Instituto Superior de Engenharia de Lisboa
Área Departamental de Engenharia Eletrónica, Telecomunicações e Computadores (ADEETC)
Mestrado em Engenharia Informática e Computadores

Infraestruturas de Sistemas Distribuídos (IESD)

Semestre de verão 2021/2022

Professor: Eng.º Luís Osório

Relatório do Trabalho Prático 2

Elaborado por:
Tiago Conceição, Nº47611
Ricardo Candeias, Nº42087
André Mendes, Nº30569
Grupo 08

Índice

1.Introdução

Este capítulo tem o propósito de apresentar sumariamente o problema, assim como, a abordagem feita. É também apresentada a estrutura do documento.

Pretende-se dar continuidade ao Trabalho.1 de IESD, fazendo as melhorias propostas (pelo professor), e a instanciação dos serviços (previamente implementados) num contexto de contentores e de rede Kubernetes (cloud computing).

O desenvolvimento efetuado no trabalho.1 de IESD foi a implementação/concretização de um cenário de validação de conceitos de coordenação no acesso concorrente a elementos serviço, conforme proposto na figura 1 do enunciado daquele trabalho.

O foco principal da abordagem é a discussão e o desenvolvimento de estratégias tendo em conta a coordenação de acesso concorrente a elementos serviço distribuídos, assim como, conceitos fundamentais de estruturação e utilização de uma rede Kubernetes e como é que estes influenciam a discussão das estratégias de controlo transacional e concorrência.

Este documento é estruturado em 5 secções: Introdução; Estado do Conhecimento e Análise e Discussão do Problema; Demonstrador Centrado na Coordenação; Cloud Computing; e Conclusões.

No tópico da Introdução é apresentado o problema, a abordagem elaborada, a estrutura do relatório, e é feita a interligação do trabalho.1 com o trabalho.2 de IESD.
No tópico Estado do Conhecimento e Análise e Discussão do Problema, será descrito o desenvolvimento de sistemas informáticos na base de elementos *Service*[1]. Será enunciada e justificada a coordenação de transações distribuídas. Serão descritos os quadros tecnológicos utilizados. Por último, será discutido o *deployment* dos serviços previamente desenvolvidos numa rede Kubernetes.
No tópico Demonstrador Centrado na Coordenação, será descrita a forma como foi desenvolvido o demonstrador, tendo em conta os desafios da **estratégia de implementação do serviço de coordenação de transações e da concorrência.**
No tópico Cloud Computing, serão descritas as alterações elaboradas nos serviços para garantir o seu correto funcionamento ao serem executados em contentores numa rede Kubernetes. Será também descrita sucintamente todos os procedimentos para a criação e configuração da rede.
Por fim, no tópico Conclusões, será apresentado de forma resumida o que foi discutido e realizado, e serão enunciadas as dificuldades e aspetos a melhorar neste trabalho prático.

2. Estado do Conhecimento, Análise e Discussão do Problema

Este capítulo tem o propósito de descrever o estado do conhecimento no âmbito do desenvolvimento de sistemas informáticos na base de elementos Service[1]. É também analisado e discutido o problema, nomeadamente, a gestão de transações distribuídas e concorrência no contexto Kubernetes e contentores. Serão descritos os quadros tecnológicos utilizados.

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

O desenvolvimento de um sistema informático na base de elementos Service (SOA - Arquitetura Orientada a Serviços)[2], permitiu-nos abordar cada elemento do cenário proposto no trabalho.1 como sendo um serviço.

Concretização de uma Arquitectura Orientada a Serviços:
Segundo o modelo ISoS, é um sistema informático (ISystem), composto por CES (Cooperation Enabled Services), e cada CES irá ter elementos que se designam por Service.
Entende-se por Serviço, uma identidade computacional independente, que tem a lógica computacional necessária para implementar uma determinada responsabilidade. Tem capacidades de comunicação com outros elementos atómicos, através de uma interface exposta pelo elemento com quem está a comunicar. Ao contrário daquilo que a arquitetura SOA define no seu modelo mais genérico, um cliente pode não ser um serviço - o cliente apenas acede ao serviço. Isto é assumido, por exemplo, na especificação JINI. Nesta situação, podemos dizer formalmente que um cliente é um serviço com 0 (zero) interfaces. (interface define-se como um *end-point* para alguma funcionalidade).

Tendo em conta este modelo, foi desenvolvido um sistema informático composto por 3 CES: CesVector, CesTC (CesTransanctionCoordination) e CesRegistry. O CES CesVector irá conter 2 serviços, estes responsáveis pelo servidor vector (SerVector) e o respetivo cliente (SerVectorCli). O CES CesTC é constituído por 2 serviços: um serviço destinado a coordenar as transações (SerTM) e um segundo serviço responsável pela coordenação do acesso a recursos partilhados, através de uma abordagem 2Phase-Lock. Por último, o CesRegistry contém um único serviço, SerRegistry que é destinado a possibilitar transparência à localização para os restantes serviços do sistema informático.

2.2 Coordenação de transações distribuídas

Conforme foi falado nas aulas, identifica-se dois sub-problemas nas propriedades ACID[3], no contexto da coordenação transacional: o problema da **atomicidade** e o problema da **concorrência**.

Considera-se o conceito de transação, como um conjunto de operações que têm de ser executadas de forma atómica. Num contexto de serviços, em que temos um sistema distribuído, ocorrem em tempo real inúmeras transações distribuídas, e existe a necessidade de coordenar as transações de modo a manter atomicidade e consistência.

2.3 *Deployment* dos serviços num sistema distribuído

No âmbito do trabalho 2, é introduzido o desafio de instanciar os serviços num contexto de contentores num ambiente *cloud (cloud computing).* Este ambiente poderá ter características que facilitam o *deployment* e gestão de um sistema distribuído, constituído por vários serviços, podendo assegurar a disponibilidade dos mesmos.

2.4 Solução a Implementar

Quando um serviço gestor de recursos (adiante designado de Resource Manager (RM)) recebe, de um serviço cliente (adiante designado de Application Program (AP)), uma operação no contexto de uma transação, este tem de se registar como participante nessa transação, no serviço de coordenação das transações. Esse serviço que tem a responsabilidade de coordenar as transações é o **Transaction Manager (TM)**. O Transaction Manager vem resolver a necessidade de atomicidade – ou a transação é realizada do início até ao fim, ou não é realizada. Para tal, a lógica de implementação da coordenação de transações distribuídas é feita na base do modelo **X/Open** (o qual define um modelo de coordenação de transações distribuídas). Este define um conjunto de especificações:

Commitment Protocol - Processo de coordenação quando a transação é terminada: O modelo X/Open sugere o protocolo **Two-Phase Commit Protocol (2-PC).**

Application Program (AP) - Estabelece a fronteira de uma transação, a que associa um conjunto de operações a executar de forma atómica (todas ou nenhuma).

Resource Managers (RM) – Serviço que gere um conjunto de recursos (bases de dados, ficheiros, etc.). Regista-se no TM como participante numa transação.

Transaction Manager (TM) - Gere transações com um identificador único associado, monitoriza as, é responsável pela sua conclusão e coordenação de falhas e recuperação.

Temos então as interfaces:

- XA - Comunicação entre serviço cliente (**AP**) ou **TM** e o serviço gestor de recursos (**RM**);
- TX - Comunicação entre o **TM** e o serviço cliente (**AP**);

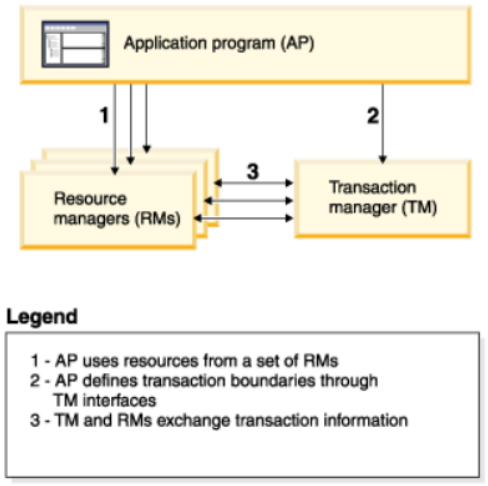

Figure 1 - Modelo de transações distribuídas X/Open

Existe a necessidade de coordenar a concorrência no acesso. Essencialmente, a estratégia consiste em serializar as transações concorrentes. O serviço responsável pela gestão desta concorrencia é o **Two-Phase Lock Manager (TPLM)**[5].

Por fim, para dar reposta ao *deployment* do sistema distribuído, pretende-se criar, segundo a indicação do enunciado do trabalho 2, uma rede Kubernetes para a orquestração dos contentores.

2.5 Descrição de quadros tecnológicos utilizados

Utilizou-se o quadro tecnológico gRPC[6], o qual é a implementação da Google do modelo RPC (Remote Procedure Call) que permite a comunicação entre dois pontos. Desta forma, opta-se assim por um modelo de comunicação directa entre os serviços, em alternativa ao modelo de comunicação indirecta – MOM (Message Oriented Midleware), na interação entre as entidades.

O gRPC é um quadro de comunicação baseado em contratos, implementados numa linguagem própria (*protobuf*). A utilização do gRPC permite que os serviços possam ser implementados em linguagens de programação diferentes, desde que estes tenham acesso ao contrato na fase de desenvolvimento dos mesmos, ou seja, o quadro tecnológico não é uma dependência para realizar uma integração entre serviços deste sistema informático.

Numa abordagem Collaborative Mobility Services Provider (CMSP)[1], serviços integrantes noutros sistemas informáticos, apenas têm de conhecer os contratos de comunicação *protobuf*. E conhecer o endereço IP e o porto do serviço de diretoria (SerRegistry) já enunciado no ponto 2.1.

O deployment e orquestração do sistema foi realizado recorrendo ao quadro tecnológico Kubernetes, no qual os serviços serão lançados em contentores respetivos ao quadro tecnológico Docker.

A criação e gestão da rede Kubernetes é realizada através da ferramenta Minikube, este quadro tecnológico open-source permite criar um cluster Kubernetes com um ou mais nós e gerir os mesmos de forma simplificada.

A rede Kubernetes, dependendo da sua configuração (do número de nós) será executada em uma ou mais máquinas virtuais, estas máquinas virtuais serão criadas e lançadas recorrendo ao quadro tecnológico open-source Virtualbox, sendo o quadro tecnológico proprietário Hyper-V uma alternativa viável para o mesmo fim.

3. Demonstrador Centrado na Coordenação

Neste tópico será apresentada a descrição da implementação do demonstrador centrado na coordenação.

3.1 Descrição do demonstrador

Implementou-se um Serviço adicional (SerRegistry) com a função de registar a participação dos elementos do sistema distribuído (TM, TPLM e Vector Services). Através deste serviço é possível adquirir ou registar outros serviços de forma genérica e parametrizável.

Assim existe um certo nível de transparência à localização[7] pois cada serviço que tente aceder às funcionalidades do sistema, apenas precisa de comunicar com este serviço para que este lhe forneça a informação de comunicação com os outros serviços do sistema distribuído.

O serviço SerRegistry facilita uma abordagem Collaborative Mobility Services Provider (CMSP) pois, para que uma integração entre um serviço do sistema e um serviço externo fosse realizada com sucesso, apenas seria necessário que o componente externo em questão tivesse posse do contrato de comunicação com o serviço pretendido, o contrato do SerRegistry e do endereço IP e porto do mesmo.

Conceção da comunicação entre serviços:

A comunicação entre os serviços TM, VectorClient (AP) e Vector (RM) seguem os standards referidos no de modelo X/OPEN[8], no qual foram implementadas as especificações TX e XA.

A especificação TX foi concretizada na comunicação entre o serviço VectorClient (AP) e o serviço TM e foram implementados os métodos: tx_begin, tx_commit e tx_rollback.

A especificação XA foi associada na comunicação entre o serviço TM e o serviço Vector (RM) no qual foram implementados os métodos: ax_reg, xa_prepare, xa_commit e xa_rollback.

Estratégia de implementação do serviço de coordenação de transações e da concorrência:

O **Transaction Manager (TM)** implementa o protocolo **Two-Phase Commit**[4] sugerido no modelo X/Open. Este protocolo funciona com um processo de votação, e consiste, como o próprio nome diz, em executar o processo de *commit* de uma transação em duas fases:

1ª - fase - *Prepare*, na qual o TM vai percorrer todos os serviços (RM) que se registaram como participantes naquela transação, e vai "avisá-los" que se devem "preparar" para o Commit. De certa forma, é uma maneira do mediador perguntar aos serviços: "Correu tudo bem? Preparar para o Commit."
Todos aqueles serviços vão responder/votar.

2ª – fase - *Commit*. Se todos os serviços (RM) responderem *Ready-to-commit*, então isso significa que correu tudo bem, e o TM vai percorrer novamente os serviços para dar ordem de Commit.
Se algum serviço (RM) responder *abort*, ou nem sequer responder, então a transação deve ser abortada. O TM vai percorrer novamente os serviços (RM) para dar ordem de Rollback à transação.

As transações são identificadas por um ID único, gerido pelo TM, o qual é fornecido ao serviço cliente (AP). O objetivo do ID único é identificar o contexto das operações.

A responsabilidade de coordenação de transações e da concorrência foi estruturada enquanto abstração CES (Cooperation Enabled Services), composto por dois elementos serviço:

- **TM (Transaction Manager)** – implementação do conceito de transação – tem a responsabilidade de coordenação de transações distribuídas. Foi desenvolvido na base do modelo X/Open. Todos os serviços que são gestores de recursos (neste caso, o recurso é o vector), vão participar numa transação. Formalmente, no modelo X/Open, esses serviços gestores de recursos chamam-se **Resourse Manager (RM)**.

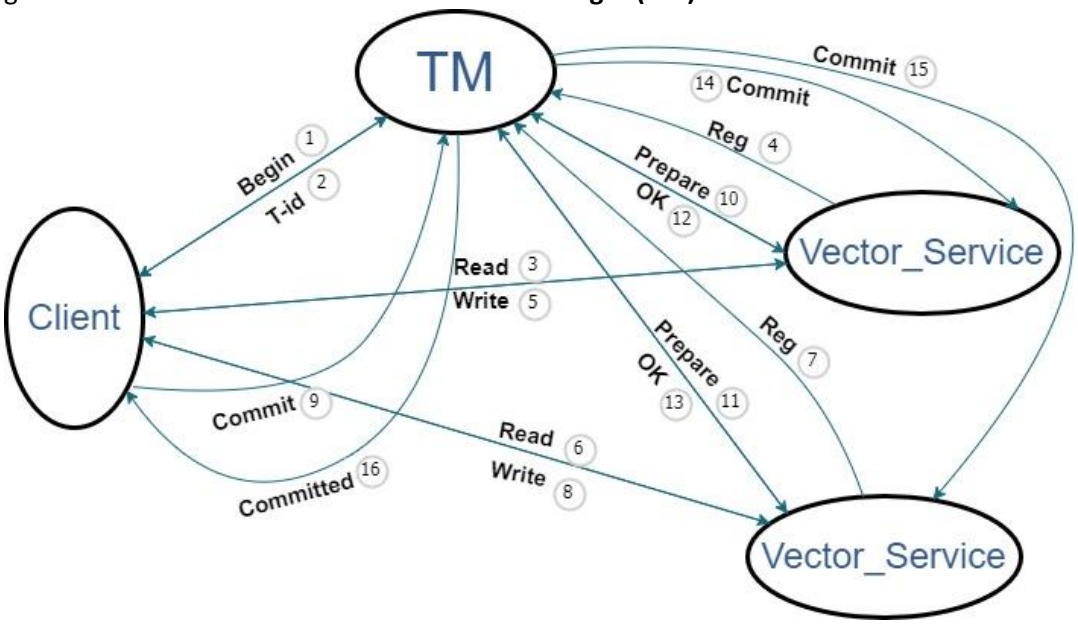

Figure 2 - Esquema ilustrativo do algoritmo do TM

- **TPLM (Two-Phase Lock Manager)** – implementação da coordenação da concorrência – tem a responsabilidade de garantir o isolamento no acesso aos recursos.

  Foi tida em conta a importância da granularidade do *Lock[9]*, que deverá ser aquela que maximiza a concorrência. Caso contrário, introduz-se latência. Como tal foi adotado o protocolo de *two-phase locking*. Para o efeito, definiu-se que o *lock* terá dois tipos, um que controla o acesso a um recurso de leituras e outro de escritas. Este paradigma permite bloquear um recurso apenas o quanto estritamente necessário.

  A granularidade do *Lock* pode ser tão mais importante quanto maior for a dimensão do recurso (vector).
  Por exemplo: se tivermos um recurso com dimensão elevada, e um (serviço) cliente pretender aceder apenas a duas ou três posições, então não queremos que todo o recurso fique bloqueado para aquele cliente, deixando todos os outros clientes em espera, quando provavelmente as posições pretendidas pelos outros clientes nem são as mesmas.

  O serviço cliente, após obter o ID da transação, vai solicitar ao TPLM os Locks que necessita para a transação. Os Locks são caracterizados/identificados da seguinte forma:
  - [Resourse Manager].[posição específica no vector]
  - Tipo de Lock:
    - Lock para leitura (lock partilhado)
    - Lock para escrita (lock exclusivo)

  A implementação do protocolo **Two-Phase Locking** exige que uma transação deve obter todos os *locks* antes de executar as suas operações. Apenas no fim da execução das mesmas vai liberar os *locks*.
  Existem, portanto, duas fases bem definidas:

  1ª fase – (fase ascendente) aquisição de todos os Locks necessários à transação. O TPLM apenas atribui os Locks ao (serviço) cliente quando todos estiverem disponíveis. Se algum Lock não estiver disponível, então o cliente ficará em espera.

  2ª fase – libertação de todos os Locks. O serviço cliente liberta todos os Locks após concluída a transação.

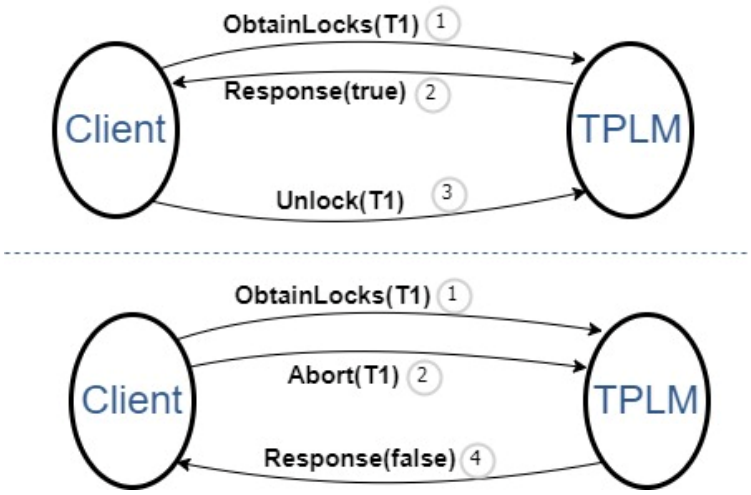

Figure 3 - Esquema ilustrativo do algoritmo do TPLM

Com esta estratégia, conseguimos evitar o problema do *deadlock[10]*, uma vez que não existe aquisição de Locks durante as operações da transação. Por outro lado, esta estratégia não favorece a concorrência, porque há Locks que a determinado momento já não são necessários, mas só serão libertados no final.

Implementação:

Em relação ao modelo de two-phase locking implementado criou-se um elemento que representa o estado de disponibilidade, quer seja para escrita quer seja para leitura, de cada elemento individual de todos os Vector Services. O acesso à estrutura de dados que contém esta informação é mediada por um lock, ou seja, apenas um pedido de cada vez poderá verificar a disponibilidade de locks.

Um pedido de lock de um determinado elemento para leitura, irá deixar outras leituras disponíveis, mas impedirá escritas para o mesmo elemento.

Um pedido de lock de um determinado elemento para escrita, impedirá quaisquer leituras ou escritas adicionais para o mesmo elemento.

Qualquer pedido que não lhe seja concedido os locks aos elementos que pretendia ficará à espera sobre o lock. Quando uma transação libertar os locks dos elementos que precisou para a sua operação, todos os outros pedidos à espera de locks serão notificados e tentaram novamente obter os locks para os seus elementos.

[Qualquer espera sobre qualquer lock neste trabalho tem um timeout. Podendo falhar desse modo.]

Validação:

A validação consiste em garantir o invariante, conforme explicado no enunciado do trabalho.
Esta operação é totalmente independente ao funcionamento do sistema. A verificação do invariante só é feita quando todas as transações estão *commited[11]*, de forma a ser garantida consistência no resultado.

4. Cloud Computing

Este capítulo tem o propósito de abordar os conceitos fundamentais de estruturação e utilização de contentores e de configuração de uma rede Kubernetes.

Vantagens da instanciação dos serviços num contexto de contentores e de rede Kubernetes (cloud computing)

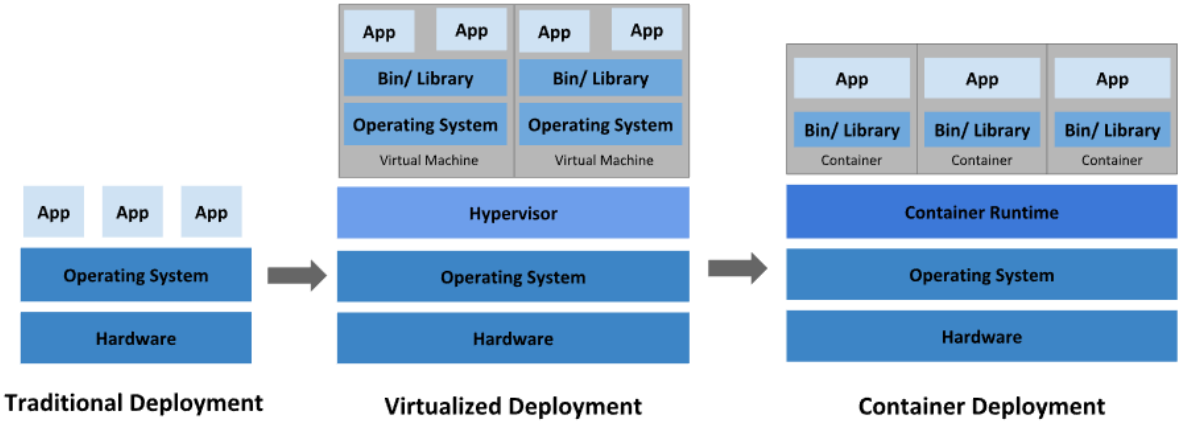

Figura 4. Fonte: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Na virtualização, cada VM tem o seu próprio Sistema Operativo (SO).

Em comparação com a virtualização, na contentorização vamos ter o *Container Runtime* (o Docker neste caso) que vai simular um SO para cada *container*. Na verdade, ele vai usar o SO do host. O *Container Runtime* isola esse processo. As vantagens de usar contentores: isolamento das aplicações; otimização de recursos (consegue-se montar um ambiente mais rapidamente).

O *Minikube* é uma ferramenta open-source que possibilita de forma simples a criação de uma rede Kubernetes, em que a criação da rede resultará na criação de máquinas virtuais, recorrendo a diferentes drivers, onde os nós K8s serão instanciados.

Kubernetes é uma plataforma open-source para gerir processos e serviços em contentores. Facilita a configuração e automação declarativa dos mesmos. Permite correr sistemas distribuídos de forma resiliente. Toma conta da escalabilidade e controlo de falhas das aplicações, bem como providencia padrões de *deployment*. O Kubernetes providência as seguintes funcionalidades:

- Descoberta de serviços e balanceamento de carga;
- Orquestração de armazenamento;
- Automação de *rollouts* e *rollbacks;*
- Empacotamento automático de containers em nós;
- Auto-Regeneração;
- Gestão secreta da configuração;

Numa rede Kubernetes existem vários tipos de objetos, dos quais cada um contém responsabilidades únicas, desta forma apresentamos os objetos considerados mais relevantes para o deployment do sistema distribuído:

- Pod: Componente que abstrai um conjunto de contentores, consiste em um ou mais contentores que partilham o mesmo host system e recursos. Dentro da rede K8s, cada Pod contém um endereço IP único. Estes contentores são criados e executados através do Docker.
- Service: Expõe uma interface, tornando um Pod acessível dentro (através de variáveis de ambiente) e fora da rede K8s, pois nesta rede todos os serviços K8s são do tipo NodePort, no qual vão expor o serviço K8s em cada IP dos nós K8s num determinado porto estático.
- ReplicaSet: Gerência a disponibilidade do número necessário de Pods idênticos, garantindo a disponibilidade dos Pods.

Os objetos Kubernetes são criados através da interpretação de ficheiro de configuração com a extensão yaml. Desta forma é possível elaborar o deployment do sistema distribuído no qual optamos por construir a seguinte configuração:

- Um Pod e um serviço k8s para o serviço Registry
- Um Pod e um serviço k8s para o serviço TM
- Um Pod e um serviço k8s para o serviço TPLM
- Um Pod e um serviço k8s para cada instância do serviço Vector

Segue uma figura que ilustra a configuração da rede Kubernetes, de modo a facilitar a compreensão da mesma, é possível verificar os efeitos dos serviços k8s do tipo NodePort, que permitem criar entry points para os serviços Client externos ao cluster:

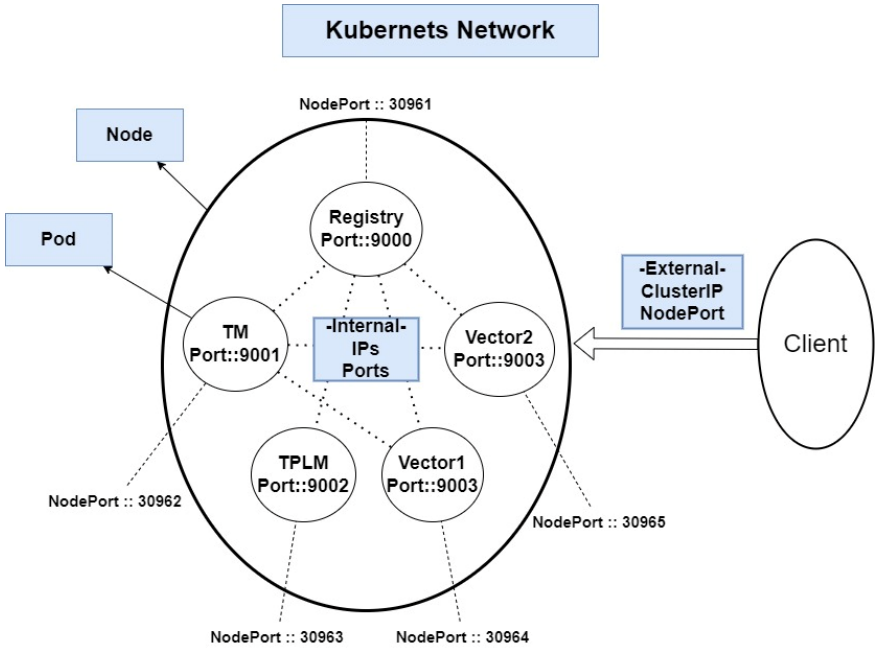

Figure 5 - Configuração da rede Kubernetes

(O ficheiro de configuração da rede estará disponível na diretoria raiz do projeto).

5. Conclusões

5.1 Resumo do que foi discutido e realizado

Os conceitos e as estratégias mais importantes que foram analisados, discutidos, e implementados, estiveram relacionados com a coordenação de transações distribuídas e da concorrência.

Conforme foi discutido anteriormente, identificamos 2 sub-problemas nas propriedades ACID, no contexto da coordenação transacional: o problema da atomicidade e o problema da concorrência.

Foi também constatada a importância da granularidade do *Lock*, que deverá ser aquela que maximiza a concorrência. Caso contrário, estaremos a introduzir latência, especialmente em recursos de elevada dimensão.

Foi criada uma rede Kubernetes com um nó. Cada serviço foi instanciado num *pod* individual. A comunicação externa é efetuada através do *IP* do nó e um *nodePort* que expõe o serviço para fora da rede Kubernetes. Internamente os serviços comunicam entre si com *IP*s & *Port*s locais e conhecidos do cluster Kubernetes. Desta forma obteve-se o sistema informático distribuído desenvolvido no âmbito do trabalho 1, *deployed* numa rede kubernetes. O que consiste no objetivo do trabalho 2.

5.2 Dificuldades e aspetos a melhorar

Ficaram claras as dificuldades e os desafios no desenvolvimento de sistemas informáticos na base de elementos (Service) autónomos, com interdependências.

Foram discutidos cenários de escalabilidade, com um número elevado de serviços cliente e vector.
Com um número elevado de serviços vector, o modelo de comunicação teria de ser repensado para uma abordagem mais dinâmica.
Com um número elevado de serviços cliente, iriamos ter um aumento da latência como consequência direta do aumento da concorrência. Lembremos que as transações correm de forma serializada.
Talvez fosse necessário distribuir a responsabilidade dos serviços, aumentando o número de réplicas destes serviços, para distribuição de carga. Nesse caso, teríamos de assegurar o sincronismo em relação ao estado dessas réplicas através de algoritmos de consenso, garantido a tolerância a falhas neste caso.

A plataforma Kubernetes contém objetos denominados por StatefulSets, mesmo não sendo abordados neste trabalho prático, com algum estudo poderiam possibilitar uma abordagem interessante no que diz respeito à tolerância a falhas.

6. Referências

[1] - **On Reliable Collaborative Mobility Services: 19th IFIP WG 5.5 Working Conference on Virtual Enterprises, PRO-VE 2018, Cardiff, UK, September 17-19, 2018, Proceedings**

[2] – What is Service-Oriented Architecture?, 2019, Software Development Community

[3] – ACID, 2021, Wikipedia

[4] – Two-phase commit protocol, 2022 , Wikipedia

[5] – Two-phase locking, 2021, Wikipedia

[6] - gRPC, 2022

[7] – Location transparency, 2021, Wikipedia

[8] – X/Open, 2021, IBM

[9] – Lock, En-Academic

[10] – DeadLock, En-Academic

[11] – Commit, 2011, Techopedia