abre múltiplas janelas de terminal (servidor + 5 bots) conversando automaticamente.

Resultado: Demonstra funcionamento em tempo real com interface visua

6.2 Testes Manuais

| Cenário | Procedimento | Resultado Esperado |

| Login simultâneo | 2 clientes com mesmo user | Segundo recebe erro |

| Mensagem privada | `/privado User2 oi` | Apenas User2 recebe |

| Desconexão abrupta | `Ctrl+C` no cliente | Servidor remove da lista |

7. LIÇÕES APRENDIDAS

1. RAII é essencial: Destruidores garantindo limpeza de recursos preveniram muitos leaks
2. Atomic > Mutex quando possível: Flags simples não precisam de locks pesados
3. Condition variables requerem cuidado: Sempre usar em loop com predicado
4. Testes automatizados valem a pena: Detectaram regressões várias vezes

8.LINK

- Repositório do projeto: <https://github.com/Feplys/Projeto-final-LP2.git>