# Sistema de Chat Distribuído com Múltiplos Clientes


- Douglas Cabral Pereira de Araújo
- Lucas Mateus Alves Luna


## Resumo


Este artigo apresenta o desenvolvimento de um sistema de chat distribuído baseado na arquitetura cliente-servidor implementado com Node.js e TypeScript. A solução suporta múltiplos usuários simultâneos organizados em salas, presença em tempo real e canal de transferência de arquivos. Foram aplicados conceitos de concorrência (event loop, filas assíncronas, sincronização de broadcast) e paralelismo (Worker Threads e Cluster API) para garantir escalabilidade e disponibilidade na comunicação. São relatadas as principais decisões arquiteturais, o fluxo de mensagens, o desenho dos componentes, bem como resultados preliminares de testes funcionais e de carga. O trabalho visa demonstrar, em um cenário acadêmico, a aplicação prática de tópicos de sistemas distribuídos.


## 1. Introdução

A popularização de aplicações colaborativas em tempo real demanda arquiteturas distribuídas robustas, capazes de lidar com múltiplos clientes, baixa latência e falhas intermitentes. No contexto da disciplina de Sistemas Distribuídos, propõe-se o desenvolvimento de um chat multiusuário que consolide os conceitos estudados em sala: comunicação via sockets, concorrência, paralelismo e sincronização. O objetivo deste artigo é documentar a implementação realizada, justificando decisões técnicas, analisando resultados e apontando futuras evoluções.

Este documento está organizado da seguinte forma: a Seção 2 descreve a metodologia adotada e os componentes da solução; a Seção 3 apresenta os resultados obtidos, incluindo evidências de funcionamento e testes; a Seção 4 resume as conclusões e trabalhos futuros.

## 2. Metodologia

### 2.1 Visão Geral da Arquitetura

O sistema segue o modelo cliente-servidor utilizando WebSockets sobre HTTP. O servidor centraliza a orquestração de salas, gerenciamento de sessões, presença e broadcast de mensagens. Os clientes interagem por meio de uma interface em linha de comando (CLI) que fornece comandos para ingresso em salas, envio de mensagens e transferência de arquivos.

- **Servidor**: composto por módulos especializados (`connection-manager`, `room-manager`, `message-broker`, `file-processor`) que encapsulam responsabilidades e promovem coesão. O servidor expõe um endpoint

HTTP simples para verificação de saúde e promove o upgrade para WebSocket.

- **Cliente**: implementado em TypeScript, utiliza `readline/promises` para a CLI e `ws` como biblioteca WebSocket. Suporta reconexão automática, heartbeat periódico e comandos para gerenciar salas.
- **Shared**: camada de recursos compartilhados responsável pelo protocolo de mensagens JSON (tipos, validação e serialização), constantes e carregamento de configuração via `.env`.

> ***Figura 1.*** *Diagrama da arquitetura cliente-servidor (inserir figura em* `docs/imagens/arquitetura.png`*).*

## 2.2 Concorrência e Paralelismo Aplicados

- **Concorrência**: uso intensivo do event loop do Node.js para lidar com múltiplas conexões simultâneas sem bloqueio. O `MessageBroker` utiliza filas assíncronas e `Promise.all` para sincronizar o broadcast de mensagens por sala.
- **Paralelismo**: o módulo `file-processor` instancia um pool de Worker Threads para compressão de arquivos antes da transmissão, preparando o caminho para tratamento de dados CPU-bound. A `Cluster API` foi configurada para habilitar múltiplos processos de servidor, escalando horizontalmente conforme o número de núcleos disponíveis.
- **Sincronização**: os acessos às estruturas de dados (por exemplo, salas e sessões) são serializados através de mapas imutáveis dentro do event loop principal. O envio de mensagens utiliza uma fila customizada (`serialization-queue`) que garante ordem e evita condições de corrida na escrita do socket.

## 2.3 Protocolo de Comunicação

As mensagens são serializadas em JSON e possuem os campos: `type`, `sender`, `room`, `timestamp`, `content` e `metadata`. Os tipos implementados incluem `MESSAGE`, `JOIN`, `LEAVE`, `HEARTBEAT`, `FILE`, `PRESENCE`, `SYSTEM` e `ERROR`. O módulo `shared/protocol.ts` realiza a validação dos payloads recebidos e padroniza a geração de timestamps e envelopes de mensagens.

## 2.4 Ferramentas e Stack Tecnológico

- **Linguagem**: Node.js 18+ com TypeScript (transpilação via `tsc`).
- **Bibliotecas**: `ws` (WebSocket), `winston` (logging), `chalk` (CLI), `dotenv` (configuração), `uuid` (identificadores), `artillery` (carga), `jest/ts-jest` (testes unitários), `tsx` (desenvolvimento com reload).
- **Execução**: scripts npm para desenvolvimento (`dev:server`, `dev:client`), produção (`start:server`, `cluster`) e testes (`test`, `test:load`).
- **Configuração**: `.env.example` documenta as variáveis principais, incluindo porta padrão (9090), limites de conexão, intervalos de heartbeat e diretórios de log.

# 3. Resultados

## 3.1 Funcionalidades Implementadas

- Autenticação por nickname único com validação de duplicidades.
- Sala geral padrão e suporte à criação dinâmica de salas adicionais.
- Broadcast de mensagens com identificação de remetente e carimbo de tempo ISO 8601.
- Sistema de presença com notificações de entrada, saída e desconexão forçada por timeout.
- Cliente CLI com comandos `/join`, `/leave`, `/rooms`, `/sendfile` e `/quit`.
- Transferência de arquivos no cliente com divisão em chunks, checksum por fragmento e feedback de progresso na CLI.
- Pool de Worker Threads para compressão (pronto para integração ao pipeline de upload).
- Script de teste de carga (`artillery.yml`) que simula conexões WebSocket, heartbeat e troca de mensagens.

## 3.2 Evidências de Execução

- A execução do servidor (`npm run dev:server`) exibe logs estruturados indicando conexões, autenticações e timeouts detectados.
- Clientes múltiplos podem ser executados simultaneamente (`npm run dev:client -- --nickname usuarioX`) demonstrando isolamento entre salas e atualizações de presença.
- O comando `/sendfile caminho sala` evidencia o pipeline de envio com percentual de progresso emitido na interface.

> ***Figura 2.*** *Captura de tela da CLI exibindo mensagens, presença e progresso de upload (inserir figura em `docs/imagens/cli.png`).*

## 3.3 Testes Realizados

- **Teste funcional**: execução local com três clientes simultâneos validando broadcast, heartbeat e reconexão após desligamento do servidor.
- **Teste de carga preliminar**: execução de `npm run test:load` (Artillery) por 60 segundos com taxa de chegada de 5 usuários/s, confirmando estabilidade do servidor e ausência de erros de protocolo.
- **Cobertura automatizada**: placeholders configurados em `tests/` para futura expansão dos cenários unitários.

## 3.4 Métricas e Observações

- Latência média observada nas mensagens em ambiente local permaneceu abaixo de 50 ms.
- O consumo de memória por cliente manteve-se inferior a 50 MB durante os testes preliminares.
- A compressão de arquivos em Worker Threads ainda não foi integrada ao fluxo completo; a etapa encontra-se preparada e será medida em trabalhos futuros.

# 4. Conclusão

A implementação do chat distribuído atendeu ao objetivo principal de aplicar conceitos de arquitetura distribuída, concorrência e paralelismo em um cenário prático. A modularização do servidor, aliada à padronização do protocolo, favoreceu a extensibilidade e a clareza na manutenção do código. Os testes executados demonstram a viabilidade da solução e apontam caminhos para evolução, tais como: finalizar o ciclo completo de transferência de arquivos (compressão + persistência no servidor), ampliar a suíte de testes automatizados, incorporar métricas e dashboards de monitoramento e produzir as figuras/diagramas finais para o artigo.

Trabalhos futuros incluem a implementação de uma interface TUI com `blessed`, coleta sistemática de métricas de desempenho, integração contínua e documentação fotográfica para composição do PDF final exigido na avaliação.

# Referências


1. Node.js Documentation. *Cluster Module*, *Worker Threads*. Disponível em: https://nodejs.org/api/ (https://nodejs.org/api/).
2. Tilkov, S., & Vinoski, S. (2010). Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14(6), 80-83.
3. Artillery Documentation. Disponível em: https://www.artillery.io/docs (https://www.artillery.io/docs).
4. Winston Logger. Disponível em: https://github.com/winstonjs/winston (https://github.com/winstonjs/winston).
5. WS WebSocket Library. Disponível em: https://github.com/websockets/ws (https://github.com/websockets/ws).