Relatório Técnico Final – Programação Concorrente

Aluno: ANTÔNIO AUGUSTO DANTAS NETO

Matrícula: 20230012215

Projeto: Tema A - Servidor de Chat Multiusuário (TCP)

Data: 06 de Outubro de 2025

1. Diagrama de Sequência Cliente-Servidor

O diagrama a seguir ilustra o fluxo de comunicação e interação entre um cliente, o servidor principal e a thread de sessão dedicada que gerencia a conexão do cliente.

```
sequenceDiagram
    participant Client as Cliente
    participant Server as Servidor Principal
    participant ClientSession as Sessão do Cliente (Thread)

    Client->>Server: 1. Tenta se conectar na porta especificada
    Server->>Server: 2. Aceita a nova conexão (cria socket do cliente)
    Server->>ClientSession: 3. Inicia uma nova thread para a sessão do cliente
    activate ClientSession
    ClientSession->>Client: 4. Confirma que a conexão foi bem-sucedida

    loop Troca de Mensagens
        Client->>ClientSession: 5. Envia uma mensagem de texto
        ClientSession->>Server: 6. Repassa a mensagem para a função de broadcast
        Server->>ClientSession: 7. Distribui a mensagem para todas as sessões ativas
        ClientSession->>Client: 8. Recebe a mensagem (originada por si ou por outros)
    end

    Client->>ClientSession: 9. Desconecta (fecha o terminal ou envia comando /quit)
    ClientSession->>Server: 10. Notifica o gestor sobre a desconexão
    Server->>Server: 11. Remove o cliente da lista de sessões ativas
    deactivate ClientSession
    ClientSession-->>Client: 12. Conexão é formalmente encerrada
```

2. Mapeamento dos Requisitos para o Código

A tabela abaixo detalha como os requisitos obrigatórios do projeto foram implementados no código-fonte, mapeando cada funcionalidade aos arquivos e componentes responsáveis.

Requisito	Arquivo(s) de Implementação e Descrição da Solução
Servidor TCP Concorrente	**src/ChatServer.cpp** e **src/main_server.cpp**: A classe ChatServer encapsula a lógica do socket do servidor (criação, bind, listen). O main instancia e inicia o servidor.
Múltiplos Clientes (Thread por Cliente)	**src/ChatServer.cpp**: O método startAccepting() executa um loop que aceita novas conexões. Para cada cliente, uma nova ClientSession é criada e sua execução é delegada a uma std::thread, garantindo o processamento paralelo.
Broadcast de Mensagens	**src/ClientManager.cpp**: O método broadcastMessage() itera sobre o vetor de clientes ativos e envia a mensagem recebida para cada um deles, exceto o remetente original.
Logging Concorrente (libtslog)	**Integrado em todo o projeto**: A biblioteca tslog é utilizada em classes como ChatServer, ClientSession e ChatClient para registrar eventos importantes (conexões, mensagens, erros) de forma segura em um ambiente multi-thread.
Proteção de Estruturas Compartilhadas	**src/ClientManager.h** e **src/ClientManager.cpp**: A estrutura compartilhada principal, _clients (um std::vector), é protegida por um std::mutex.
Exclusão Mútua (std::mutex)	**src/ClientManager.cpp**: Todos os métodos que acessam ou modificam o vetor _clients (addClient, removeClient, broadcastMessage) utilizam um

	std::lock_guard. Isso garante o acesso atômico e previne condições de corrida.
Sockets	**src/ChatServer.cpp** e **src/ClientSession.cpp**: A API de sockets é utilizada para criar o servidor, aceitar conexões (ChatServer) e para enviar/receber dados (ClientSession).
Build (CMake)	**CMakeLists.txt**: O arquivo de build define os alvos para o servidor (chat_server), cliente (chat_client) e a biblioteca de núcleo (chat_core), gerenciando as dependências e o processo de compilação.
Tratamento de Erros e Desconexão	**src/ClientSession.cpp**: A desconexão do cliente (quando recv retorna 0) ou erros de envio (send retorna -1) são detectados, resultando na remoção segura do cliente da lista do ClientManager e no encerramento da thread.

3. Relatório de Análise de Concorrência com Auxílio de IA

Esta seção documenta a análise realizada com o auxílio de um Large Language Model (LLM) Gemini para identificar e mitigar potenciais problemas de concorrência no projeto.

Prompt de Análise:

> *"Analise o seguinte código C++ de um servidor de chat multi-thread. O ClientManager gerencia uma lista de ClientSession usando um std::vector e um std::mutex. Identifique potenciais condições de corrida (race conditions), deadlocks e problemas de starvation. Avalie a robustez da estratégia de sincronização ao adicionar, remover e iterar sobre a lista de clientes para broadcast de mensagens."*

Resumo das Sugestões da IA e Ações Tomadas:

1. **Risco Identificado: Condição de Corrida no ClientManager**
   - **Análise da IA:** O LLM identificou que a principal estrutura de dados compartilhada, o vetor _clients, seria acessada por múltiplas threads simultaneamente para operações de adição (novos clientes), remoção (clientes que se desconectam) e iteração (broadcast de mensagens). Sem um mecanismo de sincronização, essas operações poderiam levar à corrupção da memória e comportamento indefinido.

- **Mitigação Implementada:** Foi adotada a sugestão de proteger **todos** os acessos ao vetor _clients com um std::mutex. A utilização de std::lock_guard nos métodos addClient, removeClient e broadcastMessage garante que o mutex seja adquirido no início da operação e liberado automaticamente no final do escopo, prevenindo condições de corrida e garantindo a integridade dos dados.

2. **Risco Identificado: Deadlock**
   - **Análise da IA:** O modelo alertou para o risco de deadlocks caso múltiplos mutexes fossem utilizados no sistema e adquiridos em ordens inconsistentes entre as threads.
   - **Mitigação Implementada:** O projeto foi mantido com uma estratégia de bloqueio simples, utilizando um único mutex principal no ClientManager. Isso elimina a possibilidade de deadlocks cíclicos por aquisição de múltiplos locks, já que não há um segundo mutex para ser adquirido em ordem diferente.
3. **Risco Identificado: Tratamento de Desconexões Abruptas**
   - **Análise da IA:** Foi levantada a questão de como o servidor lidaria com um cliente que se desconecta abruptamente. Se o cliente não fosse removido da lista, o servidor tentaria continuamente enviar mensagens para um socket inválido, desperdiçando recursos e gerando erros.
   - **Mitigação Implementada:** A lógica em ClientSession::handleClient foi projetada para tratar essa situação. Quando recv() retorna 0 (conexão fechada pelo cliente) ou send() falha (por exemplo, com um "Broken pipe"), a sessão do cliente aciona sua própria remoção da lista gerenciada pelo ClientManager. O teste de estresse automatizado validou que essa abordagem funciona de maneira robusta, com o servidor se auto-recuperando de múltiplas desconexões simultâneas sem travar.