

# Relatório do Trabalho Prático 2 - Mensageiro Rudimentar

Arthur Barbosa de Andrade<sup>1</sup>, Fernanda das Neves Merqueades Santos<sup>1</sup>,  
Jenniffer Oliveira Checchia<sup>1</sup>

<sup>1</sup>Universidade Federal de Mato Grosso do Sul – Campus Cidade Universitária  
Curso Bacharel em Ciência da Computação  
Campo Grande, MS – Brasil

2

{arthur\_barbosa, fernanda\_neves, jenniffer\_checchia}@ufms.br

**Abstract.** *This report presents the development of a rudimentary messenger using TCP transport. It also describes the development environment, the overall project organization, and the main implemented functionalities. The system was built in C++, adopting a modular architecture composed of client, server, and interface layer subsystems responsible for the communication protocol. Furthermore, the tests performed demonstrated that the messenger is functional, robust, and allows for future evolution. The code standardization and corrections applied during the debugging process ensured a clean, consistent system aligned with good programming practices.*

**Resumo.** *Este relatório descreve o desenvolvimento de um mensageiro simples baseado em comunicação TCP, utilizando um protocolo próprio estruturado em JSON. O projeto foi implementado em C++ e dividido em três partes principais: cliente, servidor e a camada de interface com o cliente. No cliente, foram desenvolvidas as conexões, envio e recebimento assíncrono de mensagens, já no servidor, foi implementado o processamento dos comandos, gerenciamento de usuários, sessões e entrega de mensagens. Ele também apresenta alguns testes realizados, que confirmaram o funcionamento correto das funcionalidades obrigatórias e de bônus, além da robustez obtida após diversas etapas de depuração. O resultado final é um sistema funcional, organizado e preparado para extensões futuras.*

## 1. Introdução

O Trabalho 2 da disciplina de Redes de Computadores (2025/2) teve como objetivo desenvolver um mensageiro simples utilizando comunicação TCP e um protocolo próprio baseado em mensagens JSON. Todas as funcionalidades obrigatórias foram implementadas, assim como os itens bônus propostos.

Para lidar com as mensagens no formato JSON, utilizamos a biblioteca `nlohmann_json`, que facilitou bastante a criação, interpretação e validação das estruturas usadas na troca de dados entre cliente e servidor.

A organização do projeto foi distribuída em três frentes: implementação do cliente, implementação do servidor e a etapa de testes, análise da arquitetura e depuração. Cada uma delas foi desenvolvida de forma modular, priorizando clareza e manutenibilidade, em alinhamento com as orientações do trabalho.

## 2. Desenvolvimento

### 2.1. Organização do Projeto

A organização do projeto segue uma estrutura modular que separa claramente os papéis do cliente, servidor e componentes compartilhados. Assim, o sistema é dividido da seguinte forma:

- `client/` — Implementa toda a lógica do cliente TCP, incluindo a classe `Client` e a camada de interface em `interface.cpp`. Esse módulo lida com a conexão ao servidor, envio e recebimento de mensagens JSON, as threads utilizadas no cliente (CLI e thread receptora) e as funções auxiliares relacionadas ao framing.
- `server/` — Contém a implementação completa do servidor. Inclui o ponto de entrada `main.cpp`, a classe `Server` responsável por aceitar conexões e criar threads dedicadas para cada cliente, além do módulo `command_handler` que processa os comandos enviados.
- `common/` — Reúne os módulos compartilhados entre cliente e servidor, como o protocolo de comunicação (`protocol.hpp/cpp`) e utilitários de socket (`socket_utils.hpp/cpp`). Esses componentes centralizam funcionalidades essenciais e evitam duplicação de código.
- `libs/` — Armazena dependências externas utilizadas pelo projeto, como a biblioteca `nlohmann_json`, empregada para manipulação das mensagens no formato JSON.

### 2.2. Implementação do Servidor

Para assegurar o isolamento entre as camadas, a estrutura do servidor é dividida em:

- `server.cpp`: organiza e gerencia as threads modulares;
- `command_handler.cpp`: abriga a lógica de aplicação, interpretando comandos, validando estados e gerenciando usuários, sessões e mensagens pendentes;
- `main.cpp`: ponto de inicialização, cria a instância do servidor e inicia sua execução.

A aplicação segue um modelo multi-threaded composto por:

- **Thread principal**: inicializa o socket e executa `bind()` e `listen()`;
- **Thread acceptor**: aceita novas conexões;
- **Thread worker**: recebe mensagens, as processa via *CommandHandler*, envia respostas e monitora desconexões.

Vale ressaltar que a thread worker é dedicada, ou seja, uma nova thread é criada para cada cliente conectado.

Uma estrutura global também é mantida de modo que haja controle de usuários registrados, sessões ativas, filas de mensagens pendentes e mapeamentos `nickname-socket` e `socket-nickname`. A manipulação dessas estruturas é protegida por `mutex`, evitando *race conditions*.

A arquitetura inicia com a função `Server::start()`, responsável pela configuração do ambiente de rede: criação do socket TCP, habilitação de reutilização

rápida de porta e conversão para *network byte order*. Após a associação do socket à porta (`bind()`) e o início do modo de escuta (`listen()`) com fila de até 10 conexões, o servidor confirma a inicialização exibindo:

```
[Server] Mensageiro iniciado na porta TCP: <porta>.
```

Uma vez iniciado, o servidor dedica uma thread para `acceptorLoop()`, que permanece temporariamente bloqueada aguardando conexões. Quando um cliente se conecta, o servidor obtém seu endereço IP, registra o evento e cria uma thread worker dedicada, imediatamente desanexada, permitindo que a thread acceptor retorne à espera por novas conexões.

A thread worker configura o `client_sockfd` para o modo não bloqueante, evitando travamentos enquanto aguarda dados e permitindo detecção eficiente de desconexões. Em seu loop principal, tenta receber mensagens completas do cliente; quando uma mensagem é recebida, ela é enviada ao *CommandHandler*, que processa o comando JSON e gera a resposta correspondente. A detecção de desconexão utiliza:

```
recv(client_sockfd, &test, 1, MSG_PEEK | MSG_DONTWAIT)
```

onde um retorno igual a zero indica que o cliente encerrou a conexão.

Em caso de desconexão — voluntária ou por erro — a função `cleanupSession(client_sockfd)` é acionada, envolvendo a recuperação do nickname associado ao socket, marcação do usuário como offline, remoção da sessão das estruturas globais e fechamento seguro do socket. Toda a lógica de aplicação reside no *CommandHandler*, que processa comandos como REGISTER, LOGIN, LOGOUT, LIST\_USERS, DELETE\_USER e SEND\_MSG. No último caso, se o destinatário estiver offline, a mensagem é armazenada em uma fila (`messageQueues[nick]`), implementando um mecanismo de *store-and-forward*.

Quando o usuário volta a ficar online, o sistema garante a entrega de mensagens pendentes por meio de `deliverPendingMessages(client_sockfd, nickname)`, chamada durante o comando LOGIN, que esvazia a fila e envia todas as mensagens ao cliente.

Por fim, a concorrência é gerenciada com o uso de `std::mutex`, protegendo o acesso às estruturas de estado globais (mapas de sessões, usuários e filas de mensagens), assegurando integridade dos dados em ambiente multithread.

### 3. Implementação do Cliente

A implementação do cliente segue uma arquitetura modular composta por duas unidades principais:

- `client.cpp`: responsável pela lógica fundamental do cliente, incluindo a conexão com o servidor, o envio e recebimento de mensagens, o controle das sessões e o gerenciamento das threads dedicadas.
- `interface.cpp`: camada de interação com o usuário, responsável por interpretar os comandos digitados, exibir as respostas recebidas e conduzir o fluxo geral do mensageiro.

O arquivo `client.cpp` gerencia toda a comunicação do lado do cliente. Ele cria o socket TCP, estabelece a conexão com o servidor e cuida do envio e recebimento das mensagens — ou seja, tudo o que o usuário manda ou recebe passa por ele.

As mensagens são trocadas no formato JSON, utilizando o caractere `\n` como delimitador. O `client.cpp` também oferece funções para enviar mensagens de forma bloqueante (esperando até o envio terminar) ou sem travar a interface. Além disso, ele transforma os dados recebidos em estruturas adequadas para uso pelos outros módulos do programa.

Já o módulo `interface.cpp` é a parte que o usuário interage. Ele interpreta os comandos digitados, organiza tudo em uma estrutura interna padronizada e envia a solicitação para o cliente. Também cuida de exibir mensagens de ajuda, erros, confirmações e respostas formatadas.

A aplicação funciona com duas threads principais:

- **Thread da interface (CLI):** responsável por ler os comandos do usuário, interpretar cada um e repassá-los ao servidor usando o módulo `client`.
- **Thread receptora:** dedicada exclusivamente a receber mensagens do servidor, garantindo que respostas, notificações e mensagens de outros usuários apareçam imediatamente, sem depender de uma nova ação do cliente.

A thread receptora foi essencial para resolver o problema de atraso nas mensagens, que fazia as respostas só aparecerem depois que o usuário digitava outro comando.

A conexão com o servidor é feita por `Client::connectToServer()`, que cria o socket, chama `connect()`, configura o uso de `\n` como delimitador e inicia a thread responsável por ouvir o servidor. O envio de mensagens usa JSON seguindo o protocolo definido em `common/protocol.hpp`, permitindo que comandos como `login`, `logout`, `send`, `list` e `delete` sejam convertidos pela interface e enviados de forma padronizada.

O gerenciamento da sessão também foi ajustado para evitar problemas. Um dos erros iniciais era o fechamento imediato do socket após o `logout`, o que deixava o terminal inutilizável. Isso foi corrigido, permitindo que o usuário continue na interface mesmo após sair da conta. O comportamento do comando `delete` também foi revisado para funcionar corretamente com o servidor.

O fluxo de mensagens recebidas usa leitura não bloqueante e separação por delimitador, garantindo que cada mensagem completa seja tratada antes de chegar à interface. A thread receptora fica ativa enquanto a conexão estiver aberta, monitorando desconexões e exibindo avisos quando necessário.

Por fim, a separação entre *interface* e *cliente* mantém as responsabilidades bem definidas: a interface cuida da experiência do usuário, enquanto a classe `Client` lida com toda a comunicação com o servidor, controle do socket, threads e conversão das mensagens JSON.

### 3.1. Testes e Resultados

Inicialmente, a thread responsável por receber respostas do servidor só imprimia as mensagens após uma nova entrada ser digitada no terminal. Isso ocorria porque a thread de recepção dependia do loop principal para liberar o buffer de saída.

Para corrigir esse comportamento, foram aplicados os seguintes ajustes:

- reorganização da thread de recepção para impressão imediata das respostas;
- desacoplamento entre leitura do teclado e leitura do socket;
- tratamento de fim de conexão para evitar acesso a sockets fechados.

Após essas alterações, as mensagens passaram a aparecer no momento correto, garantindo uma experiência mais próxima de clientes reais de chat.

Foram realizados testes manuais cobrindo os comandos principais: `register`, `login`, `logout`, `delete`, `list` e `send`. Em conjunto com o servidor, verificou-se que:

- o parser do cliente está robusto e impede usos incorretos;
- mensagens JSON são montadas de forma consistente;
- a interação pós-logout foi corrigida com sucesso;
- não ocorrem mais travamentos ao receber respostas do servidor;
- erros são tratados e exibidos adequadamente ao usuário.

## 4. Conclusão

O desenvolvimento deste mensageiro rudimentar permitiu a aplicação prática dos conceitos de redes de computadores, consolidando o entendimento sobre a comunicação via transporte TCP e a estruturação de protocolos de aplicação. A arquitetura modular adotada, separando claramente as responsabilidades entre cliente, servidor e interface, aliada ao uso da biblioteca `nlohmann.json`, mostrou-se eficaz para garantir a organização e a manutenibilidade do código em C++.

Um dos principais desafios superados durante o projeto foi o gerenciamento da concorrência e o tratamento de operações bloqueantes de I/O. A implementação de um modelo *multi-threaded* robusto (com a *thread* receptora no cliente desacoplada da leitura do teclado e *threads worker* dedicadas no servidor) foi essencial para solucionar os problemas iniciais de atraso na exibição de mensagens. Além disso, o uso correto de *mutexes* para proteção das estruturas globais assegurou a integridade dos dados, prevenindo *race conditions* no gerenciamento de sessões e filas de mensagens.

Em suma, o sistema atende a todos os requisitos funcionais e bônus propostos, incluindo o mecanismo de *Store-and-Forward* para entrega de mensagens pendentes. O resultado é uma aplicação estável, que não apenas cumpre os objetivos da disciplina, mas também serve como uma base sólida para futuras implementações, demonstrando a importância de boas práticas de engenharia de software no desenvolvimento de sistemas distribuídos.

## Agradecimentos

Agradecemos ao professor Irineu Sotoma pela orientação e suporte durante a disciplina.

## References

- [1] [cppreference.com](https://en.cppreference.com/), *C++ Reference: STL, Threads e Sockets*. Disponível em: <https://en.cppreference.com/>. Acesso em: 25 nov. 2025.

- [2] Niels Lohmann, *JSON for Modern C++ – Repositório GitHub*. Disponível em: <https://github.com/nlohmann/json>. Acesso em: 25 nov. 2025.
- [3] Linux man-pages project, *System Calls and Networking API Documentation*. Disponível em: <https://man7.org/linux/man-pages/>. Acesso em: 25 nov. 2025.
- [4] James F. Kurose; Keith W. Ross, *Redes de Computadores e a Internet – Abordagem Top-Down (referência técnica)*. Disponível em: <https://www.pearson.com/>. Acesso em: 25 nov. 2025.