Relatório de Correção de Regressão de Desempenho em Linux

Leandro Afonso

11 de Dezembro de 2025

1 Resumo do Problema

A simulação distribuída de tráfego demonstrou uma regressão significativa de desempenho em ambiente Linux nativo quando comparada com a execução em Windows/Wine.

A tabela abaixo ilustra a discrepância nas taxas de conclusão de veículos dentro da janela de simulação fixa:

Ambiente	**Taxa de Conclusão**
Windows / Wine	$\sim 95 - 100\%$
Linux (OpenJDK Nativo)	$\sim 44\%$
Linux (com `strace`)	$\sim 91\%$

Table 1: Comparação de desempenho por ambiente.

O *insight* crucial surgiu ao descobrir que a execução sob `strace` recuperava a taxa de conclusão para 91%. O `strace` introduz *overhead* em cada *syscall*, o que, paradoxalmente, estabilizou o sistema ao forçar um abrandamento natural (*throttling*).

2 Causa Raiz

O Linux executa demasiado rápido.

O Coordenador gera veículos a uma velocidade superior à capacidade de processamento das interseções distribuídas e da pilha de rede. Em Windows/Wine, o *overhead* inerente à emulação e ao agendador do SO limita naturalmente a taxa de transferência do sistema.

Em Linux nativo, a execução mais célere provoca uma condição de corrida sistémica:

- A geração de veículos excede a capacidade de processamento imediato dos nós.
- As filas de eventos congestionam (*back up*) rapidamente nas interseções.
- Veículos gerados tardiamente não dispõem de tempo de CPU suficiente para concluir o percurso antes do fim da simulação.

3 Solução Implementada

A correção consistiu na introdução de micro-atrasos (*micro-throttles*) utilizando `LockSupport.parkNanos()`. Esta abordagem simula o *overhead* natural presente no ambiente Windows, permitindo o escoamento das filas de E/S.

3.1 Alterações no Código

1. Ficheiro: `SocketConnection.java`
Adicionado um atraso de 50μs após operações de E/S para permitir o processamento da pilha TCP.

```
        // Em sendMessage() após o flush:
        dataOut.flush();
        LockSupport.parkNanos(50000); // 50 us delay

        // Em receiveMessage() após readFully:
        dataIn.readFully(data);
        LockSupport.parkNanos(50000); // 50 us delay
```

SocketConnection.java (Excerto)

2. Ficheiro: `CoordinatorProcess.java`
Adicionado um atraso de 100μs na geração de veículos para limitar a taxa de produção.

```
        // Em generateAndSendVehicle():
        sendVehicleToIntersection(vehicle, entryIntersection);
        LockSupport.parkNanos(100000); // 100 us delay
```

CoordinatorProcess.java (Excerto)

Nota: Em ambos os ficheiros deve ser importado java.util.concurrent.locks.LockSupport.

4 Resultados e Validação

A aplicação dos atrasos sintéticos restaurou a paridade de desempenho entre os sistemas operativos.

Ambiente	Antes da Correção	Após Correção
Linux Nativo	$\sim 44\%$	$\sim$ **92**%

4.1 Por que funciona?

- **Precisão:** `LockSupport.parkNanos()` oferece um atraso preciso e não bloqueante, com impacto mínimo no agendador do SO, ao contrário de `Thread.sleep()`.
- **Ritmo de E/S (50μs):** Abranda a comunicação via *socket* o suficiente para evitar a saturação dos *buffers* de receção das interseções.
- **Controlo de Fluxo (100μs):** Limita a produção do Coordenador, garantindo que o sistema a jusante consegue processar os eventos em tempo útil.

4.2 Verificação

Para validar a correção no ambiente de desenvolvimento:

```
        mvn clean compile
        mvn javafx:run
```

Resultado Esperado: Taxa de conclusão superior a 90%.

5 Abordagens Alternativas (Falhadas)

As seguintes tentativas foram realizadas antes da solução final, sem sucesso:

- **Thread.sleep(1):** Demasiado impreciso (granularidade mínima de ∼1ms em Linux), causando atrasos excessivos.
- **Thread.yield():** Sem efeito prático no agendador CFS do Linux neste contexto.
- **Garbage Collectors:** A alteração entre G1, Parallel e Shenandoah não surtiu efeito.
- **Versão Java:** Testes com Java 17 e 25 mostraram o mesmo comportamento.
- **Prioridade de Threads:** Ajustes de prioridade na JVM foram ignorados pelo SO.