

Sistemas Distribuídos

Curso de Análise e Desenvolvimento de Sistemas
Instituto Federal da Paraíba
Campus Cajazeiras

Concorrência em Java

- Java possui uma série de mecanismos para:
 - Sincronização
 - Controle de Concorrência em Thread
- Principais conceitos ligados a isto são:
 - Monitor
 - Semáforo
 - Bloqueadores

Concorrência em Java (Monitores)

- Monitor
 - é um mecanismo de mais alto nível cujo objetivo é impedir o acesso concorrente inadequado
 - obriga uma thread esperar por outra terminar o que está fazendo
 - ele é um objeto específico que controla esse acesso a um outro objeto principal que precisa da proteção de concorrência

Concorrência em Java (Monitores)

- É um mecanismo mais pronto, você só precisa dizer que precisa monitorar este objeto, em alguns casos precisa dizer que não precisa mais.
- Em Java, diz que existe um monitor quando utiliza-se a palavra reservada **synchronized** em um bloco de código
- O monitor pode atuar sobre:
 - Métodos ou
 - Objetos

Concorrência em Java (Monitores)

■ Sincronização de Método

```
public synchronized void metodo (int param) {  
    // código protegido  
}
```

- O objeto usado na invocação é bloqueado para uso da thread que invocou o método
- Se o método for static, a classe é bloqueada
- Métodos não sincronizados e atributos ainda podem ser acessados

Concorrência em Java (Monitores)

■ Sincronização de Bloco

```
synchronized (objeto) {  
    // código protegido  
}
```

- O objeto especificado na abertura do bloco é bloqueado para uso da thread corrente
- O código fica protegido de acesso concorrente
- Qualquer outra thread que tentar bloquear o mesmo objeto entrará em uma fila de espera

Concorrência em Java (Monitores)

- Java permite que sejam definidas condições de acesso dentro de monitores
 - Uma thread em um monitor deve chamar o método `wait()` se não puder prosseguir devido a alguma condição necessária não-satisfieta; o acesso ao monitor é então liberado
 - Sempre que uma condição de espera for modificada, podemos notificar uma thread em espera na fila escolhida aleatoriamente com o método `notify()`, ou notificar todas as threads na fila chamando o método `notifyAll()`
 - Chamá-los fora do monitor resulta em exceção

Concorrência em Java (Semaphore)

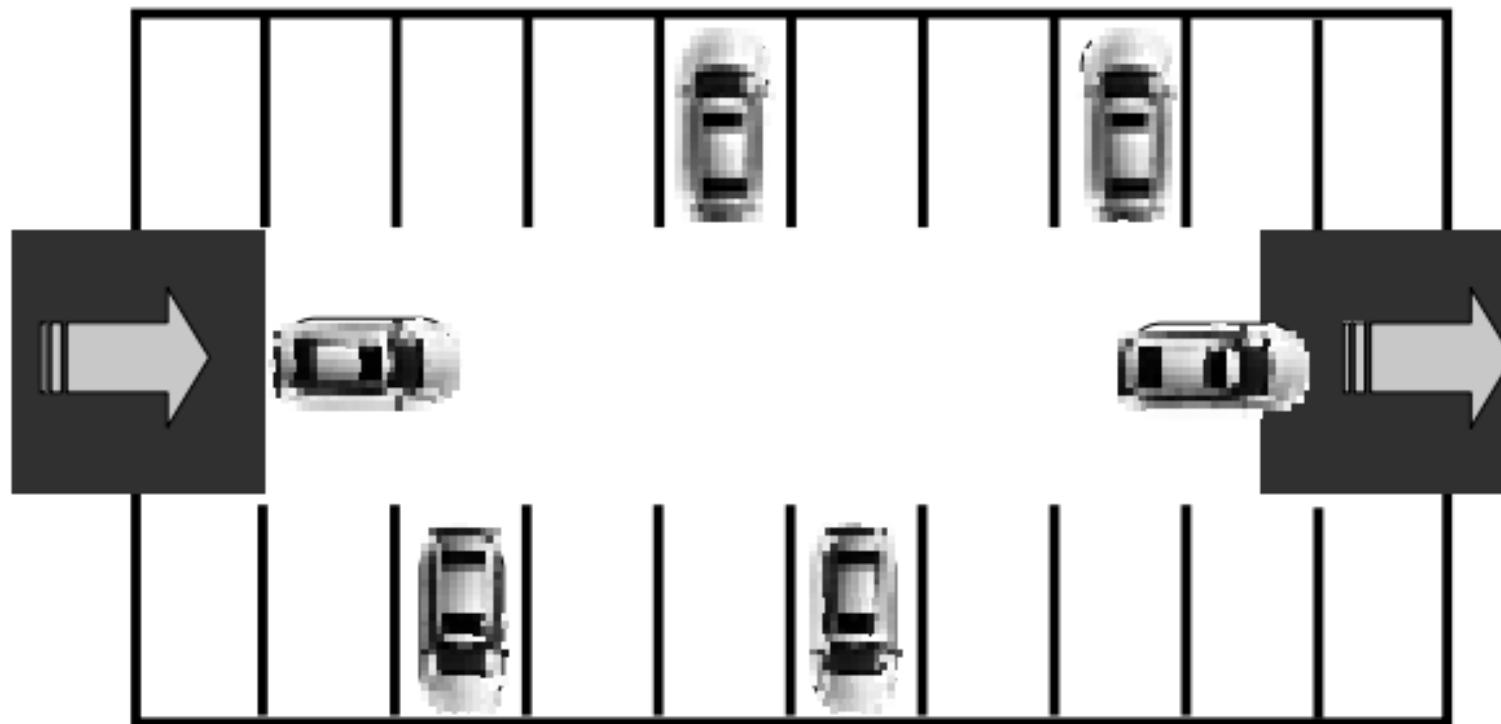
- Semáforo:
 - Um semáforo é uma estrutura de dados que controla o acesso de aplicações aos recursos,
 - Baseia-se em um número inteiro, que representa a quantidade de acessos que podem ser feitos.
 - Utiliza-se semáforos para controlar a quantidade de acesso a determinado recurso.

Concorrência em Java (Semaphore)

- Permite controlar o número de acessos simultâneos a um dado ou recurso
- Métodos da classe Semaphore
 - Semaphore(int acessos [, boolean ordem]): construtor; parâmetros definem o número de acessos simultâneos possíveis e se a ordem de liberação de threads em espera será FIFO
 - acquire(): solicita acesso a um dado ou recurso, entrando em espera se todos os direitos de acesso estiverem sendo usados
 - release(): libera um direito de acesso

Concorrência em Java (Semaphore)

- Exemplo: Estacionamento



Concorrência em Java (Semaphore)

```
import java.util.concurrent.*;
public class Carro extends Thread {
    private static Semaphore estacionamento = new Semaphore(10,true);
    public Carro(String nome) { super(nome); }
    public void run() {
        try {
            estacionamento.acquire();
            System.out.println(getName() + " ocupou vaga.");
            sleep((long)(Math.random() * 10000));
            System.out.println(getName() + " liberou vaga.");
            estacionamento.release();
        } catch(InterruptedException ie){ ie.printStackTrace(); }
    }
    public static void main(String args[]) {
        for (int i = 0; i< 20; i++)
            new Carro("Carro #"+i).start();
    }
}
```

Concorrência em Java (Locks)

- Interface Lock
 - Mecanismo de exclusão mútua
 - Permite somente um acesso por vez
 - Caso o dado/recurso esteja em uso, a thread que tentar bloqueá-lo entra numa fila
- Principais Métodos:
 - lock(): primitiva de bloqueio; deve ser chamada antes do acesso ao dado/recurso
 - unlock() : primitiva de desbloqueio; usada para liberar o acesso o dado/recurso

Concorrência em Java (Locks)

- Outros métodos da interface Lock:
 - tryLock(): bloqueia e retorna *true* se o *lock* estiver disponível, caso contrário retorna *false*
 - getHoldCount(): retorna número de threads que tentaram obter o *lock* e não o liberaram
 - isHeldByCurrentThread(): retorna true se a thread que fez a chamada obteve o bloqueio
 - isLocked(): indica se o *lock* está bloqueado
 - getQueueLength(): retorna o número de threads que aguardam pela liberação do lock

Concorrência em Java (Locks)

■ Classe ReentrantLock

- Implementa mecanismo de bloqueio exclusivo
- Por *default* a retirada de threads da fila não é ordenada, ou seja, não há garantias de quem irá adquirir o *lock* quando este for liberado
- O construtor ReentrantLock(true) cria um *lock* com ordenação FIFO da fila, o que torna o acesso significativamente mais lento

Escalonamento de Tarefas

Executores e Reservatórios

Escalonamento de Tarefas

- Para abstrair o conceito de tarefas, em Java, utilizá-se a interface Runnable.
- Com esta interface é possível separar a tarefa dos seus executores
- Usando Runnable é desenhar tarefas que podem ser compartilhadas por diversos executores

Escalonamento de Tarefas

- Em Java, três interfaces representam os tipos de executores:
 - **java.util.concurrent.Executor:**
 - representa um executor de tarefas em geral, sem se preocupar em como ocorre esta execução
 - **java.util.concurrent.ExecutorService:**
 - Um tipo de executor que gerenciar a inicialização e finalização de tarefas, além de poder fornecer meios de acompanhamento de progresso.
 - **java.util.concurrent.ScheduledExecutorService:**
 - um ExecutorService que pode agendar tarefas para executar após um determinado atraso ou para executar periodicamente.

Escalonamento de Tarefas

- Existem, ainda, duas interfaces que representam outras formas de tarefas:
 - Tarefas que representam operações com retorno futuro:
 - java.util.concurrent.Future
 - Tarefas que representam operações que podem retornar exceções:
 - java.util.concurrent.Callable
- Estas duas interface são utilizadas para realizar tarefas assíncronas

Enfileiramento de Tarefas

- Padrão produtor-consumidor usando filas e mapas concorrentes
- Duas das principais interfaces principais são:
 - ConcurrentMap
 - BlockingQueue

ConcurrentMap

- Similar a `java.util.Map`, mas utiliza uma estratégia de travamento mais eficiente para concorrência e escalabilidade
- Não permite o travamento do de acesso exclusivo
- Usado para modificações continuas
- Diferente de `SynchronizedMap` (de Collections)

BlockedQueue

- Similar a `java.util.Queue`, mas enfileira as threads e não os objetos da fila
- Principais implementações são:
 - `SynchronousQueue`
 - Usanda quando o número de produtores é suficiente para atender os consumidores
 - `ArrayBlockingQueue`
 - Similar a um `java.util.Array`, mas que controla o acesso de forma a usara estratégia FIFO (First-In-First-out)

Lei de Amdahl

- A aceleração de um programa depende da sua proporção de execução serial e paralela.
- Aumentar o número de recursos não acelera o programa na mesma proporção
- É importante determinar qual a proporção do código poder ser paralelizada e com isto melhorada
- A lei de Amdahl pode ser usada como métrica para medir a eficiência de uso de processadores e da aceleração máxima que pode ser alcançada

Lei de Amdahl

- Seja $0 \leq f \leq 1$ a fracção da computação que só pode ser realizada sequencialmente. A lei de Amdahl diz-nos que o *speedup* máximo que uma aplicação paralela com p processadores pode obter é:

$$S(p) \leq \frac{1}{f + \frac{1-f}{p}}$$

- A lei de Amdahl dá-nos uma medida do *speedup* máximo que podemos obter ao resolver um determinado problema com p processadores.
- A lei de Amdahl também pode ser utilizada para determinar o limite máximo de *speedup* que uma determinada aplicação poderá alcançar independentemente do número de processadores a utilizar.

Lei de Amdahl

- Suponha que pretende determinar se é vantajoso desenvolver uma versão paralela de uma determinada aplicação sequencial. Por experimentação, verificou-se que 90% do tempo de execução é passado em procedimentos que se julga ser possível parallelizar. Qual é o *speedup* máximo que se pode alcançar com uma versão paralela do problema executando em 8 processadores?

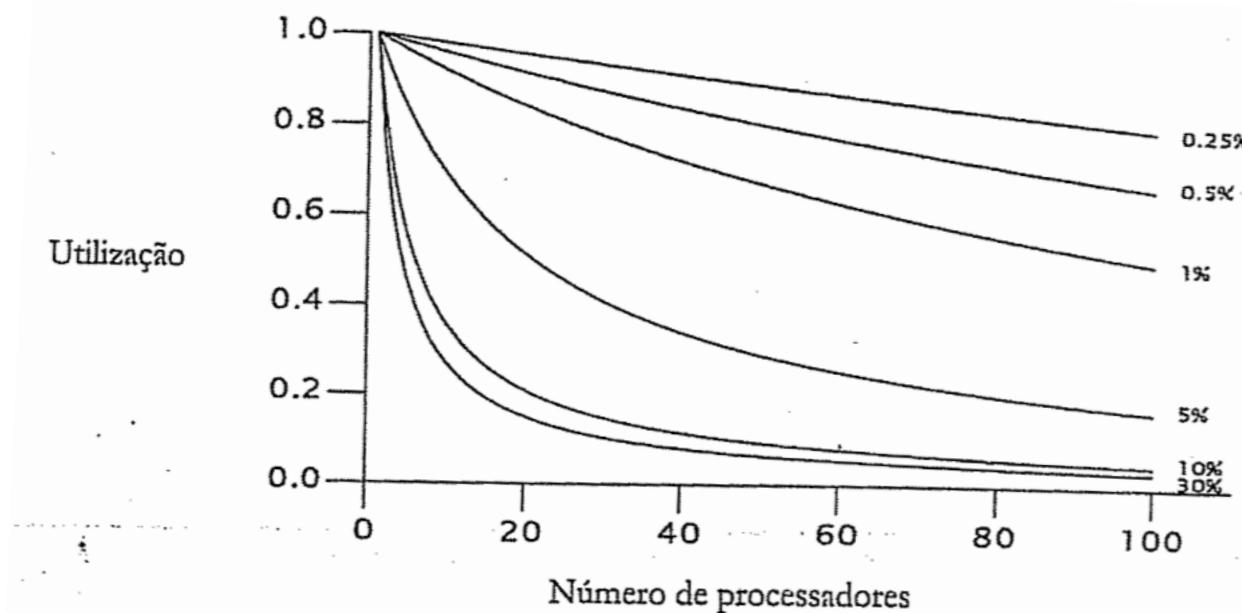
$$S(p) \leq \frac{1}{0,1 + \frac{1-0,1}{8}} \approx 4,71$$

- E o limite máximo de *speedup* que se pode alcançar?

$$\lim_{p \rightarrow \infty} \frac{1}{0,1 + \frac{1-0,1}{p}} = 10$$

Lei de Amdahl

- De nada adianta adicionar threads se não utilizarmos de forma eficiente os processadores.



Atores

- Usa a semântica de **Message Passing**
- Atores são modelos de controle de concorrência e isolamento.
- O modelo Actor força todos os acessos a um objeto a serem isolados por padrão.
- O objeto faz parte do estado local de um ator e não pode ser acessado diretamente por nenhum outro ator.

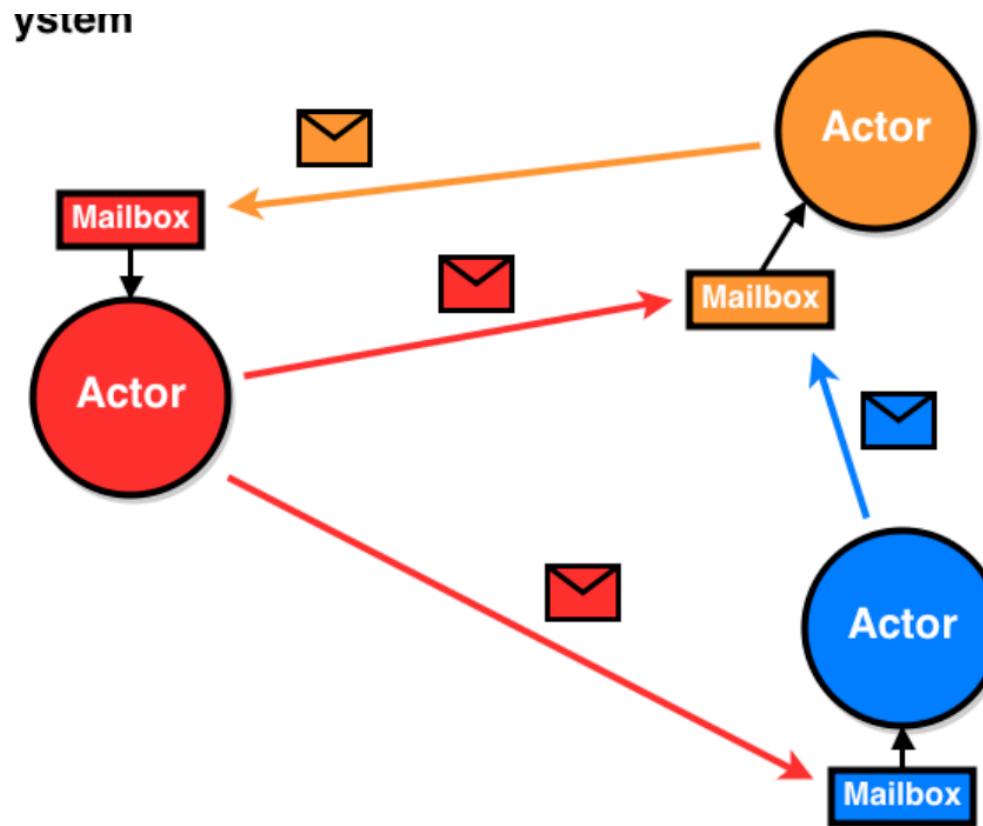
Atores

- Um Ator consiste em:
 - uma Caixa de Correio,
 - um conjunto de Métodos e
 - um Estado Local.
- O modelo de Ator é reativo, no qual os atores só podem executar métodos em resposta a mensagens;
- Os métodos podem ler e gravar no estado local e/ou enviar mensagens para outros atores.

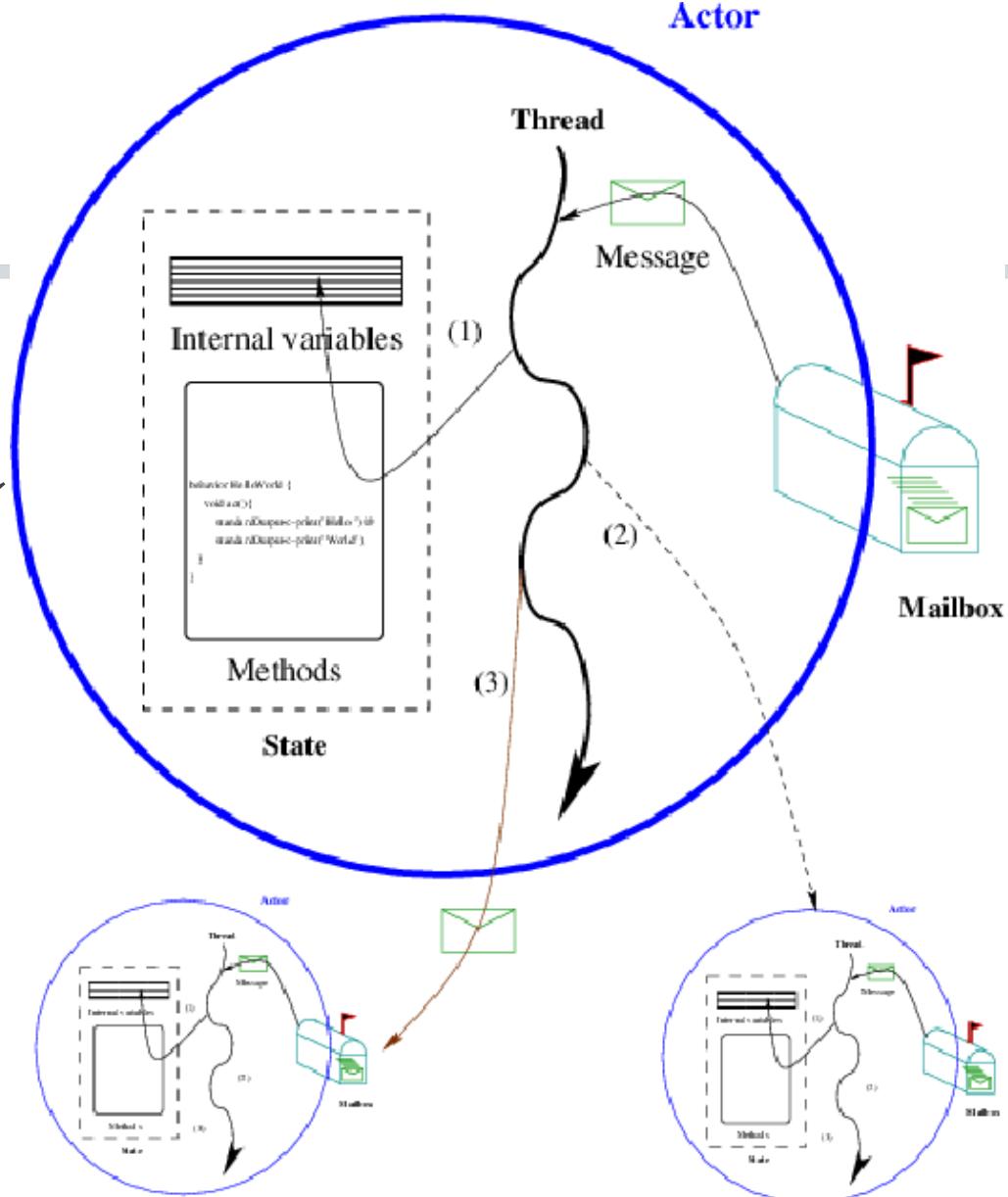
Atores

- A única maneira de modificar um objeto em um modelo de agente puro é enviar mensagens para o ator que possui esse objeto como parte de seu estado local.
- Em geral, mensagens enviadas a atores de diferentes outros atores podem ser arbitrariamente reordenadas no sistema.
- No entanto, em muitos modelos de atores, as mensagens enviadas **entre o mesmo par** de atores preservam a ordem em que são enviadas

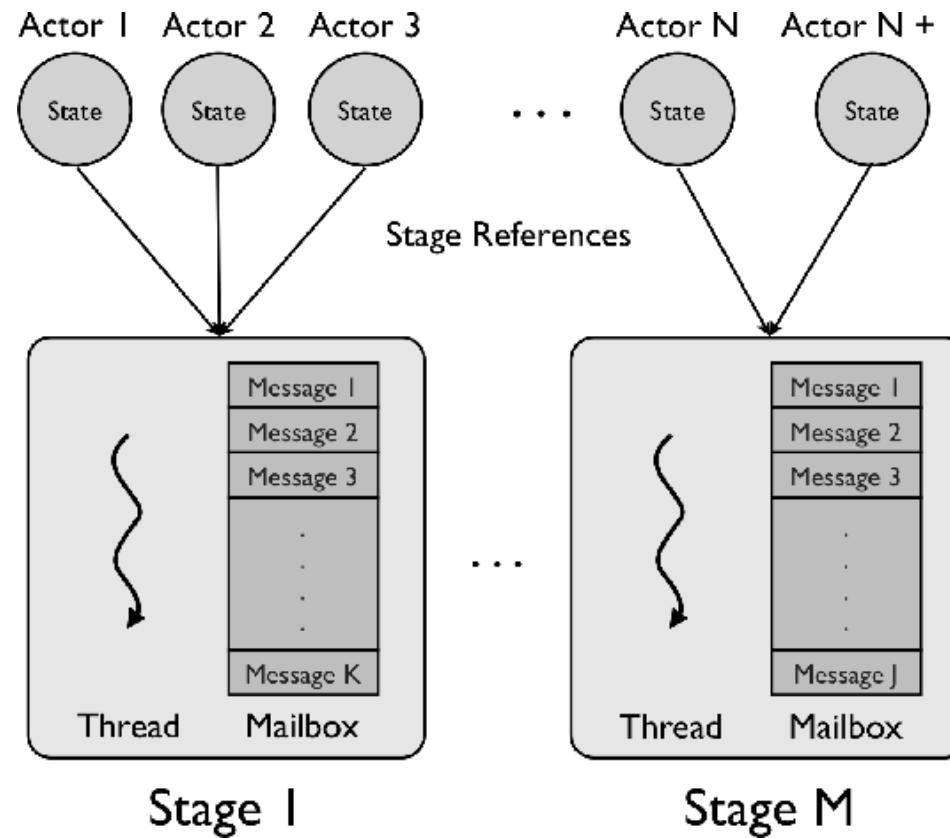
Actores



Visão Interna dos Atores



Actores



Exercício #1/2

1. Construa duas aplicações, uma cliente, que sabe enviar dois números, e outra servidora, que sabe somar e diminuir. O cliente envia dois números inicialmente aleatórios entre 0 e 9 e uma operação (9 bytes). Ao receber uma requisição, o servidor processa e devolve o resultado. De posse desse resultado o cliente envia novamente uma requisição usando o resultado e outro número aleatório entre 0 e 9 e continua fazendo isto continuamente. (use aleatoriedade para obter o tipo de operação também)

Exercício #2/2

2. Reimplemente o cliente para que este envie cerca de 10 req/s usando apenas uma única thread.
3. Para o mesmo cenário da questão 01, utilizando um repositório com 10 threads do lado do cliente e de 5 threads do lado do servidor (limitado a uma única conexão por vez), descreva o comportamento do sistema e proponha uma solução de melhoria de performance.