

Trabalho Final — Relatório Técnico

Programação Concorrente (C / C++)

- **Aluno:** Alexandre Lucas Justino Guedes
- **Matrícula:** 20230047293
- **Tema:** Servidor de Chat Multiusuário (TCP)
- **Data de Entrega:** 06 de Outubro de 2025

1. Mapeamento de Requisitos e Implementação

Esta seção documenta como os requisitos gerais do trabalho foram atendidos no código-fonte do projeto.

Requisito	Ficheiro(s) de Implementação	Breve Descrição
Threads (<code>std::thread</code>)	<code>src/server.cpp</code> (linha 194)	O servidor utiliza <code>std::thread(handle_client, ...).detach()</code> para criar uma nova thread para cada cliente que se conecta, garantindo o atendimento concorrente.
Exclusão Mútua	<code>src/server.cpp</code> (linhas 31, 55, 78, etc.)	Um <code>std::mutex</code> global (<code>client_mutex</code>) é utilizado com <code>std::lock_guard</code> para proteger todas as regiões críticas que acedem às estruturas de dados partilhadas (<code>clients</code> e <code>message_history</code>).
Sockets	<code>src/server.cpp</code> , <code>src/client.cpp</code>	Ambos os ficheiros utilizam a API de Sockets POSIX (<code>socket</code> , <code>bind</code> , <code>listen</code> , <code>accept</code> , <code>connect</code> , <code>send</code> , <code>recv</code>) para estabelecer a comunicação TCP em rede.

Logging Concorrente	<code>src/server.cpp</code> (e toda a app)	A biblioteca <code>Logger</code> (implementada na Etapa 1) é utilizada em todo o servidor para registrar eventos de forma segura a partir de múltiplas threads.
Estruturas Partilhadas	<code>src/server.cpp</code> (linhas 28, 32)	O <code>std::vector<ClientInfo> clients</code> e o <code>std::deque<string> message_history</code> são as estruturas partilhadas, protegidas pelo <code>client_mutex</code> .
Build (Makefile)	<code>makefile</code>	Um Makefile funcional foi fornecido para compilar os executáveis <code>server</code> e <code>client</code> , além de oferecer alvos para limpeza (<code>clean</code>).
Documentação	<code>README.md</code>	O ficheiro README contém instruções detalhadas sobre como compilar e executar o projeto.

2. Diagrama de Sequência Cliente-Servidor

O diagrama abaixo ilustra a sequência de interações entre dois clientes e o servidor, desde a conexão até ao envio e receção de mensagens.

participant Cliente 1
participant Servidor
participant Cliente 2

Cliente 1->>Servidor: connect() [Conexão TCP]
Servidor-->>Cliente 1: accept()
Note right of Servidor: Servidor cria Thread A para Cliente 1
Servidor-->>Cliente 1: Envia histórico de mensagens (se houver)

Cliente 2->>Servidor: connect() [Conexão TCP]
Servidor-->>Cliente 2: accept()
Note right of Servidor: Servidor cria Thread B para Cliente 2
Servidor-->>Cliente 2: Envia histórico de mensagens

Cliente 1->>Servidor: send("Olá!")
Note right of Servidor: Thread A recebe a mensagem
Servidor->>Cliente 2: broadcast("Olá!")

Cliente 2->>Servidor: send("Oi, tudo bem?")
Note right of Servidor: Thread B recebe a mensagem
Servidor->>Cliente 1: broadcast("Oi, tudo bem?")

Cliente 1->>Servidor: close() [Desconexão]
Note right of Servidor: Thread A termina e remove Cliente 1 da lista

3. Relatório de Análise de Concorrência com IA

Esta seção apresenta o resultado de uma análise crítica do código-fonte do servidor, realizada com o auxílio de um modelo de linguagem de grande escala (IA/LLM) para identificar e mitigar potenciais problemas de concorrência.

Pergunta 1: Análise de Condição de Corrida

Questão do Desenvolvedor:

"No `server.cpp`, várias threads de clientes diferentes acedem à lista `clients` e ao histórico `message_history` ao mesmo tempo. A minha preocupação é que uma thread possa tentar ler a lista enquanto outra está a adicionar um novo cliente, causando um erro ou corrompendo os dados. Como o código atual previne esse tipo de problema?"

Resposta da IA: A sua preocupação é totalmente válida e aponta para o risco mais comum em sistemas concorrentes: a **condição de corrida (race condition)**. O código implementado mitiga este risco de forma eficaz através do uso de um **mecanismo de exclusão mútua**, o `std::mutex`. Todas as secções de código que leem ou modificam os dados partilhados (`clients` e `message_history`) são protegidas pelo `client_mutex` utilizando `std::lock_guard`. Este mecanismo garante que apenas uma thread de cada vez possa aceder a estes dados, assegurando a sua consistência e integridade. O uso do `std::lock_guard` também previne erros comuns como o esquecimento de libertar o mutex, o que poderia levar a deadlocks.

Pergunta 2: Análise de Deadlock

Questão do Desenvolvedor:

"Nós usamos um único `mutex` (`client_mutex`) para proteger duas coisas: a lista de clientes e o histórico. Usar o mesmo "cadeado" para tudo não é perigoso? Não existe o risco de duas threads ficarem presas, uma esperando pela outra para sempre (um deadlock)?"

Resposta da IA: Um deadlock ocorre tipicamente quando há uma aquisição circular de múltiplos bloqueios. No seu código, a estratégia de usar um único `std::mutex` para proteger todos os recursos partilhados é conhecida como bloqueio de granularidade grosseira. Embora possa ter implicações de performance, esta abordagem é **extremamente eficaz para prevenir deadlocks**. O risco de deadlock no

seu projeto é nulo, pois uma thread nunca tenta adquirir um segundo mutex enquanto detém o primeiro; ela adquire apenas o `client_mutex`, realiza a sua operação e liberta-o. Esta simplicidade elimina a possibilidade de bloqueios circulares.

Pergunta 3: Análise de Performance e Gargalos

Questão do Desenvolvedor:

"Se o `mutex` força cada thread a esperar a sua vez para fazer qualquer coisa, isso não vai deixar o servidor muito lento se muitos clientes se conectarem? O `mutex` não se torna um gargalo?"

Resposta da IA: Sim, sob uma carga muito elevada, este `mutex` único pode tornar-se um **gargalo de performance (bottleneck)**, pois serializa o acesso aos dados partilhados. No entanto, a duração do bloqueio é muito curta, pois as operações realizadas na região crítica (como adicionar um elemento a um vetor) são extremamente rápidas. Para o âmbito deste projeto académico, este design prioriza a **correção e a segurança** sobre a máxima performance, sendo uma escolha de engenharia adequada e defensável. Para um sistema de grande escala, poder-se-ia explorar alternativas como bloqueios de granularidade fina ou estruturas de dados `lock-free`.

Pergunta 4: Análise de Vazamento de Recursos (Threads `detach`)

Questão do Desenvolvedor:

"Na função `main`, usamos `thread(...).detach()`. Se um cliente se desconectar abruptamente, a thread dele no servidor fica a correr para sempre, gastando recursos?"

Resposta da IA: O risco de uma thread "zombie" foi corretamente mitigado. A vida da thread está ligada à função `handle_client`, que passa a maior parte do tempo na chamada bloqueante `recv()`. Quando um cliente se desconecta (mesmo que abruptamente), a ligação TCP é terminada, fazendo com que `recv()` retorne `0` ou `-1`. O código verifica esta condição com `if (bytes_received <= 0)`, o que causa a saída do loop `while(true)`. A função então prossegue para a limpeza, e ao retornar, a thread termina a sua execução de forma limpa, libertando todos os seus recursos para o sistema operativo. Portanto, o uso de `detach()` é seguro neste contexto.

Pergunta 5: Necessidade de Threads no Cliente

Questão do Desenvolvedor: "Enquanto programávamos o `client.cpp`, decidimos usar duas threads: uma para receber mensagens (`receive_thread`) e a thread

`main` para enviar. No início, eu pensei em fazer tudo num único loop `while`. Por que essa abordagem com uma thread só não funcionaria e por que as duas threads são essenciais para o cliente funcionar bem?"

Resposta da IA: Essa é uma dúvida fundamental sobre I/O (Entrada/Saída) bloqueante. A abordagem de thread única não funcionaria porque tanto a leitura da rede (`recv()`) quanto a leitura do teclado (`getline()`) são operações bloqueantes. Se estivessem no mesmo loop, o programa ficaria preso: ou estaria à espera de uma mensagem da rede, impedindo-o de digitar, ou estaria à espera de que você digitasse, impedindo-o de receber mensagens de outros. Ao separar estas tarefas em duas threads, permitimos que estas operações bloqueantes ocorram em paralelo. Uma thread pode ficar bloqueada à espera da rede, enquanto a outra fica bloqueada à espera do teclado, fazendo com que o cliente seja responsivo e capaz de enviar e receber mensagens ao mesmo tempo.

Pergunta 6: Lógica de Bloqueio no Broadcast

Questão do Desenvolvedor: "Na função `broadcast_message`, nós trancamos o `mutex` uma vez, antes de começar o loop que envia a mensagem para todos os clientes. Durante a implementação, eu questionei-me se não seria melhor trancar e destrancar o `mutex` dentro do loop, apenas no momento de pegar o `client.fd`. Isso não deixaria a lista de clientes livre mais rapidamente para outras threads?"

Resposta da IA: A sua sugestão foca-se numa potencial otimização de performance, mas a abordagem atual prioriza a segurança e a consistência dos dados, o que é crucial. Se trancássemos e destrancássemos o `mutex` dentro do loop, abriríamos uma janela para uma condição de corrida perigosa: uma outra thread (por exemplo, a de um cliente que se desconectou) poderia modificar o vetor `clients` enquanto a thread do broadcast está a iterar sobre ele. Isso poderia invalidar os iteradores do loop `for`, causando um crash no servidor. Ao trancar o `mutex` antes do loop e libertá-lo depois, garantimos uma "fotografia" consistente da lista de clientes durante toda a operação de broadcast, prevenindo crashes e garantindo que a mensagem chegue a todos os que estavam conectados naquele instante.

Pergunta 7: Tratamento de Desconexão (`recv <= 0`)

Questão do Desenvolvedor: "No código, a thread do cliente sai do loop de mensagens quando `recv()` retorna um valor menor ou igual a zero (`bytes_received <= 0`). Eu entendo que isso significa que o cliente saiu, mas qual é a diferença técnica entre `recv()` retornar `0` e retornar `-1`? Por que decidimos tratar os dois casos da mesma forma?"

Resposta da IA: Essa é uma excelente pergunta sobre os detalhes da API de sockets. `recv()` a retornar `0` indica uma desconexão limpa: o cliente do outro lado da conexão fechou o socket de forma ordenada. `recv()` a retornar `-1`

indica que ocorreu um erro na conexão (por exemplo, a rede caiu ou o processo do cliente foi terminado abruptamente). Embora as causas sejam diferentes, o resultado prático para o servidor é o mesmo: a conexão com aquele cliente já não é viável. Portanto, tratar ambos os casos da mesma forma — registrando a desconexão, fechando o socket e removendo o cliente da lista — é a abordagem lógica e eficiente para a gestão do ciclo de vida do cliente.

Pergunta 8: A Real Necessidade do `memset`

Questão do Desenvolvedor: "Uma coisa que fizemos e que eu fiquei na dúvida foi usar `memset(buffer, 0, 4096)` antes de cada chamada a `recv`. O `recv` não deveria simplesmente sobrescrever o conteúdo do buffer? Esta limpeza é mesmo necessária ou é apenas uma boa prática?"

Resposta da IA: É mais do que uma boa prática; é uma medida de segurança para prevenir a corrupção de dados. A função `recv()` preenche o buffer com os bytes que chegam, mas não garante a terminação com um caractere nulo (`\0`) se a mensagem recebida for mais curta que o buffer. Imagine que o servidor recebe uma mensagem longa: "mensagem longa aqui". O buffer fica com esse conteúdo. Se a mensagem seguinte for curta, como "oi", `recv()` irá sobrescrever apenas os dois primeiros caracteres. Sem o `memset`, o buffer conteria "oi" seguido do lixo da mensagem anterior ("nsagem longa aqui"). Ao tratar o buffer como uma string, acabaríamos por transmitir "oinsagem longa aqui". O `memset` garante que o buffer está sempre limpo antes de uma nova leitura, assegurando que cada mensagem seja tratada de forma isolada e correta.

Pergunta 9: O Motivo da Struct `ClientInfo`

Questão do Desenvolvedor: "Para guardar os clientes, criámos a struct `ClientInfo { int fd; string ip; }`. Por que não usar simplesmente um `vector<int>` para os sockets? Parecia mais simples durante a implementação. Qual foi o ganho real ao usar a struct?"

Resposta da IA: A decisão de usar uma struct é um princípio fundamental de bom design de software chamado encapsulamento. Embora um `vector<int>` funcionasse, ele separaria dados que estão intrinsecamente ligados: o socket (`fd`) e o endereço de IP (`ip`) de um cliente. Ao agrupá-los na `ClientInfo`, garantimos que estes dados andem sempre juntos, o que torna o código mais limpo, mais legível e menos propenso a erros (como misturar o IP de um cliente com o socket de outro). Além disso, torna o código mais extensível. Se, no futuro, quiser.

1. Mapeamento Requisitos -> Código

Esta seção demonstra onde cada requisito técnico do trabalho foi implementado no código-fonte do projeto.

Requisitos Gerais de Programação Concorrente

1.1. Threads *Onde o paralelismo é implementado.*

```
thread(handle_client, client_fd, string(client_ip)).detach();
```

```
thread(receive_messages, client_fd).detach();
```

1.2. Exclusão Mútua *Onde os dados compartilhados são protegidos.*

```
lock_guard<mutex> lock(client_mutex);
clients.erase(
    remove_if(clients.begin(), clients.end(), [client_fd](const ClientInfo& client) {
        return client.socket_fd == client_fd;
    }),
    clients.end()
);
```

1.3. Monitores *Onde a sincronização é encapsulada em uma classe.*

```
class Logger {
public:
    static Logger& getInstance();
    void log(const std::string& message);

    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) =delete;

private:
    Logger();
    ~Logger();

    std::ofstream m_logFile;
    std::mutex m_mutex;
};
```

1.4. Sockets Onde a comunicação de rede é configurada.

```
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
    Logger::getInstance().log("ERRO: Falha ao criar o socket do servidor.");
    perror("socket failed");
    exit(EXIT_FAILURE);
}
Logger::getInstance().log("Socket do servidor criado com sucesso.");

address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(PORT);

if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
    Logger::getInstance().log("ERRO: Falha ao fazer o bind do socket na porta " + to_string(PORT));
    perror("bind failed");
    exit(EXIT_FAILURE);
}
Logger::getInstance().log("Bind do socket na porta " + to_string(PORT) + " realizado com sucesso.");

if (listen(server_fd, 10) < 0) {
    Logger::getInstance().log("ERRO: Falha ao colocar o servidor para escutar.");
    perror("listen");
    exit(EXIT_FAILURE);
}
```

1.5. Gerenciamento de Recursos Onde os recursos (sockets) são liberados.

```
close(client_fd);
```

1.6. Tratamento de Erros Onde falhas de sistema são verificadas.

```
if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
    Logger::getInstance().log("ERRO: Falha ao fazer o bind do socket na porta " + to_string(PORT));
    perror("bind failed");
    exit(EXIT_FAILURE);
}
```

1.7. Logging Concorrente Onde o logger é usado dentro de uma thread.

```
while (true) {
    memset(buffer, 0, 4096);
    int bytes_received = recv(client_fd, buffer, 4096, 0);

    if (bytes_received <= 0) {
        Logger::getInstance().log("Cliente " + client_ip + " desconectou.");
        cout << "Cliente " + client_ip + " desconectou." << endl;
        break;
    }

    string received_message(buffer, bytes_received);
    string formatted_message = "[" + client_ip + "]: " + received_message;

    Logger::getInstance().log("Mensagem recebida de " + client_ip + ": " + received_message);
    cout << formatted_message;

    add_to_history(formatted_message);

    broadcast_message(formatted_message + "\n", client_fd);
}
```


Requisitos Específicos do Tema A (Servidor de Chat)

2.1. Servidor TCP Concorrente *Onde o servidor aceita múltiplos clientes em loop.*

```
while (true) {
    struct sockaddr_in client_address;
    socklen_t client_addr_len = sizeof(client_address);
    int client_fd = accept(server_fd, (struct sockaddr *)&client_address, &client_addr_len);

    if (client_fd < 0) {
        Logger::getInstance().log("ERRO: Falha ao aceitar nova conexao.");
        perror("accept");
        continue;
    }

    char client_ip[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &client_address.sin_addr, client_ip, INET_ADDRSTRLEN);

    thread(handle_client, client_fd, string(client_ip)).detach();
}
```

2.2. Cliente Atendido por Thread e Broadcast *Onde a thread do cliente é criada e as mensagens são retransmitidas.*

```
broadcast_message(formatted_message + "\n", client_fd);
```

2.3. Cliente CLI (Linha de Comando) *Onde o cliente envia e recebe mensagens.*

```
while (getline(cin, line)) {
    if (line == "sair") {
        break;
    }

    send(client_fd, line.c_str(), line.length(), 0);
}
```

2.4. Proteção de Estruturas Compartilhadas *Onde a lista de clientes e o histórico são protegidos durante o uso.*

```
void broadcast_message(const string& message, int sender_fd) {  
    lock_guard<mutex> lock(client_mutex);  
    for (const auto& client : clients) {  
        if (client.socket_fd != sender_fd) {  
            send(client.socket_fd, message.c_str(), message.length(), 0);  
        }  
    }  
}
```