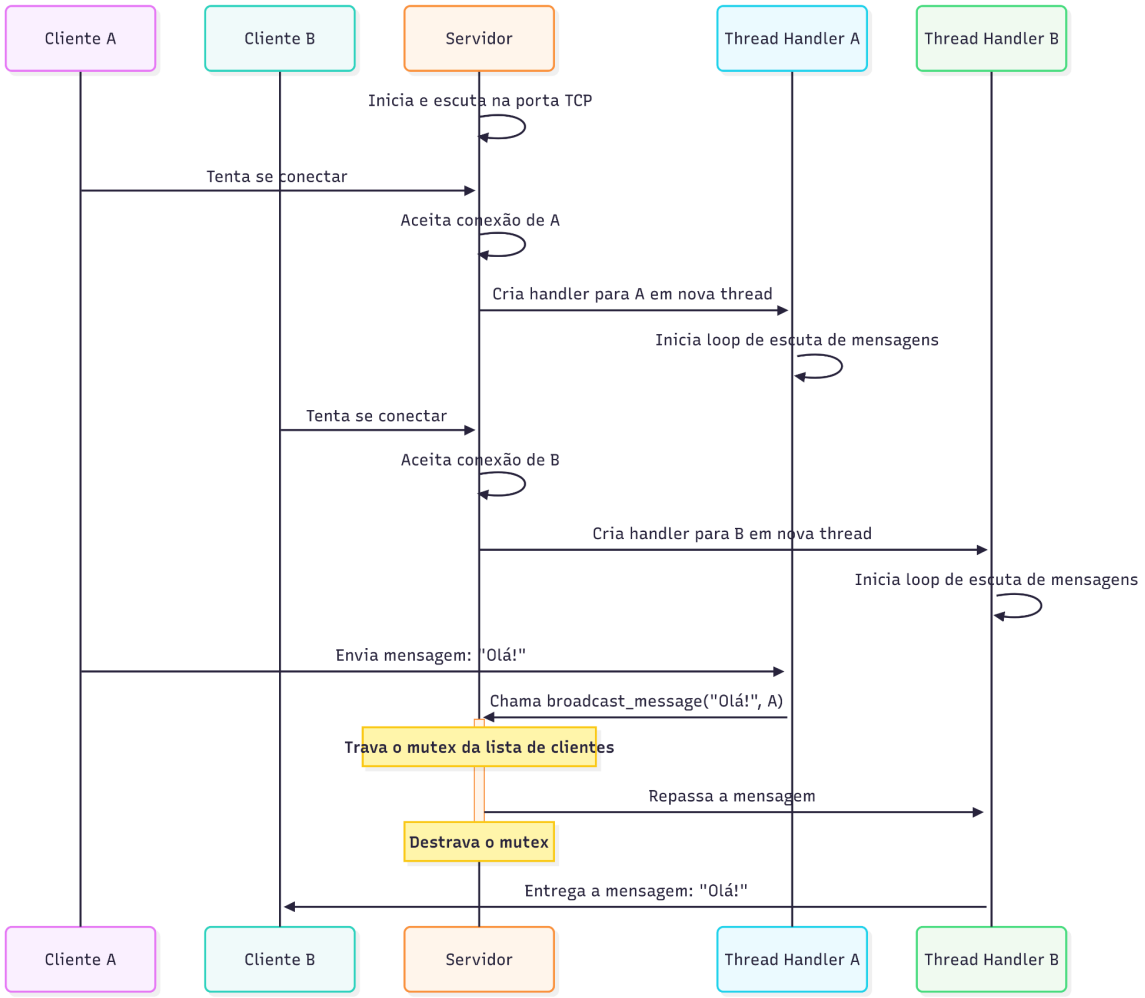
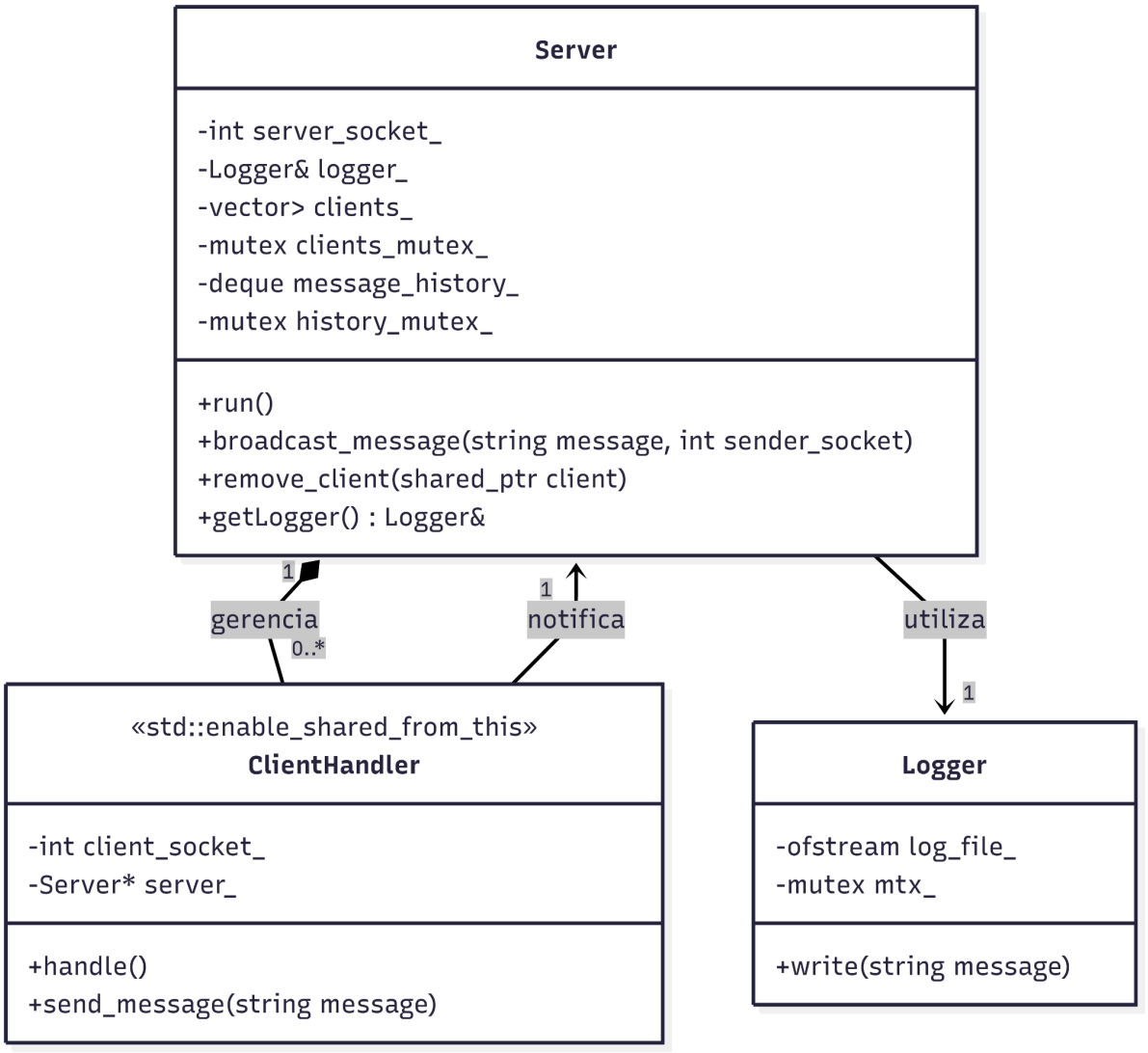
Relatório Projeto Final LPII

Maria Clara de Melo Queiroz - 20240036767
Tema: Servidor de Chat Multiusuário (TCP)

1- Diagrama de Sequência cliente-servidor e Classes





2- Mapeamento de requisitos -> código

Requisito	Implementação no código
Servidor TCP concorrente	Classe `Server` (`Server.h`, `Server.cpp`) que utiliza sockets para aceitar múltiplas conexões
Cada cliente atendido por Thread	Em `Server::accept_connections`, uma nova `std::thread` é criada para cada

	`ClientHandler`
Mensagens retransmitidas (Broadcast)	Método `Server::broadcast_message` que itera sobre a lista de clientes
Logging concorrente de mensagens	Biblioteca `libtslog` (`tslog.h`, `tslog.cpp`) utilizada em todo o sistema para registrar eventos
Cliente CLI para conectar e enviar/receber	Funcionalidade testada com clientes `telnet` e simulada com scripts (`run_clients.sh`)
Proteção de estruturas compartilhadas	`std::mutex clients_mutex_` e `std::mutex history_mutex_` na classe `Server`

3- Relatório de Análise com IA

3.1 - Ferramenta e Metodologia

Para o desenvolvimento do projeto, utilizei a IA do Google, Gemini, como assistência. A metodologia consistiu em apresentar dúvidas e problemas no meu código, sempre pedindo para não me dar a resposta direta do problema, mas sim um código de exemplo para que eu pudesse desenvolver o meu próprio. Após isso, pedia um feedback sobre a forma como construí o código e se tinha algo que podia ser melhorado.

3.2 - Prompts e Interações chaves

Separei algumas perguntas que considero importantes para o projeto para apresentar:

- **Pergunta 1:** me mostre um exemplo mínimo de servidor TCP em C++ que aceite conexão de um cliente e responda uma mensagem simples
  Resumo da resposta: A IA forneceu um código-fonte completo de um servidor TCP single-thread, explicando detalhadamente cada etapa do processo de comunicação via sockets. A resposta destacou os seguintes pontos-chave:

- A sequência fundamental de chamadas de sistema necessárias para estabelecer um servidor: socket(), bind(), listen(), e accept().
- A configuração da estrutura sockaddr_in para definir o endereço IP e a porta do servidor, incluindo a necessidade de usar htons() para a conversão correta da ordem dos bytes (endianness).
- A natureza bloqueante da chamada accept(), que pausa a execução do programa até que um cliente se conecte, e seu retorno, que é um novo descritor de arquivo exclusivo para a comunicação com aquele cliente.
- O uso da chamada send() no novo descritor de arquivo do cliente para transmitir dados, e a chamada close() para encerrar tanto o socket do cliente quanto o do servidor ao final.

Esse exemplo serviu para formular a classe Server do projeto, que usou essa lógica de uma forma mais robusta e orientada a objetos para suportar a concorrência.

- **Pergunta 2:** Ao executar o servidor, recebi o erro 'Erro fatal: std::bad_cast'. O erro ocorreu após a implementação da remoção de clientes com std::enable_shared_from_this. Meu código da função accept_connections é este: (meu código). O que pode estar causando isso?

  Tirei o código pois ficou extenso

  Resumo da resposta: "A IA explicou que o erro std::bad_cast neste contexto é sintomático de uma chamada shared_from_this() em um objeto não gerenciado por um shared_ptr. Ela sugeriu duas possíveis causas: um problema de compilação com arquivos antigos ou um erro na declaração da classe ClientHandler. A sugestão de rodar make clean && make e verificar a herança de std::enable_shared_from_thisemClientHandler.h foi a chave para resolver o problema.

3.3 - Conclusão sobre eficiência

O uso da IA acelerou significativamente o desenvolvimento e, mais importante, aprofundou o entendimento sobre tópicos complexos. A capacidade de receber explicações detalhadas e contextuais sobre erros como std::bad_cast e sobre o uso correto de std::lock_guard para evitar deadlocks foi fundamental para garantir a robustez do código final e cumprir os objetivos da disciplina.