# Políticas e consentimentos

**INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA**

**Trabalho Prático 1**

Serviço Integrado de Pagamento em Contexto de Mobilidade

Desafio de Coordenação de Transações Distribuídas

Infra-Estruturas de Sistemas Distribuídos

Luis Campanela, n.º 8600

Bruno Costa, n.º 36868

Rodrigo Pina, n.º 44178

Docente: Prof. Luís Osório

**6 de Junho de 2021**

# Introdução

No âmbito do desafio de coordenação CMSP na disciplina de IESD, foi abordado o problema das transacções e coordenação entre os serviços dos clientes e vectores. A análise das questões que se colocam neste cenário e das possíveis soluções para cada uma delas, foi feita nas aulas práticas 1 e 2 da disciplina.

A abordagem usada foi, numa fase inicial, analisar e implementar a execução de operações de leitura e escrita de um serviço cliente a um serviço Vector que disponibilizava o acesso a um *array* de 4 elementos (*Figura 1*). Posteriormente, foi-se introduzindo complexidade, de forma gradual, e analisando os novos problemas que iam surgindo. Começou por se considerar múltiplos serviços cliente, o que acrescentou problemas a nível do controlo de concorrência e, na aula prática 2, acrescentou-se a possibilidade de acesso a múltiplos vectores.

Nesta última fase, tornou-se evidente a vantagem da introdução de um serviço mediador para garantir a atomicidade e de um serviço de controlo de concorrência no acesso de múltiplos clientes a múltiplos vectores. O âmbito destas aulas práticas são os serviços distribuídos com coordenação centralizada, nas próximas será introduzida a coordenação distribuída.



*Fig. 1 Acesso de múltiplos clientes a um vector*

# Estado do conhecimento e análise e discussão do problema

o Desenvolvimento de sistemas informáticos na base de elementos Service (SOA)

o Coordenação de transações distribuídas

o Descrição de quadros tecnológicos utilizados

Num quadro de arquitectura orientada a serviços (SOA), OSGi…

O desenvolvimento de sistemas informáticos com elementos distribuídos apresenta dificuldades inerentes à concretização de diferentes mecanismos de interacção. São essas dificuldades e as possíveis abordagens para a sua resolução que se procura pôr em evidência no presente trabalho.

No quadro das arquitecturas orientadas a serviços (SOA), a complexidade dos sistemas distribuídos pode ser reduzida com a modularidade e autonomia das entidades computacionais e, tirando partido de um acomplamento fraco, contribuem para a robustez do sistema como um todo [Joachim, 2013]. Por outro, as SOA permitem a integração de módulos autónomos desenvolvidos em diferentes tecnologias, por equipas diferentes.

No entanto, as SOA colocam novos desafios em termos de coordenação de sistemas distribuídos, uma vez que, aumentando o número de elementos componentes, aumenta a complexidade da sua coordenação.

No âmbito do desafio de coordenação CMSP, foi, entre outros, abordado o problema das transacções entre os serviços distribuídos clientes e vectores, respeitando as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) [Haerder, 1983].

Para garantia da propriedade Atomicidade, foi decidido criar um elemento Transaction Manager (TM) e, para garantia das propriedades Consistência e Isolamento, um elemento Lock Manager (TPLM). O acrónimo TPLM refere-se à característica Two-Phase do Lock Manager que será explicada mais à frente.

A propriedade Durabilidade, considera-se assegurada, com a persistência dos dados em memória.

A validação das propriedades é feita através da garantia do invariante, somatório do conteúdo dos diversos vectores de cada um dos serviços.

Nos capítulos seguintes, será feita a análise e discussão do problema e a descrição da abordagem de resolução.

# Java RMI

Optou-se por utilizar o Java RMI devido à simplicidade de utilização e dependências técnológicas. Esta decisão permitiu-nos poupar tempo na criação dos projetos e na gestão dos ficheiros POM e focar no essêncial da implementação.

O RMI tem o serviço Registry, onde os serviços se registam com um determinado nome, e que permite aos clientes descobrirem esses serviços pelo nome. O serviço Registry pode ser criado através da operação LocateRegistry**.**createRegistry. Em seguida os serviços podem registar-se através da operação rebind. Os clientes encontram os serviços pela operação lookup. Os serviços devem herdar da classe UnicastRemoteObject e implementar a respetiva interface do serviço que é partilhada pelo cliente. Essa interface deve herdar da interface Remote.

O RMI tem também a vantagem de ter a possibilidade de fazer callbacks ao cliente de forma transparente, não requerendo nenhuma lógica adicional para chamar o callback do lado do servidor.

Em contrapartida o RMI é especifico da plataforma Java e não tem interopibilidade com outras linguagens/plataformas que não são suportadas pelo ambiente de execução virtual Java.

# Demonstrador centrado na coordenação

Inicialmente, começou-se por analisar e implementar a execução de operações de leitura e escrita de um serviço cliente a um serviço Vector que disponibilizava o acesso a um *array* de 4 elementos. Pretendia-se que o cliente desse indicações de leitura de uma posição do *array*, de onde fosse subtraído um determinado valor *x* e, posteriormente, lesse uma outra posição do *array*, onde seria adicionado o mesmo valor *x*. No final das 4 operações (2 leituras e 2 escritas), que compunham uma transacção, a soma de todas as posições do *array* deveria ser a mesma que antes da transacção. Foi, assim, definido o invariante que consistia naquela soma, e deveria ter sempre o mesmo valor 1379. O invariante era verificado regularmente, através de uma *thread* em execução no serviço Vector.

public interface IVector **{**

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

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

**}**

*Fig. 2 Interface do serviço Vector*

Numa fase posterior, o acesso ao serviço Vector seria feito por dois serviços Cliente, em concorrência. Tal como anteriormente, era preciso fazer a verificação do invariante, que garantia o cumprimento das propriedades ACID Atomicidade, Consistência e Isolamento.

A seguir, descrevem-se os problemas encontrados neste processo e as soluções encontradas para os resolver.

# Falha de um único cliente após uma escrita

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

Para poder verificar esta condição, consideraram-se duas possibilidades:

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.

Aqui, apresenta-se um problema de atomicidade, pois é necessário garantir que o cliente faz todas as operações, ou não faz nenhuma.

A solução *i)* foi implementada e pode ser verificada [no 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 do invariante.

No entanto, esta solução tem um problema: a verificação do invariante pode ficar bloqueada por um período de tempo excessivamente longo e, caso exista um requisito que imponha a verificação do invariante com maior regularidade, poderá não ser uma solução adequada.

A solução *ii)* é mais interessante porque permite que o invariante seja verificado em qualquer momento, porque as escritas no vetor são feitas na mesma chamada ao método *write*. Esta solução foi implementada mais tarde e pode ser verificada [no 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).

# Múltiplos Clientes

Quando existem múltiplos clientes, é necessário garantir exclusividade no acesso ao serviço entre eles, durante as quatro operações. Caso não existisse essa exclusividade, os clientes poderiam fazer operações com valores lidos que, entretanto, tinham sido atualizados por outros clientes. Este é um problema de consistência de dados que se designa *dirty read*.

Para resolver este problema, consideraram-se também duas possibilidades:

1. Implementar uma solução de exclusividade do lado do servidor;
2. Implementar uma solução de exclusividade do lado do cliente.

A solução *i)* foi implementada e pode ser verificada [no 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 única de identificar o cliente, o que foi feito através do IP e do porto usado na comunicação com o servidor. Contudo, este tipo de identificação não é viável porque a rede não é homogénea e os clientes podem, por exemplo, usar serviços de VPN, que alteram o seu IP.

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

A solução *ii)* 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 com sucesso é que pode prosseguir, fazendo os pedidos. Esta abordagem implica que todos os clientes têm que ter acesso a um sistema de ficheiros comum (e.g.um sistema de ficheiros distribuído). Tem a desvantagem de ter que confiar na concepção dos clientes porque, se não forem criados com este princípio, não poderá ser garantida a exclusividade entre eles.

# Concorrência no servidor

Para além dos problemas indicados, relacionados com a concorrência em sistemas distribuídos, também se colocam os problemas de concorrência em sistema centralizado ou *shared-memory concurrency*. Ainda que só exista um cliente, é possível que os pedidos sejam atendidos por *threads* diferentes. E, havendo *threads* diferentes a aceder ao mesmo objeto, esse acesso tem que ser sincronizado. Além disso, a verificação do invariante tem que ser feita também numa *thread* diferente das que processam os pedidos do cliente, reforçando a necessidade de haver controlo de concorrência. Por esse mesmo motivo, foram implementados os mecanismos de concorrência em sistema centralizado, como se pode verificar nas implementações referidas anteriormente.

# Atomicidade no acesso a múltiplos vectores

Com a introdução de múltiplos vectores (*Figura 3*), assegurar a atomicidade das transacções deixou de poder basear-se na verificação do número par de escritas no próprio serviço Vector, porque, em cada transacção, as escritas não têm que ser ambas no mesmo vector. A transacção tem que ser vista como atómica globalmente para o conjunto de vectores acedidos, podendo haver um número par ou ímpar de escritas em cada um dos vectores.

Assim, avaliou-se a possibilidade de ter uma entidade computacional que servisse de mediador das transacções, garantindo a execução das operações duma transacção de forma atómica. Portanto, ou se realizam ambas as escritas nos vectores (*commit*) ou não se realiza nenhuma (*abort*), mantendo-se o invariante em ambos os casos.



*Fig. 3 Acesso de múltiplos clientes a múltiplos vectores*

# Múltiplos serviços Vector

# Transaction Manager (TM)

O mediador referido no ponto anterior, trata-se de um Transaction Manager e existe em alguns quadros tecnológicos como, por exemplo, no Spring Framework. **Qual é o âmbito deste tópico?** Neste trabalho foi criado um Transaction Manager e optou-se por implementar um algoritmo do tipo two-phase commit (2PC), um dos algoritmos mais comuns, para garantir *atomic commitment* através de múltiplos nós [Gray, 1978] (*Figura 4*).



*Fig. 4 Protocolo 2PC*

No 2PC, há uma fase inicial (*prepare*) antes da segunda fase (*commit*). Quando o cliente termina as operações e está preparado para finalizar a transacção, envia uma instrução de *commit* ao TM (coordinator). Este, começa por enviar uma mensagem de *prepare* a cada um dos nós (vectores), que respondem indicando se estão prontos para fazer *commit* da transacção. O Transaction Manager, finalmente, decide se envia a instrução de commit para todos os nós ou, caso algum dos nós não esteja preparado ou não responda, dá a indicação para abortar a transacção (roll-back).

Quando a instrução de commit é enviada para os vectores, é também dada a indicação ao cliente, para este saber que a transacção terminou.

**Nota:** Podia optar-se por esperar uma confirmação dos vectores de que o *commit* tinha sido bem sucedido, para passar a informação ao cliente mas, na prática, estamos perante uma situação do tipo Problema dos Dois Generais [Gray, 1978]. Caso uma comunicação fosse perdida, não havia forma de saber se se tratava do Request ou do Reply e o TM poderia passar informação errada ao Cliente.

*Transaction Manager na perspectiva do modelo X/OPEN: two-phase commit with presumed rollback.*

# Operações

getLocks()

unlock()

# Consistência e isolamento no acesso a múltiplos vectores

O controlo de concorrência efectuado na aula prática 1 (3.2), assume que só um cliente tem acesso ao vector, para execução das operações. Há uma serialização dos acessos de cada cliente, considerado aceitável porque só existe um recurso (vector), ainda que esse recurso fique indisponível para outros clientes. Poder-se-ia considerar a possibilidade de fazer um controlo mais fino dos bloqueios, fazendo lock ao nível das posições do vector, o que permitiria que dois clientes pudessem escrever em simultâneo no vector, desde que não precisassem de escrever nas mesmas posições (em nenhuma das duas operações, porque se mantém o problema dos Dirty Reads referido em 3.2 !). Esta abordagem designa-se por Granularidade e tem impacto no desempenho porque permite afinar a que nível se deve fazer o bloqueio, por forma a não manter inutilizáveis uma parte dos recursos, desnecessariamente.

No cenário em que existem vários vectores esta solução seria inadequada uma vez que haveriam recursos disponíveis que estariam desapoveitados. Desta forma uma implemenetação com gestão dos recursos com maior granularidade seria mais adequada.

A granularidade mais baixa possível é feita ao nível de uma posição de um serviço vetor sendo necessário garantir que não podem ser efetuadas duas escritas para a mesma posição do mesmo serviço. Este nível de granularidade permite máximizar a utilização dos serviços vetor.

# Two-Phase Lock Manager

O *Lock Manager* podia ser uma função adicional do TM. A opção por criar uma entidade autónoma tem a ver com o facto de se tratar de funções com características muito distintas. Nomeadamente, o Lock Manager não tem necessidade de contactar os vectores.

O algoritmo Two-phase Lock Manager é caracterizidado por uma fase de expansão, onde são adquiridos todos os locks, na operação getLocks. E por uma fase de contração onde todos os locks adquiridos são libertados, na operação unlock, sendo necessário armazenar de forma temporária os locks pedidos.

Quando um lock é pedido pode acontecer que o recurso pedido não esteja disponível. Nessa situação o Lock Manager guarda o pedido numa lista de pedidos pendentes e quando o recurso estiver disponível, após ser chamado o unlock doutra operação, é feita uma notificação ao cliente que pediu o lock.

Garante a concorrência no acesso aos recursos.

Two-phase lock (2PL)

Os read locks (shared locks) ou write locks (exclusive locks).

Recebe os pedidos de locks dos clientes (getLocks) e atribui-os, se estiverem disponíveis. Mantém o registo dos locks atribuídos e também dos pedidos ainda não satisfeitos (locks pendentes).

Quando há a libertação de um lock num recurso, a lista de locks pendentes é consultada, para verificar se existem pedidos de lock para esse recurso. Se existir, o lock é atribuído e o cliente que o pediu é notificado.

Read locks também são designados shared-locks e Write locks também são designados exclusive locks.

# Operações

# Comunicação entre o TM e o TPLM

A comunicação entre os dois gestores de transacções pode beneficiar a robustez do sistema, considerando que pode haver actualizações de estado antecipadas por uma das entidades tomar conhecimento e transmitir à outra. Estão neste caso situações como, por exemplo, a falha de um dos clientes.

# Resource Managers (RM)

Os RM são responsáveis por receber as instruções dadas pelos clientes (leitura e escrita nos vectores) e executarem-nas quando o TM dá essa indicação, no processo de 2PC (two-phase commit).

# Verificação do invariante

# Clientes

Um cliente delimita um conjunto de operações que vai ser executado, tendo em conta as propriedades ACID. Um cliente tem que garantir que é o único que detém os locks para os elementos do vector em que pretende efectuar operações de read ou write.

O cliente inicia a transacção no TM (tx\_begin) e recebe um ID. Pede os locks ao LM (getLocks) e, assim que os obtém, inicia as operações nos vectores. Quando termina, dá indicação de commit para o TM (tx\_commit) e, após obter confirmação, dá indicação de libertação dos locks ao LM (unlock).

Se existir uma falha, o cliente pode dar uma indicação de tx\_rollback, ao TM.

Para garantir a consistência, o commit tem que ser feito antes de libertar os locks, estes têm que ser mantidos até que haja confirmação de realização das operações. A evolução de estado tem que ser sempre consistente.

# Interface entre componentes

A comunicação entre o TM e o LM pode ser vantajosa em termos de reforçar a tolerância a falhas. Por exemplo, caso a comunicação com um dos clientes com o TM falhe e este detecte essa falha ou receba uma indicação de um cliente para abortar a transacção, o TM pode informar o LM que libertará os recursos atribuídos à transacção daquele cliente. Caso contrário, o LM ficará com o espaço ocupado com os recursos da transacção.

# Callback

Por exemplo, quando um cliente falha, todos os serviços que tinham alguma forma de associação, podem libertar os recursos respectivos.

Alguns frameworks implementam funcionalidades, como callback automático no caso de um cliente cair. Pode fazer parte do runtime.

Invocação mútua?

Invocação circular?

Estamos a usar o Framework OSGi -> Callbacks para o JAX-WS, possibilidade de ter chamada assíncrona.

Usando serviços remotos baseados em JAX-WS, como é que se reflecte? A abstracção está na implementação do remoting. O modelo programático é o do OSGi, não é o do JAX-WS.

# Conclusões

* 1. o Resumo do que foi discutido e realizado
  2. o Dificuldades e aspetos a melhorar

Ao longo destas aulas práticas foram introduzidos problemas inerentes a sistemas distribuídos e foram discutidos conceitos e ideias para solucionar esses problemas, como problemas de concorrência, consistência de dados e atomicidade das transações. Para resolver o problema de consistência de dados é feita uma verificação aos vetores sempre que é realizada uma transação, para o problema de múltiplos acessos de clientes e atomicidade existem duas entidades de gestão, O Transaction Manager e o Lock Manager. O Transaction Manager faz a garantia de atomicidade de cada transação dando as ordens de commit e abort das operações. O Lock Manager faz a gestão de Locks para garantir que não há conflitos de acesso dos clientes aos vetores.

As dificuldades encontradas foi a gestão e debug do projeto em OSGI e a gestão de dependências através do Maven, devido a isso realizámos o projeto em Java RMI. Também tivemos dificuldades em encontrar uma solução para resolver o problema de concorrência entre os clientes, que foi resolvido através de uma solução fornecida pelo professor.

Adequação ao cenário CMSP

# Bibliografia

[Osório, 2021] Luís Osório. Slides da cadeira de Infraestruturas de Sistemas Distribuídos. ISEL, 2021.

[Gray, 1978] Jim N. Gray. Notes on data base operating systems. Springer, 1978.

[Kleppman, 2021] Martin Kleppman. Distributed Systems Notes. University of Cambridge, 2020/21.

[Haerder, 1983] Theo Härder, Andreas [Reuter.](https://en.wikipedia.org/wiki/Andreas_Reuter) Principles of transaction-oriented database recovery. ACM Computing Surveys, 1983.

[Joachim, 2013] Nils Joachim, Daniel Beimborn, and Tim Weitzel. The influence of {SOA} governance mechanisms on {IT} flexibility and service reuse. The Journal of Strategic Information Systems, 2013.