Relatório — Etapa 3 (v3-final)

Projeto: Servidor de Chat Multiusuário (Tema A) — *Etapa 3: Sistema completo* **Aluno:** yves ribeiro de sena
Data: 06/10/2025

Resumo

A terceira etapa do projeto teve como objetivo concluir o desenvolvimento do servidor de chat concorrente iniciado nas etapas anteriores. Nesta fase, foram integrados todos os componentes do sistema, com foco na sincronização entre threads, na organização das mensagens por meio de uma fila thread-safe e na análise crítica dos mecanismos de concorrência.

O resultado é um servidor capaz de lidar com múltiplos clientes simultaneamente, garantindo o envio ordenado de mensagens, o registro de logs de forma segura e o encerramento controlado das threads e recursos.

FLUXO DE MENSAGENS

Etapa	Descrição
1. Cliente → Servidor	Conexão via soquete TCP
2. Servidor	Cria thread dedicada para o cliente
3. Cliente	Envia mensagem para o servidor
4. Servidor	Broadcast da mensagem para todos os clientes conectados
5. Logger	Registro assíncrono das mensagens no arquivo de log

Resumo — Etapa 3: Integração e Testes do Servidor de Chat Concorrente

A **Etapa 3** teve como objetivo consolidar todas as partes desenvolvidas nas etapas anteriores, resultando em um **sistema de chat TCP totalmente funcional e concorrente**, capaz de permitir a comunicação simultânea entre vários clientes conectados a um único servidor.

Durante esta fase, foram realizadas **integrações, melhorias de sincronização e testes automatizados**, garantindo o funcionamento correto do sistema em ambiente multi-thread.

Principais Implementações

- **Servidor multi-thread**: cada cliente conectado é tratado por uma thread independente, criada com `pthread_create()` e liberada com `pthread_detach()`.
- **Controle de concorrência**: uso de `pthread_mutex_t` para evitar race conditions na lista de clientes e no sistema de logs.
- **Comunicação TCP confiável**: o servidor utiliza `socket()`, `bind()`, `listen()` e `accept()` para gerenciar conexões.
- **Broadcast de mensagens**: o servidor replica as mensagens recebidas de um cliente para todos os outros conectados.
- **Logger thread-safe (`libtslog`)**: registra todos os eventos (conexões, mensagens, desconexões) em arquivo com controle de exclusão mútua.
- **Limite de conexões**: variável `MAX_CLIENTES` define o número máximo permitido, com mensagens de aviso caso o limite seja atingido.
- **Script de teste automatizado**: o arquivo `test_chat.sh` inicia o servidor e vários clientes de forma simultânea, simulando uma conversa real.

`cliente.c`

Implementa a lógica de funcionamento do cliente.

- Se conecta ao servidor via IP e porta.
- Envia e recebe mensagens pelo socket.
- Inclui funções chamadas por `main_cliente.c`.

🔹 `main_cliente.c`

Arquivo principal do **cliente** (contém `main()` do cliente).

- Inicia o cliente com `cliente_init()` e `cliente_conectar()`.
- Cria uma thread para **receber mensagens** continuamente do servidor.
- Lê mensagens do usuário e as envia com `cliente_enviar()`.

🔹 `main_servidor.c`

Arquivo principal do **servidor** (contém `main()` do servidor).

- Inicializa o servidor (`servidor_init()`),
  executa (`servidor_executar()`),
  e gerencia o encerramento.
- É o ponto de entrada do servidor.

🔹 `servidor.c`

Implementa toda a **lógica do servidor**.

- Aceita conexões de clientes (via `accept()`).
- Cria threads para tratar cada cliente.
- Realiza **broadcast** de mensagens entre clientes.
- Usa mutexes para proteger a lista de clientes.
- Remove clientes desconectados e registra logs.

Análicse Crítica com IA

⚠️ **Pontos Fracos / Limitações Identificadas**

Categoria	Descrição	
Identificação de clientes	A associação entre "socket" e "ID lógico do cliente" ainda é confusa. O primeiro cliente conectado pode ser identificado como "Cliente 5", por exemplo.	Médio — dificulta rastreamento e depuração.
Ordem de logs	Algumas mensagens de log são registradas fora da ordem esperada (ex: "Mensagem enviada" antes da confirmação de conexão).	Baixo — apenas afeta clareza dos registros.
Broadcast simples	O envio de mensagens é direto via loop de `send()`, sem confirmação de entrega.	Médio — em redes instáveis, pode ocorrer perda parcial de mensagens.
Ausência de fila de mensagens (buffer compartilhado)	O servidor trata mensagens diretamente nas threads. Uma fila (thread-safe) permitiria desacoplar recepção e envio.	Médio — reduz escalabilidade e controle de fluxo.
Encerramento abrupto	O término do servidor com `kill` impede a liberação ordenada de recursos e threads.	Alto — pode causar vazamentos de memória ou sockets abertos.

Requisito	Arquivo	Implementação
Threads	`servidor.c`	`pthread_create()` e `pthread_detach()` para cada cliente
Exclusão Mútua (Mutex)	`servidor.c`, `libtslog.c`	`pthread_mutex_t` para proteger lista de clientes e logs
Sockets	`servidor.c`, `cliente.c`	`socket()`, `bind()`, `listen()`, `accept()`, `connect()`
Broadcast	`servidor.c`	Função `broadcast()` envia mensagem a todos os clientes conectados
Logs Concorrentes	`libtslog.c`	`tslog_write()` com `pthread_mutex_lock()`
Identificação de Clientes	`servidor.c`	Nome atribuído automaticamente ao conectar (`Cliente N`)
Limite de Conexões	`servidor.c`	Constante `MAX_CLIENTES` e verificação em `adicionar_cliente()`
Thread de Recepção	`main_cliente.c`	`pthread_create()` para receber mensagens sem travar entrada do usuário
Script de Teste Automatizado	`test_chat.sh`	Cria servidor e múltiplos clientes simultaneamente
Encerramento Seguro	`servidor.c`, `cliente.c`	Fechamento de sockets e threads com mensagens de log