# Políticas e consentimentos

**INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA**

**Trabalho Prático 1**

**Vector Service**

Infra-Estruturas de Sistemas Distribuídos

Luis Campanela, n.º 8600

Bruno Costa, n.º 36868

Rodrigo Pina, n.º 44178

Docente: Luís Osório

**Abril, 2021**

# 1. Introdução

Neste trabalho prático foi sugerida a discussão do serviço Vector. Este serviço disponibiliza o acesso a um array de 4 elementos. As operações disponibilizadas sobre este array são as operações de read duma dada posição e a escrita duma dada posição com determinado valor.

public interface IVector **{**

int read**(**int pos**);**

void write**(**int pos**,** int n**);**

**}**

Figure 1 - Interface do serviço Vector

É pretendido que os clientes subtraiam uma quantia duma posição e adicionem o mesmo valor numa posição diferente. Contudo é pretendido que, sempre que possível, seja verificada a condição em que a soma dos valores do vector dê sempre a mesmo valor

A subtração e adição de uma valor implicam duas operações, uma operação de read e uma operação de write.

# 2. Verificação da Invariante

Pretende-se que a soma da variável vector da class Vector, que disponibiliza o serviço com o mesmo nome resulte sempre o mesmo valor 1379. Esta invariante tem que se verificar sempre que possível, indepedentemente do número de clientes que acedem a este serviço e da sua localização.

## 2.1 Problema 1 – Um cliente só faz uma escrita

Se um cliente só faz uma escrita e não completa a segunda escrita então a invariante deixa de poder ser verificada. Pois, por definição, a soma do vector vai resultar na soma original menos o valor retirado.

Por isso, para poder verificar esta condição temos duas hipóteses:

1. Verificar a condição apenas quando são feitas duas escritas
2. Guardar as escritas num objeto temporário e escrever no vetor só quando é feita a segunda escrita. Esta estratégia evita que seja necessário reverter operações

Este problema é um problema de atomicidade, pois é necessário garantir que ou o cliente faz todas as opeações, ou não faz nenhuma.

A solução número 1 foi implementada e pode ser verificada [no nosso repositório git](https://github.com/brunoss/isel-iesd/blob/a6a77dfade73da1733dcedd23f0c0a38748927f8/trabalho1/iesd2021sv-master/IsyIESD/CesVector/SerVector/SerVectorOPE/src/main/java/isos/tutorial/isyiesd/cesvector/servector/Vector.java).Nesta solução é feita uma contagem do número de escritas e só quando o número de escritas é par é que é dada a possibilidade da verificação da invariante.

Esta solução tem um problema. A verificação da invariante pode ficar bloqueada por um periodo de tempo excessivamente longo e se imposto o requisito que a invariante deve ser verificada com maior regularidade possivel, então poderá não ser uma solução adequada.

A solução número dois é mais interessante porque permite que a invariante seja verificada em qualquer momento, porque as escritas no vetor são feitas na mesma chamada ao método write.

A segunda solução foi implementada mais tarde e pode ser verificada [no nosso repositório git](https://github.com/brunoss/isel-iesd/blob/8a2f96cbb250566dbd626b791b90c91e5569338f/trabalho1/iesd2021sv-master/IsyIESD/CesVector/SerVector/SerVectorOPE/src/main/java/isos/tutorial/isyiesd/cesvector/servector/Vector.java)

## 2.2 Problema 2 – Múltiplos Clientes

Se existem múltiplos clientes então é necessário garantir exclusividade no acesso ao serviço entre os diversos clientes, durante as quatro operações. Se não existisse essa exclusividade, então os clientes poderiam fazer operações com valores que já foram atualizados por outros clientes. A este problema chama-se um Dirty Read e é um problema de consistência.

Para resolver este problema temos também duas hipóteses:

1. Implmentamos uma solução de exclusividade do lado do servidor
2. Implementamos uma solução de exclusividade do lado do cliente

A solução número 1 foi implementada e pode ser verificada [no nosso repositório git](https://github.com/brunoss/isel-iesd/blob/8a2f96cbb250566dbd626b791b90c91e5569338f/trabalho1/iesd2021sv-master/IsyIESD/CesVector/SerVector/SerVectorOPE/src/main/java/isos/tutorial/isyiesd/cesvector/servector/Vector.java). Nesta solução é necessário ter uma forma únca de identificar qual é o cliente. Na nossa implementação os clientes são identificados através do IP e da porta que usam para comunicar com o servidor. Contudo **este tipo de identificação não é viável** porque a rede não é homogénea e os clientes podem usar serviços de VPN, que alteram o seu IP.

Assumindo que existe então uma forma de identificar um cliente, é depois guardada uma identificação desse cliente, e enquanto esse cliente não fizer duas escritas o servidor bloqueia os restantes clientes. Quando o cliente completa a sua transação então a identificação do cliente é libertada e é dada a oportunidade a outro cliente (ou ao memso cliente), de fazer a primeira leitura.

A solução número 2 pode ser implementada através de utilização do sistema de ficheiros como mecanismo de sincronização. Um cliente cria um ficheiro e quando completa a sua transação apaga o ficheiro. Só o cliente que cria o ficheiro é que pode fazer os pedidos. Esta abordagem implica que todos os clientes têm de ter acesso a um sistema de ficheiros comum. E tem a desvantagem que se houver clientes que não são implementados por nós, então não poderá ser mais garantida a exclusividade entre os clientes.

## 2.3 - Problema 3 – Concorrência no servidor

Mesmo só com um cliente é possível que os pedidos sejam atendidos por threads diferentes. E se há threads diferentes a acederem ao mesmo objeto então esse acesso tem de ser sincronizado. Para além disso, a verificação da invariante tem que também ser feita numa thread diferente das que processam os peddidos do cliente, reforçando a necessidade de haver controlo de concorrência. Por esse mesmo motivo, os mecanismos de concorrência estão a ser devidamente utilizados como mostrado nas implementações feitas por nós.