Relatório de Análise de Concorrência com Auxílio de IA

Servidor de Chat Multiusuário TCP - Etapa 3

Disciplina: Programação Concorrente (C/C++) **Aluno:** Eduardo Asfuri Carvalho **Matrícula:** 20230046939
Data: 04 de Outubro de 2025 **Ferramenta de IA:** Claude (Anthropic) via Perplexity AI

1. Introdução

Este relatório documenta o processo de identificação, análise e mitigação de problemas de concorrência no desenvolvimento de um servidor de chat multiusuário TCP. O sistema foi desenvolvido de forma iterativa em 3 etapas, utilizando consultas a uma IA (Claude/Perplexity) para auxiliar na identificação de race conditions, deadlocks e outros problemas típicos de sistemas concorrentes.

1.1 Visão Geral do Sistema

O sistema implementa:

- Servidor TCP concorrente (porta 8080)
- Uma thread dedicada por cliente conectado
- Broadcasting de mensagens para todos os clientes
- Histórico de mensagens thread-safe (monitor)
- Logging concorrente (padrão Producer-Consumer)
- Cliente CLI com interface interativa

2. Metodologia de Análise com IA

2.1 Abordagem Iterativa

O desenvolvimento seguiu uma abordagem incremental:

1. **Etapa 1:** Implementação da biblioteca de logging thread-safe
2. **Etapa 2:** Protótipo TCP básico com broadcasting
3. **Etapa 3:** Histórico de mensagens e melhorias de qualidade

Em cada etapa, problemas de concorrência foram identificados através de:

- Revisão manual do código
- Consultas direcionadas à IA
- Testes de stress com múltiplos clientes

2.2 Tipos de Prompts Utilizados

Os prompts seguiram três categorias principais:

- **Identificação de problemas:** "Analise este código e identifique race conditions..."
- **Validação de soluções:** "Esta implementação com mutex está correta?"
- **Sugestões de melhorias:** "Como implementar um monitor thread-safe para..."

---

3. Problemas de Concorrência Identificados

3.1 Race Condition: Lista de Clientes Conectados

Contexto do Problema

Durante a Etapa 2, foi identificado que múltiplas threads (uma por cliente) acessavam simultaneamente o vector de clientes conectados sem sincronização adequada.

Prompt Utilizado

> **Usuário:** "Quando eu rodo make test-tcp, as mensagens não tomam controle exclusivo, logo os logs ficam assim: 'Cliente 13: Cliente 9 - Mensagem 1 Cliente 6: Cliente 1 - Mensagem 2...' Em vez de haver uma mensagem por vez, no meio de uma mensagem ele pode trocar para outra, causando uma condição de corrida."

Resposta da IA

A IA identificou que o problema tinha duas causas:

Causa 1 - Broadcasting sem proteção:

```
// CÓDIGO PROBLEMÁTICO (antes da correção)
void broadcastMessage(const std::string& message, int senderSocket) {
    for (int socket : clientSockets) {  // × Acesso sem mutex
        if (socket != senderSocket) {
            send(socket, message.c_str(), message.length(), 0);
        }
    }
}
```

Causa 2 - std::cout não thread-safe no cliente:

```
// CÓDIGO PROBLEMÁTICO
void receiveMessages() {
    // ...
    std::cout << buffer << std::endl;  // × Múltiplas threads escrevendo
}
```

Análise Crítica

A resposta da IA foi **precisa e completa**. Identificou corretamente que:

1. O vector `clientSockets` era acessado concorrentemente durante broadcast
2. `std::cout` não é thread-safe e causava intercalação de caracteres
3. Era necessário proteger ambas as regiões críticas com mutex

A IA também sugeriu usar `std::lock_guard` ao invés de lock/unlock manual, seguindo boas práticas RAII.

Mitigação Implementada

Solução 1 - Proteção do broadcasting:

```
void broadcastMessage(const std::string& message, int senderSocket) {
    std::lock_guard<std::mutex> lock(clientsMutex);  // Região crítica
protegida

    for (int socket : clientSockets) {
        if (socket != senderSocket) {
            send(socket, fullMessage.c_str(), fullMessage.length(), 0);
        }
    }
}
```

Solução 2 - Proteção do std::cout:

```
// tcp_client.cpp
static std::mutex coutMutex;  // Mutex global para cout

void receiveMessages() {
    // ...
    {
        std::lock_guard<std::mutex> lock(coutMutex);  // Atomicidade
        std::cout << buffer << std::endl;
    }
}
```

Resultado: Após as correções, o teste de stress com 10 clientes simultâneos passou a funcionar corretamente, sem intercalação de mensagens.

3.2 Race Condition: Framing de Mensagens TCP

Contexto do Problema

Durante testes de stress, foi observado que múltiplas mensagens apareciam "coladas" na mesma linha, sem separação visual.

Prompt Utilizado

> **Usuário:** "Realmente não há [race condition nos mutex], porém há várias linhas que não têm parágrafo, ficando coladas."

Resposta da IA

A IA diagnosticou que o problema não era uma race condition clássica, mas sim **falta de delimitador de mensagens**:

> "O efeito de 'linhas coladas' vem do fato de o protocolo estar sem delimitador de mensagem e do TCP ser orientado a fluxo, ou seja, vários envios podem chegar juntos no mesmo recv, sem separação automática entre mensagens."

A IA explicou que TCP não preserva fronteiras de mensagens (message boundaries), logo era necessário implementar um protocolo de framing.

Análise Crítica

A resposta da IA foi **excelente**. Ela:

1. Diferenciou entre race condition (problema de sincronização) e problema de protocolo
2. Explicou o conceito de TCP como stream-oriented
3. Sugeriu usar `\n` como delimitador (line framing protocol)
4. Forneceu código completo para parsing por linhas

Esta análise demonstrou conhecimento profundo de redes TCP/IP, não apenas de concorrência.

Mitigação Implementada

Solução 1 - Envio com delimitador:

```
void sendMessage(const std::string& message) {
    std::string out = message;
    if (out.empty() || out.back() != '\n') {
        out.push_back('\n');  // Garante delimitador
    }
    send(clientSocket, out.data(), out.size(), 0);
}
```

Solução 2 - Parsing no cliente:

```
void receiveMessages() {
    std::string acc;  // Buffer acumulado
    char buf[1024];

    while (true) {
        int n = recv(clientSocket, buf, sizeof(buf), 0);
        acc.append(buf, n);

        size_t pos;
```

```
        while ((pos = acc.find('\n')) != std::string::npos) {  // Parsing
por linha
            std::string line = acc.substr(0, pos);
            acc.erase(0, pos + 1);

            std::cout << line << std::endl;  // Mensagem completa
        }
    }
}
```

Resultado: Mensagens agora aparecem sempre completas e separadas, mesmo sob alta carga.

3.3 Deadlock Potencial: Histórico de Mensagens

Contexto do Problema

Ao implementar o histórico de mensagens (Etapa 3), foi necessário garantir que múltiplas threads pudessem ler e escrever sem deadlock.

Prompt Utilizado

> **Usuário:** "Começar pelo histórico de mensagens parece uma boa ideia."

> **IA (resposta implícita através de código fornecido):** Implementação de um monitor com `mutable mutex` e métodos const para leitura.

Resposta da IA

A IA forneceu uma implementação de **monitor** (padrão de design para concorrência):

```
class MessageHistory {
private:
    std::deque<HistoryEntry> messages;
    mutable std::mutex historyMutex;  // mutable para métodos const

public:
    void addMessage(const std::string& msg, int sender) {
        std::lock_guard<std::mutex> lock(historyMutex);
        messages.push_back(entry);
        // ...
    }

    std::vector<std::string> getRecentMessages(size_t count) const {
        std::lock_guard<std::mutex> lock(historyMutex);  // Funciona com
const
        // ...
    }
};
```

Análise Crítica

A implementação sugerida pela IA foi **correta e seguiu boas práticas**:

1. **Encapsulamento:** Toda sincronização interna à classe (monitor)
2. **Granularidade:** Mutex protege apenas dados, não I/O
3. **Deadlock-free:** Apenas um mutex, sempre adquirido na mesma ordem
4. **Exception-safe:** `lock_guard` garante unlock mesmo com exceção

Um possível problema **não mencionado pela IA**: Se `getRecentMessages()` fosse chamado enquanto segurando outro mutex, poderia haver deadlock. Mas como o design garante que cada monitor tem apenas um mutex, isso não ocorre.

Mitigação Implementada

A implementação seguiu o padrão monitor sugerido pela IA, sem modificações. O design garante:

- **Atomicidade:** Cada operação é atômica
- **Isolamento:** Estado interno nunca exposto
- **Sem deadlock:** Hierarquia de locks implícita (apenas 1 mutex)

3.4 Starvation: Logger Producer-Consumer

Contexto do Problema

A biblioteca `libtslog` foi implementada na Etapa 1 usando o padrão Producer-Consumer, com limite de buffer de 1000 entradas.

Prompt Utilizado

> **Usuário:** "Você pode fazer os logs irem para uma pasta específica 'log'?"

Durante esta discussão, foi revisada a implementação do logger e identificado um potencial problema de starvation.

Análise pela IA

A IA não identificou explicitamente starvation, mas o design original tinha uma vulnerabilidade:

```
// Implementação original
void log(const std::string& message) {
    std::unique_lock<std::mutex> lock(bufferMutex);

    if (logBuffer.size() >= maxBufferSize) {
        // × O que fazer? Bloquear produtor? Descartar mensagem?
    }

    logBuffer.push(entry);
```

```
    bufferCV.notify_one();
}
```

Problema identi cado pelo aluno: Se múltiplas threads produzirem logs muito rapidamente, o buffer pode encher e:

- **Opção A (bloqueio):** Threads produtoras ficam esperando → starvation
- **Opção B (descarte):** Mensagens são perdidas → não aceitável

Análise Crítica

Este problema foi identificado **pelo aluno** durante revisão de código, não pela IA espontaneamente. Isso demonstra que:

1. A IA é útil para identificar problemas clássicos
2. Revisão humana ainda é essencial para edge cases
3. A experiência do desenvolvedor complementa a IA

Mitigação Implementada

Solução: Implementar backpressure com condition variable:

```
void log(const std::string& message) {
    std::unique_lock<std::mutex> lock(bufferMutex);

    // Espera até haver espaço no buffer (backpressure)
    bufferCV.wait(lock, [this]() {
        return logBuffer.size() < maxBufferSize || shutdownFlag;
    });

    if (shutdownFlag) return;

    logBuffer.push(entry);
    bufferCV.notify_one();
}
```

Esta solução garante que:

- Produtores não travam indefinidamente (bounded wait)
- Nenhuma mensagem é perdida
- Sistema degrada gracefully sob alta carga

3.5 Sincronização de Testes: Sleep vs Barrier

Problema Identi cado

Durante análise crítica dos testes automatizados, foi identificado que o uso de `sleep()` para coordenar clientes no teste de stress não constitui sincronização real:

```
# Código problemático
sleep 2;  # Não garante que todos conectaram
echo "Mensagem";  # Pode causar race condition
```

Análise

Este approach tem limitações fundamentais:

1. **Timing dependente de hardware**: Em máquinas lentas, o delay fixo pode ser insuficiente
2. **Não demonstra conceitos do curso**: `sleep()` não é um primitivo de sincronização
3. **Race condition potencial**: Não há garantia de ordem de execução

Solução Implementada: Barrier Pattern

Implementamos uma barreira usando `std::condition_variable`:

```
class Barrier {
    std::mutex mtx;
    std::condition_variable cv;
    int count, threshold;

    void wait() {
        std::unique_lock<std::mutex> lock(mtx);
        count++;
        if (count >= threshold) {
            cv.notify_all();  // Libera todas as threads
        } else {
            cv.wait(lock, [this]() { return count >= threshold; });
        }
    }
};
```

Funcionamento:

1. Cada cliente chama `barrier.wait()` após conectar
2. Última thread a chegar libera todas com `notify_all()`
3. Todas enviam mensagens **simultaneamente**

Resultado: Sincronização determinística e independente de hardware.

---

4. Melhorias de Qualidade com IA

4.1 Smart Pointers e RAII

Prompt Utilizado

> **Usuário:** "Vamos então para os smart pointers e RAII"

Resposta da IA

A IA sugeriu implementar:

1. **SocketGuard:** Wrapper RAII para file descriptors
2. **std::unique_ptr** para threads
3. **std::shared_ptr** para clientes compartilhados

```
// Exemplo de SocketGuard sugerido pela IA
class SocketGuard {
private:
    int socket_fd;
    bool released;

public:
    explicit SocketGuard(int fd) : socket_fd(fd), released(false) {}

    ~SocketGuard() {
        if (!released && socket_fd >= 0) {
            close(socket_fd);  // Automático
        }
    }

    int release() {
        released = true;
        return socket_fd;
    }
};
```

Análise Crítica

A sugestão da IA foi **excepcional**. Ela:

- Implementou move semantics correto
- Deletou copy constructor (resource ownership único)
- Adicionou método `release()` para transferência de ownership
- Seguiu idiomas modernos de C++17

Benefício: O código agora é exception-safe. Se uma exceção ocorrer durante setup da conexão, o socket é fechado automaticamente, evitando vazamento de file descriptors.

4.2 Prompt Visual no Cliente

Contexto

Durante testes de usabilidade, foi observado que o cliente não indicava claramente quando o usuário podia digitar.

Prompt Utilizado

> **Usuário:** "Você pode colocar alguma indicação de que é possível digitar uma mensagem? Um '>' já é bom!"

Resposta da IA

A IA implementou:

- Prompt `>` antes de cada entrada
- Limpeza de linha quando mensagem é recebida
- Reimpressão do prompt após exibir mensagem

```
std::cout << "\r" << std::string(50, ' ') << "\r";  // Limpa linha
std::cout << line << std::endl;
std::cout << "> " << std::flush;  // Reimprime prompt
```

Problema Identi cado pelo Aluno

> **Usuário:** "Tem um pequeno erro: Quando é a primeira mensagem, tem dois '>'!"

A IA forneceu uma solução, mas o aluno ajustou com uma lógica mais simples usando flag `bool firstMessage`.

Análise Crítica

Este caso demonstra **colaboração efetiva entre humano e IA**:

- IA: Forneceu estrutura e conceito geral
- Humano: Identificou edge case e simplificou lógica
- Resultado: Solução mais elegante que as duas propostas individuais

---

5. Tabela de Mapeamento: Requisitos → Código

Requisito Obrigatório	Arquivo	Linhas	Descrição da Implementação
Servidor TCP concorrente	`tcp_server.cpp`	45-90	Loop `accept()` + criação de thread por cliente
Thread dedicada por cliente	`tcp_server.cpp`	75-80	`std::thread clientThread(...)`
Broadcasting de mensagens	`tcp_server.cpp`	120-150	Método `broadcastMessage()` com mutex

Requisito Obrigatório	Arquivo	Linhas	Descrição da Implementação
Exclusão mútua	`tcp_server.cpp`	122	`std::lock_guard<std::mutex> lock(clientsMutex)`
Histórico thread-safe	`message_history.cpp`	15-60	Monitor com `mutable mutex`
Logging concorrente	`libtslog.cpp`	30-95	Producer-Consumer com `condition_variable`
Cliente CLI	`tcp_client.cpp`	50-200	Interface com prompt e comandos
Framing de mensagens	`tcp_client.cpp`	70-100	Parsing por `\n` com buffer acumulado
Proteção de std::cout	`tcp_client.cpp`	12, 85-90	Mutex global `coutMutex`
RAII para recursos	`socket_guard.h`	10-45	Destrutor fecha socket automaticamente
Smart pointers	`tcp_server.cpp`	60-65	`std::shared_ptr<ClientInfo>`

6. Análise de Padrões de Design Implementados

6.1 Producer-Consumer (libtslog)

- **Produtores:** Threads do servidor/cliente chamando `log()`
- **Consumidor:** Thread `writerLoop()` escrevendo em arquivo
- **Sincronização:** `mutex` + `condition_variable`
- **Vantagem:** Desacoplamento de logging da lógica principal

6.2 Monitor (MessageHistory)

- **Características:** Encapsulamento de sincronização dentro da classe
- **Interface pública:** Métodos thread-safe
- **Invariante:** `messages.size() <= maxSize`
- **Vantagem:** Simplifica uso correto (impossível esquecer lock)

6.3 Thread Pool Implícito

- **Implementação:** Uma thread por cliente (criação sob demanda)
- **Limitação:** Sem limite de threads (potencial problema de escalabilidade)
- **Mitigação futura:** Implementar pool com número fixo de threads

---

7. Testes de Validação

7.1 Teste de Stress

```
make stress-test  # 10 clientes simultâneos
```

Resultados:

- 10 conexões aceitas
- 20 mensagens processadas (2 por cliente)
- 0 race conditions detectadas
- 0 deadlocks
- Logs consistentes

7.2 Análise de Logs

Após stress test, verificação manual dos logs revelou:

- **Ordenação temporal:** Timestamps corretos e monotônicos
- **Completude:** Todas as mensagens registradas
- **Formato:** Sem intercalação de caracteres

---

8. Lições Aprendidas

8.1 Eficácia da IA

Pontos Fortes:

- Identificação rápida de race conditions clássicas
- Sugestão de padrões de design (monitor, producer-consumer)
- Código idiomático e moderno (C++17, RAII, smart pointers)
- Explicações didáticas sobre conceitos (TCP framing, thread-safety)

Limitações:

- Não identificou spontaneamente alguns edge cases (starvation, duplo prompt)
- Sugestões às vezes mais complexas que o necessário
- Necessita validação humana para contexto específico

8.2 Complementaridade Humano-IA

O melhor resultado foi obtido através de **colaboração iterativa**:

1. **IA:** Fornece estrutura e boas práticas
2. **Humano:** Adapta ao contexto, simplifica, testa
3. **IA:** Valida mudanças, sugere melhorias
4. **Humano:** Decisão final sobre trade-offs

---

9. Conclusão

O desenvolvimento do servidor de chat TCP demonstrou que **IA é uma ferramenta valiosa mas não substitutiva** no desenvolvimento de sistemas concorrentes. A IA (Claude/Perplexity) foi eficaz para:

1. **Identificar problemas:** Race conditions, falta de sincronização
2. **Sugerir soluções:** Padrões de design, código idiomático
3. **Educar:** Explicações sobre TCP, threading, sincronização

Porém, a **experiência humana foi essencial** para:

- Identificar edge cases não óbvios
- Simplificar soluções propostas
- Validar correção através de testes
- Tomar decisões de design contextual

Recomendação: Utilizar IA como **assistente inteligente** em ciclo de desenvolvimento iterativo, sempre validando sugestões através de revisão de código e testes extensivos.

---

10. Referências

- **Código fonte:** https://github.com/Asfuri/Servidor-Chat-TCP (tag v3-final)
- **Ferramenta de IA:** Claude (Anthropic) via Perplexity AI
- **Material de referência:** Notas de aula de Programação Concorrente (UFPB)
