Atividade 2: Sistema de Vendas Concorrente com Estado Compartilhado

Prof. Alcides / Prof. Mário

2026-02-24

Índice

1 Introdução e Contexto

No desenvolvimento de sistemas distribuídos, o gerenciamento de um **estado compartilhado** é um dos maiores desafios. Imaginem um sistema de venda de ingressos ou um e-commerce: múltiplos usuários (clientes) acessam um servidor central simultaneamente para realizar operações sobre um recurso escasso (o estoque).

Como vocês viram em Programação Paralela com C/C++, quando múltiplos fluxos de execução tentam alterar a mesma variável na memória, ocorre uma **Condição de Corrida (Race Condition)**. Nesta atividade, vamos transpor esse conceito para o modelo de Redes, utilizando **Sockets TCP** e **Multithreading** em Python.

2 O Problema: Concorrência em Redes

Um servidor iterativo (que processa um cliente por vez) é simples, mas ineficiente. Se um cliente demora para enviar um comando, todos os outros ficam bloqueados. Para resolver isso, utilizaremos o modelo **Thread-per-Connection**.

No entanto, ao usar Threads para manipular o `estoque`, precisamos garantir a **Exclusão Mútua**. Em Python, utilizamos o objeto `threading.Lock` para proteger a **Seção Crítica**.

3 Especificação Técnica

3.1 Requisitos do Servidor

1. **Protocolo:** Escutar na porta TCP `12346`.
2. **Estado Interno:** Uma variável inteira `estoque` iniciada em 10.
3. **Concorrência:** Para cada nova conexão aceita (`accept`), o servidor deve disparar uma nova `Thread`.
4. **Sincronização:** O acesso à variável `estoque` deve ser protegido por um **Lock**.

3.2 Comandos do Protocolo (Texto Simples)

Os clientes enviarão strings via socket. O servidor deve interpretar:

- `CONSULTAR`: Retorna o valor atual. Ex: `"Estoque atual: 10"`.
- `COMPRAR`:
  - Se `estoque > 0`: decrementa e responde `"Compra realizada. Estoque restante: X"`.
  - Se `estoque == 0`: responde `"ERRO: Produto esgotado"`.

3.3 Dicas de Implementação em Python

Para quem está migrando do C/C++, o Python simplifica muito a manipulação de threads e sockets.

Exemplo de uso de Lock:

```python
import threading

lock = threading.Lock()
estoque = 10

def realizar_venda():
    global estoque
    with lock: # O 'with' garante que o lock seja liberado mesmo se houver erro
        if estoque > 0:
            estoque -= 1
            return True
        return False
```

Exemplo de Servidor Multithread:

```python
import socket
import threading

def handle_client(conn, addr):
    print(f"Conectado por {addr}")
    # ... lógica de recepção de dados ...
    conn.close()

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('0.0.0.0', 12346))
server.listen()

while True:
    conn, addr = server.accept()
    thread = threading.Thread(target=handle_client, args=(conn, addr))
    thread.start()
```

---

4 Introdução: O Teste de Estresse

Em um cenário real de e-commerce (como uma Black Friday), não temos um único cliente, mas milhares. Para validar se o seu servidor é robusto, vamos desenvolver um script que emula essa "multidão". Cada thread no código do cliente funcionará como um processo independente, seguindo o protocolo definido.

5 Especificação do Cliente de Simulação

O cliente deve ser capaz de:

1. **Instanciar N Threads:** Cada thread representa um usuário único.
2. **Manter a Conexão:** Diferente de um script simples que conecta e desconecta, este deve manter o socket aberto por alguns segundos para ser capaz de receber o **Broadcast** de esgotamento.
3. **Executar o Protocolo:** Cada "usuário" deve:
    - Conectar ao servidor.
    - Enviar `CONSULTAR`.
    - Aguardar um tempo aleatório (simulando o tempo de decisão humana).
    - Enviar `COMPRAR`.
    - Ficar em modo de escuta por 5 a 10 segundos para capturar mensagens de aviso.
    - Repetir esses passos um número aleatório de vezes, entre 10 e 20.

6 Lógica de Comunicação Assíncrona no Cliente

Para que o cliente consiga enviar comandos e, ao mesmo tempo, ficar "ouvindo" o servidor sem travar a thread de simulação, utilizaremos um **timeout** no socket ou uma thread dedicada por conexão.

7 O que observar durante o experimento

Para que o projeto seja considerado bem-sucedido, o aluno deve observar no console do **Servidor**:

1. **Múltiplas conexões simultâneas:** O servidor não deve travar; ele deve processar as consultas de uns enquanto outros estão no `time.sleep`.
2. **Consistência do Estoque:** Mesmo com 50 pessoas tentando comprar ao mesmo tempo, o estoque nunca deve ficar negativo (ex: -1, -5). Isso prova que o `Lock` está funcionando.
3. **Broadcast em Tempo Real:** No momento em que o 10º item for vendido, todos os outros usuários ainda conectados (que estão no loop de escuta) devem imprimir a mensagem de "AVISO" quase instantaneamente.

8 Desafio Prático para o Aluno

Modifique o servidor removendo o `Lock` e execute a simulação.

- O que acontece com o contador de estoque?
- É possível que o servidor envie "Compra realizada" para 12 pessoas se o estoque inicial era 10?
- Relate suas observações no relatório final.

9 Desafio Extra (Diferencial)

Para os grupos que buscam excelência, implementem:

1. **Log Detalhado:** Exibir no console do servidor o timestamp, IP e porta do cliente para cada comando.
2. **Broadcast de Esgotamento:** Mantenha uma lista de conexões ativas. No momento exato em que o estoque chegar a zero, envie uma mensagem `"AVISO: O estoque acabou de esgotar!"` para todos os clientes conectados.

10 Critérios de Avaliação

Item	Descrição	Peso
Multithreading	Servidor aceita conexões simultâneas sem bloquear.	3.0
Exclusão Mútua	Uso correto de Locks; ausência de inconsistência no estoque.	4.0
Protocolo	Comandos COMPRAR/CONSULTAR funcionam conforme especificado.	2.0
Robustez	Tratamento de erros (ex: cliente desconecta abruptamente).	1.0