

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel Jorge C. Lima P. Valente

Mobilidade de micro serviços em datacenters suportada por Software Defined Networks (SDN)

Janeiro 2020



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel Jorge Valente

Mobilidade de micro serviços em datacenters suportada por Software Defined Networks (SDN)

Dissertação de Mestrado
Mestrado em Engenharia de Redes e Serviços Telemáticos

Trabalho realizado sob orientação de:
Prof. Dr. Pedro Nuno Miranda de Sousa

Supervisor na empresa:
Eng. Rui Jorge Augusto

Janeiro 2020

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.
Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.
Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Atribuição

CC BY

https://creativecommons.org/licenses/by/4.0/

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

RESUMO

O termo microserviços não é propriamente recente, existem inúmeras referências ao longo da última década sobre este conceito, no entanto não existe um verdadeiro consenso sobre quem foi o primeiro a introduzir esta abordagem. Independentemente da indefinição sobre o autor, as vantagens e os desafios da sua utilização como base ao desenvolvimento de novas aplicações são hoje bem conhecidos. É também possível verificar que esta arquitetura de software, que inicialmente era mais utilizada em desenvolvimentos nativos para a Cloud, é cada vez mais utilizada em centros de dados locais, o que lança novos desafios às infraestruturas de rede dos centros de dados.

O simples facto dos microserviços serem independentes entre si, permite que sejam desenvolvidos, distribuídos e atualizados individualmente, desta forma conseguimos atualizações mais rápidas e com maior frequência, endereçando a constante mudança de requisitos aplicacionais que se verifica em variadíssimas áreas de negócio.

No entanto a adoção de novas plataformas deve garantir que estes novos paradigmas integram, e idealmente beneficiam de tecnologias ou soluções já existentes. Num ambiente altamente distribuído, como é o caso de arquiteturas baseadas em microserviços, é evidente que a componente de comunicações tem um papel preponderante na qualidade do serviço, pelo que nos casos em que o centro de dados onde se pretende utilizar a plataforma de orquestração utiliza redes baseadas em software (SDN), o ideal é que as soluções integrem de forma bastante profunda. Esta integração é ainda mais relevante se o referido centro de dados apresentar uma arquitetura híbrida, isto é, composto por capacidade de computação em múltiplos datacenter físicos, mas também em provedores de Clouds públicas (Azure, AWS, Google Cloud, etc.).

Este trabalho pretende enumerar os principais desafios à utilização de containers em centros de dados, bem como descrever a melhor forma de integrar a solução de gestão de rede de centros de dados do fabricante Cisco (ACI - Application Centric Infrastructure) com a solução de orquestração de containers mais utilizada atualmente (Kubernetes). É também âmbito deste trabalho apresentar uma proposta à integração do ambiente descrito anteriormente (ACI+Kubernetes) com soluções de orquestração de containers alojados em Clouds públicas, nomeadamente na cloud da Microsoft (Azure).

Palavras-Chave: ACI, Cloud, Kubernetes, Microserviços, SDN

ABSTRACT

The term microservices isn't exactly new, there are innumerous references in the last decade about this concept, however there isn't a true understanding about who was the first to introduce this approach. Nevertheless, the advantageous and challenges of adopting this methodology, that started to be use more in cloud native applications, but rapidly extended to all type of applications regardless of the location where they will be executed, creating a new challenge to the network infrastructure in the datacenter.

The simple fact that the microservices are independent among themselves, allows them to be developed, distributed and updated individually, this way we can have faster and frequent updates, addressing the constant need of reviewing application requisites that we see happening in several business areas.

However, the adoption of this new platforms must consider or even benefit, with the integration with the already existing technologies. In a highly distributed environment, like the microservices architecture, it's clear that the communication aspect has a preponderant role in the overall quality of service, so in scenarios where the data center already as software defined network solution has, the ideal scenario is that this solution integrates with every solution that we add to the datacenter. This integration is even more relevant if the data center already has an hybrid architecture, combining several physical locations with public cloud providers (Azure, AWS, Google Cloud, etc.).

This paper pretends to identify the main challenges to the usage of containers in data centers, as well as describe the best way of integrating the datacenter network management tool from Cisco (ACI - Application Centric Infrastructure) with the main container orchestrator (Kubernetes). It will also be presented in this paper a proposal on how to integrate the previous mentioned environment (ACI+Kubernetes) with orchestrators solutions managed by public cloud providers, namely Microsoft Azure.

Key-Words: ACI, Cloud, Kubernetes, Microservices, SDN

ÍNDICE

1. INTRODUÇÃO 1
1.1. Enquadramento e Motivação 1
1.2. Objetivos 2
1.3. Sumário das principais contribuições 3
1.4. Estrutura da dissertação 3
2. ESTADO DA ARTE 5
2.1. Introdução ao conceito de microserviços 5
2.1.1. Transformação de aplicações monolíticas em microserviços 5
2.1.2. Providenciar um ambiente consistente para cada aplicação 8
2.2. Tecnologia baseada em Containers 8
2.2.1. O que são Containers? 8
2.2.2. Introdução ao Docker como plataforma para Containers 12
2.3. Orquestração de containers com Kubernetes 15
2.3.1. A origem do Kubernetes 15
2.3.2. Visão geral do Kubernetes 16
2.3.3. Descrição da arquitetura de um cluster Kubernetes 17
2.3.4. Instalar uma aplicação num cluster de Kubernetes 18
2.3.5. Os benefícios de utilizar Kubernetes 20
2.3.6. Principais componentes do Kubernetes 22
2.4. A componente de rede de um cluster Kubernetes 28
2.4.1. Requisitos de rede do Kubernetes 28
2.4.2. Como funciona a componente de rede do Kubernetes 29
2.5. Software Defined Network 32
2.5.1. Evolução histórica das SDN 34
2.5.2. Programabilidade da rede 35
2.5.3. SDN como parte de uma estratégia de Orquestração 35
2.5.4. Soluções SDN para o centro de dados 36

2.5.5. Introdução à solução SDN para centros de dados da Cisco 40
2.5.6. Descrição do funcionamento do ACI 46
2.6. Arquiteturas híbridas (Hybrid Cloud) 48
2.6.1. Principais desafios das arquiteturas hibridas 49
2.6.2. Soluções de interligação com cloud's públicas 50
2.6.3. Funcionalidades de controlo de acessos nas Clouds públicas 51
2.6.4. Controlo de acesso em datacenters locais (Cloud's Privadas) 52
2.6.5. Containers em arquiteturas hibridas 53
3. ARQUITETURA E SOLUÇÃO PROPOSTA 58
3.1. Introdução ao problema 58
3.2. Descrição dos casos de uso/validação 59
3.2.1. Aplicação de apoio ao processo de venda (POS) 59
3.2.2. Facilitar a movimentação de aplicações para a cloud 59
3.3. Descrição da arquitetura pretendida 60
4. DESCRIÇÃO DA BANCADA DE TESTES 62
4.1. Instalação de um cluster de Kubernetes 62
4.1.1. Instalar um cluster Kubernets na cloud pública Microsoft Azure 63
4.1.2. Instalar um cluster localmente no centro de dados 75
4.2. Implementação da integração do ACI com Kubernetes 91
4.2.1. Preparação do ACI (*acc-provision*) 91
4.2.2. Preparação da componente de rede dos nós do cluster (Linux) 95
4.2.3. Instalação do plug-in CNI do ACI 97
4.3. Integração do ACI com a cloud Azure 100
4.3.1. Ligação entre o centro de dados local e a cloud 101
4.3.2. Tradução do modelo de politicas do ACI para o modelo do Azure 103
5. TESTES E RESULTADOS 105
5.1. Validação da inter-operabilidade entre o ACI e o Kubernetes 105
5.1.1. Sincronismo e visibilidade (aplicação *guest-book)* 105
5.1.2. Balanceamento de tráfego por hardware 107

5.1.3. Politica de controlo de acessos (aplicação MyHero) 110
5.2. Validação da integração entre o ACI e a Cloud Azure 118
5.2.1. Conectividade entre o centro de dados e o Azure. 118
5.2.1. Cloud APIC (CAPIC) 119
5.2.2. Multi-site Orchestrator (MSO) 120
6. CONCLUSÕES 124
6.1. Resumo do trabalho 124
6.2. Principais contribuições 125
6.3. Trabalho Futuro 125
6.3.1. Integração de storage através de CSI (Container Storage Instance) 125
6.3.2. Integração de um cluster Kubernetes em simultaneo com F5 e ACI 126
7. ANEXOS 132
7.1. Instalar um cluster K8S local com um nó usando o Minikube 132
7.1.1. Instalar o Minikube 132
7.1.2. Lançar uma aplicação no cluster através de comando Imperativos 134
7.1.3. Executar uma aplicação no cluster através de objetos declarativos 136
7.1.4. Escalar uma aplicação 137

LISTA DE FIGURAS

Figura 1-Aplicação monolítica versus microserviços 6
Figura 2-Cada microserviço pode ser escalado individualmente 7
Figura 3-Várias aplicações a correr no mesmo servidor podem ter conflitos nas dependências 8
Figura 4-Usar MV's para isolar aplicações VS usar containers 10
Figura 5-Diferenças entre o uso do CPU Físico 10
Figura 6 - Imagens, Registos e Containers em Docker 13
Figura 7 - Seis aplicações a correr em três MV vs. correrem em containers Docker 14
Figura 8 - O Kubernetes abstrai todo o centro de dados numa única plataforma 16
Figura 9 - Componentes de um cluster Kubernetes 17
Figura 10 - Arquitetura básica de um cluster de Kubernetes (quatro nós) com apenas uma aplicação 19
Figura 11 - Relação entre Pod's, Containers e Worker Nodes 23
Figura 12 - Representação de um POD com múltiplos containers 24
Figura 13 - Representação da utilização de Serviços 26
Figura 14 - Relação entre os PersistenVolumes e PersistenteVolumeClaim 28
Figura 15 - Requisitos da componente de rede do Kubernetes 29
Figura 16 - Comunicação entre POD’s do mesmo nó 30
Figura 17 - Representação do endereçamento IP de um cluster de Kubernetes 30
Figura 23 - Software Defined Network 33
Figura 24 - Descrição do pacote VXLAN 37
Figura 25 - Representação do conceito de VTEP 38
Figura 26 - Formato do pacote NVGRE 39
Figura 27 - Representação da rede Underlay e Overlay 39
Figura 28 - Componentes do ACI (retirado de: “Cisco ACI Starter KITS2) 41
Figura 29 - Política no ACI (retirado de: ACI policy driven datacenter [24]) 41
Figura 30 - Arquitetura spine/leaf (retirado de: refreshing my memory on ACI [25]) 42
Figura 31 - Comunicação entre EPG's (adaptado de: Cisco ACI Test Drive) 44
Figura 32 - Estrutura lógica do ACI (adaptado de: Cisco ACI Test Drive) 44
Figura 33 - Exemplo multicliente (adaptado de: Cisco ACI Test Drive) 46
Figura 34 - Integração com motores de virtualização (retirado de: Apresentação do Cisco Live) 48
Figura 35 – Exemplo de arquitetura/ligação VPN (retirado da página da MSFT [27]) 51
Figura 36 - Exemplo de arquitetura/ligação dedicada (retirado da página da MSFT [28]) 51
Figura 37 - Comparativo de componentes de rede 54
Figura 38 - Arquiteturas de rede, modelo nativo 55
Figura 39 - Arquiteturas de rede, modelo overlay 56
Figura 40 - Arquitetura de rede, integração com Cisco ACI 57
Figura 41 - Representação da mobilidade entre locais considerados 59
Figura 42 - Representação da integração de rede K8S com ACI 60
Figura 43 - Integração Kubernetes com centro de dados local 60
Figura 44 - Integração do Kubernetes com solução IaaS 61

Figura 45 - Integração do Kubernetes com solução PaaS 61
Figura 71 - Configurações iniciais do cluster em Azure 64
Figura 72 - Escalabilidade para ACI (retirado de PPT da Microsoft) 65
Figura 73 - Configurações de escalabilidade em Azure 65
Figura 74 - Configurações de autenticação em Azure 66
Figura 75 - Configurações de rede do cluster em Azure 67
Figura 76 - Configurações da monitorização de cluste em Azure 67
Figura 77 - Acesso ao AKS através da Cloud Shell 68
Figura 78 - Configuração do contexto de acesso ao AKS 68
Figura 79 - Output dos comandos kubectl 69
Figura 80 - Acesso ao AKS através de uma máquina Windows 69
Figura 81 - Expor um pod alojado em AKS 70
Figura 82 - Descrição de um deployment em AKS 71
Figura 83 - Aceder a serviços através de variáveis de ambiente 71
Figura 84 - Pod's do Namespace kube-system 72
Figura 85 - Aceder ao Dashboard nativo do Kubernetes do AKS 73
Figura 86 - Representação AKS (Node, Pod, Persistent Volume) - Retirado de doc. MSFT 73
Figura 87 - AKS StorageClasses 74
Figura 88 - Criação de um Persistent Volume Claim 74
Figura 89 - Associar um PVC a um POD [39] 74
Figura 90 - Arquitetura de alta disponibilidade (retirada de [42]) 75
Figura 91 - Conclusão da instalação do Master 77
Figura 92 - Estado do cluster após o "kubeadm init" 77
Figura 93 - Confirmação do Worker 78
Figura 94 - Integração BIG-IP com K8S modo Nodeport (retirado de [45]) 79
Figura 95 - Integração BIG-IP com K8S modo Cluster (retirado de [45]) 80
Figura 96 - Código para a criação do ClusterRole e do ClusterRoleBinding 81
Figura 97 - Código para a criação do BIG-IP Controller 82
Figura 98 - Ambiente de teste da F5 – deployment 82
Figura 99 - Ambiente de teste da F5 - service 83
Figura 100 - Ambiente de teste da F5 – Configmap 83
Figura 101 - Ambiente de teste da F5 – Confirmação do LB 84
Figura 102 - Configuração do CNI Flannel 85
Figura 103 - Mapeamentos no túnel VXLAN (adaptado de [47]) 86
Figura 104 - Comandos do F5 para a criação do túnel VXLAN 86
Figura 105 - Status do nó F5 no cluster 87
Figura 106 - Política de balanceamento no YAML 87
Figura 107 - Balanceamento com integração ClusterMode 88
Figura 108 - Política de balanceamento em Clustermode 88
Figura 109 - Criação de Persistent Volume para NFS 90
Figura 110 - Criação de Persistent Volume Claim para NFS 90
Figura 111 - Criação de POD com volume NFS 91
Figura 112 - Verificação da criação do volume persistente 91
Figura 113 - Ficheiro de preparação do ACI 92
Figura 114 - Verificação no APIC dos Bridge Domains 94

Figura 115 - Verificação no APIC dos Contracts 94
Figura 116 - Verificação no APIC do Application Profile criado 95
Figura 117 - Visualização no APIC do domínio associado ao cluster criado 95
Figura 118 - Rede de interligação apenas com a placa de gestão configurada 96
Figura 119 - Verificação no APIC da integração da rede 97
Figura 120 - Rede de interligação após a configuração da segunda placa de rede 97
Figura 121 - Resultado do comando de instalação do ACI-CNI 97
Figura 122 - Validação da integração no Kubernetes 98
Figura 123 - Logs de sincronização do cluster com o APIC 98
Figura 124 - Validação no ACI da integração kube-system 99
Figura 125 - Validação no ACI da integração para balanceamento e redireccionamento 99
Figura 126 - Validação no ACI da integração para visibilidade 99
Figura 127 - Representação da arquitetura ACI mais Azure (retirado de [51]) 100
Figura 128 - Componente underlay da arquitetura ACI mais Azure (adaptado de [51]) 101
Figura 129 - Conectividade dentro do Azure (retirado de [51]) 102
Figura 130 - Arquitetura de rede da ligação do ACI ao Azure (retirado de [51]) 103
Figura 131 - Política de segurança em ACI (retirado de [51]) 104
Figura 132 - Política de segurança em Azure (adaptado de [51]) 104
Figura 133 - Correspondência entres os objetos ACI e Azure (adaptado de [51]) 104
Figura 134 - Arquitetura da aplicação de teste guestbook 105
Figura 135 - Criação da aplicação de teste guestbook 105
Figura 136 - Aceder à aplicação guest-book 106
Figura 137 - Validação no ACI da aplicação de teste guest-book 106
Figura 138 - Validação no ACI da aplicação de teste guestbook (2) 106
Figura 139 - Arquitetura da aplicação de teste MyHero 107
Figura 140 - POD's criados para a aplicação MyHero 107
Figura 141 - Acesso à aplicação MyHero 108
Figura 142 - Aplicação MyHero 108
Figura 143 - Adição de entrada no L3Out (common) para a aplicação MyHero 108
Figura 144 - Contrato no ACI 109
Figura 145 – Service Graph no ACI 109
Figura 146 - Política de encaminhamento de tráfego no ACI 109
Figura 147 - Representação da entrada de tráfego na solução K8s+ACI 110
Figura 148 - Representação dos diferentes tipos de isolamento 111
Figura 149 - Novo EPG para isolamento baseado em Namespace 112
Figura 150 - Criação de um novo Namespace 112
Figura 151 - Adicionar annotation ao namespace 112
Figura 152 - Criação da aplicação com isolamento namespace 113
Figura 153 - POD trojanapp no EPG kube-default 113
Figura 154 - Validação de acessos no isolamento baseado em namespace 113
Figura 155 - POD pwndui no EPG ns-myhero 114
Figura 156- Validação de acessos no isolamento baseado em namespace (2) 114
Figura 157 - Application Profile da aplicação MyHero no ACI 115
Figura 158 - Detalhe referente aos acessos ao EPG myhero-data 115
Figura 159 - Annotation adicionado ao Deployment do myhero-data 116

Figura 160 - POD's do EPG myhero-ui com isolamento por deployment 117
Figura 161 - Visibilidade do ACI sobre os objetos do Kubernetes 117
Figura 162 - Criação do container de teste da política baseada em deployment 118
Figura 163 - Validação da política de controlo por deployments 118
Figura 164 - Validação do OSPF entre CSR's físicos e virtuais 119
Figura 165 - Validação do Overlay BGP-EVPN 119
Figura 166 - Interface gráfico do cAPIC 120
Figura 167 - Dashboard inicial do MSO 120
Figura 168 - MSO Criação de EPG 121
Figura 169 - Rede Azure aprendida via BGP 121
Figura 170 - Selectors criados no EPG 122
Figura 171 - Validação de conectividade 122
Figura 172 - Máquina virtual no EPG de SHARED_TEST2 123
Figura 173 - Container no EPG SHARED_TEST2 123
Figura 62 - Definições de uma máquina virtual 133
Figura 63 - Informação referente ao arranque do Minikube 134
Figura 64 - Listagem de serviços expostos num cluster Kubernetes 135
Figura 65 - Endereço externo do container e endereço de rede 135
Figura 66 - Output do comando kubectl get pods 136
Figura 67 - Output do comando kubectl describe pod 136
Figura 68 - Exemplo 1 de um ficheiro YAML 137
Figura 69 - Ficheiro YAML de uma solução com replicas 138
Figura 70 - Demonstração do balanceamento automático 138

LISTA DE ABREVIATURAS E SIGLAS

ACI	Application Centric Infrastructure
AMQP	Advanced Message Queuing Protocol
API	Applications Program Interface
APIC	Application Policy Infrastructure Controller
ARP	Address Resolution Protocol
ASG	Application Security Group
AWS	Amazon Web Services
BD	Bridge Domains
BGP	Border Gateway Protocol
CAPIC	Cloud Application Policy Infrastructure Controller
CLI	Command Line Interface
CNI	Container Network Interface
DCI	Data Center Interconnect
DNS	Domain Name System
EPG	End Point Group
EVPN	Ethernet Virtual Private Network
GRE	Generic Routing Encapsulation
HTTP	ypertext Transfer Protocol
IaaS	Infrastructure as a Service
IP	Internet Protocol
JSON	JavaScript Object Notation
K8S	Kubernetes
L4-L7	Layer 4 to Layer 7
LAN	Local Area Network
MAC	Media Access Control
MSO	Multi-Site Orchestrator
MV	Máquina Virtual
NAT	Network Address Translation
NETCONF	Network configuration
NFS	Network File System
NSG	Network Security Group
NVGRE	Network Virtualization using Generic Routing Encapsulation
OSI	Open System Interconnect

POS	Point of Sail
PV	Persistent Volume
PVC	Persistent Volume Claim
REST	Representational State Transfer
RESTful	Capacidade de usar REST
RFC	Request for comment
SDN	Software Defined Networks
SOA	Service Oriented Architecture
TCP	Transport Control Protocol
TNI	Tenant Network ID (NVGRE)
UDP	User Datagram Protocol
UTS	UNIX Timesharing System
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNET	Virtual Network
VPN	Virtual Private Network
VRF	Virtual Routing Forwarding
VTEP	VXLAN Tunnel End Point
VXLAN	Virtual Extensible Local Area Network

1

1. INTRODUÇÃO

1.1. Enquadramento e Motivação

Antes do aparecimento das abordagens aplicacionais baseadas em microserviços [1], as aplicações eram criadas em ambientes monolíticos, tipicamente executados num único processo ou num número reduzido de processos espalhados por alguns servidores. Estes sistemas são atualmente vistos como legacy, no entanto existem imensas aplicações empresariais em produção baseadas nestes paradigmas. Estas aplicações têm ciclos de atualização lentos e muito pouco frequentes, na prática o que vemos é que no fim de cada ciclo de desenvolvimento a equipa de desenvolvimento empacota a aplicação e encaminha-a para a área de operações/infraestruturas que procede à instalação e consequente monitorização. Em muitas situações, verificamos uma total ausência de redundância, pelo que em causa de falha de hardware é necessário efetuar intervenções manuais no sentido de colocar os serviços novamente disponíveis.

Uma arquitetura baseada em micro serviços destaca-se das restantes (Monolítica ou SOA) por conferir uma maior granularidade através do completo isolamento da funcionalidade num serviço totalmente independente, que comunicará com os restantes serviços através de uma infraestrutura de rede utilizando protocolos standard, como por exemplo o HTTP. A flexibilidade e agilidade que esta arquitetura proporciona levou à sua massificação, principalmente em aplicações nativas na Cloud [2].

Por outro lado, a adoção de arquiteturas baseadas em microserviços, pode levar ao aumento significativo do numero de componentes de software, principalmente em ambientes com centros de dados de dimensão considerável, o que levanta novos desafios às equipas de configuração e gestão das componentes que suportam as aplicações, nomeadamente nas decisões de “onde” colocar os componentes de forma a conseguir uma utilização do hardware existente de forma eficiente.

A única forma de endereçar esta necessidade é através da automatização dos processos de distribuição, monitorização e recuperação em caso da falha, na prática necessitamos de uma plataforma de orquestração das tarefas associadas à gestão destes microserviços.

Adicionalmente, temos vindo a assistir a uma mudança sobre a adoção de Cloud na maioria das organizações, isto é, são conhecidos inúmeros exemplos em que é tomada a decisão de migrar as aplicações para centros de dados próprios, o que nos obriga a preparar as infraestruturas locais para receber estas novas arquiteturas [3], bem como garantir o seu correto funcionamento.

A implementação destas plataformas traz novos desafios às infraestruturas de rede tradicionais, uma vez que têm de conferir o mesmo grau de flexibilidade e agilidade existente nas Clouds públicas. Tendo em conta que a adoção de redes baseadas em software (SDN) já é, aos dias de hoje, uma realidade em muitos centros de dados este trabalho propõe identificar/apresentar uma solução que permita a completa integração destas plataformas em centros de dados locais e em cloud 's públicas.

À data de hoje é inquestionável que as redes baseadas em SDN, em ambientes de centros de dados, conferem às organizações a possibilidade de simplificar drasticamente algumas operações [4]. Por um lado, a existência de uma camada de controlo baseada em politicas, facilita drasticamente a gestão da infraestrutura e o controlo de acessos aos serviços alojados dentro e fora do centro de dados [5], por outro lado a capacidade de manipulação dos fluxos tendo por base parâmetros que vão além das tradicionais referências de rede (e.g. *tags* ou características funcionais) permitem encaminhar o trafego para destinos alternativos (e.g. firewall 's, antivírus, etc.) sem necessitarmos de alterar a arquitetura da rede e apenas quando pretendido. Esta capacidade de manipulação dos fluxos de rede é especialmente relevante quando operamos em ambientes de alta disponibilidade [6], isto é, quando estamos perante um cenário em que consideramos mais do que um centro de dados "ativo" em detrimento do tradicional cenário ativo/passivo frequentemente utilizado em cenários de recuperação de desastre. Na pratica a grande vantagem de termos uma arquitetura que permita termos serviços ativos em mais do que um centro de dados está diretamente relacionada com a rentabilização do equipamento que nas soluções tradicionais se encontra desaproveitado, neste sentido a capacidade de movimentar máquinas virtuais entre centros de dados, sem perda de conectividade e sem alteração das configurações de rede é altamente simplificada pelo uso das funcionalidades SDN descritas anteriormente. Estas funcionalidades estão totalmente dependentes da capacidade do controlador SDN manipular a infraestrutura de rede do centro de dados, o que nos leva a questionar se a abstração adicional ao nível da rede que as plataformas que suportam micro serviços implementam não irá comprometer as atuais arquiteturas que suportam centros de dados em ambientes ativo/ativo.

1.2. Objetivos

Este projeto tem como objetivo principal a identificação, especificação e implementação, de mecanismos de integração entre uma plataforma que suporte e orquestre containers, serviços de

comunicações no centro de dados local e serviços de containers em provedores de cloud 's públicas. Neste contexto pretende-se avaliar as capacidades que advêm desta integração, nomeadamente visibilidade, ganhos em contexto operacional, capacidade de aplicar as mesmas politicas de segurança independentemente do local onde o container se encontra.

Neste sentido, para alcançar o objetivo anteriormente mencionado, definem-se como objetivos parcelares deste trabalho os seguintes pontos:

- Investigação e levantamento inicial bibliográfico sobre as duas principais áreas que são abrangidas neste trabalho (Arquiteturas baseadas em microserviços, Containers e plataformas de orquestração, *Software Defined Networks*)
- Familiarização com as tecnologias utilizadas pela empresa SONAE nas duas componentes
- Especificação e implementação da solução proposta em cenário real
- Testes, validações de funcionamento e identificação de limitações do protótipo implementado

1.3. Sumário das principais contribuições

Como principais contribuições deste trabalho podemos referir a implementação e estudo dos mecanismos de integração da solução SDN da Cisco para centros de dados, com a plataforma de orquestração de containers Kubernetes, bem como a definição dos requisitos para a realização de uma integração semelhante, mas com plataformas de orquestração alojadas e geridas em Clouds públicas. Para tal efeito, foram utilizadas as infraestruturas produtivas da empresa proponente, tanto na componente do centro de dados, como na componente de Cloud pública. No final, foi possível verificar pela combinação das tecnologias mencionadas a possibilidade de movimentar *containers* entre *clusters* alojados no centro de dados local e a cloud pública, mantendo a mesma política de controlo de acessos e a mesma visibilidade. Foi também possível confirmar a capacidade de integração com soluções presentes no centro de dados, identificando alguns desafios futuros.

1.4. Estrutura da dissertação

Este documento está organizado em 6 capítulos. A descrição de cada um dos capítulos é a seguinte:

- Introdução: este capítulo fez o enquadramento e a contextualização do trabalho, ao apresentar o tema geral e os objetivos do trabalho a desenvolver.
- Estado da Arte: Descreve o conceito de microserviços e consequentemente de *containers*, no sentido de enquadrar a sua necessidade no contexto atual. É

identificado o Kubernetes como plataforma de orquestração, bem como alguns desafios à sua implementação. É apresentado o conceito de SDN e a sua importância para a implementação de infraestruturas híbridas.

- Arquitetura e solução proposta: descreve o problema que nos propomos a resolver, no contexto da empresa proponente.
- Descrição da bancada de testes: aborda todo o trabalho desenvolvido, nomeadamente configurações, topologia de rede e os mecanismos implementados.
- Testes e Resultados: Demonstra evidencias sobre o ambiente implementado, sobre a visibilidade obtida e sobre a capacidade de definir centralmente políticas de acesso.
- Conclusões: Apresentadas as principais conclusões obtidas de todo o trabalho, bem como eventuais futuros desenvolvimentos.

2. ESTADO DA ARTE

Neste capítulo iremos abordar os principais conceitos relacionados com o desenvolvimento aplicacional baseado em microserviços (secção 2.1). Em seguida descrevemos o conceito de *Container* como sendo um mecanismo que permite isolar de forma mais eficiente um microserviço, apresentando também a tecnologia *Docker* como uma plataforma que permite distribuir e executar aplicações baseadas em *Containers* (secção 2.2). Iremos também descrever o principal orquestrador de *Containers* (secção 2.3) com especial foco na sua componente de rede (secção 2.4). Finalmente, introduzimos o tema das SDN (secção 2.5), concluindo com algumas considerações sobre ambientes híbridos (secção 2.6), isto é, ambientes compostos por centros de dados interligados com cloud 's públicas.

2.1. Introdução ao conceito de microserviços

2.1.1. Transformação de aplicações monolíticas em microserviços

Com o intuito de permitir um melhor entendimento relativamente às vantagens da utilização de microserviços, é necessário descrever os conceitos da abordagem *legacy* – aplicações monolíticas.

Aplicações monolíticas são normalmente constituídas por vários componentes ligadas diretamente, que na prática são tratados como um só. Alterações numa das componentes obriga a reinstalação da aplicação completa, alem disso verifica-se que ao longo do tempo a falta de limites e de mecanismos formais de integração entre as diferentes componentes aumenta a complexidade da solução e muitas vezes à deterioração da qualidade do sistema como um todo.

Uma aplicação monolítica normalmente obriga à utilização de um número reduzido de servidores que disponibilizem a capacidade necessária à aplicação. Na eventualidade de ser necessário um aumento de capacidade, podemos eventualmente optar por um crescimento vertical (adicionando mais recursos de hardware – processador, memória, etc.) ou crescimento horizontal com a parametrização de novos servidores com cópias da aplicação. Escalar verticalmente normalmente não causa qualquer impacto nas aplicações, no entanto os custos eventualmente ficam bastante elevados e no limite podemos atingir a capacidade máxima do equipamento. Escalar horizontalmente, apesar de consideravelmente mais barato pode lançar desafios à componente aplicacional, uma vez que este método nem sempre é possível utilizar, por exemplo se o problema de capacidade estiver num motor de base de dados relacional.

Na prática, se algum dos componentes de uma aplicação monolítica não for escalável, a solução como um todo torna-se ela também não-escalável, a não ser que seja possível dividir a aplicação monolítica.

Dividir aplicações em microserviços

Estes e outros problemas forçam-nos a equacionar a divisão das atuais aplicações monolíticas em componente mais pequenos e independentes, aos quais damos o nome de microserviços. Cada microserviço é executado como um processo independente (Figura 1) e comunica com os outros microserviços através de interfaces simples e bem definidas (API's).

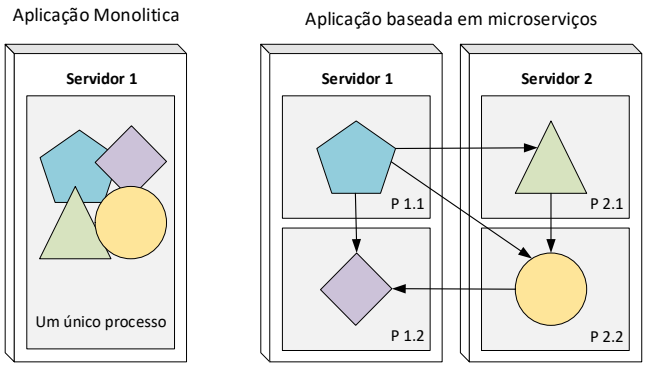


Figura 1-Aplicação monolítica versus microserviços

Microserviços comunicam através protocolos síncronos, como por exemplo HTTP, através dos quais normalmente expõem API's RESTful (Representational State Transfer), ou protocolos assíncronos como o AMQP (Advanced Message Queueing Protocol). Estes protocolos são considerados de simples implementação e não são proprietários de nenhuma linguagem de programação, o que permite que cada microserviço seja desenvolvido na linguagem mais apropriada à implementação do serviço em causa.

Estas características, nomeadamente o facto de que os microserviços disponibilizam API's relativamente estáticas, permite que cada microserviço seja desenvolvido de forma independente, o que posteriormente permite que as suas atualizações e/ou alterações sejam igualmente independentes, necessitando apenas de assegurar compatibilidade com os restantes microserviços com os quais comunica. A Figura 2 mostra-nos um exemplo. Certos componentes são replicados e executados como múltiplos processos distribuídos por vários servidores, enquanto que outros microserviços executam apenas um processo.

Escalar microserviços

Escalar microserviços, ao contrário das aplicações monolíticas, onde temos de escalar os serviços como um todo, é feito serviço a serviço, o que na prática permite alocar mais recursos apenas

ao serviço que necessita. Considerando a Figura 2 como um exemplo, é possível verificar que determinados componentes são replicados e correm como múltiplos processos.

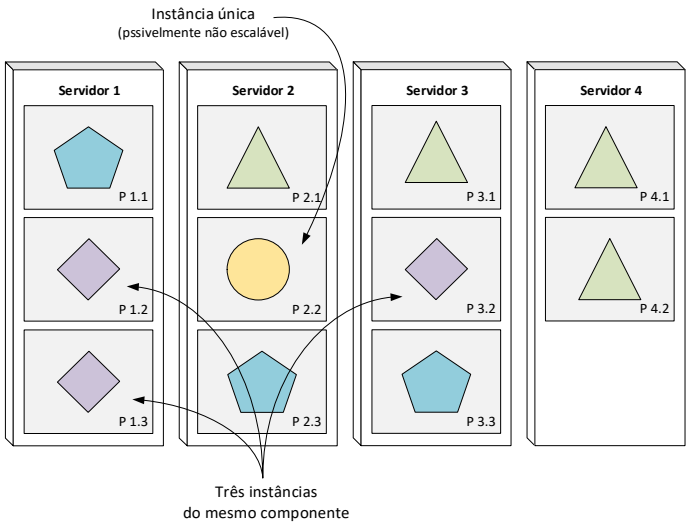


Figura 2-Cada microserviço pode ser escalado individualmente

Implementação de microserviços

Como em qualquer outra arquitetura, os microserviços também têm desvantagens. Quando o seu sistema consiste em apenas alguns componentes, a gestão é relativamente simples, até mesmo o processo de decidir onde colocar o microserviço em execução, do ponto de vista dos servidores que temos disponíveis, uma vez que as opções são reduzidas.

Quando o número de componentes aumenta, decisões relacionadas com a implementação tornam-se cada vez mais difíceis, não só aumenta o número de combinações como também o número de interdependências entre os componentes, o que aumenta consideravelmente a complexidade.

Os microserviços só conseguem ser consequentes se trabalharem em equipa, pelo que a capacidade de identificar os restantes microserviços e de trocar mensagens entre eles é absolutamente indispensável. Ao implementar uma solução baseada em microserviços, alguém ou alguma coisa tem de garantir que todas as configurações necessárias à colaboração estão corretamente efetuadas, de forma a permitir que os microserviços operem como um sistema único.

A necessidade de ambientes divergentes

Como já referimos anteriormente uma das grandes vantagens de uma arquitetura baseada em microserviços é o facto de os componentes não serem apenas implementados independentemente, eles são também desenvolvidos independentemente. Esta independência, associada ao facto de que normalmente até temos diferentes equipas de desenvolvimento responsáveis por cada componente, nada nos garante que as diferentes equipas utilizem as mesmas livrarias ou até mesmo as mesmas

versões de uma mesma livraria. A divergência de dependências entre os componentes aplicacionais, como os que vemos na Figura 3, são inevitáveis.

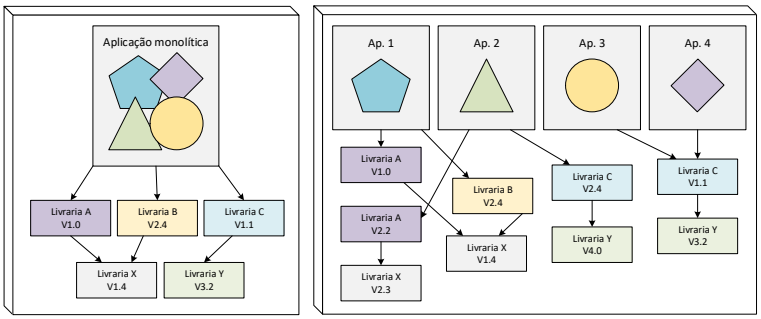


Figura 3-Várias aplicações a correr no mesmo servidor podem ter conflitos nas dependências

2.1.2. Providenciar um ambiente consistente para cada aplicação

Independentemente de quantos componentes estamos a desenvolver ou a instalar, um dos maiores desafios que programadores e equipas de operações enfrentam é a diferença entre os ambientes onde as aplicações são executadas. Não só é evidente as diferenças entre ambientes de desenvolvimento e de produção, como muitas vezes até os ambientes de produção têm diferenças consideráveis.

Estas diferenças podem estar relacionadas com hardware, sistemas operativos ou até mesmo livrarias disponíveis em cada máquina. Tipicamente ambientes produtivos são geridos por uma equipa de operações, ao passo que muitos desenvolvimentos são efetuados nos equipamentos dos próprios programadores, o que leva a grandes diferenças entre estes dois sistemas.

Estas diferenças normalmente surgem apenas na fase final de entrada em produção, uma vez que o normal é encontrar ambientes produtivos que servem várias aplicações, que pela heterogeneidade de dependências lançam enormes desafios à sua implementação.

Para reduzir estas situações, o ideal era que as aplicações fossem executadas exatamente no mesmo ambiente em que foram desenvolvidas, desta forma mantínhamos consistente o sistema operativo, livrarias, configurações de sistema assim como as definições de rede e comunicações. Alem disso, do ponto de vista da equipa de operações, é necessário garantir que as aplicações seguintes não colocam em risco as já existentes.

2.2. Tecnologia baseada em Containers

2.2.1. O que são Containers?

Anteriormente vimos que diferentes componentes a correr na mesma máquina poderiam necessitar de diferentes livrarias, no limite até livrarias incompatíveis de coexistir na mesma máquina.

Quando uma aplicação é composta por poucos, mas grandes componentes, é compreensível que o desenho da infraestrutura contemple uma máquina virtual (MV) para cada componente e

atingindo o isolamento pretendido através da utilização de instâncias diferentes do sistema operativo. Mas quando estes componentes começam a ficar mais pequenos e a sua quantidade cada vez maior, não podemos atribuir uma MV a cada componente, uma vez que seria um desperdício de capacidade. Mas não é só o desperdício de capacidade, e consequentemente desperdício financeiro que devemos considerar, uma vez que cada MV tem de ser configurada e gerida individualmente, o que manifestamente se prova como não sendo uma solução escalável. Assim que o número de MV começar a aumentar é necessário aumentar também o tamanho da equipa operacional, no sentido de garantir o correto funcionamento das aplicações.

Isolar componentes através da tecnologia *Linux Containers*

Em vez de utilizarmos MV para isolar os ambientes de cada microserviço (ou processo de uma aplicação), os programadores estão a recorrer a uma tecnologia denominada *Linux Containers*. Esta tecnologia permite executar vários serviços na mesma máquina, possibilitando não só a utilização dos componentes que cada microserviço necessite, mas também garantindo que um componente não impacta os outros componentes alojados na mesma máquina, na prática podemos compara-los com as MV mas com muito menos sobrecarga.

Um processo que está a ser executado num *Container* é visto pelo sistema operativo como um processo normal, no entanto mantém o isolamento dos restantes processos. Do ponto de vista do próprio processo existe a perceção de que é o único processo naquela máquina.

Comparar máquinas virtuais com *Containers*

Comparando com MV, os *Containers* são muito mais leves, o que permite executar um número muito superior de componentes por determinado hardware, principalmente pelo facto de que uma MV necessita de alocar recursos aos seus próprios componentes de sistema. Um *Container* por outro lado, é apenas um processo isolado que está a ser executado no sistema operativo da máquina, consumindo apenas os recursos necessário pela aplicação, sem a sobrecarga de qualquer processo adicional.

A sobrecarga que existe nas MV, leva a que se instale múltiplas aplicações na mesma máquina, pela razão de que muitas vezes os componentes não necessitam de toda a capacidade existente na MV. Quando utilizamos *Containers*, podemos e devemos ter apenas uma aplicação por *Container*, como vemos na Figura 4. Resumindo esta tecnologia permite a colocação de mais componentes por servidor do que aqueles que conseguimos recorrendo apenas a MV.

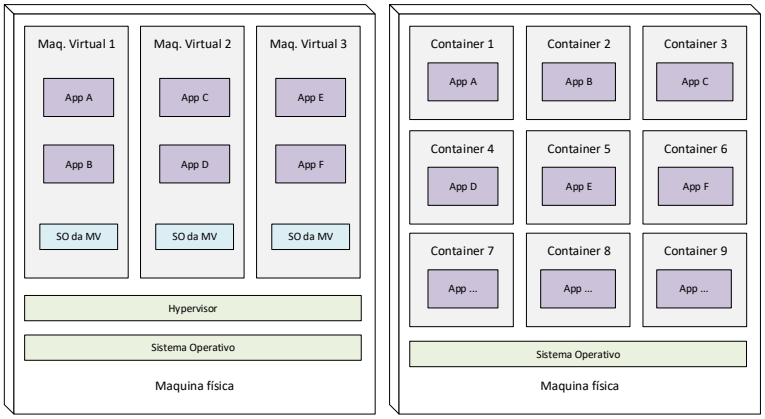


Figura 4-Usar MV's para isolar aplicações VS usar containers

Quando executamos três máquinas virtuais numa máquina física, na prática temos três sistemas operativos completamente distintos a partilhar o mesmo hardware. Por "baixo" dessas máquinas virtuais, normalmente temos o sistema operativo da máquina física e um qualquer *hypervisor*, que é responsável por dividir o hardware existente pelas máquinas virtuais. As aplicações que se encontram dentro das MV's executam chamadas ao kernel do sistema operativo que se encontra dentro da MV, e o kernel executa as respetivas instruções x86 no CPU da máquina física através do *hypervisor*.

Os containers por outro lado, executam chamadas diretamente ao kernel que está a ser executado na máquina física. Este kernel é o único que interage com o hardware físico, deste modo não é necessária qualquer virtualização ao nível do CPU. A Figura 5 representa a interação descrita anteriormente.

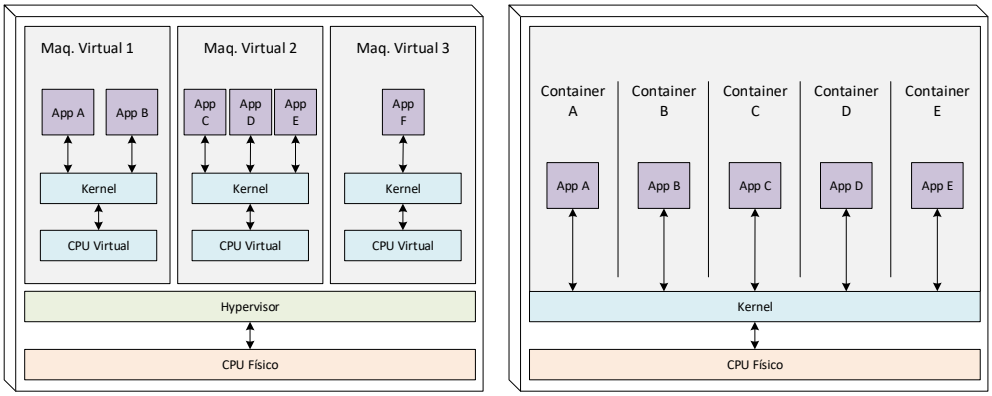


Figura 5-Diferenças entre o uso do CPU Físico

A principal vantagem em utilizarmos MV's é o completo isolamento que conseguimos obter, uma vez que cada MV executa o seu próprio *kernel*, enquanto nos containers todas as chamadas são efetuadas ao mesmo *kernel*, o que se pode revelar como uma falha de segurança. Na eventualidade de termos um número reduzido de servidores, as MV podem ser uma alternativa caso o número de

processos a isolar seja também reduzido. Para executar um número elevado de processos isolados, os containers serão a escolha mais acertada, principalmente pela reduzida sobrecarga, visto que as MV's executam os seus próprios serviços de sistema operativo e os containers não. Esta característica vai também permitir executar containers sem necessitar de carregar um novo sistema operativo, o que reduz o tempo de recuperação em comparação com as MV's.

Mecanismo que permite o isolamento baseado em Containers

Há dois mecanismos que tornam possível o isolamento baseado em containers. O primeiro, denominado *Linux Namespaces*, permite que cada processo tenha a sua própria visão do sistema (ficheiros, processos, interfaces, hostname, etc.). O segundo é denominado por *Linux Control Groups (cgroups)*, que limitam a quantidade de recursos que o processo pode consumir (CPU, memória, largura de banda, etc.).

Isolar processos utilizando *Linux Namespaces*

Por defeito, cada sistema Linux tem apenas um *Namespace*. Todos os recursos do sistema, como sistemas de ficheiros, identificadores de processos, identificadores de utilizadores, interfaces de rede, etc. pertencem a esse mesmo *Namespace*. No entanto é possível criar *Namespaces* adicionais e assignar recursos a cada um. Quando queremos executar um processo, o processo é executado dentro de um *Namespace*, o que na prática permite que este processo veja apenas os recursos afetos a esse mesmo *Namespace*. Na realidade existem vários tipos de *Namespace*, pelo que um processo não pertence apenas a um *Namespace*, mas sim a um de cada tipo, por exemplo:

- Mount (mnt)
- Process ID (pid)
- Network (net)
- Inter-process Communication (ipc)
- UTS
- User ID (user)

Cada tipo de Namespace é usado para isolar um determinado grupo de recursos. Por exemplo, o *Namespace* UTS determina o *hostname* e o *domain name* que é visível pelo processo que está a ser executado dentro deste *Namespace*. Ao assignarmos dois *Namespaces* UTS diferentes a um par de processos, eles vão "ver" *hostnames* diferentes. Na prática, aos olhos dos processos, vai haver a sensação de estarem a ser executados em máquinas distintas.

À semelhança do *Namespace* UTS, ao utilizarmos diferentes *Namespaces* do tipo *Network* conseguimos controlar quais as interfaces de rede que o processo "vê". Cada interface de rede

pertence apenas a um *Namespace*, mas pode mudar de *Namespace*. Cada container usa o seu próprio *Namespace* do tipo *Network*, pelo que apenas vê o seu próprio conjunto de interfaces.

Limitar os recursos disponíveis a um determinado processo

Conforme referido anteriormente a outra funcionalidade que permite o isolamento baseada em container é a capacidade de limitar o número de recursos que um container pode utilizar. Esta limitação é conseguida através de *cgroups*, uma funcionalidade nativa do *kernel* do Linux que permite limitar os recursos que determinado processo pode utilizar. Um processo não consegue usar mais do que o limite estipulado de CPU, memória, largura de banda, etc. Desta maneira, garantimos que nenhum processo consome recursos necessários para outro processo.

2.2.2. Introdução ao Docker como plataforma para Containers

Apesar do conceito de containers já existir há bastante tempo, a crescente utilização da plataforma Docker nos últimos anos tem vindo a demonstrar as mais valias deste tipo de abordagem. Docker foi o primeiro sistema que permitiu a portabilidade de containers entre máquinas. Na prática simplificou o processo de empacotamento (*packaging),* não só ao nível da aplicação mas também de todas as livrarias e outras dependências, no limite podemos adicionar um sistema operativo completo num simples pacote que pode ser executado em qualquer máquina que tenha Docker.

Quando executamos uma aplicação através de um container de Docker, vemos os ficheiros correspondentes aos pacotes que adicionamos, esteja o container a correr numa máquina de desenvolvimento ou numa máquina produtiva, independentemente da distribuição de Linux que a máquinas físicas tenha, o que nos torna verdadeiramente agnósticos relativamente às livrarias instaladas nas máquinas físicas.

Este resultado acaba por ser semelhante ao que obtemos usando máquinas virtuais. Se instalarmos um sistema operativo numa MV e instalarmos a aplicação nessa MV, atingimos o mesmo grau de isolamento, no entanto, utilizando Docker despendemos menos recursos para atingir o mesmo resultado.

Conceitos introdutórios do Docker

Docker é uma plataforma que permite a pacotização (packaging), distribuição e execução de aplicações. Alem destas funcionalidades o Docker também permite transferir o pacote criado para um repositório central, no sentido de posteriormente ser descarregado para qualquer máquina com Docker e executado nesse local.

Os principais conceitos de Docker que permitem o que descrevemos são:

- *Images* – um container baseado numa imagem de Docker é um pacote criado por nós com a aplicação e as respetivas dependências. Contém o sistema de ficheiros que vai estar disponível para a aplicação e o caminho para o executável que será usado no arranque do container.
- *Registries* – Um registo de Docker é um repositório que armazena imagens Docker e facilita a partilha dessas imagens com outras pessoas ou equipamentos. Quando criamos a nossa própria imagem podemos executa-la na máquina onde criamos a imagem ou carregar (*push*) a imagem para um registo e depois efetuar o download (*pull*) noutra máquina. Podemos ter registos públicos, que permitem que qualquer pessoa faça download das imagens, ou privados, apenas disponíveis para determinadas pessoas ou máquinas.
- *Containers* – Um container baseado em Docker é um container Linux normal que foi criado a partir de uma imagem de Docker. Um container em execução é um processo que está a ser executado na máquina real, mas completamente isolado da própria máquina e dos processos paralelos existentes na mesma máquina. O processo tem recursos limitados, isto é, só pode utilizar os recursos (CPU, RAM, etc.) que lhe foram atribuídos.

Criar, distribuir e executar imagens Docker

A Figura 6 mostra os três conceitos e como se relacionam entre si. Um programador constrói a imagem e carrega-a para um repositório. Desta forma a imagem fica disponível para qualquer pessoa que tenha acesso ao repositório. Essas pessoas podem descarregar a imagem para qualquer máquina com Docker e executar a imagem. O Docker cria um container isolado baseado na imagem e executa o binário especificado na criação da imagem.

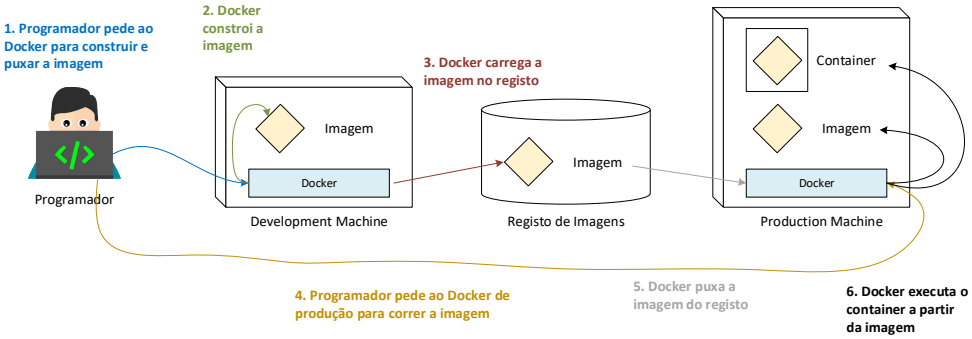


Figura 6 - Imagens, Registos e Containers em Docker

Comparação entre máquinas virtuais e containers Docker

Anteriormente vimos a comparação entre máquinas virtuais e o conceito teórico de containers, agora pretendemos acrescentar alguns pontos relativamente aos containers criados com Docker. A Figura 7 mostra-nos seis aplicações a correr em 3 máquinas virtuais, à esquerda, e à direita as mesmas aplicações, mas segmentadas por containers Docker.

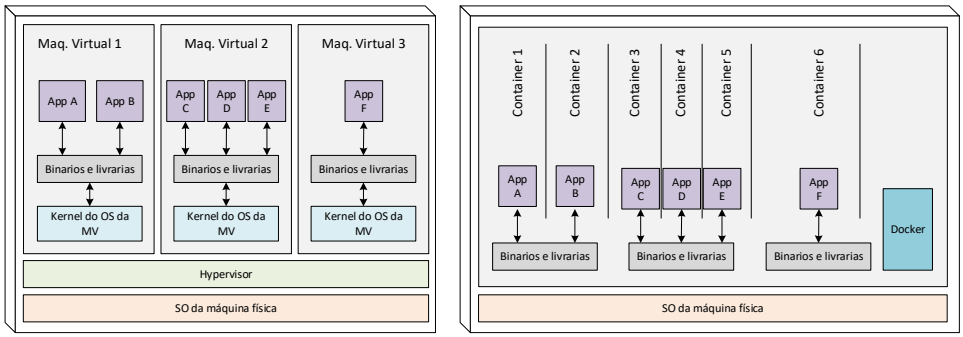


Figura 7 - Seis aplicações a correr em três MV vs. correrem em containers Docker

De salientar que as aplicações A e B têm acesso aos mesmo binários e livrarias tanto estando a correr em MV's como em containers. No caso da MV esta questão já não levanta dúvidas, no entanto nos containers é suposto cada um ter o seu próprio sistema de ficheiros, pelo que de forma a compreender o formato correto de um container de Docker devemos descrever o conceito de "camadas" da imagem.

Camadas das imagens de Docker

Diferentes imagens podem conter exatamente as mesmas camadas porque todas as imagens Docker são construídas tendo por base outra imagem, pelo que duas imagens podem ter sido criadas tendo por base a mesma imagem de referência. Esta característica permite acelerar consideravelmente a distribuição de novas imagens via rede de dados, uma vez que uma camada que já foi transferida, provavelmente como parte da imagem inicial, não necessita de ser transferida novamente.

Mas as vantagens de usar camadas não se limita à aceleração dos processos de distribuição, são também relevantes no objetivo de reduzir o espaço em disco ocupado, uma vez que camada só é guardada uma vez. Dois containers criados tendo por base a mesma imagem podem aceder aos mesmos ficheiros, mas se algum deles escreve num desses ficheiros, o outro container não vê essas alterações. O que na prática significa que mesmo que partilhem alguns ficheiros, os containers mantém o seu isolamento. Isto só é possível porque as permissões que cada container tem em cada camada é apenas de leitura, no entanto, ao executar o container é criada uma nova camada, com permissões de escrita. Se um container tenta alterar um ficheiro que está numa camada inferior,

apenas de leitura, o ficheiro é todo copiado para a camada superior e é nessa camada que as alterações são escritas.

Limitações conhecidas relativamente à portabilidade de containers

Teoricamente, um container pode ser executado em qualquer máquina Linux que tenha o Docker a correr, no entanto não podemos esquecer que todos os containers usam o Kernel da MV onde estão. Se um container necessitar de uma versão específica do Kernel poderá não ser possível executar a aplicação.

Apesar dos containers serem muito mais "leves" existem algumas restrições relativamente às aplicações que suportam, que vão mais além do que apenas versões do Kernel. Um container é construído para uma determinada arquitetura de hardware, por exemplo, se a imagem foi construída para ser executada numa arquitetura x86 não será possível usar a mesma imagem numa arquitetura ARM.

2.3. Orquestração de containers com Kubernetes[1]

Já referimos que à medida que o número de containers aumenta na infraestrutura, a tarefa de os gerir convenientemente fica bastante complexa. A Google foi provavelmente a primeira empresa a sentir a necessidade de um mecanismo mais eficiente relativamente às tarefas de gestão das suas componentes de gestão, derivado da sua dimensão e dispersão geográfica. A sua dimensão (centenas de milhares de servidores), obrigou-os a desenvolver uma solução que permitisse gerir os milhares de componentes de software a um custo aceitável.

2.3.1. A origem do Kubernetes

Ao longo da sua existência a Google desenvolveu um sistema interno denominado *Borg* que mais tarde evoluiu para o sistema *Omega*, que ajudava os seus programadores e administradores de sistemas a gerir os tais milhares de componentes. Além de simplificar o desenvolvimento e a gestão, permitiu-lhes uma utilização do hardware existente de forma muito mais eficiente, o que para uma infraestrutura do tamanho da Google é de extrema importância.

A Google manteve estes projetos apenas para utilização interna por mais de uma década, no entanto em 2014 apresentou o Kubernetes, uma plataforma aberta baseada na experiência adquirida durante o desenvolvimento do *Borg,* do *Omega* e de outros projetos internos.

[1] Kubernetes é uma palavra com origem grega, significa piloto ou timoneiro.

2.3.2. Visão geral do Kubernetes

O Kubernetes é uma plataforma que nos permite de forma fácil implementar e gerir aplicações baseadas em containers [7]. Baseia-se nas funcionalidades nativas do sistema operativo Linux para executar aplicações heterogéneas sem qualquer preocupação com os detalhes internos dessas aplicações, e sem necessitar de intervenções manuais no processo de instalação em múltiplos servidores. O simples facto destas aplicações serem executadas em containers, evita que uma nova instalação impacte as restantes aplicações que estão a ser executadas no mesmo servidor, o que é uma vantagem enorme quando a mesma infraestrutura é partilhada por várias organizações, ou até simplesmente por várias aplicações.

Uma plataforma como o Kubernetes permite criar uma abstração relativamente à infraestrutura física, simplificando o desenvolvimento, a implementação e a gestão. Na prática obtemos a perceção de estarmos a implementar componentes num enorme computador.

A implementação de aplicações através do Kubernetes é sempre igual, isto é, não é relevante se a infraestrutura física tem apenas alguns nós, centenas ou até milhares de nós físicos. Outra das vantagens é a facilidade com que adicionamos novos equipamentos ao cluster, uma vez que simplesmente representam capacidade adicional para implementar aplicações.

Descrição do *Core* do Kubernetes

A Figura 8 mostra-nos uma representação, muito simplista, de um sistema Kubernetes. O sistema é composto por um nó denominado *Master*, e alguns nós denominados *Workers*. Quando o programador submete uma lista de aplicações ao *Master*, o *Kubernetes* implementa essa aplicação no cluster de nós do tipo *Worker*. Que componente fica a ser executada em que nó não é relevante nem para o programador nem para o administrador de sistemas.

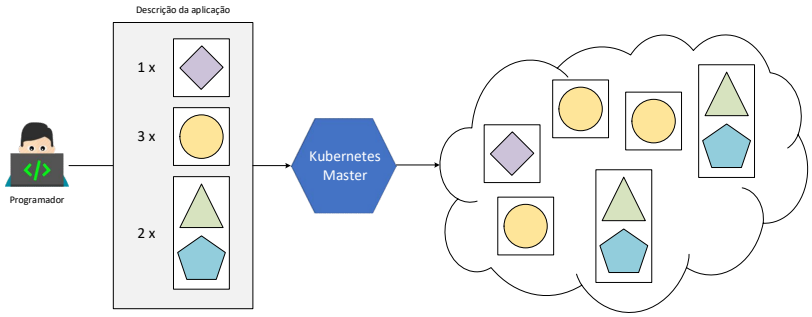


Figura 8 - O Kubernetes abstrai todo o centro de dados numa única plataforma

O programador pode inclusive definir que certos componentes devem permanecer juntos, o que faz com que o Kubernetes os coloque na mesma máquina física. Os outros vão ser espalhados pelas máquinas existentes, sem nunca perderem a capacidade de "falar" uns com os outros, independentemente do nó em que cada componente esteja a ser executada.

Maior foco dos programadores no desenvolvimento das funcionalidades aplicacionais

O Kubernetes é visto como um sistema operativo de todo o cluster, evita que os programadores tenham de implementar funcionalidades relacionadas com infraestrutura, nomeadamente:

- Descoberta dos serviços/componentes
- Escalar os recursos que cada componente dispõe
- Balanceamento de carga
- Mecanismos de resolução automática de problemas

Melhor gestão dos recursos por parte das equipas de operações

O Kubernetes vai executar os containers em qualquer nó do cluster, assegurando a comunicação entre as componentes e mantendo os containers sempre disponíveis de forma totalmente automática, permitindo que os recursos físicos de cada nó sejam geridos de forma mais eficiente, quando comparado com uma gestão manual dos mesmos recursos.

2.3.3. Descrição da arquitetura de um cluster Kubernetes

No que diz respeito à componente de hardware, um cluster de Kubernetes é composto por vários nós, que se dividem em dois tipos:

- O *Master*, que representa o plano de controlo do Kubernetes que na prática controla e gere todo o cluster
- Os *Workers*, nós onde as aplicações são executadas

Na Figura 9 vemos uma representação de algumas componentes dos dois tipos de nós, que detalharemos de seguida.

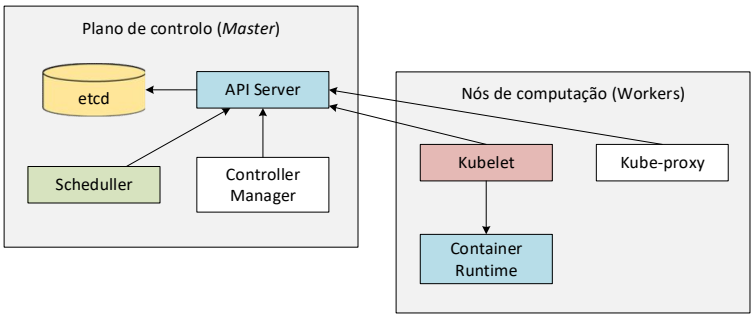


Figura 9 - Componentes de um cluster Kubernetes

O plano de controlo (*Master*)

O plano de controlo é o que na prática controla o cluster e garante o seu funcionamento. Consiste num conjunto de componentes que podem ser executados apenas num nó ou serem divididos por múltiplas instâncias, é também possível termos um cluster em que, por motivos de redundância, todos os componentes estão em todos os nós do tipo *Master*. Estas componentes são:

- *API Server*, permite que os utilizadores e os outros componentes interajam com o cluster.
- *Scheduller*, escalonador de componentes, na prática é quem define em que nó de trabalho vai ser executada cada componente.
- *Controller Manager*, funções especificas do cluster, como por exemplo, replicação de componentes, gestão dos nós de trabalho, resolução de problemas relacionados com falhas nos nós de trabalho, etc.
- *etcd,* sistema de ficheiros distribuído que armazena, de forma persistente, a informação de configuração do cluster.

Estas componentes asseguram o controlo e gestão do cluster, mas não asseguram a execução de qualquer componente aplicacional, tarefa que é assegurada pelo nós de trabalho.

Nós de trabalho (*Workers*)

É neste tipo de nós que são executados os containers. As tarefas associadas à execução dos containers, monitorização e disponibilização dos serviços existentes no cluster são asseguradas por:

- *Docker*, ou qualquer outro *container runtime*, que execute containers.
- *Kubelet*, que interage com o *API Server* e mantém os containers no seu nó
- *Kubernetes service proxy*, responsável pelo balanceamento do tráfego aplicacional entre os diferentes componentes de uma aplicação

2.3.4. Instalar uma aplicação num cluster de Kubernetes

Para conseguirmos instalar uma aplicação num cluster de Kubernetes primeiro temos de segmentar essa aplicação numa ou mais imagens de containers, depois enviamos essas imagens para um registo central e finalmente enviamos uma descrição da nossa aplicação para o *Kubernetes API Server*.

Essa descrição inclui informação sobre a imagem ou imagens que contêm as componentes aplicacionais, como é que esses componentes se relacionam entre si e quais devem ser executados no mesmo nó (*co-located*). Para cada componente é também possível definir quantas replicas do mesmo componente pretendemos que sejam criadas. Adicionalmente, a descrição também inclui quais desses componentes disponibilizam algum serviço internamente (a outros containers) ou externamente o que obriga a expor esses mesmos serviços através de um endereço IP e a garantir a sua descoberta pelos restantes componentes.

Como é que uma descrição resulta num container já em execução

Quando o servidor de API's processa a descrição que criamos, o *Scheduller* distribui os grupos de containers definidos pelos nós (*worker nodes)* disponíveis, tendo em conta os recursos que cada container vai necessitar e os recursos livres em cada nó.

O *Kubelet* que está presente em cada nó, dá instruções ao *Container Runtime* (Docker, por exemplo) para puxar a imagem e executar o *Container*.

A Figura 10, representa uma possível arquitetura *Kubernetes* com uma aplicação composta por vários *containers* em execução.

A descrição que é enviada ao *Control Master* (*Master Node*) contém quatro containers agrupados em 3 *pods*. Os primeiros dois contêm apenas um *container,* no entanto o *pod* número três contém 2 containers que devem ser executados no mesmo nó. Ao lado de cada *pod* encontramos um número que define o número de replicas que pretendemos que sejam criadas e executadas em paralelo.

Ao enviarmos a descrição para o *Kubernetes*, ele vai distribuir o número de replicas solicitado pelos nós que tem disponível. O *Kubelet* em cada nó interage com o Docker para puxar a imagem do *registry* e executar o *container.* A Figura 10 representa o fluxo descrito anteriormente.

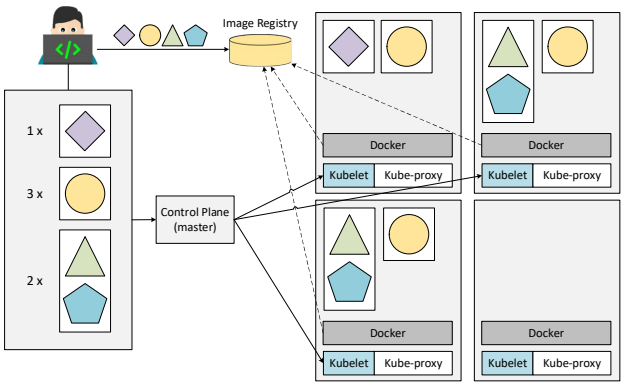


Figura 10 - Arquitetura básica de um cluster de Kubernetes (quatro nós) com apenas uma aplicação

Manter os *containers* em execução

Quando a aplicação está em execução, o *Kubernetes* assegura que o que está distribuído pelos diferentes nós é exatamente o que o *Master* recebeu no momento inicial. Por exemplo, se definirmos na descrição que pretendemos cinco servidores web, o Kubernetes vai garantir que tem sempre cinco instâncias. Se uma instância deixar de funcionar corretamente, um processo que para ou simplesmente deixa de responder, o Kubernetes reinicia-o automaticamente.

É possível verificar este comportamento mesmo quando a falha é ao nível do próprio nó, isto é, em caso de falha de hardware o Kubernetes vai garantir que todos os containers que estavam a ser executados nesse nó são criados nos nós operacionais.

Escalar o numero de cópias

Mesmo com a aplicação em produção, podemos decidir aumentar ou diminuir o número de cópias, nestes casos o *Kubernetes* cria as novas instâncias ou elimina as que houver em excesso. É ainda possível confiar ao próprio *Kubernetes* a definição do número ideal de cópias, tendo em conta a ocupação do CPU, consumo de memória, pesquisas por segundo, ou qualquer outra métrica que a aplicação disponibilize.

Manter a consistência no acesso do exterior

Conforme referimos anteriormente o *Kubernetes* pode necessitar de mover os containers entre os nós do cluster. Estas movimentações ocorrem, por exemplo, quando um nó fica indisponível, ou porque determinado container é "convidado" a sair desse nó no sentido de melhorar a ocupação do cluster como um todo. Se o container está a disponibilizar algum serviço para o exterior ou para outros containers dentro do cluster, como garantimos a disponibilidade desse serviço se o container está em constante movimento? E como é que os clientes acedem a um determinado serviço, quando esses containers são replicados e espalhados por todo o cluster?

Para permitir que os clientes localizem os containers que disponibilizam determinados serviços, podemos informar o *Kubernetes* quais os containers que disponibilizam o mesmo serviço e o *Kubernetes* vai expor todos esses containers através de um único endereço IP estático, esse endereço fica disponível para todas as aplicações dentro do cluster. Isto é conseguido através de variáveis de ambiente, no entanto os clientes também podem utilizar um serviço de DNS para localizar o endereço IP de determinado serviço. A componente denominada k*ube-proxy* vai permitir balancear o tráfego por todos os containers que disponibilizam o mesmo serviço, desta forma o endereço IP mantém-se constante permitindo que os clientes utilizem sempre o mesmo endereço independentemente da localização do container dentro do cluster.

2.3.5. Os benefícios de utilizar Kubernetes

Se conseguirmos atingir um ponto em que todos os servidores pertencem a clusters *Kubernetes*, a equipa de operações nunca mais necessitará de se preocupar com os desafios de gestão do ciclo de vida das aplicações, uma vez que as aplicações vão incluir todas as componentes necessárias ao seu correto funcionamento.

Sinmplificação do processo de instalação de uma aplicação

Uma vez que o *Kubernetes* expõe todos os seus nós de trabalho (*worker nodes*) como uma plataforma única, os programadores podem começar a desenvolver de forma totalmente autónoma, sem se preocuparem com os servidores que compõem o cluster.

Na prática todos os nós são vistos como um conjunto de recursos de computação, à espera de serem consumidos pelas aplicações. No entanto há situações em que o programador necessita de se preocupar com estes pontos, principalmente em ambientes em que os nós não são todos iguais, poderá fazer sentido que determinada aplicação só seja executada em nós com determinadas características. Por exemplo podemos pretender que uma aplicação só seja executada em nós com discos SSD e não HDD, nestes casos temos de garantir que estes containers só são criados em nós com esta capacidade.

Melhor utilização do Hardware

Ao adotarmos uma solução como o Kubernetes ganhamos a capacidade de desacoplar a camada aplicacional da infraestrutura. Quando instruímos o Kubernetes no sentido de executar uma aplicação, na prática estamos a deixar que o Kubernetes selecione o nó mais apropriado, sendo que esta escolha é efetuada com base nos requisitos definidos na descrição da aplicação e com os recursos que cada nó tem disponível.

Pelo facto de usarmos *containers* e por não associar diretamente uma aplicação a um nó físico, permitimos que a aplicação mude de nó dentro do cluster livremente, o que permite que em qualquer momento o *Kubernetes* faça a gestão mais eficiente possível de todo o cluster.

Verificação do estado e auto recuperação

Termos um sistema que nos permite mover as aplicações dentro de um cluster sempre que necessário é também útil em caso de falha. Independentemente da dimensão do cluster, vão existir sempre falhas de hardware, e nestes casos o *Kubernetes* também consegue ajudar.

O *Kubernetes* monitoriza as componentes aplicacionais e os nós onde as aplicações são executadas, e automaticamente recria as aplicações em caso de falha de um nó. Esta capacidade permite que a equipa de operações não tenha que se preocupar com a recuperação da componente aplicacional, permitindo que se foque de imediato na recuperação do nó propriamente dito, com o intuito de restabelecer a capacidade total do cluster. Em casos em que o cluster tenha capacidade suficiente para suportar todas as aplicações com menos um nó, a equipa de operações nem necessita de reagir imediatamente ao incidente.

Escalar automáticamente

Conforme já referimos anteriormente, o Kubernetes pode estar a monitorizar constantemente os recursos utilizados por cada aplicação, e tomar ações no sentido de escalar esses mesmo recursos, permitindo acomodar picos de utilização sem qualquer intervenção manual.

Na eventualidade de estarmos a usar Kubernetes numa infraestrutura em cloud, em que adicionar nós adicionais ao cluster é tão simples como efetuar uma chamada a uma API, é possível escalar o próprio cluster no sentido de disponibilizar os recursos necessários às aplicações.

Simplificação do desenvolvimento aplicacional

As vantagens descritas até agora beneficiam principalmente as equipas operacionais, mas o Kubernetes também apresenta vantagens para as equipas de desenvolvimento. Se nos lembrarmos que os ambientes, sejam produtivos ou de desenvolvimento, são executados no mesmo contexto, isto tem um efeito enorme relativamente à altura em que detetamos eventuais erros.

Outra mais valia é que esta plataforma liberta os programadores de terem que desenvolver funcionalidades de apoio ao funcionamento da solução, nomeadamente, descoberta de serviços ou outras aplicações no mesmo cluster, uma vez que o Kubernetes já disponibiliza estas funcionalidades embutidas na plataforma. Normalmente a aplicação apenas pesquisa por determinadas variáveis de ambiente ou efetua um pedido de resolução de nomes, no máximo poderá ter de efetuar um pedido à API do Kubernetes para obter a informação que necessita.

Finalmente, é também relevante mencionar que o Kubernetes aumenta consideravelmente a confiança que os programadores têm no momento da disponibilização da aplicação, visto que têm a garantia de que o Kubernetes vai automaticamente perceber se a aplicação não funciona e reverter todo o processo de distribuição. Esta crescente confiança permite a muitas organizações adotar praticas de *Contínuos Delivery*, o que beneficia toda a organização.

2.3.6. Principais componentes do Kubernetes

Pod

O pod é o objeto mais importante de um cluster Kubernetes, na prática são a base de toda a operação, todos os restantes objetos servem para gerir ou expor pod's ou são usados pelos pod's.

Um pod é um grupo de um ou mais containers que têm uma relação entre si e vão ser executados no mesmo *worker node* e no mesmo *Namespace*. Cada pod é como uma máquina lógica independente com o seu endereço IP, nome, processos, etc. que executa uma única aplicação. A aplicação pode ser um único processo a correr num único container, ou pode ser uma aplicação principal com múltiplos processos de suporte ao seu funcionamento, cada um a correr no seu container. Todos os containers que estão no mesmo pod vão ter a perceção de estar a ser executados na mesma máquina lógica, enquanto os containers a correr noutros pod's, mesmo que estejam no mesmo *worker node*, vão ter a perceção de estar em máquinas diferentes.

Olhando para a Figura 11 é possível perceber melhor a relação entre pod's, *worker nodes* e containers.

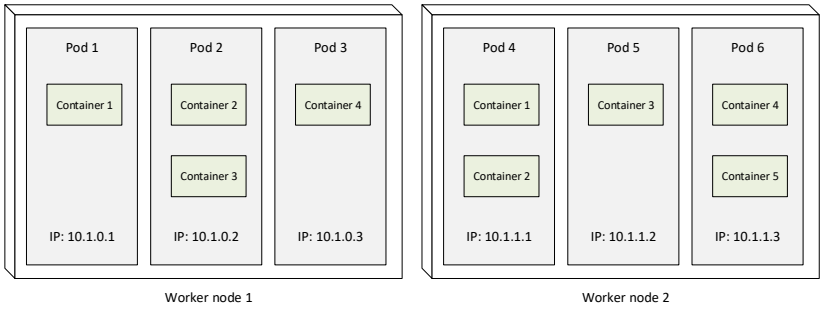


Figura 11 - Relação entre Pod's, Containers e Worker Nodes

Independentemente da visibilidade que cada container tem, na prática se os containers forem criados no mesmo POD, o Kubernetes vai colocar esses containers no mesmo nó, uma vez que um POD nunca se estende ente múltiplos nós.

Mas porquê utilizar POD's

Na realidade é possível incluir num container múltiplos processos, no limite é perfeitamente possível termos uma aplicação inteira dentro de um único container, no entanto essa abordagem contraria as arquiteturas aplicacionais baseadas em microserviços, e seriamos forçados a implementar serviços de monitorização, mecanismos automáticos de escalabilidade e eventualmente automatismos de recuperação em caso de falha, que podem ser assumidos pelo próprio Kubernetes.

Uma vez que pelas razões expostas acima não devemos colocar diferentes processos num único container, é necessário criar uma forma que permita agrupar diferentes containers que pela sua afinidade aplicacional necessitem de estar mais próximos, daí a existência dos POD's. Desta forma conseguimos tirar partido de todas as vantagens de uma arquitetura baseada em containers e ao mesmo tempo os processos têm a perceção de estar juntos.

Isolamento parcial entre containers no mesmo POD

Ao colocarmos mais do que um container num único POD estamos a assumir que pretendemos que alguns recursos sejam partilhados, no entanto mantendo toda a flexibilidade já conhecida. Esta funcionalidade é obtida através da partilha dos *namespaces* de Linux, levando a que todos os *containers* partilhem o mesmo *hostname* e interfaces de rede. No entanto, por defeito não partilham o sistema de ficheiros, mas em muitos casos vai ser necessário ter ficheiros partilhados, para isso vai ser necessário utilizar *Volumes*.

Situações que justificam a colocação de vários containers num único POD

A principal razão para colocarmos no mesmo POD vários containers é quando a aplicação consiste num processo principal e um ou vários processos complementares, como representado na Figura 12.

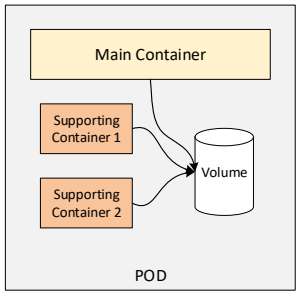


Figura 12 - Representação de um POD com múltiplos containers

Por exemplo, o container principal pode ser um servidor web que disponibiliza ficheiros de um determinado diretório, enquanto um outro container descarrega periodicamente conteúdo de uma fonte externa e armazena-o nesse mesmo diretório.

Rede (Network)

Introdução à arquitetura de rede *Flat Inter-Pod*

Todos os POD's de um cluster Kubernetes têm um endereço IP de uma única rede sem qualquer segmentação, o que significa que cada POD pode aceder diretamente ao endereço IP de outro POD do cluster. Não é necessário efetuar qualquer tradução de endereços (NAT). Quando um POD envia um pedido a outro POD, o POD de destino tem visibilidade sobre o IP real do POD que efetuou o pedido.

Esta arquitetura permite a que a comunicação entre POD's seja sempre simples. É irrelevante se o POD está escalonado para um ou vários nós, em ambos os casos os containers dentro dos POD's conseguem comunicar uns com os outros através da rede, independentemente da arquitetura de rede real de interligação entre os nós. Podemos inclusivamente comparar este funcionamento com o que encontramos numa rede local (LAN), uma vez que cada POD vai receber um endereço IP e consegue aceder a todos os outros POD's. Esta capacidade é obtida com a adição de uma camada lógica ao nível da rede através de tecnologias baseadas em software (*Software Defined Networks – SDN*).

Partilha da rede por containers dentro do mesmo POD

Não podemos esquecer que containers que estão no mesmo POD partilham a componente de rede, isto é, vão partilhar um único endereço IP e conjunto de portas TCP/UDP. Esta partilha obriga a que containers que estejam a ser executados num único POD não possam utilizar as mesmas portas TCP/UDP para disponibilizar os seus serviços. Alem do endereço IP vão também partilhar a interface de *loopback*, pelo que containers dentro do mesmo POD podem comunicar uns com os outros através do nome *localhost*.

Serviços (Services)

Um serviço em Kubernetes é um recurso que criamos com o intuito de obter um ponto único e constante de acesso a um grupo de POD's que disponibilizam o mesmo serviço. A cada serviço é atribuído um endereço IP e uma porta que nunca muda. Quando os clientes efetuam pedidos a esse IP/porta os pedidos são roteados para um dos POD's que suportam o serviço. Esta camada intermédia permite que os clientes não necessitem de conhecer a localização individual dos POD's que disponibilizam o serviço, permitindo movimentar os POD's dentro do cluster a qualquer momento.

Considerando uma aplicação a duas camadas (front-end e base de dados)

Se considerarmos uma arquitetura aplicacional a duas camadas, onde temos um serviço que está exposto ao cliente (front-end) suportado por uma base de dados (back-end) é perfeitamente possível ter inúmeros POD's a desempenhar a função de front-end e apenas um a disponibilizar a base de dados. Nestes casos temos de garantir dois tipos de conectividade:

- Clientes externos necessitam de fazer pedidos à camada de front-end sem se preocuparem se há um ou vários servidores web.
- A camada de front-end tem de interagir com a base de dados. Uma vez que a base de dados está a ser executada num POD não temos visibilidade sobre qual o nó em que a base de dados está a ser executada, inclusivamente pode ter de mudar de nó ao longo do tempo o que origina a alteração do endereço IP do POD.

Ao criarmos um serviço para a camada *front-end* e parametrizando-o para estar disponível do exterior do cluster, obtemos um endereço único que os clientes podem utilizar para aceder ao serviço, independentemente do número de POD's ou do nó em que o POD esteja a ser executado. Podemos adotar exatamente a mesma abordagem para a base de dados, se criarmos um serviço para o serviço de *backend*, conseguimos disponibilizar um endereço estável para os POD's da camada de *front-end* utilizar nas chamadas à base de dados. Alem de conferirmos a estabilidade necessária, a criação do serviço vai permitir aceder à base de dados através do seu nome, recorrendo ao serviço de DNS ou a variáveis de ambiente. Na Figura 13 podemos ver uma representação gráfica da arquitetura descrita anteriormente.

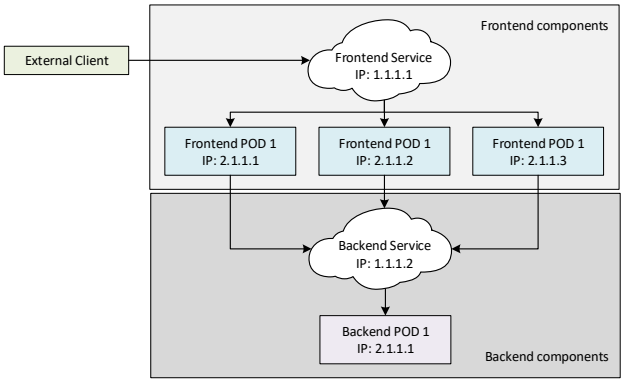


Figura 13 - Representação da utilização de Serviços

Aceder a serviços usando um nome

Ao criarmos um serviço conseguimos ter um único endereço IP que representa um conjunto de POD's, independentemente da dimensão do conjunto. Mas como é que os clientes descobrem o IP e a porta do serviço? O Kubernetes também endereça esta necessidade nativamente através de variáveis de ambiente ou um serviço de resolução de nomes interno (DNS[2]).

Armazenamento (Volumes)

Todos os containers arrancam exatamente com os mesmos ficheiros que foram adicionados à imagem no momento da construção. Combinando este pressuposto com o facto de que os POD's são reiniciados por variadíssimas razões, é necessário termos um mecanismo que permita que um container tenha acesso à informação gerada pelos containers anteriores, estando no mesmo POD ou não. O Kubernetes endereça esta necessidade criando *volumes de armazenamento*. Os volumes muitas vezes não são vistos como um recurso principal do cluster pelo facto de serem considerados como parte integrante de um POD. Na prática um volume é criado quando o POD arranca e é destruído quando o POD é eliminado, o que lhe dá total independência relativamente aos containers que estão a ser executados dentro desse mesmo POD. Um volume está disponível para todos os containers do POD, no entanto é necessário montá-lo em cada container. Em cada container podemos montar o mesmo volume em localizações do sistema de ficheiros diferentes.

Tipos de volumes

Existe uma enorme variedade de tipos de volume disponíveis no Kubernetes, muitos são genéricos, mas outros são específicos à tecnologia de armazenamento existente.

[2] DNS – Domain Name System

É muito improvável que numa implementação se utilize todos os tipos existentes, mas para referência listamos os tipos que atualmente são suportados:

- `emptyDir` – Um diretório vazio para armazenar dados temporários
- `hostPath` – Utilizado para mapear diretórios do nó real para o POD
- `gitRepo` – Mapear um repositório Git
- `nfs` – Partilha de rede NFS mapeada no POD
- `azureDisk` (Microsoft Azure), `awsElastic-Blockstore` (AWS), `gcePersistentDisk` (GCP) – Utilizados para mapear armazenamento proprietário das clouds públicas.
- `Cinder, cephs, iscsi, flocker, vsphere-Volume` – Tipos de armazenamento por rede
- `PersistenteVolumeClaim` – Utilizado para mapear armazenamento aprovisionado de forma dinâmica

Armazenamento persistente

Quando uma aplicação que está a ser executada num POD necessita de ter informação persistente, isto é, disponível mesmo que o POD seja escalonado para outro nó, temos de recorrer a algum tipo de armazenamento acessível através da rede. O exemplo mais relevante onde necessitamos de armazenamento persistente é um POD que utiliza uma base de dados.

Eliminar a dependência da tecnologia de armazenamento utilizada

O facto de termos de conhecer as tecnologias de armazenamento existentes na infraestrutura no momento em que criamos um POD contraria um pouco o grau de abstração que pretendemos obter ao utilizar Kubernetes. Na prática, ao criarmos os volumes desta forma estamos dependentes da tecnologia disponível neste cluster, logo não conseguimos reutilizar as definições noutro cluster, pelo que é necessário encontrar uma forma de nos mantermos agnósticos à tecnologia de armazenamento existente.

Introdução aos *PersistentVolumes* e *PersistentVolumeClaims*

Os nomes dos objetos propriamente ditos podem ser um pouco enganadores, uma vez que como já vimos anteriormente um volume normal também pode armazenar informação persistente.

Na Figura 14 ilustramos de que forma os POD's, *PersistentVolumeClaims* e *PersistentVolumes* se relacionam entre si.

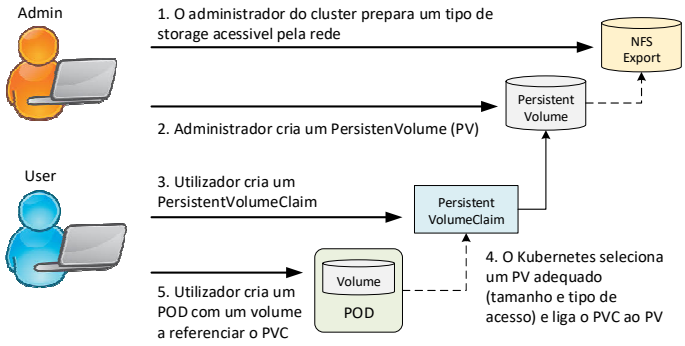


Figura 14 - Relação entre os PersistenVolumes e PersistenteVolumeClaim

Em vez de ser o programador a adicionar os detalhes referentes à tecnologia especifica de armazenamento, é o administrador do cluster que deve efetuar essas definições e registá-las no Kubernetes com a criação de um *PersistentVolume*. O administrador do cluster na criação do próprio objeto define o tamanho e o modo de acesso.

Quando um utilizador do cluster necessita de utilizar um determinado espaço de armazenamento num dos seus POD's, primeiro criam um *PersistentVolumeClaim*, especificando o tamanho mínimo pretendido e o tipo de acesso, o Kubernetes liga o volume ao pedido.

2.4. A componente de rede de um cluster Kubernetes

A componente de rede, em qualquer ambiente distribuído, tem uma preponderância enorme, e um cluster de Kubernetes não é exceção. O facto de cada POD necessitar de um endereço IP único no cluster, e de ser necessário garantir comunicação com os outros POD's existentes sem recorrermos a tradução de endereços (NAT), obriga a alguns cuidados adicionais, principalmente quando temos múltiplos nós e eventualmente múltiplos clusters.

O Kubernetes propriamente dito não obriga nem implementa, de forma nativa, qualquer solução que resolva estas questões, pelo que essa definição cabe ao administrador do cluster ou eventualmente a um plugin do tipo *Container Network Interface* (CNI).

2.4.1. Requisitos de rede do Kubernetes

Apesar do Kubernetes não forçar a utilização de uma componente/arquitetura de rede especifica, existem requisitos, como por exemplo, os POD's têm de comunicar uns com os outros independentemente de estarem a ser executados no mesmo nó ou não. A configuração da rede que é utilizada para cumprir com este requisito tem de permitir que o endereço IP que um POD utiliza para comunicar com outro tem de ser o mesmo IP que o próprio POD identifica como sendo o seu endereço.

Tendo como referência a Figura 15, quando o POD A comunica (envia um pacote de rede para) com o POD B, o IP de origem que o POD B vê é o mesmo IP que o POD A identifica como sendo o seu IP. Não pode haver qualquer tradução de endereços (NAT) entre os POD's.

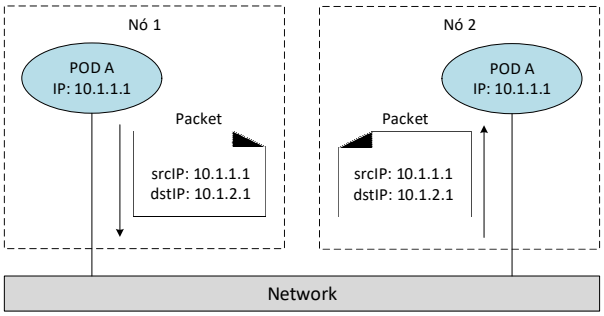


Figura 15 - Requisitos da componente de rede do Kubernetes

Este requisito condiciona as possíveis arquiteturas de rede mais tradicionais, mas simplifica a comunicação dos POD's dentro do próprio cluster, o que na prática facilita o desenvolvimento dos micro serviços. O facto de não haver NAT entre POD's permite que as aplicações se auto registem noutros POD's. Por exemplo, se tivermos um POD X que disponibiliza um serviço de notificações baseado em registos. O processo de registo pode ser simplesmente um POD Y enviar uma mensagem ao POD X informando o seu próprio endereço, uma vez que o POD X vai conseguir enviar diretamente para o POD Y as informações necessárias.

Estes requisitos que acabamos de descrever não se referem apenas à comunicação dentro do mesmo POD (*inter-pod*) é também necessário garantir que a comunicação entre POD's de diferentes nós também é possível sem recorrer à tradução de endereços. A única situação em que vamos ter tradução de endereços de tráfego gerado pelos POD's é quanto o destino é a Internet, como os endereços utilizados no interior do cluster são endereços privados [RFC1918], é efetuada uma tradução do endereço de origem para o IP do nó em que o POD está a ser executado.

2.4.2. Como funciona a componente de rede do Kubernetes

Para cada POD é criado um par de interfaces virtuais (veth), uma interface é visível no próprio *host* (ao executarmos o comando `ifconfig` vemos interfaces com o nome `vethXXX`), enquanto o par desta interface virtual é criado no próprio *Namespace* do container e renomeado para `eth0`, conforme vemos na Figura 16. As interfaces virtuais funcionam como uma ligação ponto a ponto ente dois equipamentos de rede, isto é, o tráfego que é colocado numa interface sai forçosamente pelo par correspondente e vice-versa.

A interface do lado do *host* está associado a uma rede local (*bridge*), o que permite que a interface `eth0` que se encontra dentro do container receba um endereço desta rede. Qualquer informação que seja colocada no `eth0` sai pelo `veth` do *host* que está associado à rede local, o que

na prática significa que conseguimos ter conectividade com qualquer interface que esteja ligado a esta rede.

Todos os containers de um mesmo nó, estão associados à mesma rede local, o que permite a comunicação intra-pod sem qualquer dificuldade, no entanto para comunicarmos com containers que estão noutros POD's (inter-pod) temos de interligar as redes locais de cada nó.

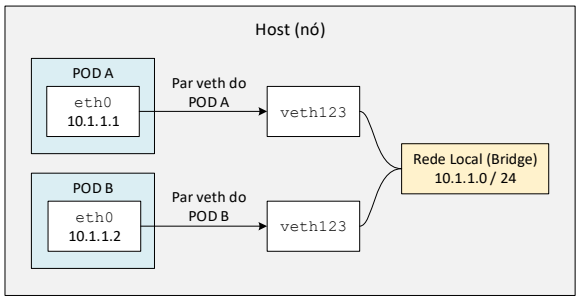


Figura 16 - Comunicação entre POD's do mesmo nó

Comunicação entre POD's de nós diferentes

Existem várias maneiras de ligar as *bridge's* dos diferentes nós, podemos adotar uma solução baseada em redes *overlay* ou *underlay,* ou até mesmo optando por mecanismos mais tradicionais de roteamento de pacotes de rede (*layer 3)*.

Conforme já referimos os endereços IP dos POD's devem ser únicos, logo os endereços de rede de cada *bridge* não se podem sobrepor. Como podemos ver na Figura 17 os POD's do nó 1 estão a usar endereços da rede 10.1.0.0/24 enquanto os POD's do nó 2 utilizam a rede 10.1.1.0/24. Neste cenário a tabela de roteamento do nó 1 tem de ser configurada de forma a que todos os pacotes que tenham como destino a rede 10.1.1.0/24 sejam encaminhados para o no 2 e vice-versa.

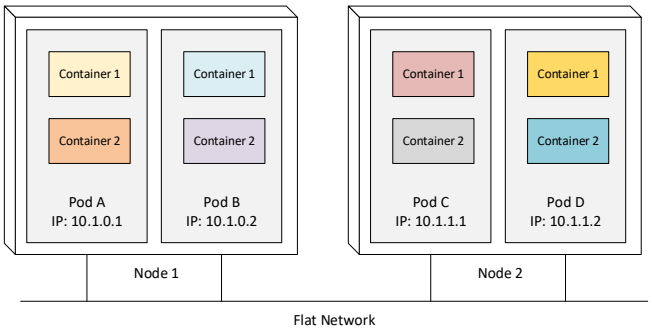


Figura 17 - Representação do endereçamento IP de um cluster de Kubernetes

Esta arquitetura só é possível se os nós do cluster estiverem ligados à mesma rede IP, sem qualquer equipamento externo de roteamento. É certo que conseguiríamos incluir nesta solução equipamento de roteamento externo, no entanto esta solução ficaria demasiado complexa à medida que o número de equipamento entre os nós fosse aumentando. É esta a razão que nos leva a afirmar de que uma solução baseada em tecnologias Software Defined Network (SDN), que vai dar a sensação

aos nós do cluster de estarem ligados no mesmo domínio *layer 3*, independentemente da rede física existente (*underlay*). Os pacotes de rede à saída do POD são encapsulados e enviados pela rede até ao nó que tem o POD de destino, onde o pacote é desencapsulado e entregue ao destino final.

Introdução ao Container Network Interface

De forma a simplificar o modo de integrar containers nas redes tradicionais, foi criado um projeto denominado *Container Network Interface (CNI)*. Na prática o CNI permite que o Kubernetes seja parametrizado para utilizar plugins externos, como por exemplo [8]:

- ACI - Integração com a solução Cisco ACI com vista à gestão da componente de rede e política de segurança.
- Calico - Solução de rede baseada em L3 com funcionalidade de gestão de políticas de acesso.
- Canal - Conjugação do Flannel com o Calico, disponibilizando a componente de rede e política de acesso.
- Cilium - Solução de rede L3 com capacidade para influenciar tráfego *layer 7* (HTTP/API). Suporta implementação baseada em roteamento ou *overlay*.
- CNI_Genie - Permite que o Kubernetes integre de forma transparente com outros plugins CNI, como por exemplo, Calico, Flannel, Romana ou Weave.
- Contiv - Arquitetura de rede flexível, suportando múltiplas implementações (L3 nativo com BGP, *Overlay* com *VXLAN*, L2 clássico e *Cisco-SDN/ACI*). Projeto completamente *opensource*.
- Contrail - Baseado na solução Tungsten Fabric, é um projeto *opensource* preparado para ambientes multi-cloud com funcionalidades de controlo de políticas de acesso.
- Flannel - Solução baseada em *overlay* para Kubernetes
- Knitter - Solução que suporta múltiplas redes para clusters Kubernetes
- Multus - Multi plugin que suporta vários plugins CNI, por exemplo Calico, Cilium, Contiv, Flannel
- NSX-T Container Plug-In (NCP) - Integração entre a solução SDN da VMWare (NSX) e clusters Kubernetes.
- Nuage - Plataforma SDN que através de uma abordagem baseada em políticas de acesso integra o mundo Kubernetes com o mundo exterior. Facilidade de integração com a solução SDN da Nokia (ex. *Nuage Networks*)
- Romana - Plugin que adota uma abordagem *layer* 3 mas que suporta a implementação de politicas.

- Weave Net - Disponibiliza funcionalidades de rede e de controlo de acessos baseado em políticas. Não necessita de uma base de dados externa.

Controlo de acessos entre POD's (Network Policy)

A escolha de qual CNI utilizar não deve ser efetuada apenas considerando questões de conectividade, uma vez que em certos cenários poderá ser importante controlar o tráfego entre POD's tirando partido da componente de rede do cluster, e neste caso teremos de implementar um CNI que suporte a definição/implementação de políticas de acesso à rede (*Network Policy*).

Uma política de acesso à rede em Kubernetes é aplicada aos POD's, identificando-os através de uma descrição. Podemos definir quem pode aceder ao POD ou a que recursos o referido POD tem acesso. Na prática criamos regras que definem que trafego pode entrar (*ingress*) ou sair (*egress*) do POD. Podemos criar regras usando POD's, os próprio *Namespace's* ou blocos de endereçamento através de notações CIDR (*Classless Inter-Domain Routing*), como por exemplo 192.168.32.0/24.

2.5. Software Defined Network

As abordagens baseadas em modelos em que o controlo e as funcionalidades de transporte estão distribuídas pelos diferentes ativos de rede ainda são hoje bastante utilizadas. Este modelo, apesar de bastante maduro, revela-se muitas vezes pouco flexível. Outra desvantagem neste modelo é que na grande maioria das soluções existe uma associação direta entre o hardware e o software, isto é, ao adquirirmos determinado equipamento estamos limitados ao software que o fabricante desenvolve, o que obriga a conhecimento específico nesse mesmo software.

O paradigma das redes baseadas em software, apresenta um conceito em que há uma clara segmentação entre a lógica de controlo dos fluxos de informação (plano de controlo), e os equipamentos que efetuam o encaminhamento propriamente dito (plano de dados). Esta separação leva a que os equipamentos de rede se tornem apenas em máquinas de encaminhamento de tráfego, ao passo que a lógica de controlo será implementada centralmente, ou num equipamento físico externo (controlador de SDN) ou no próprio sistema operativo de rede.

Ao separarmos as camadas, é absolutamente indispensável que continue a haver comunicação entre as duas, no entanto não podemos estar dependentes do que cada fabricante possa implementar, daí o aparecimento do protocolo Openflow.

Um equipamento de rede que suporte *Openflow*, na prática, vai disponibilizar uma API (*application programming interface*) que permite que um controlador externo interaja diretamente com o switch.

As API's que permitem a comunicação entre os controladores e os equipamentos de rede são conhecidos como *southbound* API's, em oposição às API's conhecidas como *northbound,* que são API's que o próprio controlador disponibiliza para que aplicações terceiras interajam com a infraestrutura.

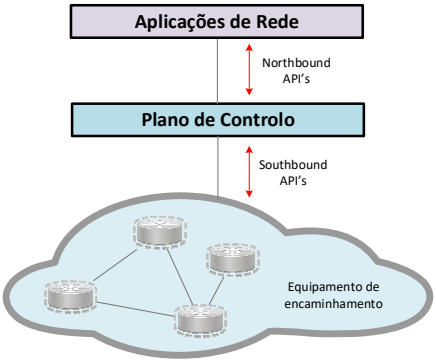


Figura 18 - Software Defined Network

Uma importante consequência da adoção desta abordagem foi a introdução de uma divisão de responsabilidades, relativamente a quem define as políticas da rede, quem implementa essas políticas nos equipamentos, e finalmente quem efetua propriamente dito o encaminhamento dos pacotes. Este novo grau de abstração leva à simplificação da gestão, e facilita a evolução e consequentemente a inovação da rede.

Apesar de comercialmente as soluções SDN serem relativamente recentes [9] (solução atual da Cisco começou a ser comercializada em 2014), o paradigma das SDN's e o próprio Openflow, que contribuiu significativamente para a disseminação das SDN, foram objeto de estudo no mundo académico vários anos antes.

Atualmente alguns fabricantes têm tentado introduzir uma visão um pouco diferente no que diz respeito ao SDN, denominada *Software Driven Network*.

Na prática não se trata de um conceito novo, mas sim uma forma diferente de interpretar a utilização das SDN. Numa visão *software-driven* considera-se o próprio protocolo *Openflow* e a arquitetura apresentada anteriormente como uma opção, isto é, como se de uma funcionalidade se tratasse, considerando que existem outras abordagens diferentes. Em vez de olharem para a rede como algo composto por um controlador central, onde guardamos/definimos toda a logica, e equipamentos de rede sem qualquer inteligência, defendem uma abordagem um pouco mais híbrida entre as arquiteturas tradicionais e a arquitetura SDN. Esta abordagem assenta na convicção de que não é aceitável pensar que as redes tradicionais vão ser todas substituídas por redes SDN. Alem disso ao longo dos últimos anos têm havido avanços consideráveis noutras tecnologias que não devem ser desconsiderados. No fundo o que esta abordagem descreve é que o ideal será um ambiente misto, em

que uma parte da rede será controlada centralmente por um controlador, mas outra terá uma abordagem mais tradicional, o que obriga a que ambos os mundos sejam interoperáveis desde o início.

A título de resumo, podemos definir as *Software Defined Networks* como uma arquitetura que otimiza e simplifica a operação, através da aproximação entre aplicações, serviços e dispositivos, sejam eles reais ou virtuais. Este ganho é muitas vezes atingido através da implementação de um ponto central de controlo – controlador SDN – que orquestra e facilita a comunicação entre aplicações que pretendem interagir com elementos da rede, mas também quando os próprios elementos da rede pretendem enviar informação para essas aplicações. O controlador tem também a capacidade de expor e abstrair as funcionalidades da rede via interfaces aplicacionais modernos (API's).

2.5.1. Evolução histórica das SDN

É particularmente interessante, num contexto[3] em que as primeiras implementações de soluções SDN em centros de dados produtivos têm menos de um ano, constatar que os conceitos não são, de maneira nenhuma, recentes.

É possível recuar mais de 20 anos (1998) e encontrar artigos [10] que descrevem a criação de *frameworks* ligadas à programabilidade da rede e consequente alteração das configurações de forma automática, tendo em conta as necessidades ou o tipo de utilização que está a ser dada à rede num determinado momento. No entanto, o aparecimento de interfaces abertas, como o Openflow (2008) [11], permitiram uma maior inovação do que aquela que era possível com soluções fechadas. Inclusivamente o termo Software Defined Network surge apenas em 2009, num artigo [12] da revista da universidade de Massachusetts.

No período de 2009 a 2013/2014, momento em que surgem as principais soluções SDN ditas comerciais (Cisco ACI, Nuage Nokia [13], VMware NSX [14]), ocorreram importantes acontecimentos que contribuíram significativamente para a evolução do paradigma SDN. Importante salientar o lançamento do *Open vSwitch* [15] e a criação da *ONOS* [16], que apesar de estarem mais relacionados com redes virtuais deram um importante contributo para as soluções SDN atuais.

[3] Primeiras implementações de soluções SDN em Portugal iniciaram no final de 2018

2.5.2. Programabilidade da rede

O conceito de programabilidade da rede surge constantemente associado ao paradigma SDN. Apesar de não ser um conceito propriamente recente, desde sempre existiu a necessidade de gerir equipamentos de rede de forma sistematizada, o que muda com o aparecimento das redes SDN é a forma como gerimos e/ou interagimos com os dispositivos. O objetivo principal é conseguir uma comunicação bidirecional, entre os equipamentos e outros softwares de forma simples.

Este objetivo é atingido pela disponibilização de interfaces programáveis, com capacidades para se auto descreverem de forma a permitir que outras aplicações consigam de forma fácil aprender/entender as capacidades de um elemento da rede.

Aplicação VS Rede

Numa abordagem mais tradicional os ativos de rede (ex. routers, switchs ou firewalls) suportam algumas opções de comunicação com o próprio dispositivo, tipicamente disponibilizam uma linha de comandos (CLI), acesso via SNMP [RFC1098] e mais recentemente NETCONF [RFC6241]. Estas opções são baseadas em código específico destas interfaces, que tipicamente está inserido diretamente no software do equipamento, o que significa que estas funcionalidades foram previamente programadas e são estáticas. Na grande maioria dos casos, uma aplicação que tivesse permissões para interagir com um equipamento de rede, teria de utilizar um destes protocolos, ou no limite teria de comunicar com um equipamento que atuasse como um reverse proxy.

Atualmente o que se pretende é consumir interfaces que sejam mais fáceis de utilizar por parte de quem desenvolve as aplicações, daí a adoção, por parte da grande maioria de fabricantes, de API's RESTful [17] (*Representational State Transfer*). Estas interfaces são geralmente definidas utilizando ficheiros JSON (*JavaScript Object Notation*). A utilização de ficheiros em formato JSON, resolve muitas das questões que referimos anteriormente.

2.5.3. SDN como parte de uma estratégia de Orquestração

Principalmente em ambientes de datacenter, tem sido cada vez mais evidente a necessidade de automatizar tarefas. No entanto a automatização do aprovisionamento de alguns elementos que compõem um datacenter não é suficiente, pelo que surge a necessidade de orquestrar vários elementos. Ser capaz de entregar de forma rápida, com baixo custo e sem erros, torna-se uma necessidade critica para qualquer ambiente de datacenter que se pretende o mais otimizado possível.

Quando observamos a componente de computação, em que a virtualização é já uma tecnologia extremamente madura, ou até a componente de armazenamento, que através das soluções de *hiperconvergência* [18] e/ou protocolos de armazenamento distribuído (ex. CEPH [19], já

disponibilizam interfaces que facilitam a orquestração, é inevitável exigir da rede o mesmo grau de abstração e a mesma facilidade de interação, que como já vimos anteriormente é atualmente conseguido recorrendo à utilização de API's, disponíveis nos controladores de SDN.

2.5.4. Soluções SDN para o centro de dados

Um dos grandes desafios relativamente à adoção de soluções SDN para centros de dados, é que na realidade não existe um standard propriamente dito, o que permite que diferentes fabricantes adotem/implementem as suas soluções de forma bastante autónoma, inclusivamente recorrendo a protocolos proprietários que complicam a interoperabilidade. Conforme vimos anteriormente, existe uma panóplia considerável de protocolos que podem ser utilizados, não só para DCI, mas também, como vamos ver em seguida, virtualizar a componente de rede do centro de dados. Na prática vamos introduzir o conceito de network overlay, como algo que permite a virtualização da arquitetura física da rede, também conhecida como *network underlay*.

Network Overlay

Com a generalização de ambientes de computação virtualizados, as redes virtuais têm vindo a ganhar cada vez mais importância em ambientes locais (centros de dados) ou ambientes públicos. Como resultado desta evolução, é muito frequente vermos associado aos paradigmas SDN o conceito de *Overlay*, ou rede virtual. Na prática uma rede virtual define uma nova topologia "por cima" de uma rede tradicional. Esta abordagem permite criar múltiplas redes separadas virtualmente, utilizando a mesma infraestrutura física. Normalmente encontramos estas soluções em ambientes em que é necessário separar clientes ou aplicações, que apesar de consumirem os mesmo recursos físicos necessitam de um elevado grau de isolamento.

A utilização de *overlays* de rede ajudam a resolver alguns desafios que encontramos na gestão de um centro de dados:

- Escalabilidade e flexibilidade – A utilização de tecnologias overlay cria um certo distanciamento entre os equipamentos físicos e a informação de encaminhamento da informação, o que permite por um lado efetuar ações de manutenção de forma independente, e por outro ganhamos uma enorme liberdade sobre onde ligar os dispositivos à rede, não necessitando de ter em consideração a arquitetura física.
- Conectividade layer 2 mesmo através de rede layer 3 – outro benefício é a capacidade de estender domínios de layer 2 através de redes layer 3, estendendo para outros centros de dados a flexibilidade que descrevemos no ponto anterior.

- Sobreposição de endereçamento – A grande maioria das tecnologias de *overlay* utilizam um identificador para cada rede virtual, o que permite que outra rede virtual consiga utilizar o mesmo espaço de endereçamento sem qualquer dificuldade.
- Segmentação de funções e responsabilidades – principalmente em ambientes multiutilizador, é possível ter equipas responsáveis pela gestão da infraestrutura subjacente (*underlay*) e outras equipas responsáveis pela gestão da componente virtual.

VXLAN (Virtual Extensible Local Area Network)

VXLAN é um protocolo utilizado na criação de redes overlay, inicialmente foi desenvolvido pela Cisco, mas tem vindo cada vez mais a ser identificado com um standard por outros fabricantes relevantes como a VMware, a Red Hat, Juniper, Arista e a Citrix. Hoje em dia encontramos o protocolo VXLAN em muitas soluções SDN, nomeadamente na solução da VMware (NSX) [20] e em ambientes baseados no *hypervisor* KVM.

VXLAN implementa um encapsulamento de *frames ethernet* (layer 2) em pacotes UDP (layer 3). Ao implementar um encapsulamento MAC-em-UDP, o VXLAN permite criar uma abstração relativamente ao endereçamento IP e ao local onde a máquina se encontra. Na Figura 19 é possível verificar a informação que é adicionada no processo de encapsulamento. É importante salientar a existência de endereços MAC internos e externos, responsáveis pela abstração que mencionamos acima, bem como o facto de que o protocolo de transporte que é utilizado na componente externa é o protocolo IP, este facto é importante porque permite que o segmento de VXLAN se estenda a qualquer localização, desde que existe conectividade via IP. Esta capacidade é especialmente interessante em ambientes de múltiplos centros de dados.

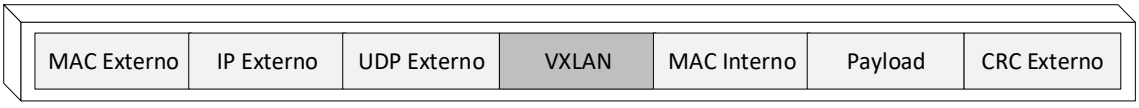


Figura 19 - Descrição do pacote VXLAN

A adoção de VXLAN também permite endereçar uma outrs limitação relevante que se prende com o número de VLAN's que é possível criar num determinado ambiente. O facto do identificador de VLAN's ser um número com apenas 12 bits, não permite que num único centro de dados se ultrapasse as 4096 VLAN's. O identificador de uma determinada VXLAN é um número com 24 bits o que permite mais de 16 milhões de VXLAN's num mesmo ambiente.

A operação do VXLAN consiste em equipamentos, físicos ou virtuais, com capacidade de disponibilizar acessos layer 2 a dispositivos finais e também a capacidade de estabelecer tuneis VXLAN, estes equipamentos são denominados por VTEP (Virtual Tunnel Endpoint). Estes equipamentos

são posteriormente ligados a uma rede IP no sentido de garantir conectividade com o outro ponto do túnel, conforme vemos na Figura 20.

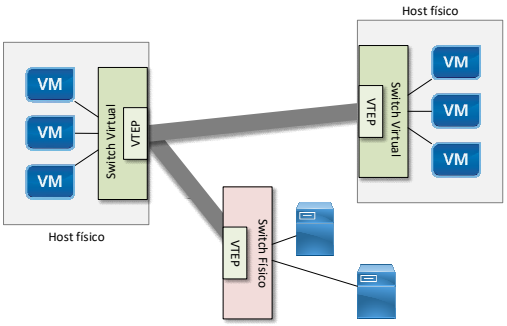


Figura 20 - Representação do conceito de VTEP

Quando uma máquina pretende comunicar com outra máquina, primeiro necessita de descobrir o endereço MAC de destino, para isso envia um pedido de ARP para a rede em broadcast. Este funcionamento numa rede puramente layer 2 não levanta qualquer desafio, já que os equipamentos de forma nativa suportam este comportamento. No entanto, num ambiente com encapsulamento VXLAN o VTEP ao receber um pacote em broadcast encaminha-o em multicast para o seu grupo de VNI (VXLAN Network Identifier). Os outros VTEP vão aprender o endereço MAC interno da máquina de origem e o endereço IP externo do VTEP correspondente. A máquina de destino responde ao pedido de ARP com o envio de uma mensagem de volta à máquina de origem, o que permite que o VTEP original aprenda o mapeamento correspondente ao destino.

Quando uma máquina se movimenta para outra localização, o mesmo método é aplicado, permitindo às restantes máquinas aprender a nova localização.

Em termos de programabilidade, o VXLAN disponibiliza uma interface única que permite criar um overlay em layer 2. Além desta facilidade o VXLAN, em determinados ambientes virtuais, é também utilizado para controlar o tráfego entre máquinas virtuais, desde o simples IO até à largura de banda ou definição de atributos de segurança.

NVGRE (Network Virtualization using Generic Routing Encapsulation)

O protocolo NVGRE foi criado com o intuito de resolver o problema de escalabilidade das soluções implementadas apenas com VLAN's – limite de 4096. À semelhança do VXLAN, o NVGRE implementa um encapsulamento de *frames* layer 2 em pacotes layer 3. Como o próprio nome indica é utilizado o protocolo GRE (*Generic Routing Encapsulation*) para a criação dos tuneis. O resultado é a capacidade de estender domínios layer 2 através de redes layer 3. O principal fabricante a utilizar NVGRE é a *Microsoft*.

NVGRE utiliza um identificador de 24 bits denominado TNI (*Tenant Network Identifier*) que é adicionado à frame ethernet. A frame é posteriormente encapsulada com um IP externo e finalmente com um endereço MAC externo, como podemos ver na Figura 21.

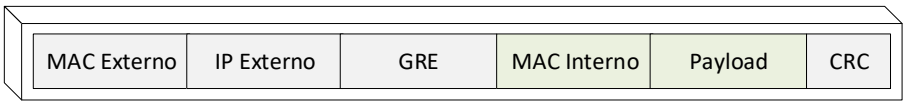


Figura 21 - Formato do pacote NVGRE

O comportamento de uma máquina ou switch que efetua o encapsulamento do tráfego utilizando NVGRE é bastante transparente. A qualquer frame gerada é adicionada o TNI e é encaminhado para o túnel GRE correspondente. No destino a *frame* é desencapsulada permitindo o encaminhamento para o endereço MAC original.

Tanto um como o outro são exemplos de protocolos com capacidade de estender domínios layer 2 através de redes layer 3. Estes overlays são implementados através de um encapsulamento adicional, que pode ou não ser reconhecido pelos equipamentos de transporte, isto é, apenas os dispositivos que se encontram no início e no fim do percurso (equipamento que vão encapsular e desencapsular o protocolo) necessitam de suportar o protocolo.

Network Underlay

A camada inferior da solução pode incluir um sem número de tecnologias. Na prática, independentemente das soluções adotadas, esta camada tem apenas uma função, transportar tráfego entre servidores dentro do centro de dados. Tipicamente a camada superior (*overlay*) é criada utilizando elementos como routers virtuais, switchs ou tuneis lógicos. A rede da camada inferior (*underlay*) é uma rede clássica, como por exemplo, Ethernet, MPLS ou IP.

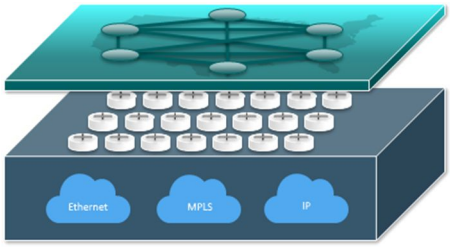


Figura 22 - Representação da rede Underlay e Overlay

Na realidade a grande maioria das soluções atuais pretendem disponibilizar uma rede *underlay* que suporta um ou mais *overlays*. A questão que se coloca é que solução é mais adequada a determinado desafio, isto é, enquanto uma abordagem pode escalar de forma muito eficiente no que toca à descoberta de endereços MAC, pode ser ineficiente na movimentação de máquinas virtuais entre centros de dados, ou integração com Clouds públicas. Outra questão relevante é a capacidade

de escalar/adaptar a alterações na rede de forma automática, em oposição a soluções que necessitem de intervenção manual.

2.5.5. Introdução à solução SDN para centros de dados da Cisco

Conforme referido anteriormente os conceitos em torno das soluções SDN atuais há muito que são tema no mundo académico, no entanto, só em 2013 é que a Cisco anuncia [21] que está a desenvolver um produto, com foco nos centros de dados, com esta abordagem. Em novembro de 2013 é anunciado o Cisco ACI (*Application Centric Infrastructure*). Esta data é identificada pela Cisco como a data em que nasce a primeira solução comercial para centros de dados, no entanto é importante referir que nesta altura apenas se tratava de um anuncio comercial, visto que em 2014 é anunciado que o protocolo *OpFlex* [22] será utilizado para a comunicação *southbound* entre o controlador SDN, conhecido como APIC (*Application Policy Infrastructure Controller*), e os equipamentos de rede, colocando assim disponível para o mercado a primeira [23] versão da sua solução SDN.

Apesar de estar claro que esta seria a estratégia da Cisco para o futuro das redes dos centros de dados, a adoção da solução não foi grande nos primeiros anos. Por um lado, a disrupção face às soluções tradicionais, geraram alguma resistência por parte do mercado, e por outro, a falta de maturidade da solução fez com que só com o lançamento da versão 2 em julho de 2016 a perceção começasse a mudar, colocando o ACI como uma verdadeira alternativa às arquiteturas de datacenter tradicionais. Atualmente podemos contar já com a versão 4.2.

Conceitos e princípios inerentes ao ACI

O ACI surge num momento em que a pressão sobre as equipas que gerem e operam os centros de dados é cada vez maior. As empresas vivem momento de enorme dinamismo tecnológico, o que resulta numa necessidade crescente de novos sistemas de informação. O ACI ao disponibilizar um modelo operacional baseado em políticas, com a pretensão de se estender ao longo de todo um ecossistema de soluções, simplifica a complexidade e reduz o custo de operação. Na prática ao simplificarmos todo o ciclo de vida das aplicações, seja no centro de dados ou na Cloud, estamos a adicionar agilidade e a criar valor para o negócio.

O ACI é uma combinação de hardware específico Cisco (família *Nexus* 9000), o controlador APIC, e uma panóplia de soluções terceiras, que integram nativamente com a rede, com o objetivo de construir uma solução coesa com a maior amplitude possível. A sua capacidade nativa de automação e aplicação de políticas, permite a gestão centralizada de todas as regras de conectividade, qualidade e níveis de serviço, bem como endereçar vários requisitos de segurança. A Figura 23 representa os principais componentes da solução SDN da Cisco.

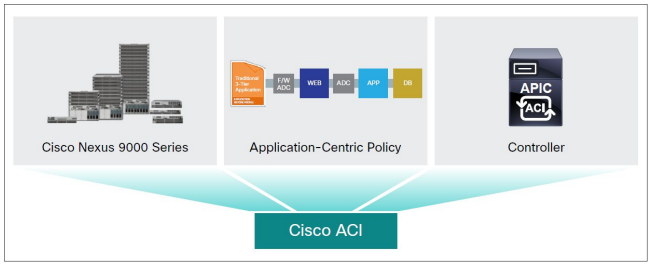


Figura 23 - Componentes do ACI (retirado de: "Cisco ACI Starter KITS2)

Estas funcionalidades podem ainda ser estendidas a outros componentes do centro de dados, como por exemplo, computação e armazenamento, permitindo um aprovisionamento de soluções totalmente baseado em políticas do ACI.

O ACI pretende que a configuração da rede seja efetuada tendo sempre presente a visão da aplicação que pretendemos suportar, permitindo nativamente a descrição de aplicações multicamada, e dos seus requisitos de comunicação, isto é, são criados objetos denominados "end-point-groups" que posteriormente são interligados por outro objeto, denominado "contrato". Os contratos vão definir explicitamente os fluxos aplicacionais, bem como outros elementos de rede, como por exemplo políticas de qualidade de serviço.

Na Figura 24 podemos ver a representação de uma aplicação com uma arquitetura composta por 3 camadas (Web, App e DB). É também possível verificar que os pedidos iniciarão do exterior diretamente à camada Web, e consequentemente a camada Web, encaminhará pedidos para a camada App, que por sua vez, caso necessite, poderá fazer consultas ou alterações à componente DB (base de dados). É ainda importante referir que entre estas camadas foram criados contratos, que além de permitirem a conectividade, definem políticas de qualidade de serviço e de filtragem.



Figura 24 - Política no ACI (retirado de: ACI policy driven datacenter [24])

Do ponto de vista físico, isto é, do hardware de rede propriamente dito, uma solução ACI apenas permite a utilização de equipamentos da família Cisco Nexus 9000. Estes equipamentos são interligados de forma a criar uma estrutura conhecida por *spine/leaf* (espinha/folha). Esta arquitetura é composta apenas por duas camadas, em oposição à arquitetura tradicional que permite três camadas (*core, distribution, access*). A vantagem desta arquitetura é que todos os serviços estão exatamente a dois "saltos" (*hops*) de distância, o que permite uma maior previsibilidade da performance aplicacional, e confere às equipas de operação uma enorme flexibilidade no que diz

respeito ao local físico onde são instalados os recursos computacionais. A Figura 25 representa uma arquitetura *spine/leaf*.

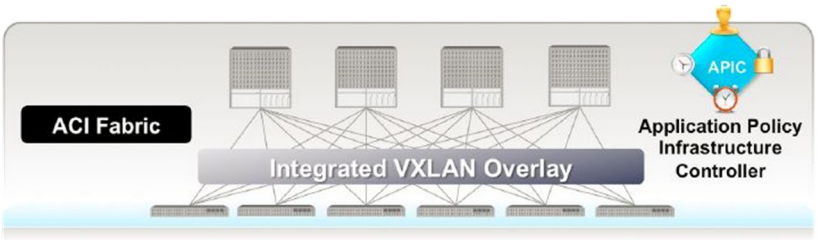


Figura 25 - Arquitetura spine/leaf (retirado de: refreshing my memory on ACI [25])

Numa arquitetura *spine/leaf* todos os dispositivos são ligados nas *leaf*, de forma a garantir os já referidos dois saltos de distância. À semelhança do que acontece com qualquer solução SDN, a política é inicialmente criada no controlador, que por sua vez traduz essa mesma política em configurações de rede.

Esta abordagem permite olhar para os switchs de rede, não como equipamentos individuais, mas sim como um "*fabric*", um sistema único, independentemente da sua dimensão ou densidade de portas.

As infraestruturas de rede dos centros de dados, têm vindo a ficar cada vez mais complexas, com centenas de ACL's, protocolos para redundância, múltiplos caminhos, etc. o que nos leva a uma relação muito rígida entre o endereçamento IP e a política implementada. O ACI, com o objetivo de disponibilizar um modelo simplificado, começa por criar uma abstração relativamente à configuração lógica da aplicação, da instanciação propriamente dita das configurações do hardware. Esta capacidade permite-nos ser agnósticos à localização dos recursos, sejam eles físicos, virtuais ou até em diferentes *hypervisors*. Na prática conseguimos desfazer a relação entre o endereçamento IP e as políticas de encaminhamento ou segurança, conferindo uma enorme mobilidade ao sistema. Todas estas alterações ao nível da descrição da aplicação, têm de ser refletidas nas configurações da rede automaticamente, como referimos anteriormente esta interação é assegurada pelo APIC.

O APIC não desempenha apenas as funções de controlador SDN, disponibiliza a interface que permite a criação dos perfis aplicacionais (descrição das aplicações) e é o ponto central para a gestão de toda a infraestrutura, nomeadamente:

- Monitorização dos equipamentos
- Monitorização das aplicações (*health-scores*)
- Disponibiliza ferramentas que facilitam a resolução de problemas
- Integração com aplicações externas
- Gestão de *firmware* dos equipamentos

- Inventário da infraestrutura

Definição de políticas e a configuração da rede

Nas redes Ethernet clássicas, definimos a configuração lógica da rede com a definição de VLAN's, instâncias de VRF's (Virtual Routing Forward), pontos de saída, etc. Estas configurações garantem a conectividade entre os diferentes locais. Após garantirmos a conectividade temos de assegurar a implementação de regras de segurança, pelo que recorremos a ACL's (Access Control Lists), colocando-as o mais perto possível da origem, deixando sempre passar o tráfego importante para o negócio.

Em infraestruturas de grande dimensão, o número de dispositivos que temos de configurar aumenta consideravelmente. Definir VRF's, VLAN's, endereçamento IP específico revela-se uma tarefa que consome demasiado tempo, e é bastante passível a erros. Esta abordagem tem também bastantes desafios quando pretendemos remover algum dos componentes referidos anteriormente.

O conceito de VLAN foi inicialmente pensado para limitar domínios de broadcast, no entanto, atualmente as VLAN's são também utilizadas para controlar tráfego entre servidores. Os endereços IP são utilizados para aceder a recursos de computação, mas atualmente são também utilizados para filtrar tráfego e organizar recursos.

Com o ACI, deixamos de necessitar de dividir a rede em VLAN's ou utilizar as redes IP como utilizávamos nas redes clássicas.

Agrupar servidores em diferentes domínios é o equivalente em ACI a colocar os servidores num EPG (*End Point Group*). Um EPG é um grupo de conexões no *fabric* que têm os mesmos atributos, como por exemplo, um conjunto de servidores que desempenham a mesma função, por exemplo, *front-end*'s aplicacionais.

O ACI utiliza um objeto denominado *Bridge Domain* quando existe o requisito de conectividade *layer 2* entre dois dispositivos, isto é, teremos de garantir que estes dispositivos se encontram no mesmo *bridge domain*.

Podemos ter um único *bridge domain*, com vários dispositivos alocados a múltiplas sub-redes, organizados em EPGs. Posteriormente podemos definir quais os EPG's que conseguem falar entre si. Um *bridge domain* pode ser configurado, por exemplo, para encaminhar tráfego em *unicast* ou para *flooding*. Se tivermos múltiplos EPG's no mesmo *bridge domain* configurado para suportar o envio de pedidos ARP (Address Resolution Protocol) em *broadcast*, então todas as sub-redes vão usar ARP *flooding* para conseguir falar com os dispositivos na mesma sub-rede. No caso do *bridge domain* estar configurado para *unicast, não vai existir flooding nem pedidos ARP em broadcast.* Qualquer tráfego *unicast* e até os pedidos ARP são enviados em *unicast*. A associação do EPG ao *bridge domain* é

utilizada pelo sistema para perceber quem é o *default gateway* de cada dispositivo dentro do EPG. Esta associação, não obriga a que todos os dispositivos estejam na mesma rede IP, ou que os dispositivos só conseguem comunicar em *layer 2*. A decisão de se o pacote é roteado ou apenas entregue em *layer 2*, depende do destino que o dispositivo de origem coloca no pacote. Se o pacote for enviado para o *default-gateway* do dispositivo de origem, então o pacote vai ser roteado.

Ao separarmos os servidores em EPG's, temos de criar regras (*Contracts*) que descrevam a intenção da aplicação. Uma vez que estes contratos são criados ao nível do EPG, as regras são aplicadas a todos os servidores que estiverem no EPG. No exemplo da Figura 26 podemos verificar que o EPG vermelho e o EPG verde conseguem falar entre si e para o exterior, mas o EPG laranja apenas fala com o exterior.

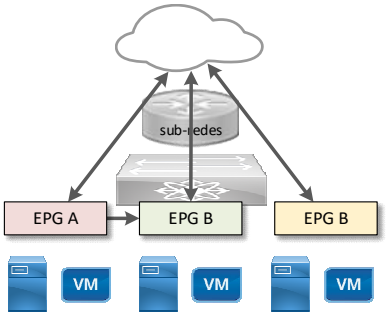


Figura 26 - Comunicação entre EPG's (adaptado de: Cisco ACI Test Drive)

Mapear a logica aplicacional numa política

Um dos principais elementos de uma arquitetura ACI é o conceito de *Tenant*. Os *tenant's* permitem criar diferentes segmentos dentro do ACI, sejam eles referentes a unidades de negócio, clientes, grupos, empresas, etc. A Figura 27 representa a estrutura hierárquica dos objetos do ACI por *tenant*. A figura pretende fazer um paralelismo entre os objetos nativos do ACI e os conceitos das redes tradicionais, facilitando desta forma a compreensão da relação existente.

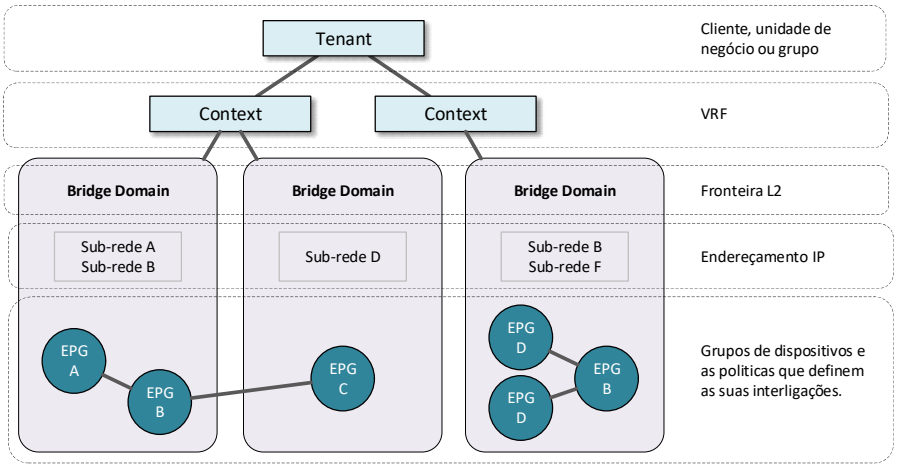


Figura 27 - Estrutura lógica do ACI (adaptado de: Cisco ACI Test Drive)

Todas as políticas são definidas dentro do *tenant* correspondente, sendo que um *tenant* pode incluir vários objetos, como por exemplo, filtros, contratos, redes externas, *bridge domains* e perfis aplicacionais que incluem os EPG's.

Um *contexto* é bastante semelhante a um VRF, permitindo diferentes instâncias de roteamento, e pode ser usado como ponto de separação entre perfis de gestão da solução. Tipicamente a utilização deste objeto serve o mesmo propósito dos tradicionais VRF's, isto é, um *tenant* pode representar uma organização e dentro dessa organização existem múltiplos departamentos. Esta abordagem permite que todas as organizações partilhem o mesmo hardware físico, mas com uma segmentação logica que confere privacidade e uma partilha clara das responsabilidades de gestão.

Um *bridge domain* não é propriamente o equivalente a uma VLAN, é simplesmente um objeto que agrupa sub-redes. Um *bridge domain* pode ser utilizado para definir uma fronteira ao nível da camada dois. Na prática um *bridge domain* é um *overlay* que não ultrapassa o contexto onde foi criado. Este overlay é criado através do protocolo VXLAN. A utilização de VXLAN permite que um equipamento na sub-rede A comunique diretamente com um device na sub-rede B. É inclusivamente possível ter um dispositivo na sub-rede A a falar com um dispositivo na sub-rede D, tratando-se de VXLAN's distintas temos de recorre ao roteamento entre VXLAN's. A utilização de VXLAN *bridging* e *routing* permite a comunicação entre qualquer dispositivo. As redes IP a que os dispositivos pertencem passam a ser perfeitamente irrelevantes. Em redes tradicionais a rede é importante porque o próprio roteamento é efetuado com base na rede, no entanto com ACI o roteamento passa a ser baseado no IP do host *host-based*, o que confere uma enorme mobilidade aos dispositivos, uma vez que não têm de se preocupar com a localização da rede a que pertencem.

Um EPG (*end point group*) é um contentor lógico que agrega objetos que partilham a mesma política, sejam camadas aplicacionais ou serviços.

A Figura 28 descreve um cenário com dois clientes. O cliente A tem dois contextos (VRF's), um para ambientes de desenvolvimento e teste, e outro para ambientes produtivos. O cliente B tem apenas um contexto denominado Serviços Web, que é subdivido em *bridge domain*, ou domínios *layer* 2, que posteriormente são ainda divididos por sub-rede. Podemos definir uma política de encaminhamento no *bridge domain* que permita o encaminhamento de tráfego desconhecido *unicast* e tráfego ARP, conferindo-lhe um funcionamento equivalente a uma VLAN.

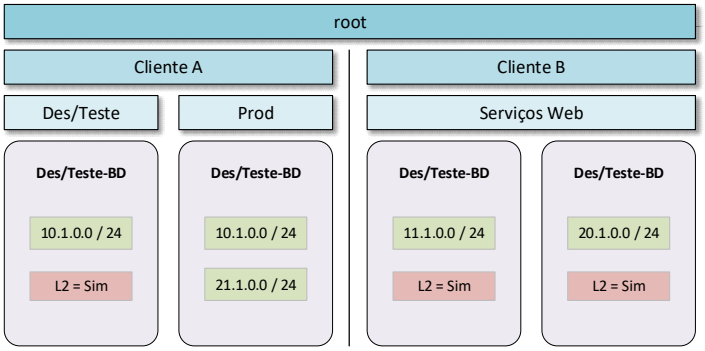


Figura 28 - Exemplo multicliente (adaptado de: Cisco ACI Test Drive)

A logica aplicacional é mapeada na política do ACI, isto é, ao definirmos quem pode falar com quem, na realidade, estamos a mapear os fluxos aplicacionais. Na Figura 24 vimos uma representação do que poderia ser uma política, no entanto é relevante acrescentar que as ligações entre os EPG's podem ser unidirecionais ou bidirecionais.

Atualmente um centro de dados não se pode limitar a disponibilizar uma rede com equipamentos de computação, é necessário incluir capacidades de priorização de tráfego (QoS), controlos de acesso (firewall's), deteção e proteção contra intrusos (IDS/IPS), etc. Estas necessidades, pelo facto de necessitarem que o tráfego atravesse as respetivas appliances, condicionaram durante anos as arquiteturas tradicionais de centros de dados. Num centro de dados com ACI é possível, recorrendo aos contratos que criamos entre os EPG's, definir diferentes fluxos aplicacionais sem necessitar de alterar a arquitetura, esta funcionalidade é conhecida como *service insertion*. Na prática o que acontece é que ao definirmos um contrato entre dois EPG's estamos a permitir que os dispositivos que compõem esses EPG's comuniquem uns com os outros. Nesse contrato é também possível definir políticas de redireccionamento do tráfego, ou de qualidade de serviço. No caso do redireccionamento, é possível forçar o tráfego a passar num equipamento de segurança (firewall, IDS, IPS, etc.) sem necessitar de alterar qualquer roteamento ou endereço IP dos intervenientes. No caso da politica de QoS ao definirmos as configurações no contrato, esta politica é aplicada nos equipamentos físicos em que o fluxo irá passar, no entanto se houver alguma alteração na rede, forçando a que o caminho seja outro, a politica de QoS é automaticamente configurada nesses novos equipamentos, garantindo mais uma vez uma enorme mobilidade.

2.5.6. Descrição do funcionamento do ACI

Como já vimos anteriormente a arquitetura física de uma solução ACI adota uma topologia *spine-leaf*. Todos os equipamentos são ligados em *leaf's*, camada com maior densidade de portas, enquanto as *spine* são utilizadas para interligar as *leaf's*, sendo que a largura de banda "interna" do *fabric* vai estar diretamente relacionado com o número de interligações existente entre as *leaf's* e as

spines, e claro com a largura de banda dessas mesmas ligações. No momento do lançamento do ACI estas ligações eram de 40Gbps, atualmente é possível implementar com ligações a 100Gbps.

Tráfego em que a origem e o destino estão na mesma *leaf* é tratado localmente, quando o destino se encontra noutra *leaf*, o tráfego terá de atravessar um *switch spine*. Apesar da arquitetura física apresentar dois saltos entre origem e destino, logicamente isso não acontece, isto é, todo o *fabric* é visto pelos dispositivos externos como um único *switch* com capacidades de roteamento (*layer 3 switch*).

Este tipo de arquitetura, apresenta-se como sendo mais fácil de construir, testar e manter [26]. A escalabilidade, referente ao número de portas, é assegurada pela simples capacidade de adicionar mais *leaf's*, e se pretendermos aumentara a largura de banda entre dispositivos ligados em *leaf's* diferentes, apenas necessitamos de adicionar mais interligações (fibras) entre as leaf's e as spines. É também relevante mencionar que nesta arquitetura todos os dispositivos estão apenas a dois saltos de distância, o que permite uma maior previsibilidade da performance da solução.

A abstração referida anteriormente só é conseguida com a criação de um *overlay* baseado em tuneis VXLAN entre as *leafs*. Cada *leaf* tem um endereço virtual privado, que é utilizado como terminação dos tuneis (VTEP – VXLAN Tunnel End Point) VXLAN. De forma a garantir uma descoberta automática e a conectividade entre todas as *leaf's* que compõem o *fabric* é utilizado o protocolo de roteamento IS-IS (*Intermediate System to Intermediate System*).

Todo o tráfego que entra no *fabric* é marcado com um cabeçalho VXLAN que identifica a política, isto é, grupo de origem, *tenant*, *bridge domain*, política de balanceamento de carga ou de telemetria.

Integração com Hypervisors

O ACI permite nativamente uma integração direta com os principais motores de virtualização utilizados nos grandes centros de dados. Essa integração é conseguida precisamente através da extensão do *overlay* descrito anteriormente até ao *switch* virtual do motor de virtualização. Como podemos ver na Figura 29 a integração com VMWARE e KVM é através do uso de VLANs e VXLAN, no entanto a integração com a Microsoft é através de NVGRE (*Network Virtualization Generic Routing Encapsulation*).

Esta integração permite que todas as funcionalidades descritas anteriormente, como a aplicação de políticas, a mobilidade, etc. sejam aplicadas de igual modo a dispositivos físicos diretamente ligados ao *fabric,* ou a máquinas virtuais alojadas em motores de virtualização com os seus *switchs* virtuais nativos.

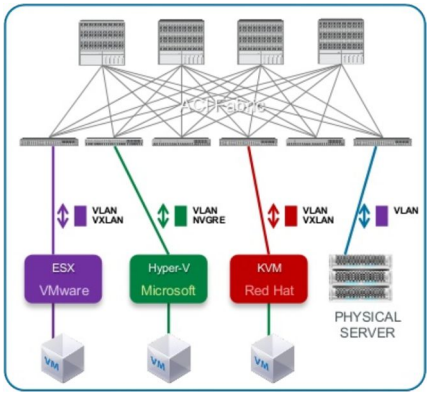


Figura 29 - Integração com motores de virtualização (retirado de: Apresentação do Cisco Live)

2.6. Arquiteturas híbridas (Hybrid Cloud)

O conceito de Cloud pública já existe há quase uma década. Tem sido interessante assistir à evolução, e até aceitação desta abordagem no tecido empresarial. Na realidade, num muito curto espaço de tempo, passamos de uma posição de algum ceticismo para intenções de experimentação, e atualmente vivemos num momento de total aceitação, onde a grande questão é qual o melhor local para alojar determinada aplicação, numa cloud pública, ou *on-prem*[4]. Durante anos esta decisão foi muitas vezes influenciada por características relacionadas com agilidade, velocidade de implementação ou automação, que são características nativas nas *Clouds* públicas. No entanto, hoje em dia com o amadurecimento do conceito das *Clouds* privadas, estas facilidades são conseguidas também *on-prem*, pelo que a decisão ficou um pouco mais difícil de tomar.

Independentemente das razões que cada organização considere relevantes para esta tomada de decisão, é inevitável depararmo-nos com situações em que será mais vantajoso colocar a aplicação A na Cloud e a aplicação B *on-prem*. Esta situação rapidamente evolui para um cenário em que uma aplicação alojada na Cloud necessita de partilhar informação com uma aplicação *on-prem*, levando-nos com naturalidade para um ambiente híbrido.

Uma arquitetura híbrida, é por definição mais complexa do que uma baseada apenas num dos tipos de *Cloud*, no entanto é a única abordagem que nos permite verdadeiramente tirar partido do melhor que cada abordagem disponibiliza.

Por um lado, uma *Cloud* privada, apenas operada internamente pela organização, permite o cumprimento de normas regulatórias, ou do país de residência ou do próprio negócio, relativamente ao local físico onde a informação é armazenada. Tem, no entanto, a desvantagem de ter um limite no que

[4] *On-premises* – alojar no centro de dados local

diz respeito aos recursos de computação e do modelo de custeio obrigar a um grande investimento no início do processo.

Por outro, uma *Cloud* pública, utiliza infraestrutura gerida por terceiros, o que em determinadas situações poderá levantar questões referentes à privacidade dos dados, no entanto disponibiliza uma capacidade quase infinita e um modelo de custeio baseado em *pay-as-you-go*, isto é, apenas pagamos a capacidade que consumimos. Outra grande vantagem que encontramos nas *Clouds* públicas são os serviços geridos. A sua utilização muitas vezes obriga a alterações aplicacionais, mas tipicamente com ganhos significativos do ponto de vista da resiliência da aplicação, e uma redução do esforço de gestão.

No entanto, é claro que as arquiteturas hibridas são o único caminho capaz de endereçar todas as necessidades de uma organização, o que levanta inúmeros desafios não só à componente de comunicações, pela obvia necessidade de garantir que uma infraestrutura local comunique com uma infraestrutura remota, mas também pela necessidade de movimentar aplicações de um "mundo" para o outro, mantendo o mesmo controlo, visibilidade e politica de controlo de acessos. Um tema também bastante relevante é a dificuldade que as organizações sentem, no momento de adoção de novas tecnologias. Garantir que as suas equipas têm o conhecimento técnico necessário para gerir infraestruturas locais e simultaneamente infraestruturas de múltiplos operadores de cloud distintos.

2.6.1. Principais desafios das arquiteturas hibridas

Apesar de conseguirmos interligar os dois mundos, recorrendo a tecnologias de comunicações já conhecidas, é necessário considerar outros desafios inerentes à adoção de uma arquitetura híbrida, uma vez que apesar de ser possível as trocas de informação continuam a ser dois mundos independentes, muitas vezes utilizando tecnologias distintas, o que levanta inúmeras questões, nomeadamente:

- Aumento da superfície exposta a ataques – ao estender o nosso centro de dados para uma cloud pública temos de conhecer bem as responsabilidades de cada interveniente na componente de segurança. O operador de cloud não se responsabiliza por uma configuração incorreta feita pelo cliente que inadvertidamente exponha serviços críticos na Internet.
- Dificuldade na gestão de identidades – uma das formas mais fáceis de endereçar este tema é estender a solução de gestão de identidade para a cloud, no entanto é necessário garantir que essa partilha de informação não compromete as políticas de segurança da empresa.

- Privacidade dos dados – com a implementação do Regulamento Geral sobre a Proteção de dados (RGPD), este tema toma ainda mais relevância no que concerne aos dados pessoais dos cidadãos europeus. No entanto, sempre houve, em determinadas áreas de negócio, regras claras sobre a privacidade da informação, nomeadamente área da saúde e finanças, que colocam entraves ao armazenamento de determinada informação fora dos países de origem. É também pertinente referir que a própria gestão de chaves de encriptação é muitas vezes considerada como relevante numa análise de risco. Ou usamos soluções/chaves distintas, o que tem o ónus de aumentar a complexidade, ou confiamos no provedor no sentido de permitir que as mesmas chaves que usamos no nosso centro de dados, sejam também utilizadas pelos motores de base de dados da cloud pública.
- Política unificada de controlo de acessos – pelo facto das tecnologias utilizadas em cada *Cloud* serem diferentes, torna-se muito complicado fazer cumprir com a política de controlo de acessos definida pela organização.

O grande foco deste trabalho é precisamente nas arquiteturas de comunicação que suportam não só os ambientes híbridos, mas também a movimentação das componentes aplicacionais, garantindo a correta implementação dos controlos de acesso, visibilidade e gestão centralizada, independentemente do local ou do tipo (máquina virtual ou *container*) onde a componente aplicacional se encontre.

2.6.2. Soluções de interligação com cloud's públicas

Ligação VPN

Para estabelecer uma ligação privada entre as redes de um centro de dados e a cloud temos de cumprir com alguns requisitos. Do lado do centro de dados temos de ter um equipamento que suporte o estabelecimento de tuneis encriptados, e do lado da cloud temos de configurar um serviço denominado *VPN Gateway*. Esta abordagem, pelo facto de utilizar a própria internet como meio de transporte, é recomendada para fluxos menos críticos ou em que a aplicação consegue suportar algum acréscimo de latência e não necessita de uma total previsibilidade na qualidade da ligação. Atualmente os *VPN Gateway* de Azure ou AWS não ultrapassam os 1.25Gbps de *throughput*.

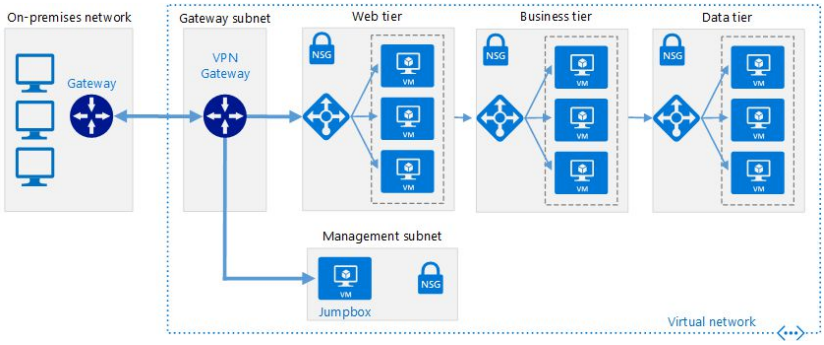


Figura 30 – Exemplo de arquitetura/ligação VPN (retirado da página da MSFT [27])

Ligação dedicada

Este método passa pela contratação de um circuito dedicado (*internacional*) entre o centro de dados local e os centros de dados dos operadores de cloud's públicas. Esta solução é ideal para cenários com uma grande distribuição das componentes aplicacionais, com a necessidade de grandes larguras de banda ou de total previsibilidade da qualidade da comunicação. Dependendo do operador internacional de comunicações selecionado, conseguimos ter ligações até 10Gbps, ou larguras de banda escaláveis com o intuito de adaptar a capacidade às necessidades momentâneas da organização.

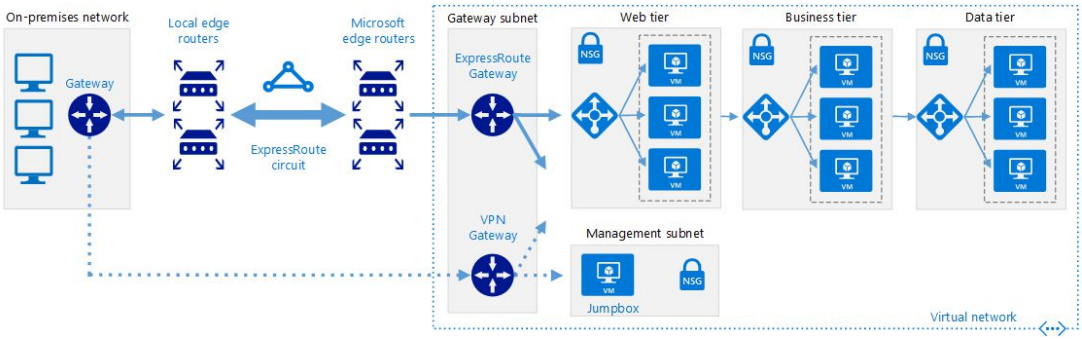


Figura 31 - Exemplo de arquitetura/ligação dedicada (retirado da página da MSFT [28])

Ao dia de hoje já é possível beneficiar de ligações dedicadas entre operadores distintos de cloud 's públicas, isto é, podemos contratar uma ligação dedicada entre a cloud da Microsoft e a cloud da Oracle, permitindo utilizar os melhores serviços de cada operador.

2.6.3. Funcionalidades de controlo de acessos nas Clouds públicas

As redes virtuais dentro das cloud's públicas são normalmente isoladas entre si através de encapsulamentos, recorrendo a protocolos conhecidos e amplamente utilizados por provedores de serviços de comunicações, como por exemplo NVGRE (*Network Virtualization using Generic Routing Encapsulation*), como é o caso da Microsoft.

A implementação deste tipo de encapsulamento confere um grau de isolamento lógico por cima de uma infraestrutura comum, o que nos permite criar redes virtuais com diferentes propósitos

(desenvolvimento, teste ou produção) mas com endereçamento idêntico se o pretendermos. Como seria de esperar, se criarmos redes virtuais com blocos de endereçamento distintos é possível permitir conectividade direta entre as duas redes virtuais. À semelhança do que fazemos em ambientes não cloud, aqui também conseguimos segmentar as redes virtuais em sub-redes mais pequenas, permitindo a implementação de mecanismos de controlo de acesso em diferentes pontos da rede.

Mecanismos de controlo de acesso na rede

A capacidade de segmentar a rede conforme descrevemos acima, não confere automaticamente a capacidade de impedir que determinada máquina virtual ou serviço Cloud tenha conectividade um com o outro, este controlo é efetuado através de *Network Security Groups.*

Utilizando NSG é possível permitir ou negar a passagem de informação entre dois pontos dentro da própria Cloud e também controlar o tráfego que entra ou sai da Cloud. Na prática conseguimos aplicar regras de entrada e/ou saída nas próprias máquinas virtuais ou nas sub-redes como um todo. Estas regras têm a capacidade de analisar:

- Endereço IP de origem e destino
- Porta de origem e destino
- Protocolo utilizado ao nível da camada 4 do modelo OSI (TCP/UDP)

Estes mecanismos, apesar de amplamente utilizados, apenas permitem implementar regras de segurança baseadas em parâmetros encontrados ao nível das camadas 3 e 4 do modelo OSI, o que em variadíssimas situações se verifica como insuficiente.

A forma de conseguir implementar políticas baseadas em fatores mais relevantes, como por exemplo, logica aplicacional ou parâmetros aplicacionais (nível 7 do modelo OSI) obriga a recorrer a soluções de terceiros.

2.6.4. Controlo de acesso em datacenters locais (Cloud's Privadas)

O controlo de acesso em centros de dados locais consegue-se também através da definição de algum tipo de isolamento entre as camadas aplicacionais. Este isolamento, até ao aparecimento das redes baseadas em software (SDN), era conseguido através da criação de VLAN's (*Virtual Local Area Network*) e em casos mais complexos recorrendo também a múltiplos VRF's (*Virtual Routing Forward*). Estes mecanismos de isolamento têm uma correspondência direta com os mecanismos utilizados na Cloud's públicas, isto é, uma VLAN pode ser perfeitamente comparada com uma sub-rede (*Subnet*) e um VRF com uma rede virtual (*VNET*), sendo que os mecanismos de controlo nativos na infraestrutura permitem apenas a criação de regras baseadas em parâmetros ao nível 3 e 4 do modelo OSI.

Apesar da obvia semelhança entre conceitos e capacidades, o facto de interligarmos as redes do centro de dados com as redes virtuais da cloud pública, em nada simplifica a gestão da

infraestrutura ou da política de controlo de acessos, uma vez que as metodologias e ferramentas de gestão são distintas.

O aparecimento de novas necessidades de controlo, como a *micro-segmentação*, e novos modelos de computação, como os *Containers,* aumentam ainda mais os desafios identificados anteriormente.

2.6.5. Containers em arquiteturas hibridas

A dicotomia *Cloud vs. On-Premises* não é propriamente um tema recente, no entanto o aparecimento dos *Containers*, trouxe de volta a questão. A crescente adoção de metodologias de desenvolvimento ágil [29], que pela necessidade de integrações continuas das sucessivas entregas de novos desenvolvimentos, apontam claramente para uma filosofia DevOps, levam a uma natural associação de todos estes conceitos às Clouds públicas, pelas capacidades nativas de automatização da infraestrutura.

No entanto, as razões que levaram as empresas a optar por implementar infraestruturas híbridas, antes da utilização dos Containers, continuam válidas e este facto associado à evolução das tecnologias utilizadas nos centros de dados locais, permitem hoje os mesmo graus de automação e a utilização das mesmas ferramentas de orquestração tanto nas Clouds públicas, como nos ambientes locais, vulgarmente denominados Clouds privadas.

Razões para utilizar Containers num centro de dados local

Utilização de recursos disponíveis exclusivamente em centros de dados locais. Esta é provavelmente a mais evidente, na eventualidade de necessitarmos que os Containers utilizem algum tipo de recurso específico que esteja disponível localmente, como por exemplo hardware de computação específico (GPU's), ou algum tipo de armazenamento não disponível numa Cloud pública.

A interação ou dependência de serviços locais. Se o serviço que está implementado em Containers necessita de consumir serviços locais ou vice-versa, em que o acréscimo de latência ou limitações de largura de banda, pelo facto de estar numa Cloud pública, prejudicam o desempenho global da aplicação é outro caso em que fará sentido alojar o Container localmente.

Localização da informação. Apesar dos containers, por natureza, serem objetos efémeros, em determinadas situações poderão necessitar de armazenar informação, o que pode levantar questões legais ou de *compliance* com normas do negócio ou do país no caso dessa informação estar em determinadas geografias.

Migração para a cloud ou infraestruturas *legacy*. Nos casos em que a organização tem interesse em migrar para a Cloud, mas o seu ambiente aplicacional é composto por várias aplicações monolíticas com inúmeras interdependências, poderá fazer sentido começar a adotar Containers com

o intuito de ao longo do tempo reduzir a complexidade desse movimento. Ou se simplesmente fizeram um investimento avultado em infraestruturas tradicionais, mas não pretendem que esse facto condiciona a evolução das arquiteturas aplicacionais, isto é, abordagens baseadas em microserviços.

Ambientes de teste nos centros de dados locais e apenas os ambientes de produção na Cloud. Ao testar num ambiente mais controlado poderá facilitar a deteção de problemas quando colocarmos a aplicação em produção num ambiente Cloud. Este cenário pode também ser considerado útil em cenários em que a organização pretende manter as funcionalidades que está a desenvolver totalmente privadas, ou em situações em que as funcionalidades de segurança não estão ainda totalmente desenvolvidas.

Opções para integrar clusters de Kubernetes com a rede do centro de dados.

Ao olharmos para a Figura 32, facilmente constatamos que nativamente a componente de rede do Kubernetes já apresenta uma maior complexidade do que os ambientes tradicionais. Toda a agilidade e flexibilidade possível no interior de um cluster só é possível com a criação de camadas de abstração de rede, normalmente usando o protocolo VXLAN [RFC7348]. É importante referir que também é possível instalar o Kubernetes diretamente numa máquina física, eliminando algumas camadas, no entanto o cenário representado na Figura 32 é o cenário mais comum, principalmente em organizações que estão a iniciar a adoção de containers.

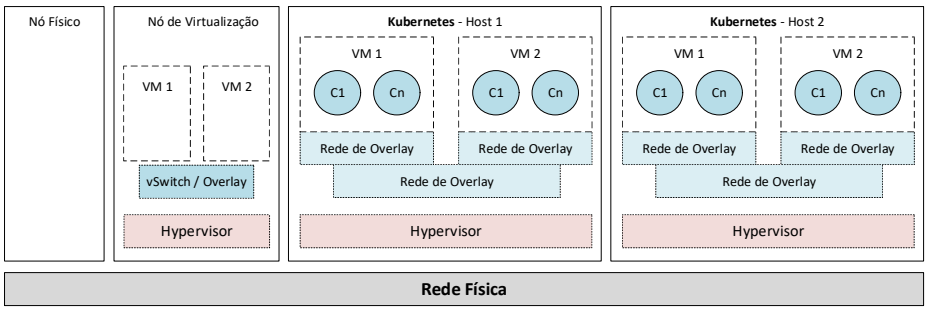


Figura 32 - Comparativo de componentes de rede

A integração das diferentes componentes de rede (Física, Virtual e Overlay), não deve comprometer a performance. Por exemplo, se o container C1 alojado no Host 1 necessita de falar com outro container Cn no mesmo Host, não fará sentido que os pacotes tenham de "descer" até à componente física de forma a permitir a comunicação entre containers dentro do mesmo host, pelo que é necessário adotar uma arquitetura que responda corretamente aos requisitos da organização.

Arquiteturas de rede possíveis

Conectividade Nativa

O caso de uso mais comum para esta implementação é em ambientes em que apenas existem clusters locais. A conectividade é assegurada através dos conceitos tradicionais de rede, como

encaminhamento/encapsulamento de *frames* (L2), roteamento de pacotes (L3) e tradução de endereços IP. Numa abordagem layer 3 (roteamento), existe uma rede IP por cada host físico o que permite adotar um dos seguintes modos de funcionamento:

- Conectividade através de tradução de endereços, este é modo por defeito do Kubernetes, a rede IP dos containers é vista apenas internamente no cluster e os pedidos ao exterior obrigam a uma tradução do IP de origem (NAT) para o IP do próprio host. O que obriga a configurações especificas quando pretendemos que um container esteja disponível externamente, por exemplo, a criação de encaminhamentos ao nível da porta de rede (TCP), na interface de rede do host físico.
- Conectividade direta, isto é, utilizando protocolos de roteamento dinâmico, de forma a permitir a descoberta automática de novas redes. Este modo necessita da utilização de um plugin de CNI como por exemplo o *Calico* [30], que utiliza o protocolo BGP.

Numa abordagem layer 2 (encapsulamento), o CNI instalado no host permite as interfaces de rede dos containers utilizem a rede física. Na prática o CNI comporta-se com um switch de rede normal onde todos os containers são ligados, permitindo que se apresentem como um dispositivo normal da rede física. Nestes casos o IP configurado como *default-gateway* nos containers estará configurado diretamente na infraestrutura física. Para este modo de funcionamento podemos utilizar o CNI *Contiv* [31].

Este modelo tem a vantagem de apresentar uma arquitetura simples, e bastante escalável do ponto de vista do tamanho do cluster. Relativamente à aplicação de políticas de acesso, o CNI *Contiv* mesmo no modelo descrito acima já permite a sua aplicação, no entanto nem o *Calico* nem o modo *default* têm qualquer funcionalidade a este nível.

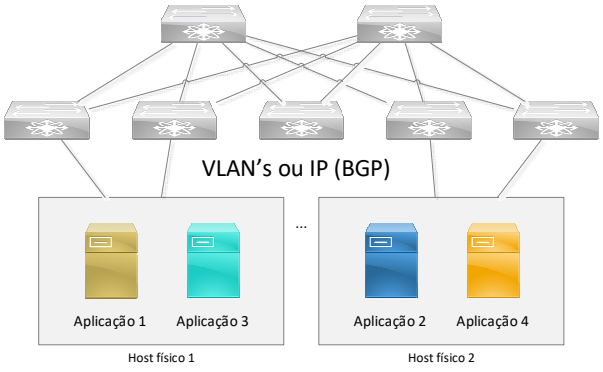


Figura 33 - Arquiteturas de rede, modelo nativo

Conectividade através de overlay

Neste modelo é criado uma rede *overlay* que interliga os hosts onde os containers são executados. Na prática podemos tirar partido de uma extensão (L2) de rede pelos hosts físicos que

compõem o cluster, permitindo uma comunicação entre containers utilizando apenas VLAN's. Para permitir a comunicação com serviços externos ao cluster, são estabelecidas adjacências de BGP entre os hosts e os equipamentos de rede físicos, permitindo uma descoberta automática de todo o espaço de endereçamento existente.

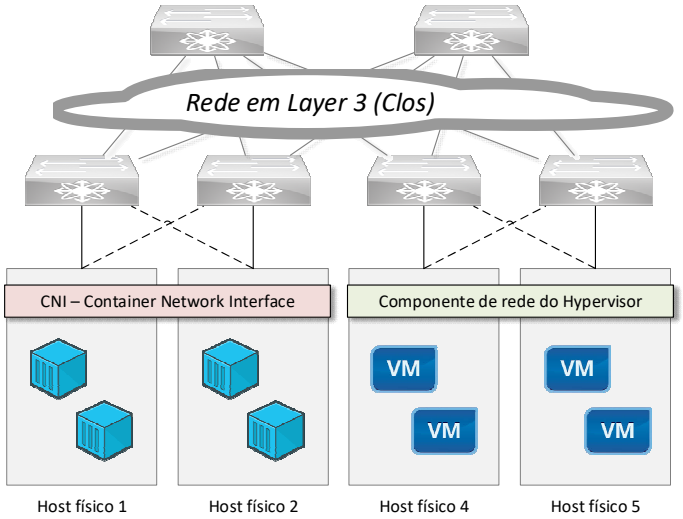


Figura 34 - Arquiteturas de rede, modelo overlay

Neste modelo já é possível mover containers dentro do cluster sem necessitar de alterar os endereços, no entanto este modelo dificilmente escala para soluções com clusters em diferentes localizações, e não permite uma integração no que concerne às políticas de acesso.

Conectividade integrada com a tecnologia existente no datacenter

Um modelo de total interoperabilidade estará sempre condicionado à compatibilidade entre as tecnologias já implementadas no centro de dados e a tecnologia que pretendemos adicionar. Neste caso, tendo em conta que a tecnologia que o centro de dados utiliza (Cisco ACI), já disponibiliza um CNI específico de interoperabilidade, deverá ser este o caminho a seguir.

O plugin ACI CNI [32] é o plugin recomendado pela Cisco, para ambientes em que o centro de dados dispõe da sua solução SDN denominada Cisco ACI.

Este plugin integra totalmente do ponto de vista de conectividade e do controlo de políticas de acesso com clusters Kubernetes. Permite fazer um mapeamento entre as políticas de rede (NetworkPolicies) do Kubernetes com as políticas do ACI, permitindo a unificação de políticas para todos os tipos de *workloads* (containers, máquinas virtuais ou máquinas físicas).

Para além de centralizarmos a componente de controlo de acessos e de conectividade, é também possível com o ACI CNI, implementar diretamente nos equipamentos Cisco, através de PBR (Policy Based Routing), os mecanismos de balanceamento de carga nativos do Kubernetes.

Estas funcionalidades permitem que as aplicações se movimentem sem qualquer alteração das suas configurações.

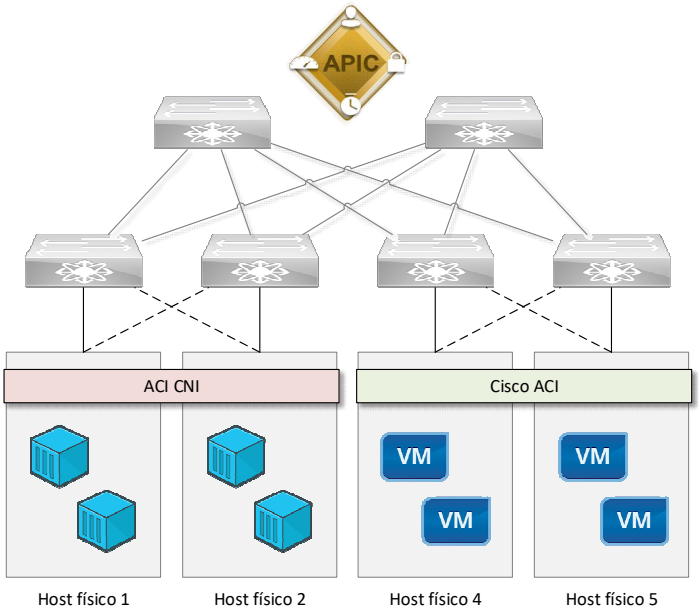


Figura 35 - Arquitetura de rede, integração com Cisco ACI

3. ARQUITETURA E SOLUÇÃO PROPOSTA

Este capítulo inicia por descrever o problema que a empresa proponente pretende endereçar (secção 3.1), descrevendo os principais casos de uso onde a solução encontrada poderá ser aplicada (secção 3.2). Finalmente, (secção 3.3) descrevemos de forma resumida, a arquitetura da solução proposta.

3.1. Introdução ao problema

Mesmo sem considerarmos o tema Containers, a necessidade de simplificar as metodologias de gestão da infraestrutura, ou uniformizar as ferramentas de monitorização e controlo em ambientes híbridos já era uma necessidade. Um ambiente verdadeiramente multicloud, isto é, que inclua uma ou mais Clouds públicas e Clouds privadas, obriga a equipas mais numerosas e com conhecimentos nos vários ambientes.

Mesmo com uma abordagem, do ponto de vista operacional segmentada, a necessidade de tirar partido das mais valias de cada Cloud é uma realidade pelo que se levantam temas relativamente à mobilidade de ambientes entre Clouds, que no caso de aplicações alojadas em máquinas virtuais se torna extremamente complicado, pela incompatibilidade entre os diferentes *hypervisor 's* adotados por cada cloud.

Esta dificuldade é muitas vezes uma das principais razões que levam as equipas de desenvolvimento a adotar Containers. A camada de abstração que adicionam permite precisamente esta compatibilidade, no entanto é necessário que a componente de rede se adapte de forma dinâmica a estas movimentações, mantendo políticas e garantindo que o utilizador final não tem qualquer perceção sobre o local onde a aplicação está a ser executada. A Figura 36 pretende representar o cenário final pretendido. Conforme referimos anteriormente, recorrendo à solução SDN da Cisco (ACI) já é possível movimentar máquinas virtuais entre centros de dados locais sem necessitar de alterar o endereço IP, e mantendo a mesma política de controlo de acessos em vigor. A integração que existe com os equipamentos que ligam os centros de dados à rede MPLS (routers Cisco ASR1001X) faz com que os pedidos das localizações onde os clientes se encontram (lojas, estruturas centrais ou armazéns) sejam automaticamente encaminhados para o centro de dados onde a máquina virtual se encontra ativa.

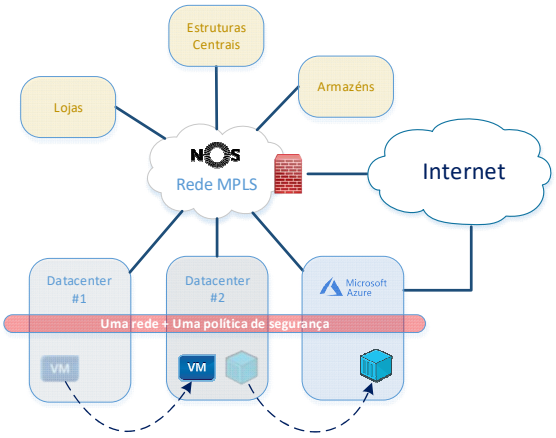


Figura 36 - Representação da mobilidade entre locais considerados

Ao conseguirmos mover um *workload* baseado em containers de um centro de dados local para uma cloud pública, leva-nos a explorar soluções que permitam, do ponto de vista de rede e segurança, operacionalizar essa mobilidade.

3.2. Descrição dos casos de uso/validação

3.2.1. Aplicação de apoio ao processo de venda (POS)

Atualmente, uma das operações mais críticas do negócio da empresa proponente está assente numa aplicação cliente/servidor que não suporta qualquer tipo de redundância ou funcionalidade de alta disponibilidade. Há já algum tempo que a empresa tem vindo a explorar diversas alternativas no sentido de conferir uma maior resiliência a este serviço, mas sem grande consequência uma vez que a própria aplicação cliente impõe restrições bastante limitativas, nomeadamente:

- Não consegue efetuar chamadas ao serviço utilizando nomes (DNS)
- O endereço IP do servidor é parametrizado no momento da instalação do equipamento cliente, não permitindo uma fácil alteração.
- O fabricante não suporta a aplicação em caso de utilização de balanceadores de carga e/ou proxys reversos.

Levando-nos ao desafio de conseguir mover o referido serviço sem alterar o endereço IP que o equipamento cliente conhece.

3.2.2. Facilitar a movimentação de aplicações para a cloud

A utilização das cloud 's públicas, é já uma realidade bastante comum na empresa proponente, seja para o alojamento de novas aplicações ou mesmo em momentos de alteração de componentes que atualmente estão alojados no centro de dados local. No entanto deparam-se muitas vezes com desafios acrescidos quando essas componentes interagem com outras aplicações que se irão manter no centro de dados, uma vez que a alteração do endereço IP do serviço, obriga a alterar

inúmeras aplicações e a atualizar a política de controlo de tráfego no interior do centro de dados. Pelo que a capacidade de mover serviços para a cloud, sem alterar endereços IP e/ou garantindo a mesma política de acesso seria de uma grande mais valia para a organização.

3.3. Descrição da arquitetura pretendida

Inicialmente será implementado no centro de dados principal um cluster de Kubernetes, com três Workers e um Master devidamente integrado na rede do centro de dados. Para isso, utilizaremos o CNI disponibilizado pela própria Cisco, permitindo que através do controlador SDN (APIC) seja efetuada a gestão da rede e o controlo de acessos das máquinas físicas, virtuais e também dos containers. Na prática, como vemos na Figura 37, esta abordagem recorre ao switch virtual Open vSwitch para criar uma rede overlay entre todos os nós do cluster, permitindo a automatização das configurações de conectividade, e de implementação de políticas de acesso unificadas.

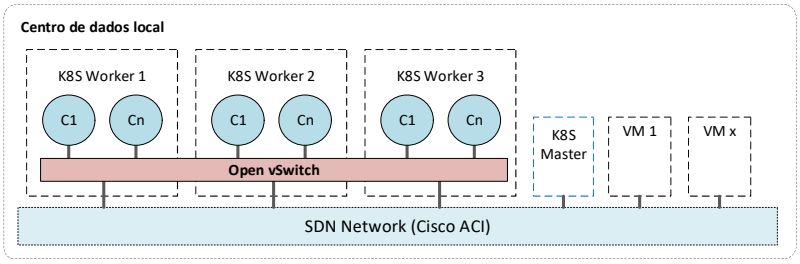


Figura 37 - Representação da integração de rede K8S com ACI

Além da integração ao nível da rede, será necessário garantir que o cluster de Kubernetes consegue utilizar os serviços de armazenamento de informação local, e também tirar partido da solução de balanceamento de carga existente no centro de dados, de forma a garantir que uma aplicação criada na cloud pública tem este tipo de recursos disponíveis quando movida para o centro de dados local.

Como vemos na Figura 38 o Kubernetes terá a capacidade de orquestrar a criação e atualização de balanceadores no equipamento local, e no caso do armazenamento, será possível criar *Persistent Volumes* que mapeiam partilhas de rede NFS.

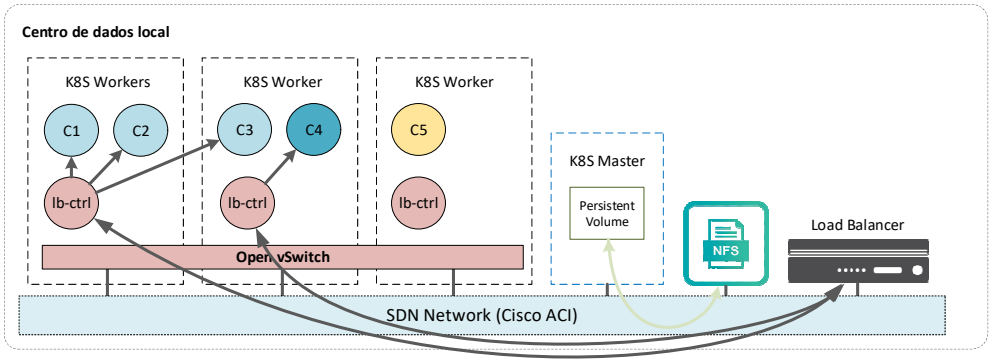


Figura 38 - Integração Kubernetes com centro de dados local

Finalmente, integraremos o ambiente local descrito acima, com as soluções de containers existentes na cloud Azure, nomeadamente:

- Solução IaaS, instalação manual do Kubernetes em máquinas virtuais Azure.

Numa abordagem IaaS (Infrastructure as a Service) em Azure, tiramos partido da cloud apenas para a disponibilização de máquinas virtuais, obrigando-nos a gerir e configurar todo o cluster. Apesar de existir nesta solução um maior esforço de gestão, do ponto de vista de integração pressupomos[5] que seja mais simples, uma vez que estendendo o Overlay criado pelo Open vSwitch, como vemos na Figura 39, deverá ser possível responder aos requisitos propostos.

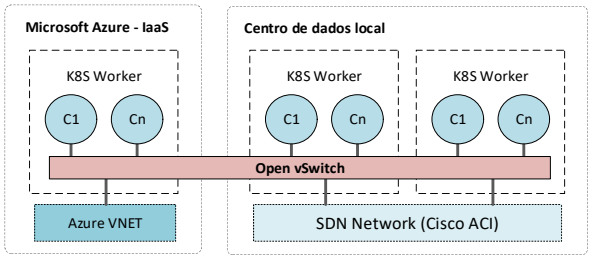


Figura 39 - Integração do Kubernetes com solução IaaS

- Solução PaaS, denominada AKS (Azure Kubernetes Service). No caso do AKS o facto de não conseguirmos controlar a componente de rede do cluster de Kubernetes obriga a considerar uma abordagem diferente. Conforme vemos na Figura 40, iremos criar o cluster AKS recorrendo às definições avançadas de rede, que possibilitam colocar os containers diretamente na rede virtual do Azure (eliminando um *overlay)*. Desta forma, esperamos que as funcionalidades de integração do ACI com a rede Azure, nos permitam endereçar os desafios propostos.

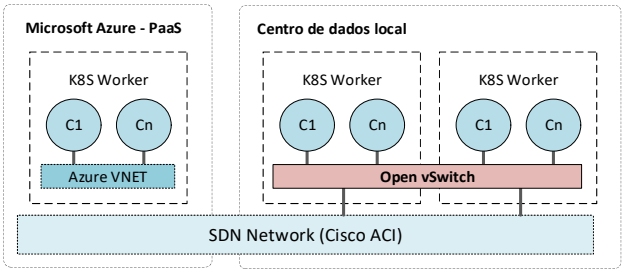


Figura 40 - Integração do Kubernetes com solução PaaS

[5] No decorrer do projeto constatamos que esta abordagem não é possível de implementar usando o CNI-ACI, uma vez que os nós do cluster têm de ter ligação em *layer2* com a VLAN *Infra* do ACI, o que não é possível atualmente.

4. DESCRIÇÃO DA BANCADA DE TESTES

Este capítulo descreve o trabalho de investigação e de implementação realizado em torno da conceção de uma bancada de testes que permitisse a validação da solução proposta. Nesse sentido a secção 4.1 descreve o processo de instalação de um cluster de Kubernetes, seja no centro de dados local ou na cloud pública da Microsoft. De seguida, na secção 4.2 abordamos a integração da rede do centro de dados com a rede do cluster de Kubernetes. Finalmente, na secção 4.3 abordamos as potencialidades de integração da solução do centro de dados diretamente com a cloud Azure.

4.1. Instalação de um cluster de Kubernetes

Uma das grandes vantagens de "pacotizar" uma aplicação em containers é a facilidade com que conseguimos disponibiliza-la em qualquer infraestrutura, inclusivamente num cluster Kubernetes.

A correta parametrização de um cluster Kubernetes com múltiplos nós exige um determinado conhecimento prévio de Linux e de redes, uma vez que a solução se estende por múltiplas maquinas (virtuais ou físicas) e a rede tem de estar corretamente configurada de forma a que os containers dentro do cluster possam "falar" entre si através de uma única rede IP.

Existem inúmeras formas [33] de implementar um cluster de Kubernetes. Apesar da lista estar em constante evolução, o que condiciona a sua análise detalhada, é bastante pertinente listar as mais relevantes.

- Implementação local, apenas na nossa máquina de desenvolvimento
- Num datacenter com inúmeras máquinas
- Em Clouds públicas utilizando máquinas virtuais
- Em Clouds públicas utilizando clusters geridos

Com o intuito de documentar as opções mais relevantes para o nosso trabalho, apresentamos abaixo a descrição da implementação de um cluster gerido em Microsoft Azure e finalmente um cluster de três nós num datacenter tradicional. Em anexo disponibilizamos também a descrição do processo de instalação num ambiente local.

4.1.1. Instalar um cluster Kubernets na cloud pública Microsoft Azure

Antes de iniciar a descrição de como implementar em cluster é importante referir que as principais Clouds públicas disponibilizam várias maneiras de "correr" containers, e o Azure neste ponto não é exceção. A cloud da Microsoft disponibiliza os seguintes modelos:

- Executar um container recorrendo apenas ao serviço PaaS [34] *Azure Web Apps*, este serviço assenta em conceitos já conhecidos pelos utilizadores da cloud pública Azure, mas não dispõe de uma integração tão eficiente com as restantes componentes.
- Modelo IaaS [35] passa pela criação de máquinas virtuais na infraestrutura da Microsoft apenas com o sistema operativo instalado, isto é, na prática o processo de instalação do Kubernetes será em tudo semelhante ao que iremos descrever no capítulo 4.1.2.
- Solução *Azure Container Instance* (ACI), neste serviço a cloud Azure permite executar containers sem qualquer esforço na preparação da infraestrutura e consequentemente na sua gestão. Permite escalar a capacidade de processamento automaticamente e apresenta um modelo de custeio baseado em utilização, o que é vantajoso em situações momentâneas de acréscimo de capacidade.
- *Azure Kubernetes Service* (AKS) é o serviço que disponibiliza clusters de *Kubernetes* geridos pelo *Azure*. Este modelo reduz a complexidade e a sobrecarga operacional de gerir o cluster, ao transferir grande parte dessa responsabilidade para o *Azure*. Tarefas como monitorização, manutenção e gestão de controlo de acessos são completamente asseguradas pelo *Azure*, inclusivamente no momento da configuração inicial as parametrizações do *Master* e a configuração dos *Worker Nodes* são também asseguradas pelo *Azure*. Será este o modelo que iremos detalhar, uma vez que é o mais flexível quando pretendemos um ambiente *multicloud* ou *hibrido* (*cloud* e *on-prem*).

Atualmente temos duas opções para criar um cluster AKS, via linha de comandos ou via o portal de gestão da cloud Azure. A grande diferença entre estes métodos, é precisamente na componente de rede, uma vez que se pretendermos usar o `kubenet` [36] em detrimento do Azure-CNI, disponibilizado por defeito, temos mesmo de criar o cluster através da linha de comandos, já que essa opção não está disponível na interface gráfica. No nosso caso, não existe qualquer vantagem em mudar o CNI usado em Azure, pelo que iremos demonstrar o processo de criação via interface gráfico.

Configurações iniciais

Acedendo ao portal e escolhendo a opção de criação de um cluster Kubernetes temos de introduzir diversos parâmetros, sendo que na primeira página definimos algumas opções nativas do Azure, como a subscrição, o *Resource Group,* a região e o prefixo de DNS. Nesta janela definimos também a versão do Kubernetes que pretendemos utilizar, o número de *Worker Nodes* e os recursos de computação de cada *Worker Node*.

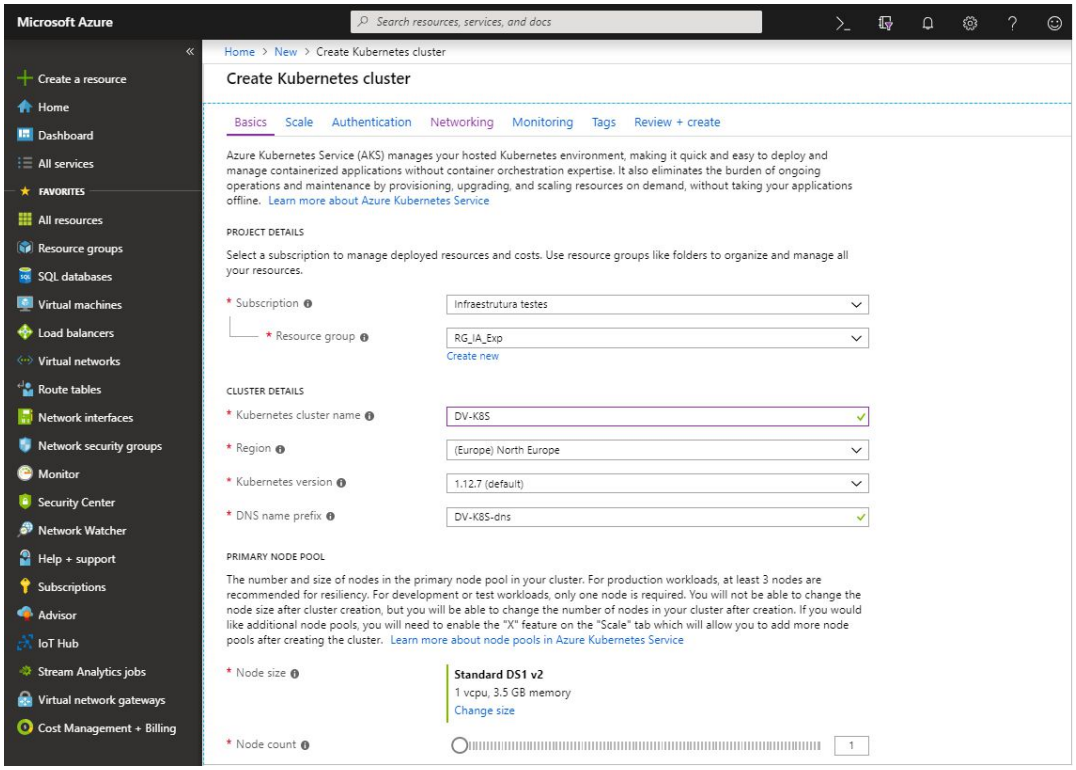


Figura 41 - Configurações iniciais do cluster em Azure

Configurações de escalabilidade

Tendo em conta que estamos a utilizar uma cloud pública podemos tirar partido da capacidade, quase infinita, que estes provedores disponibilizam. Para o serviço gerido de Kubernetes temos a possibilidade de escalar em número de nós (*Worker Nodes)* ou utilizando o serviço Azure Cluster Instance.

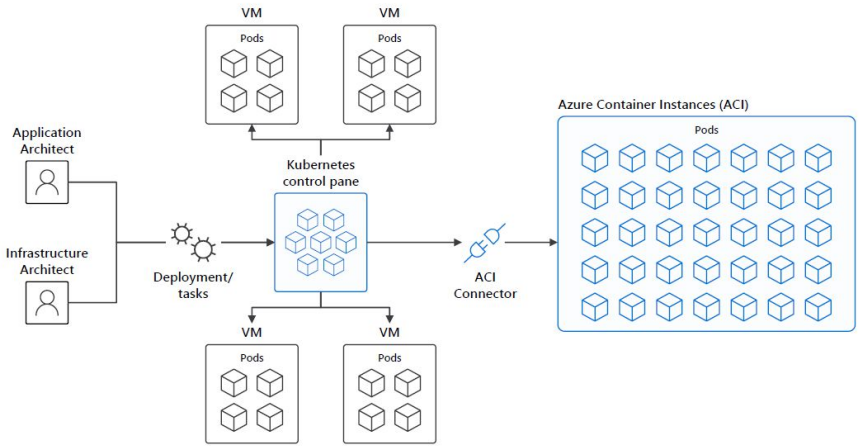


Figura 42 - Escalabilidade para ACI (retirado de PPT da Microsoft)

Mais uma vez as parametrizações são efetuadas no portal, conforme vemos na Figura 43.

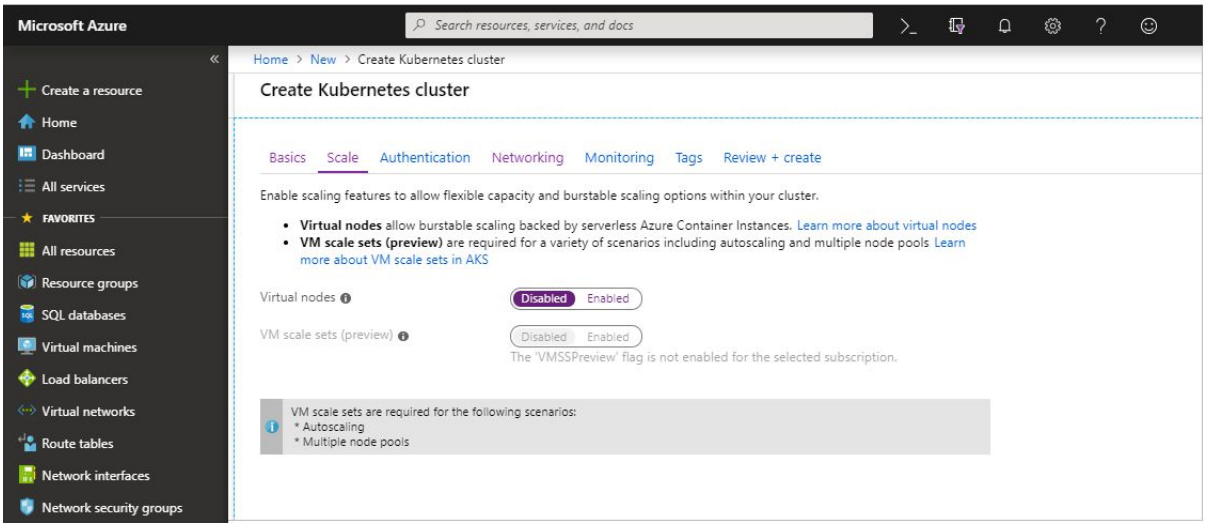


Figura 43 - Configurações de escalabilidade em Azure

Configurações de autenticação

Nesta janela definimos duas autenticações de extrema importância para o serviço. Por um lado, definimos de que forma é que o cluster de Kubernetes consegue criar outros recursos na cloud Azure, como por exemplo balanceadores de carga, áreas de armazenamento ou até registos locais de containers. Tendo em conta que esta integração é uma das principais mais valias de termos o cluster alojado numa cloud pública, esta funcionalidade é critica para tirarmos o devido partido do ambiente cloud. Por outro lado, podemos definir de que forma vamos gerir os acessos aos restantes recursos do cluster. Este tema é considerado como sendo um dos grandes desafios em ambientes Kubernetes criados em datacenters próprios, pelo que a possibilidade de integrar com os serviços de gestão de utilizadores já existentes é uma enorme mais valia.

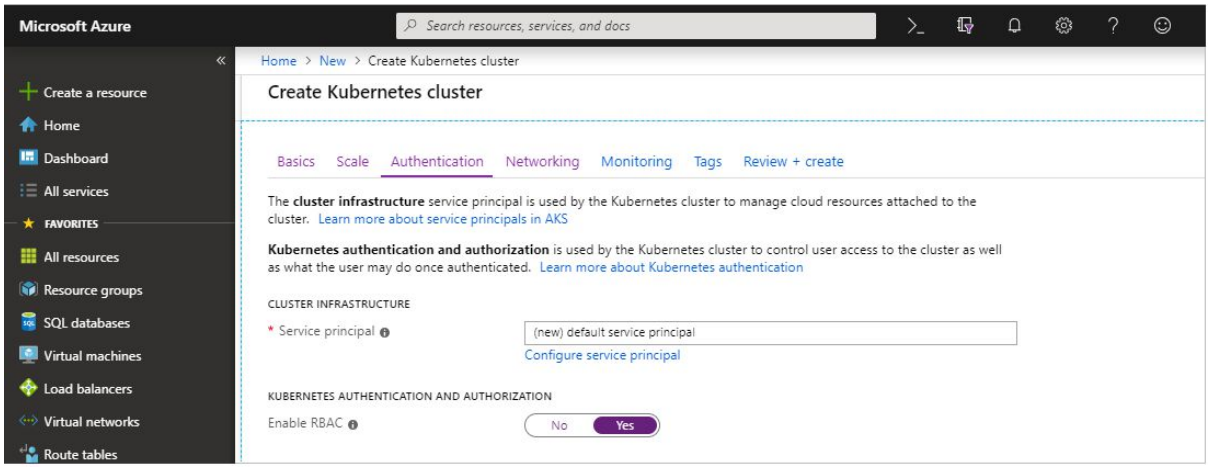


Figura 44 - Configurações de autenticação em Azure

Configurações de rede

Via interface gráfico conseguimos definir as redes IP que vão ser utilizadas pelo cluster. Definimos a gama de endereços utilizada pelos próprios POD's (*Cluster subnet*), o endereçamento para serviços internos (*Kubernetes services*) e o endereçamento de interligação entre a rede dos worker nodes e a rede de serviços (*Docker Bridge*).

O Azure CNI não cria um Overlay de rede para os POD's, o que acontece é que o Azure reserva 32 IP's da rede do Cluster para cada *worker node*, isto é, tendo em conta que cada POD vai necessitar de um endereço IP, em Azure apenas conseguimos ter 32 POD's por worker node[6].

[6] Esta limitação pode ser contornada com a adição de novas placas de rede ao worker node, uma vez que na prática a limitação está no número de IP's que cada interface de rede suporte, que são 32.

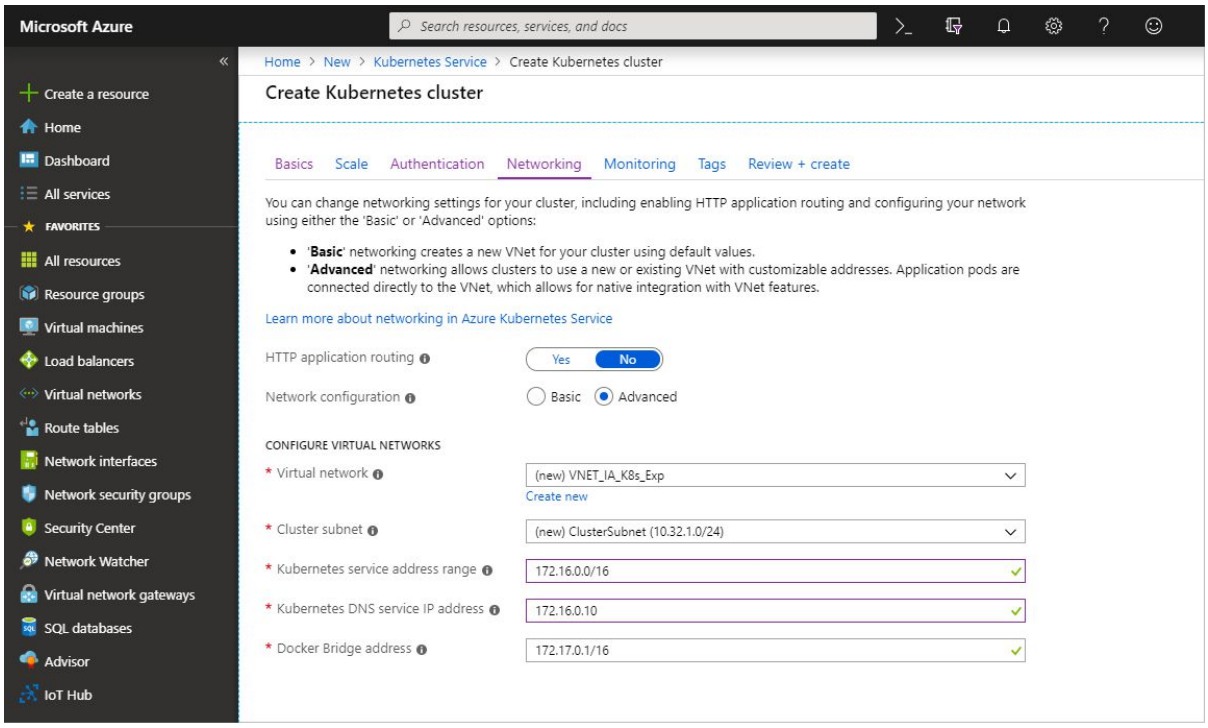


Figura 45 - Configurações de rede do cluster em Azure

Monitorização

Nesta janela parametrizamos as definições de monitorização que o próprio serviço gerido do Azure disponibiliza. Conseguimos inclusivamente visualizar num *dashboard* a ocupação de processador e memória de cada membro do cluster.

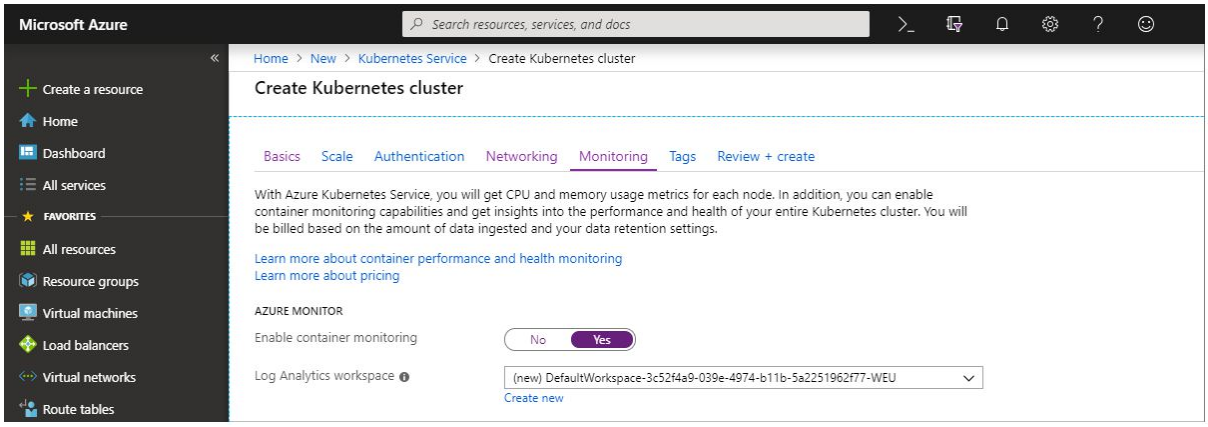


Figura 46 - Configurações da monitorização de cluste em Azure

Usar o `kubectl` para gerir o cluster através do portal Azure

A operação de um cluster gerido de Kubernetes em Azure em tudo semelhante à operação de um cluster de experimentação como o Minikube ou até de cluster instalado num datacenter local, isto é, utilizamos a ferramente `kubectl`.

Conforme vimos anteriormente, o facto de estarmos a usar um serviço gerido em Azure facilita o processo de parametrização inicial e gestão do cluster, mas também disponibiliza um acesso rápido à ferramente `kubectl`.

Basta apenas abrir uma *Cloud Shell* e executar o comando abaixo para automaticamente conseguirmos utilizar a ferramenta `kubectl` interagindo com o cluster que criamos.

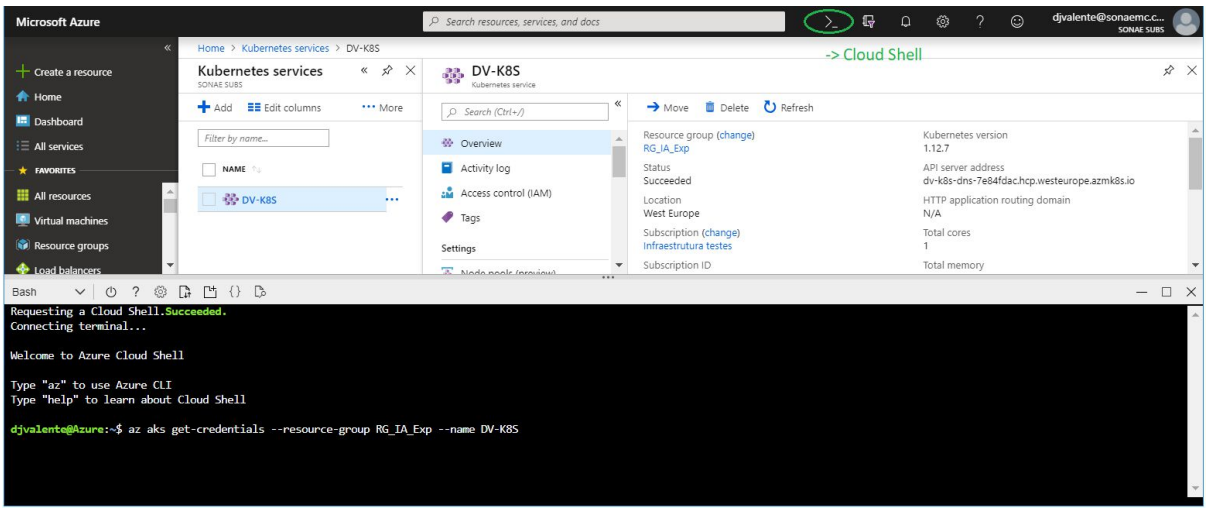


Figura 47 - Acesso ao AKS através da Cloud Shell

Este comando define como contexto atual do `kubectl` o cluster AKS indicado. Na prática é criado um ficheiro de configurações que permite aceder ao cluster.

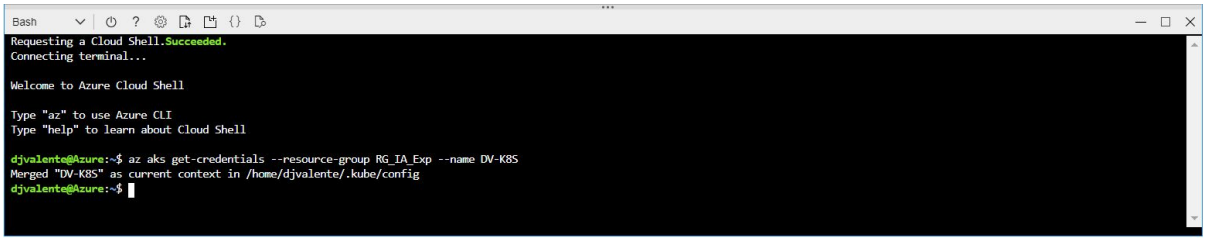


Figura 48 - Configuração do contexto de acesso ao AKS

Neste momento podemos utilizar, sem qualquer restrição, o comando `kubectl`, como podemos verificar na Figura 49.

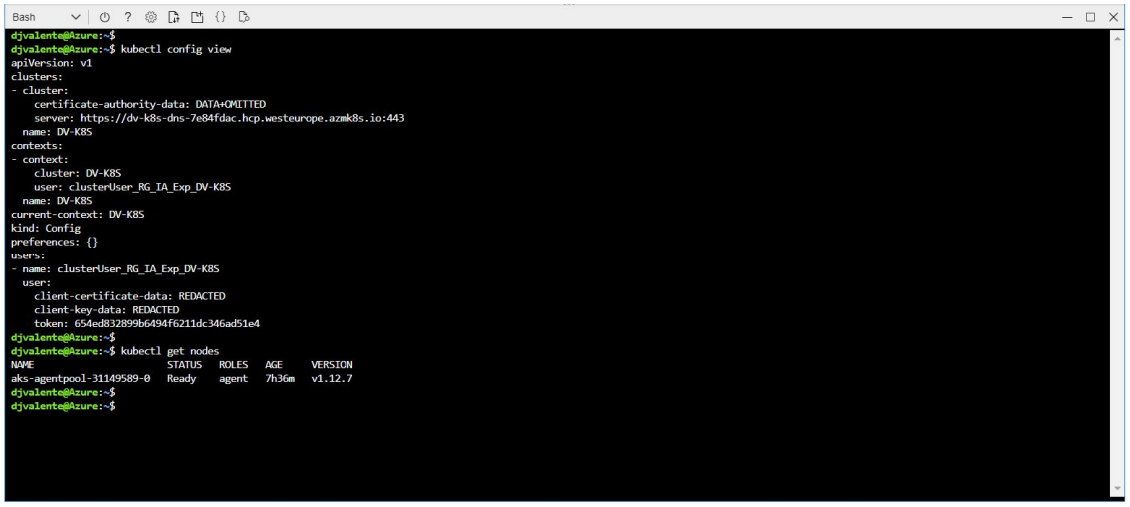


Figura 49 - Output dos comandos kubectl

Usar o `kubectl` para gerir o cluster através de uma máquina externa

Apesar da ferramenta `kubectl` estar disponível através do portal Azure, não é necessário ter este tipo de acesso de administração à subscrição para conseguir operar o cluster. Podemos utilizar o conteúdo do ficheiro de configurações gerado pelo Azure, para de qualquer máquina externa interagirmos com o cluster.

No caso de pretendermos utilizar um equipamento Windows para interagir com o cluster, podemos efetuar o download do `kubectl` para Windows diretamente do site oficial do Kubernetes [37]. À semelhança do ambiente Linux a recomendação é que o ficheiro seja colocado numa pasta que pertença à *path* do sistema operativo, e finalmente, é necessário criar uma variável de ambiente com o nome *KUBECONFIG* a referenciar o ficheiro com o conteúdo gerado pelo Azure.

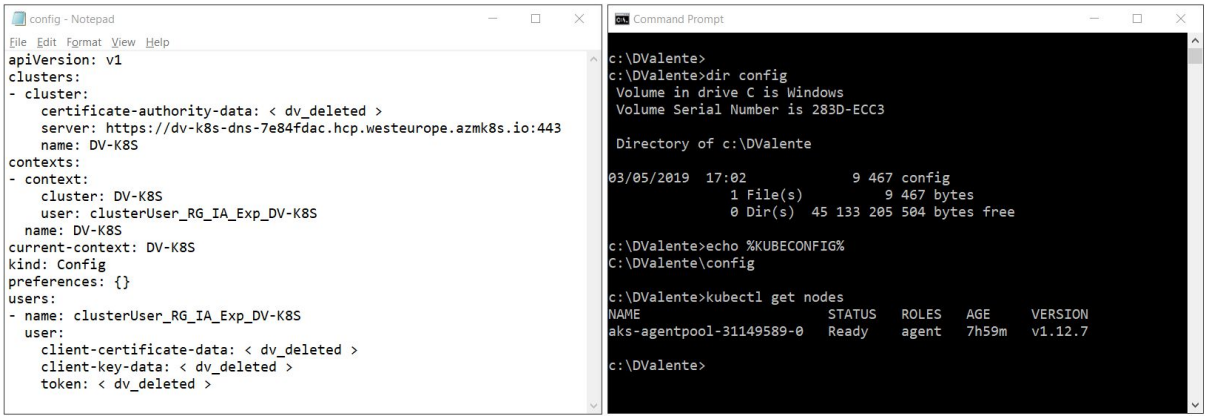


Figura 50 - Acesso ao AKS através de uma máquina Windows

Executar a aplicação de teste no AKS

Criação de um pod de forma imperativa

Uma vez que a imagem que pretendemos utilizar já se encontra num repositório publico apenas necessitamos de executar o comando:

```
kubectl run djv --image=djvalente/testes --port=88 --
generator=run-pod/v1
```

Uma vez que estamos a num ambiente isolado na cloud Azure, a única forma de conseguirmos aceder ao nosso container é através da atribuição de um endereço publico ao nosso *pod*. A forma correta de fazermos esta atribuição em Azure é através da criação de um load balancer em Azure, atribuir um IP publico ao load balancer e configurar esse mesmo load balancer para redirecionar o tráfego para o cluster de Kubernetes. Uma vez que a integração entre ambos os mundos (Azure e Kubernetes) é total, basta-nos um comando `kubectl` para que todos os passos descritos anteriormente se realizem:

```
kubectl expose pod djv --type=LoadBalancer --name djv-http
```

Conforme vemos na Figura 51 o pod foi criado e foi exposto no endereço publico 23.100.7.196:88

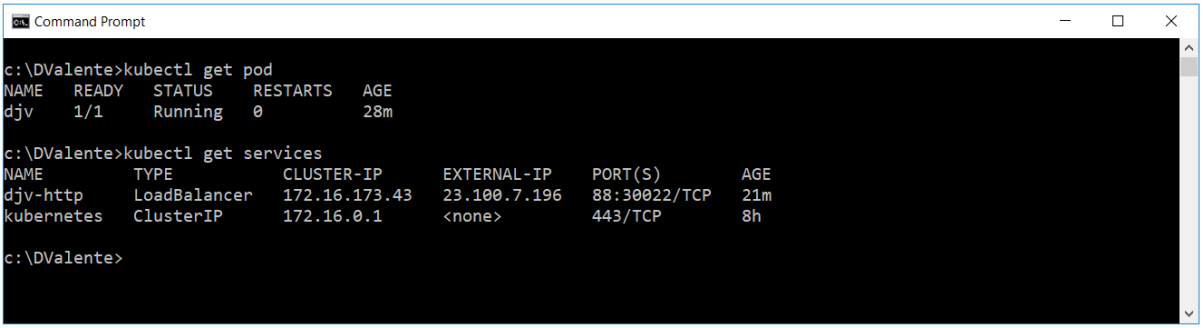


Figura 51 - Expor um pod alojado em AKS

Utilização do método declarativo em AKS

À semelhança do que referimos anteriormente também a utilização do modelo declarativo é bastante semelhante ao que descrevemos anteriormente, pelo que apenas necessitamos de executar o comando `kubectl create -f djv22.yaml`, tendo previamente efetuado uma pequena alteração ao ficheiro YAML relativamente ao tipo de serviço utilizado para expor o *deployment*. Conforme vemos na Figura 52 não podemos utilizar o tipo *NodePort* quando o cluster está em Azure, somos obrigados a recorrer ao serviço de balanceamento da cloud pública. Esta nuance é interessante do ponto de vista de resiliência da solução, uma vez que ficamos agnósticos ao IP do POD, isto é, pode haver movimentações e/ou transformações do serviço ou da infraestrutura em Azure, sem obrigar a que os clientes mudem a forma como consomem a aplicação.

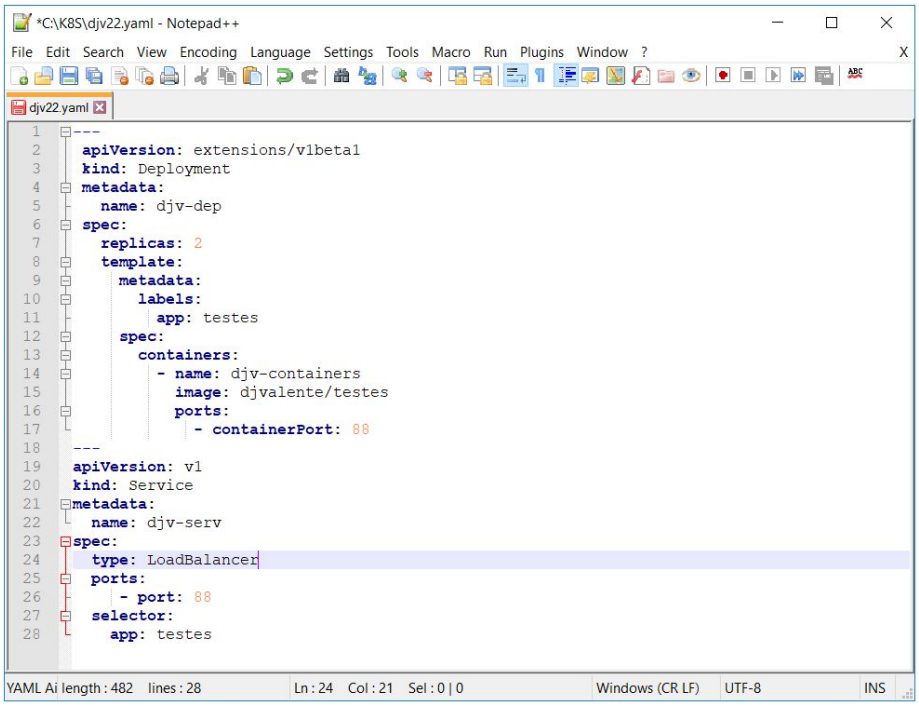


Figura 52 - Descrição de um deployment em AKS

Aceder a serviços por nome

Utilizando variáveis de ambiente

Quando um POD é iniciado, o Kubernetes inicializa um conjunto de variáveis de ambiente que contém o caminho para cada um dos serviços existente naquele momento. Se criarmos o serviço antes de criarmos o cliente, o cliente vai conseguir aceder ao serviço através das suas próprias variáveis de ambiente.

Como podemos ver na Figura 53, na altura da criação do POD existiam dois serviços disponíveis, pelo que se executarmos o comando `kubectl exec djv-dep-55f9bc6d8b-pc7qx env`, com o intuito de vermos as variáveis de ambiente do próprio POD, verificamos que existem variáveis com o sufixo _SERVICE_HOST e _SERVICE_PORT que contêm precisamente a informação necessária para o POD aceder ao serviço `djv-serv`.

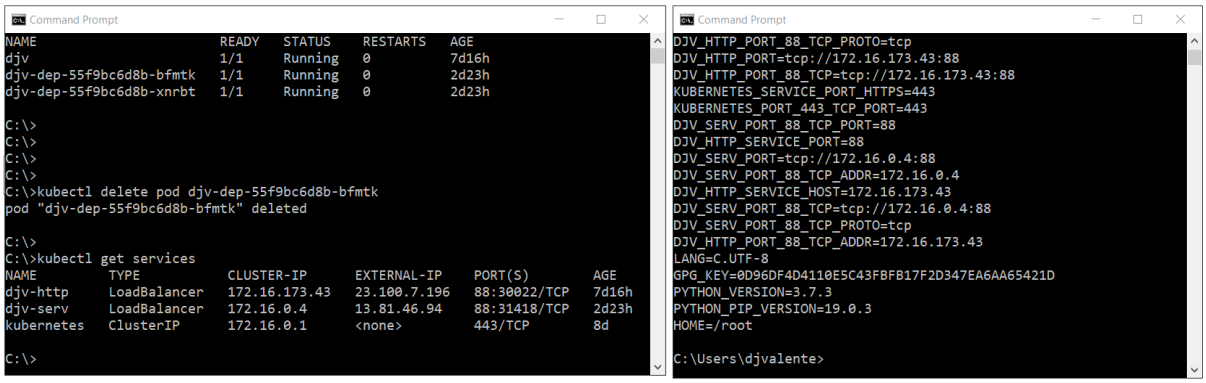


Figura 53 - Aceder a serviços através de variáveis de ambiente

De notar que ao definirmos o nome das variáveis de ambiente, além de acrescentarmos o sufixo referido anteriormente, as letras são todas capitalizadas e os hífens transformados em sublinhado.

Utilizando o serviço de DNS interno do Kubernetes

Ao listarmos os POD's que existem no *Namespace* `kube-system` rapidamente constatamos que existem POD's de sistema com a sigla "dns" no nome. Como o próprio nome sugere, estes POD's têm um servidor de DNS interno que é o servidor de DNS que todos os POD's do cluster utilizam. Este serviço de DNS conhece todos os serviços que estão a correr no cluster.

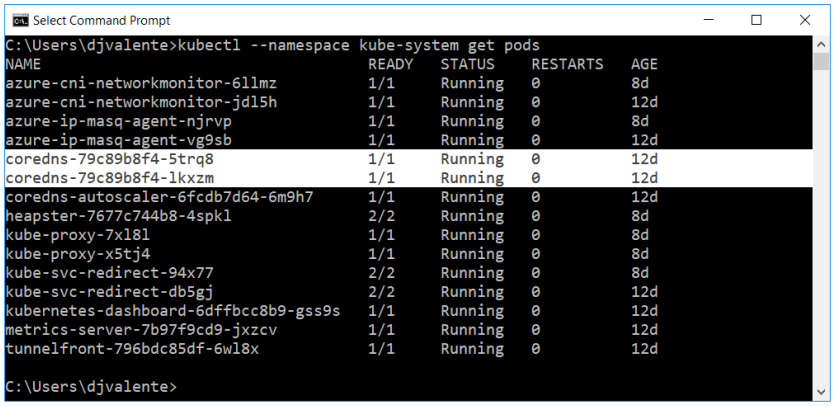


Figura 54 - Pod's do Namespace kube-system

Aceder ao Dashboard do Kubernetes em AKS

Mais uma vez verificamos que o facto de utilizarmos uma cloud pública, não nos retira qualquer funcionalidade ao cluster de Kubernetes. Apesar do próprio Azure disponibilizar funcionalidades adicionais de monitorização é possível aceder ao Dashboard nativo do próprio Kubernetes.

Para aceder ao Dashboard temos de executar o seguinte comando na *shell* do Azure:

```
az aks browse --resource-group RG_IA_Exp --name DV-K8S
```

Ao executar este comando é possível visualizar um URL que nos permite aceder diretamente ao Dashboard do nosso cluster.

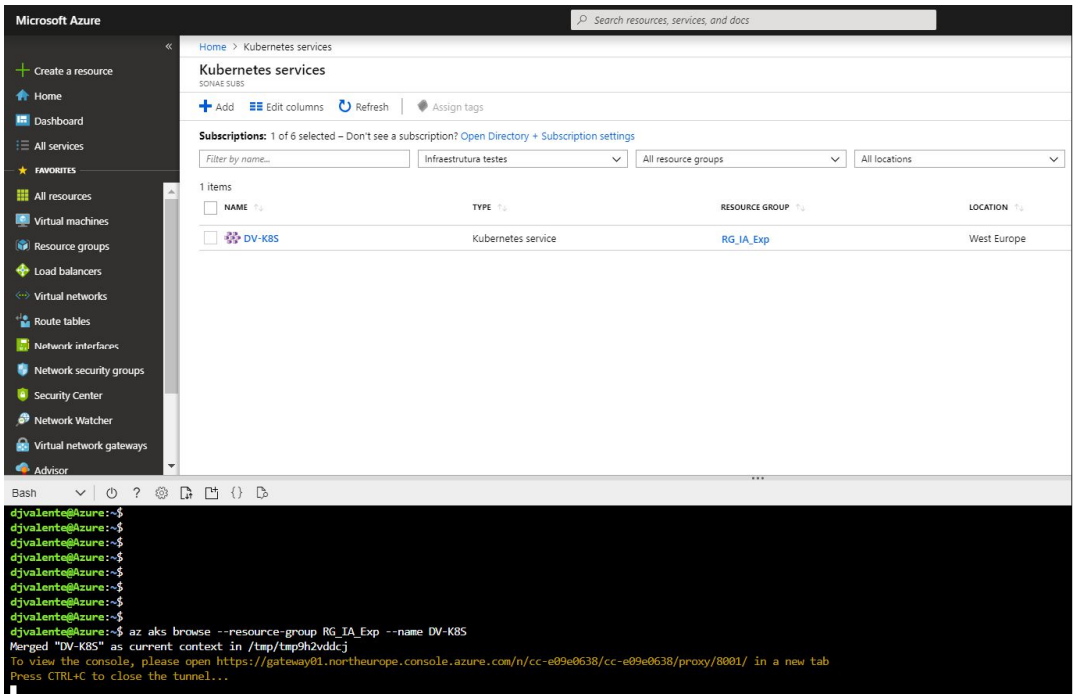


Figura 55 - Aceder ao Dashboard nativo do Kubernetes do AKS

Eventualmente podemos ter necessidade de executar um comando adicional, uma vez que se configurarmos o AKS para utilizar as funcionalidades do Azure para gestão de acessos, o POD do Dashboard não tem permissões para recolher as informações necessárias, neste caso basta executar o comando:

```
kubectl create clusterrolebinding kubernetes-dashboard --
clusterrole=cluster-admin --serviceaccount=kube-
system:kubernetes-dashboard
```

Este comando vai adicionar o role `cluster-admin` à conta do `kubernetes-dashboard`.

Armazenamento em AKS

O armazenamento, principalmente de informação que se pretende manter, é um tema que mesmo em clusters geridos, ou alojados em *Clouds* públicas deve ser cuidadosamente analisado. O facto de um POD poder simplesmente desaparecer, por variadíssimas razões, obriga a que os dados sejam armazenados fora do cluster, conforme vemos na Figura 56.

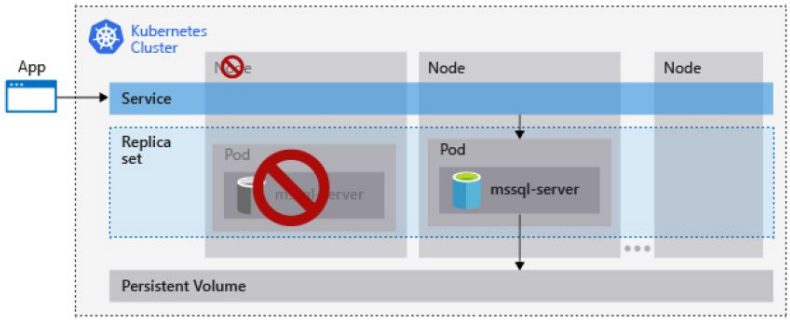


Figura 56 - Representação AKS (Node, Pod, Persistent Volume) - Retirado de doc. MSFT

Antes de conseguirmos disponibilizar armazenamento persistente aos POD's é necessário definir o que o Kubernetes denomina por *StorageClasses* [38]. Na prática este passo deverá ser efetuado pelo próprio administrador do cluster no sentido de descrever e definir os tipos de armazenamento disponíveis. Em AKS já existem dois tipos de *StorageClasses* previamente definidos, ambos utilizam *Azure Disks*, no entanto o *Default* recorre a discos do tipo HDD ao passo que o *managed-premium* utiliza discos SSD. Podemos ver na Figura 57 o comando e respetiva resposta que permite visualizar as StorageClasses existentes num cluster.

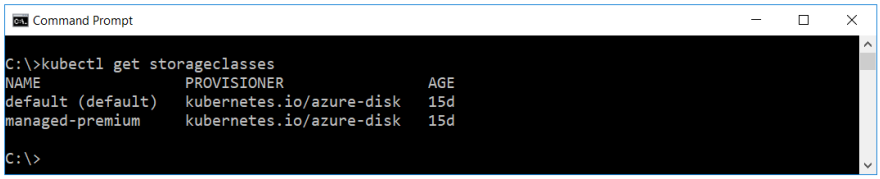


Figura 57 - AKS StorageClasses

Após definirmos as StorageClasses temos de criar um *Persistent Volume Claim* (PVC). Um PVC é utilizado para automaticamente criar um espaço de armazenamento tendo por base uma das StorageClasses existentes no Cluster. Na descrição do PVC, além de identificarmos a correspondente StorageClasse, podemos definir outros parâmetros relativamente ao volume, nomeadamente o tamanho.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azure-managed-disk
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: managed-premium
  resources:
    requests:
      storage: 5Gi
```

Figura 58 - Criação de um Persistent Volume Claim

Finalmente, associamos o PVC ao POD no momento da criação, descrevendo também o caminho pelo qual o POD pode aceder ao espaço de armazenamento.

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: nginx:1.15.5
    volumeMounts:
    - mountPath: "/mnt/azure"
      name: volume
  volumes:
    - name: volume
      persistentVolumeClaim:
        claimName: azure-managed-disk
```

Figura 59 - Associar um PVC a um POD [39]

4.1.2. Instalar um cluster localmente no centro de dados

A instalação de um cluster de Kubernetes num centro de dados local obriga a algumas considerações adicionais [40], nomeadamente, na integração com balanceadores de carga já existentes, localização física do armazenamento persistente, requisitos mínimos de hardware para cada elemento do cluster e até portas TCP que necessitam de estar disponíveis entre o(s) *Worker(s)* e o(s) *Master(s)*.

Tratando-se de uma infraestrutura que vai suportar ambientes produtivos é necessário dotar a solução de alguma resiliência, pelo que a arquitetura de alta disponibilidade recomendada [41] para o *Kubernetes* é composta por 3 *Workers* e 3 *Masters*. Tendo em conta que a existência ou não de múltiplos Master(s) não condicionam o trabalho que pretendemos desenvolver, iremos implementar um cluster com 1 Master e 3 Workers, conforme vemos na Figura 60

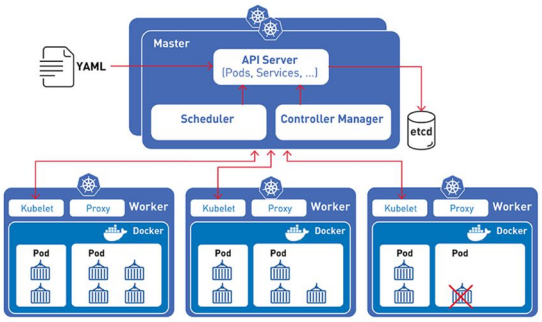


Figura 60 - Arquitetura de alta disponibilidade (retirada de [42])

Para instalarmos um cluster de múltiplos nós, o próprio Kubernetes disponibiliza uma ferramenta, denominada `kubeadm` que facilita todo o processo.

Como já vimos anteriormente um cluster de Kubernetes é composto por diversos serviços, pelo que no Master necessitamos de instalar:

- API Server – Interface que permite que elementos externos interajam com o Master. Através de pedidos HTTP e ficheiros JSON/YAML é possível alterar ou obter informação dos vários objetos.
- Scheduler – Modulo que executa tarefas, nomeadamente, criação de containers nos Workers, etc.
- Controller Manager – Monitoriza o número de replicas desejadas em cada solução no sentido de garantir sempre o estado desejado.

Num sistema altamente distribuído como é o caso de um cluster Kubernetes é necessário garantir que a informação referente à própria configuração do cluster, e sobre o estado de cada objeto, é guardado num repositório também ele distribuído, para isso utilizamos um repositório *etcd* [43].

Nas máquinas que desempenham o papel de Worker, para além do *runtime* (docker), é necessário instalar:

- Kubelet – agente que interage com o próprio docker para criar, arrancar e apagar containers.
- Kube-Proxy – responsável por rotear o tráfego para o container correto tendo por base o IP e a porta de destino do tráfego.
- Pod – grupo de containers criados num único worker.

Instalação das componentes no Master

Antes de iniciarmos a instalação propriamente dita temos de garantir alguns requisitos de conectividade entre os nós, pelo que devemos desativar o SELinux, permitir pedidos para determinadas portas TCP e garantir que as máquinas conseguem "falar" entre si através do nome standard do Kubernetes.

```
hostnamectl set-hostname 'k8s-master'
exec bash
setenforce 0
sed -i --follow-symlinks
's/SELINUX=enforcing/SELINUX=disabled/g' /etc/sysconfig/selinux

firewall-cmd --permanent --add-port=6443/tcp
firewall-cmd --permanent --add-port=2379-2380/tcp
firewall-cmd --permanent --add-port=10250/tcp
firewall-cmd --permanent --add-port=10251/tcp
firewall-cmd --permanent --add-port=10252/tcp
firewall-cmd --permanent --add-port=10255/tcp
firewall-cmd --reload
modprobe br_netfilter
echo '1' > /proc/sys/net/bridge/bridge-nf-call-iptables

nano /etc/hosts
10.126.73.141 k8s-master
10.126.58.195 worker-node1
10.126.58.196 worker-node2
10.126.58.197 worker-node3
```

Como os pacotes de instalação do Kubernetes não estão disponíveis nos repositórios por defeito do CentOS, temos de os adicionar e só depois podemos instalar o `kubeadm` e o `docker`.

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
```

```
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-
el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF

yum install kubeadm docker -y
systemctl restart docker && systemctl enable docker
systemctl restart kubelet && systemctl enable kubelet
```

Após a instalação dos pacotes necessários temos de proceder à inicialização do Master, para isso executamos o comando `kubeadm init`, que deverá concluir com a mensagem visível na Figura 61.

```
Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run (as a regular user):

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  http://kubernetes.io/docs/admin/addons/

You can now join any number of machines by running the following on each node
as root:

  kubeadm join --token a3bd48.1bc42347c3b35851
```

Figura 61 - Conclusão da instalação do Master

Na Figura 61, além da mensagem de que o Master foi inicializado com sucesso, destacamos também os comandos necessários para operar o cluster como utilizador *root,* e também o comando que teremos de utilizar nos Workers para os adicionar ao Master.

Neste ponto, já é possível verificar o estado do cluster através do comando `kubectl get nodes` e `kubectl get pods –all-namespaces`. Como podemos ver na Figura 62, o estado do cluster ainda está *NotReady*, e o estado do pod correspondente ao DNS está *Pending*.

```
[root@k8s-master ~]# kubectl get nodes
NAME         STATUS     AGE       VERSION
k8s-master   NotReady   45m       v1.7.5
[root@k8s-master ~]# kubectl  get pods  --all-namespaces
NAMESPACE     NAME                                 READY     STATUS    RESTARTS   AGE
kube-system   etcd-k8s-master                      1/1       Running   0          42m
kube-system   kube-apiserver-k8s-master            1/1       Running   0          42m
kube-system   kube-controller-manager-k8s-master   1/1       Running   0          42m
kube-system   kube-dns-2425271678-044ww            0/3       Pending   0          47m
kube-system   kube-proxy-9h259                     1/1       Running   0          47m
kube-system   kube-scheduler-k8s-master            1/1       Running   0          42m
[root@k8s-master ~]#
```

Figura 62 - Estado do cluster após o "kubeadm init"

Esta situação é ultrapassada assim que containers alojados em Workers distintos conseguirem comunicar entre si. Para isso necessitamos de adicionar um CNI (Container Network Instance) ao cluster.

Por motivos de simplificação, optamos por nesta fase utilizar o CNI Weave Net [44], posteriormente iremos descrever a integração com o CNI ACI.

```
export kubever=$(kubectl version | base64 | tr -d '\n')
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-
version=$kubever"
serviceaccount "weave-net" created
clusterrole "weave-net" created
clusterrolebinding "weave-net" created
daemonset "weave-net" created
```

Após estes comandos, o estado do master deverá ficar `Ready`, e todos os pod's deverão estar no estado `Running`.

Instalação das componentes no Worker

No Worker necessitamos de garantir os pré-requisitos iniciais. Processos idênticos aos executados no Master:

- Desativar o SELinux
- Adicionar os pacotes do Kubernetes aos repositórios
- Instalar o *kubeadm* e o *docker*, garantindo que ficam operacionais
- Garantir conectividade por nome

Finalmente temos de executar o comando `kubeadm join`, utilizando o *token* e o certificado apresentado pelo Master no final da inicialização.

```
kubeadm join 10.126.73.141:6443 --token fo13k2.r7fczvn177kjp7yz
--discovery-token-ca-cert-hash
sha256:eae855544c648042e00a652efec95cd99304531c1cbc3f275651891c
b6e7b0c0
```

A Figura 63 monstra a mensagem que deverá surgir assim que terminar o processo de adição do Worker ao Master.

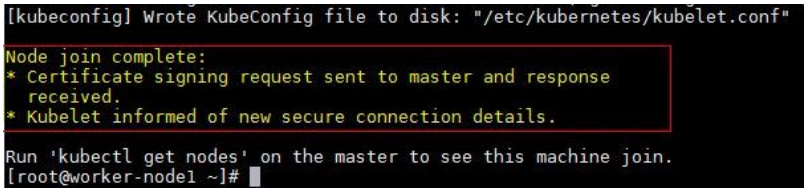


Figura 63 - Confirmação do Worker

Integração com balanceador de carga existente (F5 BIG-IP)

A função do Kubernetes, como já vimos ao descrever a sua utilização na Cloud Azure, não se limita à criação/gestão dos próprios containers, é suposto que orquestre também serviços adicionais como é o caso dos balanceadores de carga, criação de espaços de armazenamento, ou controlo de acessos entre pod's.

Ao contrário do que acontece nas Clouds públicas, onde todas as integrações estão asseguradas previamente, num cenário proprietário, como é o caso de uma instalação num centro de dados privado, é necessário garantir essa integração de modo a que um *deployment* que necessite de um balanceador consiga essa funcionalidade, proporcionando exatamente a mesma experiência ao utilizador do Cluster.

Modos de integração com o balanceador F5 BIG-IP

O fabricante F5 disponibiliza dois modos de integração com clusters Kubernetes, sendo que a principal diferença está precisamente na componente de rede.

Modo Nodeport

Este é o modo por defeito para uma integração entre o BIG-IP e Kubernetes. Do ponto de vista das configurações necessárias é o mais simples, uma vez que não está dependente do CNI que o cluster utiliza. Alem disso, este modo não obriga a licenciamento específico do BIG-IP.

Este modo, como podemos ver na Figura 64, efetua 2 balanceamentos de carga:

1. O próprio BIG-IP balanceia o pedido para os nós (*kube-proxy*)
2. Os nós (*kube-proxy*) balanceiam para os POD's

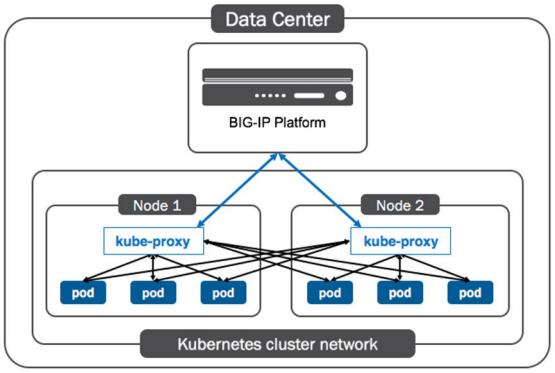


Figura 64 - Integração BIG-IP com K8S modo Nodeport (retirado de [45])

Esta integração apresenta algumas limitações, por um lado, temos obrigatoriamente de utilizar *services* do tipo *Nodeport*, isto é, uma vez que estamos a utilizar um CNI que baseia a disponibilização dos POD's num mecanismo de *port-forwarding*, o BIG-IP vai balancear o tráfego para os endereços dos próprios nós para a porta TCP onde o POD respetivo se encontra à escuta. O facto do BIG-IP não conseguir balancear o tráfego diretamente para os POD's, faz com que funcionalidades como a

persistência baseada na camada 7 não tenha o comportamento esperado. É também evidente que este mecanismo adiciona latência e reduz a visibilidade que o BIG-IP tem sobre o estado dos POD's.

Modo Cluster

A utilização do modo Cluster pressupõe que pretendemos integrar o BIG-IP diretamente com a rede interna do Kubernetes. Este modo elimina todas as desvantagens apontadas no modelo anterior, no entanto apresenta uma maior complexidade e obriga a licenciamento específico, uma vez que vamos necessitar de funcionalidades SDN e/ou roteamento avançado. A Figura 65 ilustra a arquitetura do modo *Cluster*.

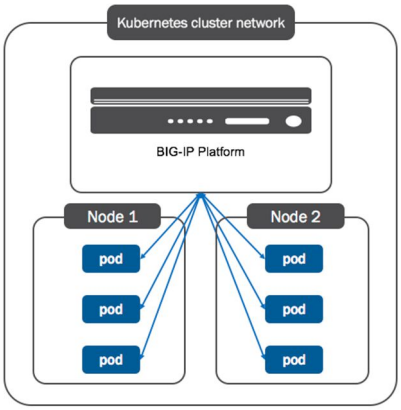


Figura 65 - Integração BIG-IP com K8S modo Cluster (retirado de [45])

A integração do BIG-IP na rede do Kubernetes pode ser conseguida através de uma arquitetura baseada em *overlay's* (VXLAN, NVGRE ou IPIP), ou através de roteamento tradicional. Na realidade os POD's recebem endereços de redes IP distintas da rede onde os nós físicos se encontram. Neste sentido temos duas opções, ou criamos um *overlay* entre o BIG-IP e os nós, ou utilizamos BGP como protocolo de roteamento dinâmico, no sentido de permitir conectividade direta entre o BIG-IP e os POD's. Para cada uma das opções apresentadas a F5 sugere um CNI específico, na abordagem baseada em *Overlays* é recomendado o CNI *flannel* que utiliza VXLAN, numa abordagem camada 3 com BGP, é recomendado o CNI *Calico.*

Configurações para a integração no modo Nodeport

Um dos requisitos iniciais de integração é o facto de não ser possível utilizar a partição */Common* do BIG-IP, o que obriga à criação de uma partição especifica para o Kubernetes. Como um dos principais objetivos é mesmo a criação de novos objetos, é necessário adicionar ao Kubernetes credenciais de acesso ao balanceador. Para isso criamos um *secret* com as respetivas credenciais.

```
kubectl create secret generic bigip-login --namespace kube-
system --from-literal=username=admin --from-literal=password=
admin
```

No Kubernetes é necessário criar uma conta de serviço para o balanceador, através do comando `kubectl create serviceaccount bigip-ctlr -n kube-system`. Para associar a esta conta os privilégios necessários temos de criar um *ClusterRole* e depois através de um *ClusterRoleBinding* associar a conta de serviço a este perfil. Estes passos são conseguidos através da aplicação de uma política definido num ficheiro YAML. Na Figura 66 podemos ver o ficheiro YAML utilizado para criar os objetos mencionados, e o comando para aplicar a politica é `kubectl create -f dv-rbac.yaml`.

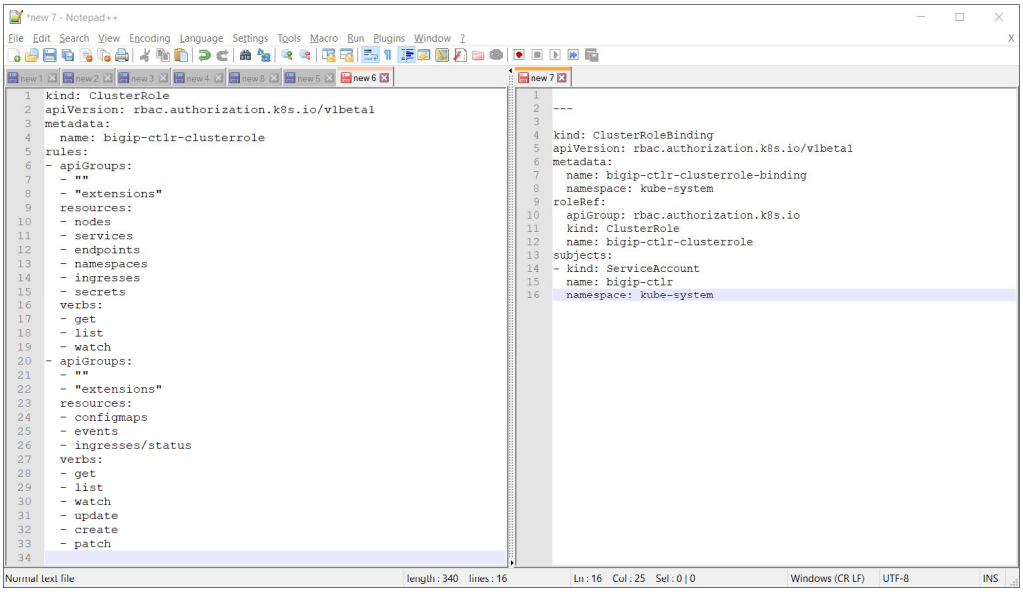


Figura 66 - Código para a criação do ClusterRole e do ClusterRoleBinding

Finalmente, é necessário proceder à instalação do *BIG-IP Controller* propriamente dito. Para isso criamos novamente um ficheiro YAML que ao ser aplicado no Kubernetes cria um POD com o controlador e parametriza as configurações do BIG-IP pretendido. A Figura 67 mostra-nos um exemplo de um ficheiro YAML de criação do controlador.

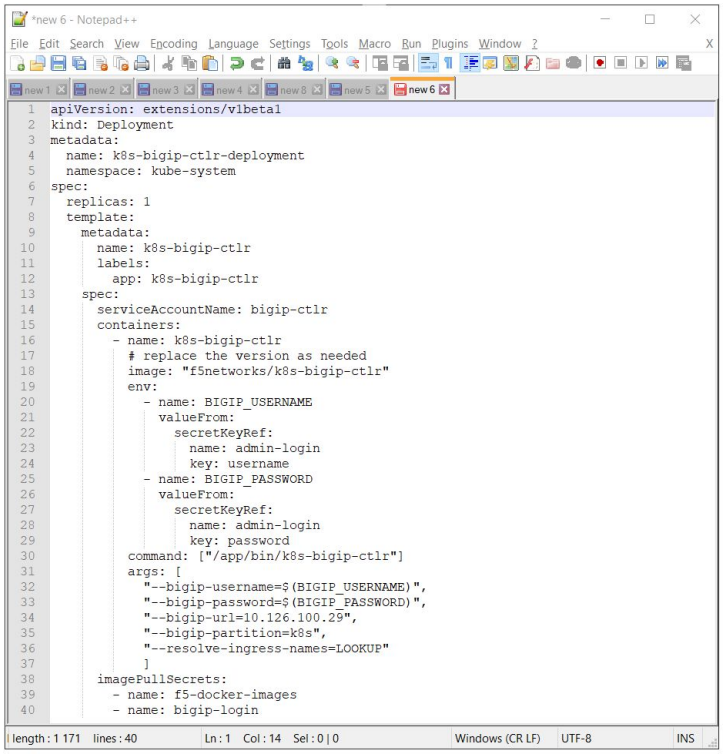


Figura 67 - Código para a criação do BIG-IP Controller

Podemos verificar o correto funcionamento da integração através da criação de um ambiente de teste disponibilizado pela própria F5. Este ambiente é composto por 3 ficheiros YAML:

- `my-frontend-deployment.yaml` – descreve os containers que compõem o ambiente. Como podemos ver na Figura 68 vamos ter duas replicas do container `chen/f5-demo-app` que está disponível no *DockerHub*. O container em questão disponibiliza o serviço na TCP 80.

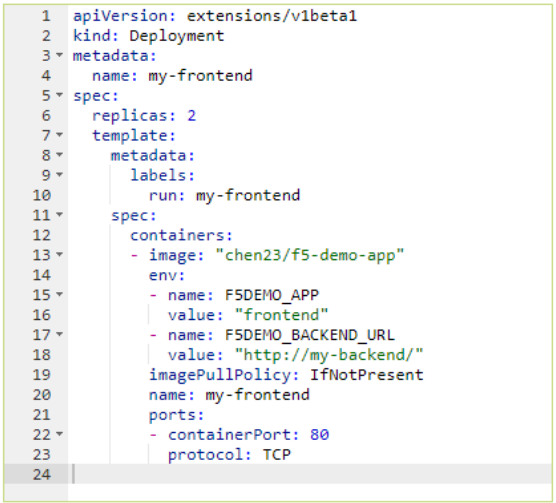


Figura 68 - Ambiente de teste da F5 – deployment

- `my-frontend-service.yaml` – que descreve a públicação do serviço (*port-forwarding)*, permitindo o acesso do exterior do cluster aos POD's.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-frontend
  labels:
    run: my-frontend
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  type: NodePort
  selector:
    run: my-frontend
```

Figura 69 - Ambiente de teste da F5 - service

- `my-frontend-configmap.yaml` – a utilização de *ConfigMaps* permite manter os detalhes específicos da implementação separados do próprio container. Desta forma conseguimos manter os containers agnósticos à configuração ou ao local onde o cluster está a ser executado. Neste caso usamos *ConfigMaps* para descrever o que é necessário criar no BIG-IP. Como vemos na Figura 70 é solicitada a criação de um `virtual-server` (10.126.97.10) que vai receber tráfego na porta 80 e vai balancear, utilizando o método *round-robin*, para os membros do cluster.

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: my-frontend
  namespace: default
  labels:
    f5type: virtual-server
data:
  schema: "f5schemadb://bigip-virtual-server_v0.1.7.json"
  data: |-
    {
      "virtualServer": {
        "frontend": {
          "balance": "round-robin",
          "mode": "http",
          "partition": "k8s",
          "virtualAddress": {
            "bindAddr": "10.126.97.10",
            "port": 80
          }
        },
        "backend": {
          "serviceName": "my-frontend",
          "servicePort": 80
        }
      }
    }
```

Figura 70 - Ambiente de teste da F5 – Configmap

É importante salientar que em nenhum ficheiro YAML é efetuada referência aos endereços dos nós do cluster, ou ao porto TCP em que os containers foram disponibilizados, no entanto ao consultar os objetos efetivamente criados (Figura 71), constatamos que esse mapeamento foi efetuado automaticamente pelo orquestrador.

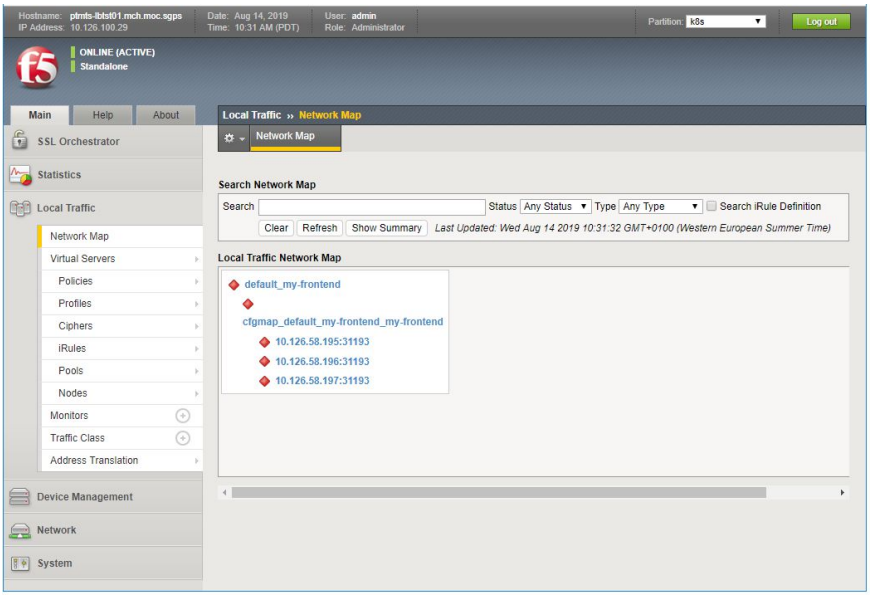


Figura 71 - Ambiente de teste da F5 – Confirmação do LB

Configurações para a integração no modo Cluster

Numa abordagem *Cluster-mode* o balanceador de carga F5 vai ser integrado diretamente na rede do cluster de Kubernetes. Esta integração vai permitir acesso direto aos POD's. Esta abordagem, conforme já referimos, não vai ter qualquer limitação referente ao tipo de serviço utilizado na criação dos POD's, e a visibilidade vai ser idêntica à já existente em ambientes apenas com máquinas virtuais.

Considerando as opções existentes para este tipo de integração, optamos pela utilização do CNI *Flannel* [46]. Esta escolha foi baseada no facto da organização onde estamos a desenvolver este estudo não pretender utilizar o protocolo de roteamento BGP no interior do datacenter, o que nos obriga a descartar o CNI *Calico*. Relativamente ao tipo de encapsulamento, optamos por VXLAN pela razão que também é suportado pela solução ACI da Cisco, com que pretendemos integrar no futuro.

Para conseguirmos utilizar corretamente o CNI *Flannel* é necessário fornecer, no momento da instalação do Master um parâmetro adicional referente ao endereçamento. Na prática não conseguimos alterar o CNI de um cluster, pelo que somos obrigados a criar um novo para testar o *Flannel*. Neste sentido o comando de criação do Master terá de ser `kubeadm init --pod-network-cidr=10.244.0.0/16`.

Para instalar o CNI *Flannel* sem qualquer alteração às configurações pré-definidas, executamos o comando abaixo no Master.

```
kubectl apply -f \
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

Ao analisar o ficheiro YAML utilizado, rapidamente verificamos as definições de rede que vão ser parametrizadas, nomeadamente o tipo de encapsulamento e o endereçamento IP que vai ser utilizado pelos POD's, como vemos na Figura 72.

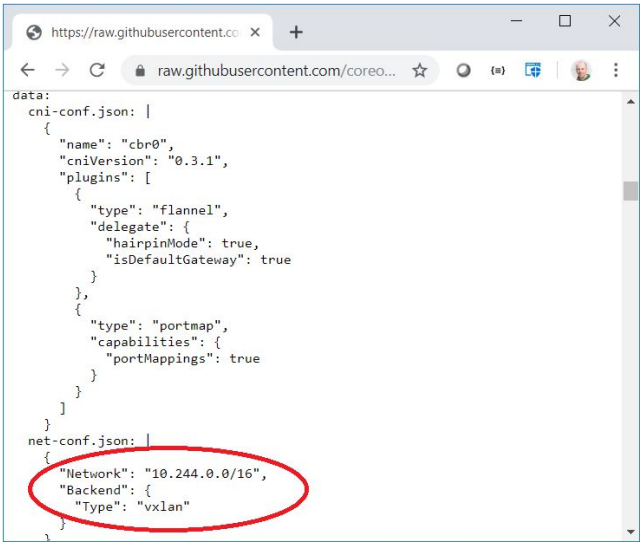


Figura 72 - Configuração do CNI Flannel

O *Flannel*, quando utilizado como CNI de um cluster de Kubernetes, cria um pod em cada nó do cluster. Este POD contém um *daemon* que é responsável pela recolha e disponibilização de informação de rede dos próprios nós do cluster. Relativamente a endereçamento IP, o *Flannel*, atribui uma sub-rede IP da rede que definimos no momento da sua instalação para cada nó. É destas sub-redes que são atribuídos endereços aos pod's. A integração em modo *cluster* obriga a que o próprio F5 seja visto pelo *cluster* como um nó adicional, desta forma, através do *Flannel* e do controlador específico da F5 vai ser possível balancear tráfego diretamente para os pod's.

Este controlador adiciona ao túnel VXLAN que liga o balanceador ao *cluster*, informação que mapeia o endereço IP de cada nó do cluster com o endereço MAC da interface *Flannel* correspondente. Vai ser também responsável por adicionar registos ARP estáticos, mapeando o endereço MAC do *Flannel* com o endereço IP dos pod's criados no cluster. Analisando o exemplo da Figura 73, podemos concluir que o balanceador vai enviar tráfego para o endereço 10.244.1.2, que na prática vai ser encaminhado para o nó 1 do *cluster*, que por sua vez vai entregar no pod alojado nesse mesmo nó.

Kubernetes Node1	
Node IP address	172.16.2.10
MAC address of Node's flannel VXLAN interface	98:ba:76:dc:54:fe
Pod public-ip address assigned by flannel	10.244.1.2

FDB record

```
flannel_vxlan {
 records [
    98:ba:76:dc:54:fe {
      endpoint: 172.16.2.10
    }
 ]
}
```

static ARP entry

```
{
   name: k8s-10.244.1.2
   ipaddress: 10.244.1.2
   macaddress: 98:ba:76:dc:54:fe
}
```

Figura 73 - Mapeamentos no túnel VXLAN (adaptado de [47])

Relativamente ao processo de configuração propriamente dito, este divide-se nas seguintes componentes:

- Configurar o túnel VXLAN no balanceador
- Adicionar o balanceador ao próprio cluster como se de um nó se tratasse
- Instalar o controlador do balanceador no cluster de Kubernetes

A configuração do túnel VXLAN no F5 passa pela criação do túnel propriamente dita e pela criação de uma nova interface com um endereço IP da rede definida na instalação do *Flannel.* Na Figura 74 vemos os comandos necessários à configuração descrita anteriormente.

```
1
2  # Criar um VTEP
3  create /net tunnels tunnel flannel_vxlan key 1 profile fl-vxlan local-address 172.16.30.3
4
5  # criar um self IP do endereçamento interno dos POD
6  create /net self 10.244.255.1/16 allow-service none vlan flannel_vxlan
7
8  # criar um floating IP nesta mesma rede
9  create /net self 10.244.255.2/16 allow-service none traffic-group traffic-group-1 vlan flannel_vxlan
10
```

Figura 74 - Comandos do F5 para a criação do túnel VXLAN

Para adicionar o próprio F5 como um nó "falso" do cluster necessitamos de conhecer o endereço físico (endereço MAC) do túnel VXLAN criado anteriormente. Esta informação é obtida através do comando `show net tunnels tunnel flannel_vxlan all-properties`. Para a criação propriamente dita devemos utilizar um ficheiro YAML, disponibilizado pela F5 [47], efetuando obviamente as devidas alterações referentes ao endereçamento físico, lógico e *Flannel Annotations*. Depois de criar o novo nó, através do comando `kubectl create -f f5-kctlr-bigip-node.yaml`, conseguimos verificar que existe verdadeiramente um novo nó no cluster. O status deste nó será sempre *NotReady, como vemos na* Figura 75*,* uma vez que na prática não se trata de um nó real.

```
[danielv@k8s-master ~]$ kubectl get nodes
NAME           STATUS     ROLES    AGE    VERSION
bigip          NotReady   <none>   13d
k8s-master     Ready      master   13d    v1.16.3
worker-node1   Ready      <none>   13d    v1.16.3
worker-node2   Ready      <none>   13d    v1.16.3
worker-node3   Ready      <none>   13d    v1.16.3
[danielv@k8s-master ~]$
```

Figura 75 - Status do nó F5 no cluster

Finalmente, é necessário instalar o controlador do balanceador no cluster. Este processo é em tudo idêntico ao processo efetuado na integração em modo *Nodeport*, nomeadamente, criar os objetos que permitirão o acesso ao balanceador (*service account, ClusterRole, ClusterRole Binding*) e criar um *Deployment* com base no *template* também disponível no site da F5 [47].

Este modo de integração recorre a recursos do F5 bastante mais flexíveis do que o modo *Nodeport*, isto é, no próprio ficheiro YAML conseguimos definir toda a política de *ingress* do balanceamento, como vemos na Figura 76. Na Figura 77 e na Figura 78 vemos os objetos e as respetivas configurações que o controlador cria automaticamente no balanceador.

```
1  ---
2  apiVersion: extensions/v1beta1
3  kind: Ingress
4  metadata:
5    name: ghost-f5
6    annotations:
7      # See the k8s-bigip-ctlr documentation for information about
8      # all Ingress Annotations
9      # https://clouddocs.f5.com/products/connectors/k8s-bigip-ctlr/latest/#supported-ingress-annotations
10     virtual-server.f5.com/ip: "10.126.99.245"
11     virtual-server.f5.com/http-port: "80"
12     virtual-server.f5.com/partition: "k8s-controller"
13     virtual-server.f5.com/balance: "least-connections-node"
14     kubernetes.io/ingress.class: "f5"
15     # Annotations below are optional
16     #  virtual-server.f5.com/balance:
17     #  virtual-server.f5.com/http-port:
18     #  virtual-server.f5.com/https-port:
19     #  ingress.kubernetes.io/allow-http:
20     #  ingress.kubernetes.io/ssl-redirect:
21     virtual-server.f5.com/health: |
22       [
23         {
24           "path":     "ghost-f5.valente.pt/",
25           "send":     "GET /\\r\\n",
26           "interval": 5,
27           "timeout":  10
28         }
29       ]
30 spec:
31   rules:
32   - host: ghost-f5.valente.pt
33     http:
34       paths:
35       - path: /
36         backend:
37           serviceName: ghost-f5
38           servicePort: 80
39 [danielv@k8s-master ~]$
```

Figura 76 - Política de balanceamento no YAML

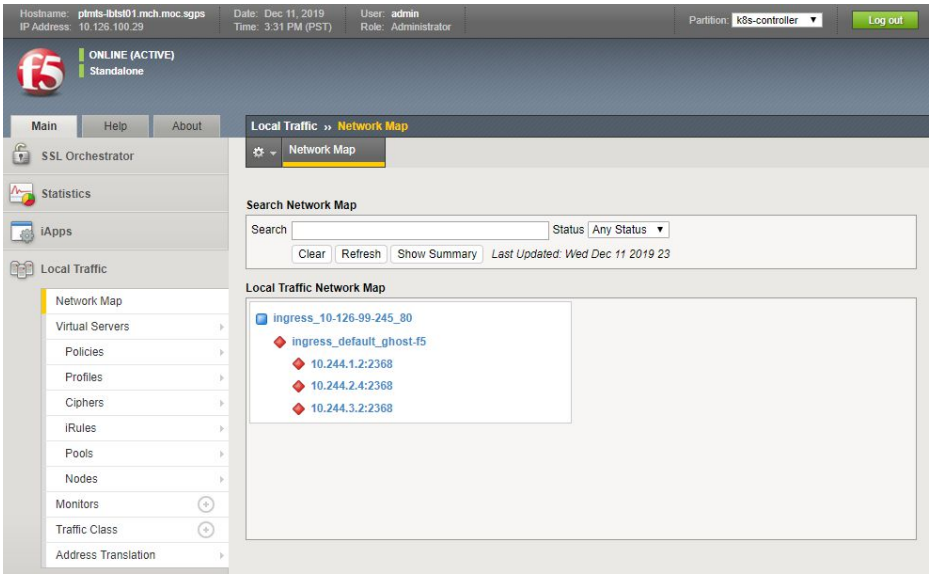


Figura 77 - Balanceamento com integração ClusterMode

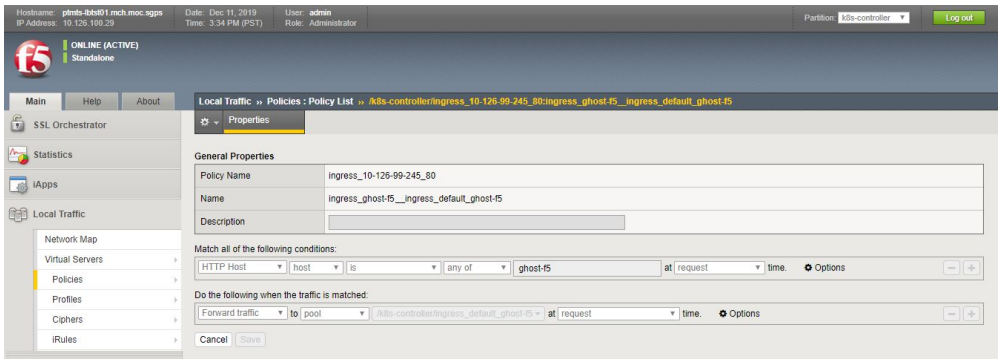


Figura 78 - Política de balanceamento em Clustermode

Disponibilização de armazenamento persistente

Conforme já referimos anteriormente o facto de os containers serem efémeros, lança um desafio relativamente ao armazenamento, isto é, um container que "desaparece" é automaticamente substituído por um novo, mas com o sistema de ficheiros no estado inicial, o que impossibilita a utilização do seu sistema de ficheiros para armazenamento permanente, como por exemplo bases de dados ou logs, mas também para partilha de informação com outros containers.

Para endereçar estes desafios o Kubernetes implementa um mecanismo de abstração ao nível do armazenamento que designamos por *Volumes* [48].

Existe uma grande variedade de tipos de *volumes* que mapeiam diretamente com protocolos ou tipos de armazenamento suportados pelo Kubernetes, os que consideramos mais relevantes para o ambiente atual são:

- awsElasticBlockStore
- azureDisk
- azureFile
- cephfs

- fc (fibre channel)
- gcePersistentDisk
- hostPath
- iscsi
- local
- nfs
- vsphereVolume

Tendo em conta a pretensão de um ambiente multicloud, será necessário considerar as opções existentes nas principais Clouds públicas (AWS, Azure e GCP). Eventualmente poderá ser necessário suportar múltiplas formas de armazenamento no centro de dados (FC, iSCSI, CEPH, NFS ou vSphere) e finalmente em determinadas situações poderá ser necessário utilizar armazenamento nos próprios nós do cluster (Local ou HostPath).

É relevante salientar a ausência do protocolo SMB, protocolo que permite a partilha de ficheiros em ambientes Microsoft. Esta ausência não significa que um cluster Kubernetes não consiga utilizar um servidor Windows Server como fonte de armazenamento permanente, visto que desde a versão 2016 o Windows Server já suporta o protocolo NFS. Existe também um projeto [49], baseado em *Flexvolume*, que pretende desenvolver uma integração direta entre o Kubernetes e partilhas em SMB/CIFS, mas ainda não é uma funcionalidade nativa da atual versão do Kubernetes.

Uma vez que já descrevemos a utilização em ambiente Cloud, iremos descrever a configuração de armazenamento no centro de dados usando NFS, concluindo os pré-requisitos para uma arquitetura híbrida.

Configurações para a utilização de NFS para armazenamento permanente

Tendo em conta que já descrevemos anteriormente os principais conceitos relacionados com armazenamento persistente, como é o caso dos *Persistent Volumes* e *Persistent Volumes Claims*, iremos apenas apresentar um exemplo de como utilizar estes objetos num cenário num centro de dados local, utilizando o protocolo NFS.

O primeiro passo é precisamente a definição do *Persistent Volume*, que como podemos ver na Figura 79, é aqui que definimos o endereço do servidor, o caminho para a partilha de rede, o tipo de armazenamento e a política de retenção.

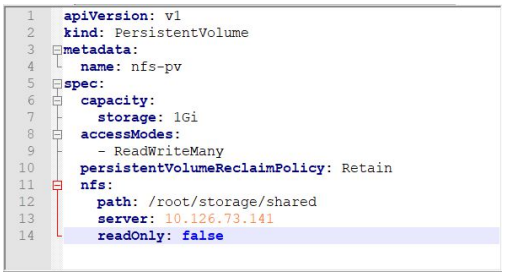


Figura 79 - Criação de Persistent Volume para NFS

O segundo passo, à semelhança do que descrevemos anteriormente, é a definição do *Persistent Volume Claim*, que definirá o tipo de acesso e a capacidade que o respetivo projeto necessita. Tendo em conta os atributos definidos e o cenário em causa, a associação entre o *Persistent Volume* criado e este *Persistent Volume Claim* será automática. Esta associação tem sempre um mapeamento 1:1, isto é, mais nenhum PVC poderá ficar associado a este PV, no entanto os vários POD's existentes neste projeto poderão aceder a este PVC, permitindo a partilha de informação entre POD's. Na Figura 80, podemos ver um exemplo do ficheiro YAML que permitiria a criação do PVC.

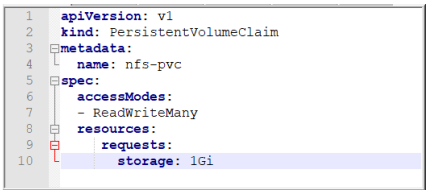


Figura 80 - Criação de Persistent Volume Claim para NFS

Os comandos que permitiriam a utilização dos ficheiros YAML descritos acima para a criação propriamente dita dos objetos seriam, `kubectl create -f nfs-pv.yaml` e `kubectl create -f nfs-pvc.yaml`.

Após termos o PV e o PVC criados no cluster de Kubernetes, é apenas necessário que o POD que pretende usar o PVC o referencie no momento da sua criação. Como exemplo adicionamos um volume ao POD `djv1` já utilizado anteriormente, como podemos confirmar na Figura 81.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: djv1
  labels:
    name: djv1
    app: teste
spec:
  containers:
    -
      name: djv1-containers
      image: djvalente/testes
      ports:
      -
        name: web
        containerPort: 88
      volumeMounts:
      -
        name: nfsvol
        mountPath: /usr/share/nginx/html
  volumes:
    - name: nfsvol
      persistentVolumeClaim:
        claimName: nfs-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: djv1-service
spec:
  type: NodePort
  ports:
    - port: 88
  selector:
    name: djv1
```

Figura 81 - Criação de POD com volume NFS

Como podemos verificar na Figura 82, ao listarmos o conteúdo do diretório partilhado no servidor (`10.126.73.141: /root/storage/shared`) com o caminho definido para o volume na criação do POD (`/usr/share/nginx/html`), constatamos que se trata precisamente do mesmo.

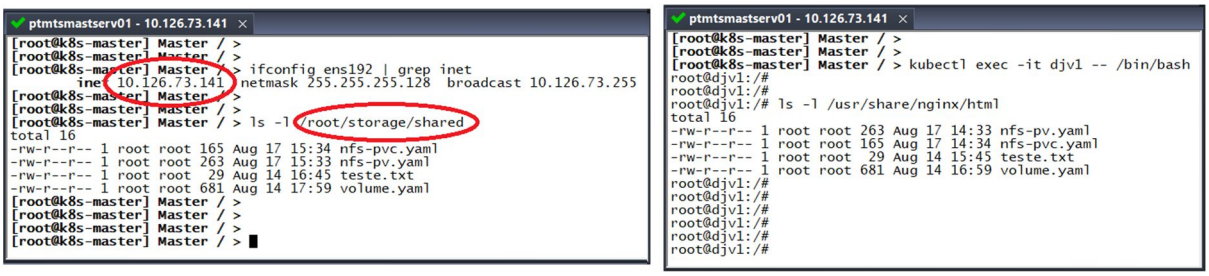


Figura 82 - Verificação da criação do volume persistente

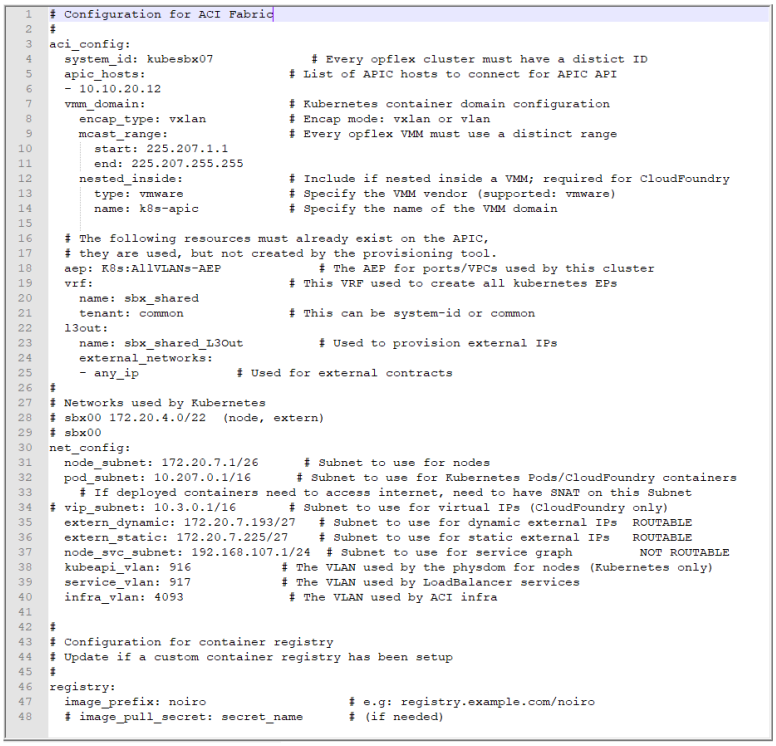
4.2. Implementação da integração do ACI com Kubernetes

O primeiro passo é precisamente preparar a solução ACI para integrar com plataformas externas de containers, para isso, a Cisco desenvolveu uma ferramenta denominada `acc-provision` que assegurará esta componente. Esta ferramenta está disponível no próprio site da Cisco (www.cisco.com) ou se preferirmos podemos instala-la através do `pip`.

Esta ferramenta, além de preparar a integração, cria também as definições aplicacionais que posteriormente aplicaremos ao Kubernetes para instalar o CNI.

4.2.1. Preparação do ACI (*acc-provision*)

Após instalarmos a ferramenta através do comando `pip install acc-provision`, executamo-la fornecendo como parâmetro um ficheiro YAML que descreve os detalhes necessários à integração. Na Figura 83 vemos o conteúdo do ficheiro que utilizamos.



```
# Configuration for ACI Fabric
#
aci_config:
  system_id: kubesbx07             # Every opflex cluster must have a distict ID
  apic_hosts:                    # List of APIC hosts to connect for APIC API
  - 10.10.20.12
  vmm_domain:                    # Kubernetes container domain configuration
    encap_type: vxlan            # Encap mode: vxlan or vlan
    mcast_range:                 # Every opflex VMM must use a distinct range
      start: 225.207.1.1
      end: 225.207.255.255
    nested_inside:               # Include if nested inside a VMM; required for CloudFoundry
      type: vmware               # Specify the VMM vendor (supported: vmware)
      name: k8s-apic             # Specify the name of the VMM domain

  # The following resources must already exist on the APIC,
  # they are used, but not created by the provisioning tool.
  aep: K8s:AllVLANs-AEP              # The AEP for ports/VPCs used by this cluster
  vrf:                           # This VRF used to create all kubernetes EPs
    name: sbx_shared
    tenant: common               # This can be system-id or common
  l3out:
    name: sbx_shared_L3Out           # Used to provision external IPs
    external_networks:
    - any_ip              # Used for external contracts
#
# Networks used by Kubernetes
# sbx00 172.20.4.0/22  (node, extern)
# sbx00
net_config:
  node_subnet: 172.20.7.1/26      # Subnet to use for nodes
  pod_subnet: 10.207.0.1/16      # Subnet to use for Kubernetes Pods/CloudFoundry containers
    # If deployed containers need to access internet, need to have SNAT on this Subnet
# vip_subnet: 10.3.0.1/16       # Subnet to use for virtual IPs (CloudFoundry only)
  extern_dynamic: 172.20.7.193/27   # Subnet to use for dynamic external IPs  ROUTABLE
  extern_static: 172.20.7.225/27    # Subnet to use for static external IPs   ROUTABLE
  node_svc_subnet: 192.168.107.1/24  # Subnet to use for service graph         NOT ROUTABLE
  kubeapi_vlan: 916             # The VLAN used by the physdom for nodes (Kubernetes only)
  service_vlan: 917             # The VLAN used by LoadBalancer services
  infra_vlan: 4093               # The VLAN used by ACI infra

#
# Configuration for container registry
# Update if a custom container registry has been setup
#
registry:
  image_prefix: noiro                 # e.g: registry.example.com/noiro
  # image_pull_secret: secret_name      # (if needed)
```

Figura 83 - Ficheiro de preparação do ACI

Consideramos relevante, para uma melhor compreensão da integração, descrever algumas parametrizações descritas no ficheiro YAML.

- **system_id**: Identificador único para o cluster Kubernetes em questão. Este nome vai ser utilizado para criar um novo *Tenant* em ACI e um novo domínio no respetivo VMM [50] (*Virtual Machine Manager*) para o Kubernetes.
- **apic_hosts**: Endereços dos controladores APIC
- **vmm_domain > nested_inside**: Se os nós do Kubernetes forem máquinas virtuais e não nós físicos, neste parâmetro indicamos o domínio VMM em que as próprias máquinas virtuais se encontram.
- **aep**: Indicamos o *Access Entity Profile* [50] que vai ser usado para ligar ao cluster.
- **vrf**: VRF que vamos utilizar para o cluster. É possível utilizar um VRF que já esteja a ser utilizado num outro cluster.
- **l3out**: No ACI o objeto l3out representa endereçamento externo à solução, neste caso este l3out será utilizado para permitir a comunicação das redes do ACI com as redes internas do cluster.
- **net_config > node_subnet**: Endereçamento IP que vai ser utilizado para as interfaces de gestão do Kubernetes. É nesta rede que vai estar disponível a API de configuração do cluster.

- **net_config > pod_subnet**: Endereçamento IP que vai ser utilizado pelos próprios pods/containers dentro do Kubernetes
- **net_config > extern_dynamic**: Endereçamento IP que vai ser utilizado quando o acesso aos pods for possível através de *load balancers*. É utilizada por serviços configurados para utilizar endereços dinâmicos
- **net_config > extern_static**: Semelhante ao ponto anterior, no entanto este endereçamento é utilizado quando é necessário endereçamento estático.
- **net_config > node_svc_subnet**: Endereçamento IP que é utilizado como parte da rede de overlay dos serviços do Kubernetes e dos *service graphs* que foram criado dinamicamente pelo CNI.
- **net_config > kubeapi_vlan**: ID da VLAN que é utilizada para identificar o tráfego para a node_subnet. Este tráfego passa entre o ACI e os nós do Kubernetes usando uma interface configurado com 802.1Q em cada nó.
- **net_config > service_vlan**: ID da VLAN que é utilizada para identificar o tráfego da rede de overlay para os balanceadores de carga do Kubernetes. Este tráfego passa entre o ACI e os nós do Kubernetes usando uma interface configurado com 802.1Q em cada nó.
- **net_config > infra_vlan**: ID da VLAN que usaremos para no ACI para as comunicações do underlay. O CNI baseia a sua operação numa extensão do próprio fabric do ACI até ao cluster de Kubernetes assumindo a gestão do switch OVS dentro dos servidores Linux. Este tráfego passa entre o ACI e os nós do Kubernetes usando uma interface configurado com 802.1Q em cada nó.

O seguinte comando prepara a infraestrutura conforme definido no ficheiro YAML `acc-provision -c aci-containers-config.yaml -o aci-containers.yaml -f kubernetes-<version> -a -u [apic username] -p [apic password].`

Este comando além de preparar o ACI para a integração, disponibiliza como output (`-o aci-containers.yaml`) um ficheiro YAML que servirá de input para a parametrização do CNI no próprio cluster de Kubernetes.

Após a execução do comando podemos observar no APIC os objetos criados, nomeadamente:

- **Bridge Domains** - um denominado `kube-node-bd` e outro denominado `kube-pod-bd`, as sub-redes associadas são precisamente as que definimos no ficheiro YAML (`node_subnet` e `pod_subnet`).

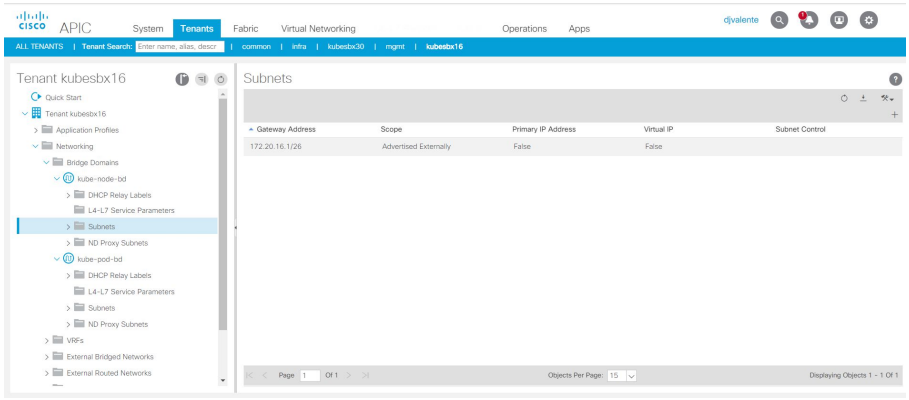


Figura 84 - Verificação no APIC dos Bridge Domains

- **Contracts (Filters)** – Criados para definir as políticas que vão permitir a comunicação das funcionalidades base do Kubernetes.

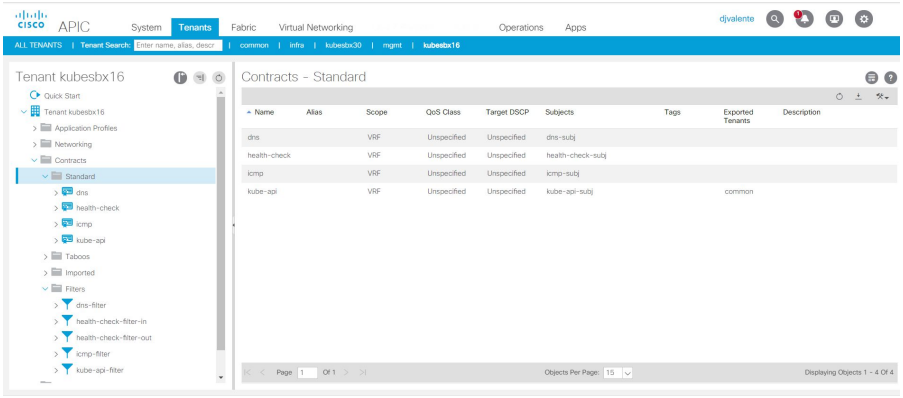


Figura 85 - Verificação no APIC dos Contracts

- **Application Profile** – Criou um Application Profile denominado Kubernetes que contém três EPG (*End Point Group*) já com os respetivos contratos associados.
    - `kube-default` é o EPG onde os PODs criados no Kubernetes sem uma política explicitamente definida serão colocados.
    - `kube-nodes` é o EPG que permitirá o acesso do ACI aos nós de gestão do Kubernetes.
    - `kube-system` é o EPG onde serão colocados os POD's que o próprio Kubernetes cria para as suas funções internas, como por exemplo o `kube-dns`.

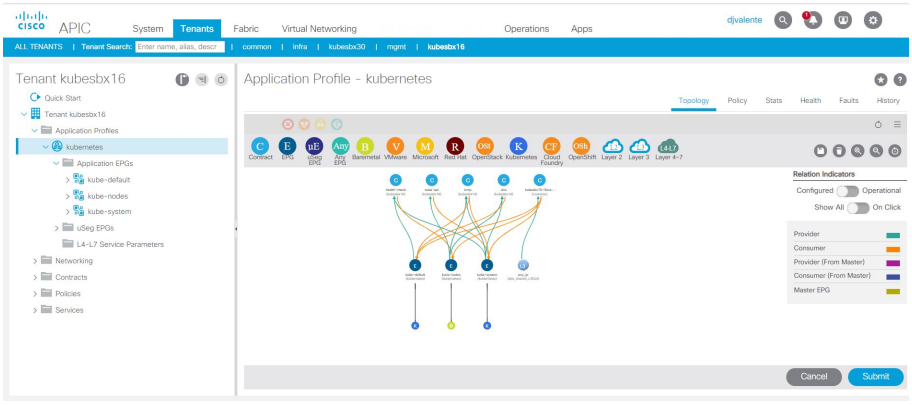


Figura 86 - Verificação no APIC do Application Profile criado

- **Container Domain** – Assim que o Kubernetes estiver operacional e o CNI instalado, este **domínio** é a forma de comunicação/integração com o ACI.

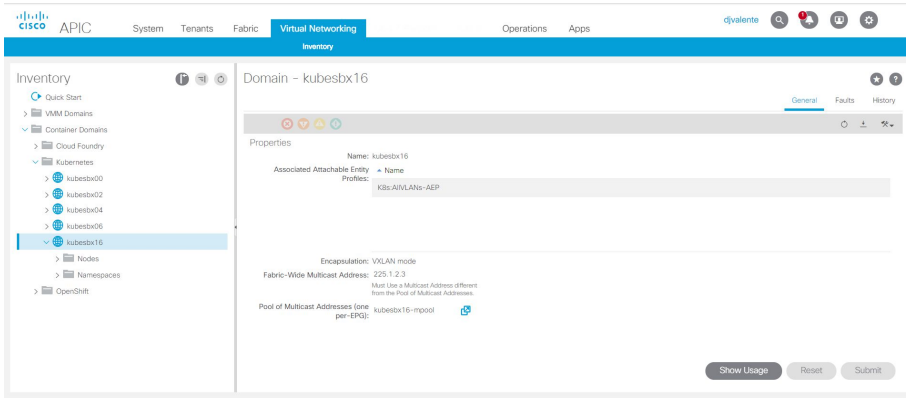


Figura 87 - Visualização no APIC do domínio associado ao cluster criado

4.2.2. Preparação da componente de rede dos nós do cluster (Linux)

É importante garantir que todos os nós que compõem o cluster dispõem de duas placas de rede distintas, sendo que uma vai ser usada para gestão dos próprios nós e a outra vai ser utilizada para a comunicação propriamente dita do Kubernetes. Esta ultima terá de ser configurada para utilizar o protocolo 802.1Q de modo a permitir múltiplas VLAN's na mesma interface.

Do ponto de vista da integração com o ACI, existe desde já uma diferença entre as duas interfaces, sendo que a interface que suportará a gestão estará associada apenas ao *Tenant Common* e a outra terá uma configuração um pouco mais complexa. No momento inicial da configuração é espectável termos as ligações conforme a Figura 88.

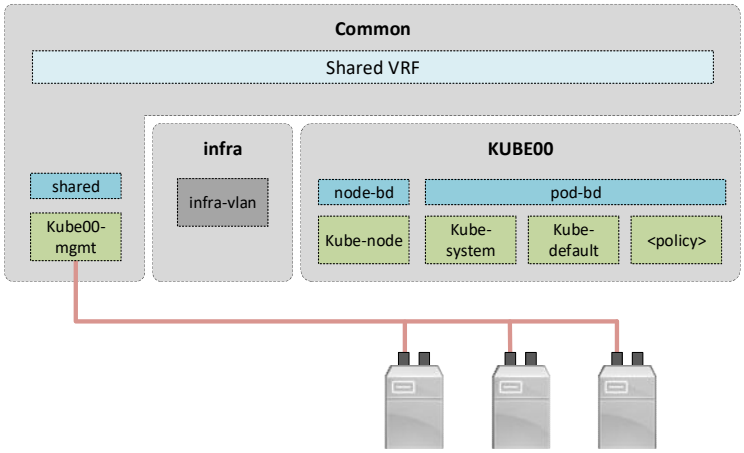


Figura 88 - Rede de interligação apenas com a placa de gestão configurada

As interfaces de gestão dos nós do cluster, do ponto de vista do ACI, vão ser "colocados" no EPG `Kube00-mgmt` que pertence ao BD `shared` do *Tennant* `Common`.

As segundas interfaces de cada nó do cluster terão de suportar duas VLAN's, sendo que os ID's respetivos foram definidos no ficheiro YAML inicial.

- Extensão do Fabric do ACI – `infra-vlan` – este subinterface estende o underlay do ACI para os nós do Kubernetes, e vai permitir que o switch OVS, que será instalado juntamente com o próprio Kubernetes, seja integrado no *Fabric* como se de uma leaf se tratasse. Este subinterface receberá um IP através de um serviço de DHCP.
- Gestão do Kubernetes – `node-vlan` – esta interface terá um endereço IP configurado manualmente.

Para fechar as configurações de rede dos nós do cluster temos de adicionar uma rota de multicast, referindo que a totalidade dos endereços reservados para multicast (224.0.0.0/4) estão disponíveis através do subinterface da `infra-vlan`. Esta configuração é importante para o tráfego gerado no *overlay* pelo ACI.

Finalmente, devemos retirar o default-gateway da interface de gestão (`kube00-mgmt`) uma vez que o subinterface de gestão dos nós já está configurado, mantendo esta interface para gestão *out-of-band*.

Após concluirmos esta configuração os nós do cluster ficam visíveis na consola de gestão do APIC como podemos ver na Figura 89, e as ligações de rede passam a estar conforme a Figura 90.

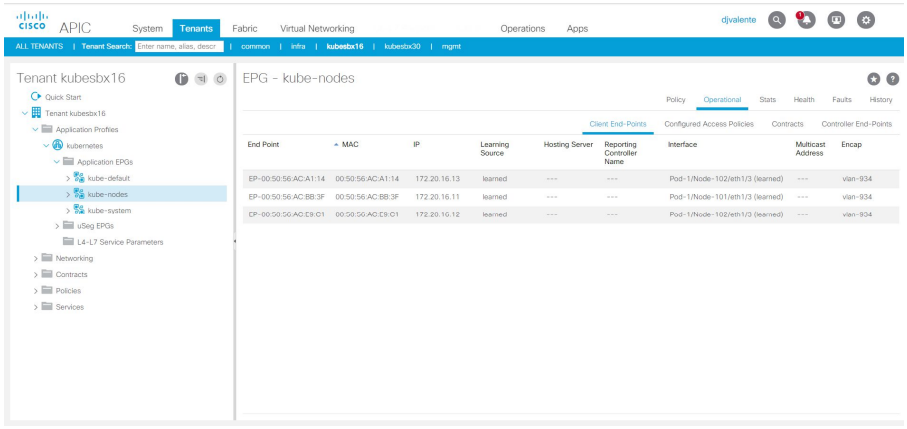


Figura 89 - Verificação no APIC da integração da rede

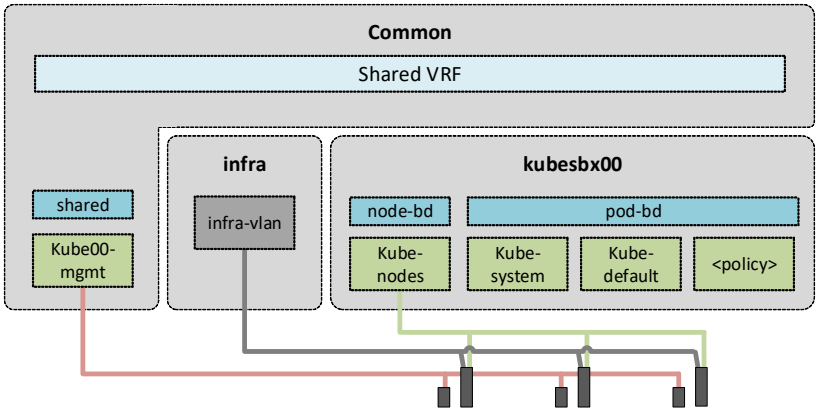


Figura 90 - Rede de interligação após a configuração da segunda placa de rede

4.2.3. Instalação do plug-in CNI do ACI

No capítulo 4.2.1, ao descrevermos a ferramenta `acc-provision`, referimos que o output da sua utilização na preparação do ACI é um ficheiro YAML que serve como input ao comando que instala o CNI-ACI no cluster. Neste sentido, o comando que instala o plugin é `kubectl apply -f aci-containers.yaml`, onde o ficheiro `aci-containers.yaml` é precisamente o ficheiro que referimos anteriormente. Na Figura 91, podemos observar o resultado da execução do comando.

```
[developer@sbx16devbox sbx16]$ kubectl apply -f aci-containers.yaml
configmap "aci-containers-config" created
secret "aci-user-cert" created
serviceaccount "aci-containers-controller" created
serviceaccount "aci-containers-host-agent" created
clusterrole "aci-containers:controller" created
clusterrole "aci-containers:host-agent" created
clusterrolebinding "aci-containers:controller" created
clusterrolebinding "aci-containers:host-agent" created
daemonset "aci-containers-host" created
daemonset "aci-containers-openvswitch" created
deployment "aci-containers-controller" created
[developer@sbx16devbox sbx16]$
```

Figura 91 - Resultado do comando de instalação do ACI-CNI

Verificação da integração no Kubernetes

Após executar o comando referido anteriormente a integração deverá estar concluída. Podemos verificar o estado da integração validando o estado dos POD's e dos nós do cluster, através dos comandos `kubectl get pods --all-namespaces` e `kubectl get nodes`. Na Figura 92 vemos um exemplo do output previsto.

```
NAMESPACE     NAME                                                READY   STATUS    RESTARTS   AGE
kube-system   aci-containers-controller-3029831268-j5x71          1/1     Running   0          10m
kube-system   aci-containers-host-67dg4                           3/3     Running   1          10m
kube-system   aci-containers-host-f2xg7                           3/3     Running   0          10m
kube-system   aci-containers-host-m8d4s                           3/3     Running   0          10m
kube-system   aci-containers-openvswitch-4vq86                    1/1     Running   0          10m
kube-system   aci-containers-openvswitch-66bx3                    1/1     Running   0          10m
kube-system   aci-containers-openvswitch-f75tw                    1/1     Running   0          10m
kube-system   etcd-sbx16kube01.localdomain                        1/1     Running   0          32m
kube-system   kube-apiserver-sbx16kube01.localdomain              1/1     Running   0          32m
kube-system   kube-controller-manager-sbx16kube01.localdomain     1/1     Running   0          32m
kube-system   kube-dns-2617979913-z6rp4                           3/3     Running   0          33m
kube-system   kube-proxy-czwcp                                    1/1     Running   0          30m
kube-system   kube-proxy-k2zh2                                    1/1     Running   0          33m
kube-system   kube-proxy-q3514                                    1/1     Running   0          31m
kube-system   kube-scheduler-sbx16kube01.localdomain              1/1     Running   0          32m

[developer@sbx16devbox sbx16]$ kubectl get nodes
NAME                       STATUS    AGE       VERSION
sbx16kube01.localdomain    Ready     34m       v1.7.15
sbx16kube02.localdomain    Ready     32m       v1.7.15
sbx16kube03.localdomain    Ready     31m       v1.7.15
```

Figura 92 - Validação da integração no Kubernetes

Podemos ainda validar os *logs* do controlador da integração através do comando `kubectl -n kube-system logs aci-containers-controller-3029831268-x0qg4`, verificando que a última mensagem é totalmente explicita referindo o sucesso da sincronização, conforme podemos ver na Figura 93.

```
[developer@sbx16devbox sbx16]$ kubectl -n kube-system logs aci-containers-controller-3029831268-j5x71
time="2019-10-15T13:42:09Z" level=info msg="Loading configuration from /usr/local/etc/aci-containers/controller.conf"
time="2019-10-15T13:42:09Z" level=info msg="Starting logLevel=info vmm-domain-type=Kubernetes
time="2019-10-15T13:42:09Z" level=info msg="Setting up Kubernetes environment" kubeconfig=
time="2019-10-15T13:42:09Z" level=info msg="Waiting for node/namespace cache sync"
time="2019-10-15T13:42:09Z" level=info msg="Updated node annotations" Node=sbx16kube01.localdomain PodNetworkRangeAnnotation="{\"V4\":[{\"start\":\"10.216.
0.2\",\"end\":\"10.216.0.129\"}]}" ServiceEpAnnotation="{\"mac\":\"e6:e8:e4:05:fc:8d\",\"ipv4\":\"192.168.116.2\"}"
time="2019-10-15T13:42:09Z" level=info msg="Updated node annotations" Node=sbx16kube02.localdomain PodNetworkRangeAnnotation="{\"V4\":[{\"start\":\"10.216.
0.130\",\"end\":\"10.216.1.1\"}]}" ServiceEpAnnotation="{\"mac\":\"4e:89:00:8c:12:9f\",\"ipv4\":\"192.168.116.3\"}"
time="2019-10-15T13:42:09Z" level=info msg="Updated node annotations" Node=sbx16kube03.localdomain PodNetworkRangeAnnotation="{\"V4\":[{\"start\":\"10.216.
1.2\",\"end\":\"10.216.1.129\"}]}" ServiceEpAnnotation="{\"mac\":\"46:e4:e8:cc:d7:43\",\"ipv4\":\"192.168.116.4\"}"
time="2019-10-15T13:42:09Z" level=info msg="Node/namespace cache sync successful"
time="2019-10-15T13:42:09Z" level=info msg="Service cache sync successful"
time="2019-10-15T13:42:09Z" level=info msg="Waiting for cache sync for remaining objects"
time="2019-10-15T13:42:09Z" level=info msg="Updated pod annotations" Eg="{\"policy-space\":\"kubesbx16\",\"name\":\"kubernetes|kube-system\"}" Sg="[]" name
=kube-dns-2617979913-z6rp4 namespace=kube-system node=
time="2019-10-15T13:42:09Z" level=info msg="Cache sync successful"
time="2019-10-15T13:42:09Z" level=info msg="Starting status server"
time="2019-10-15T13:42:09Z" level=info msg="Connecting to APIC" host=10.10.20.12
time="2019-10-15T13:42:09Z" level=info msg="Starting APIC full sync"
time="2019-10-15T13:42:09Z" level=info msg="APIC full sync completed" deletes=0 updates=20
[developer@sbx16devbox sbx16]$
```

Figura 93 - Logs de sincronização do cluster com o APIC

Verificação da integração no APIC

Como já tínhamos visto anteriormente o processo de preparação do ACI cria três EPG's onde são colocados os PODs/containers criado no cluster. No entanto, apenas o EPG `kube-nodes` tinha conteúdo (Figura 89). Nesta fase já será visível, conforme vemos na Figura 94, que o EPG `kube-system` também já tem conteúdo, neste caso o POD responsável pela resolução de nomes dentro do cluster.

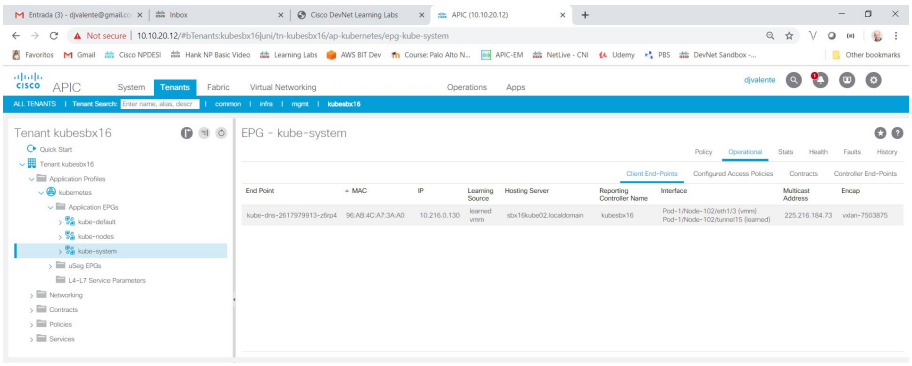


Figura 94 - Validação no ACI da integração kube-system

Este POD é também visível na Figura 92, associado ao *namespace* `kube-system`.

Outra verificação importante é validar as configurações de base que posteriormente vão permitir que o ACI controle o balanceamento de carga e o redireccionamento de tráfego L4-L7 para os serviços do Kubernetes. Na Figura 95 é possível confirmar que o processo de criação também criou um *serviço* do tipo *L4-L7* no ACI incluindo as interfaces dos nós do *cluster.*

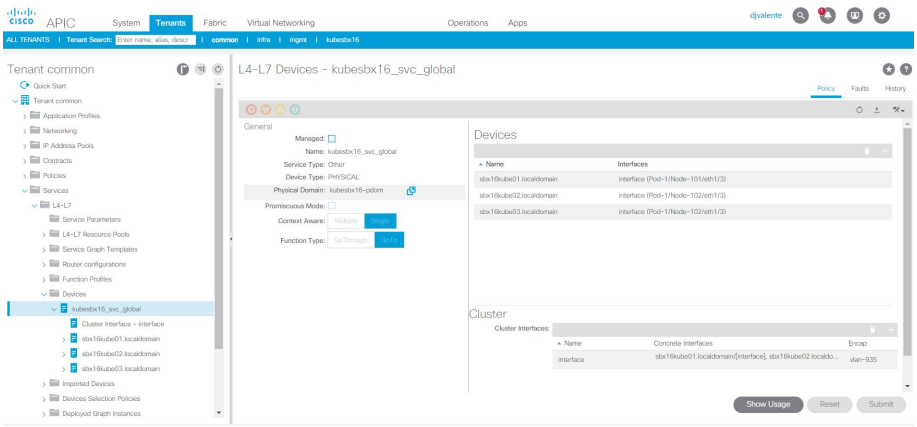


Figura 95 - Validação no ACI da integração para balanceamento e redireccionamento

Finalmente, é possível constatar que tendo a integração ficado concluída, o ACI tem uma enorme visibilidade sobre os componentes do Kubernetes, nomeadamente sobre os nós, *namespaces*, POD's, etc. A Figura 96 mostra exatamente um exemplo desta visibilidade.

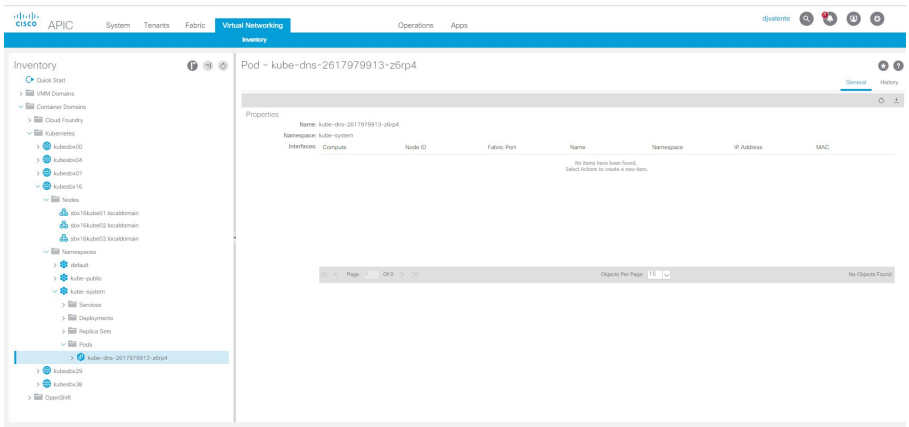


Figura 96 - Validação no ACI da integração para visibilidade

4.3. Integração do ACI com a cloud Azure

A integração do ACI com a cloud pública da Microsoft é uma funcionalidade apenas disponível na versão 4.2, lançada no final de setembro de 2019. O principal objetivo desta funcionalidade é precisamente estender o controlo que o ACI tem sobre a rede do centro de dados local para a rede do Azure, isto é, vai ser possível replicar o modelo de gestão do ACI para o Azure. Esta funcionalidade vai permitir a criação de uma política de acessos global, um único ponto de gestão, automação de várias componentes e também o consumo de serviços nativos da cloud. A Figura 97 pretende precisamente representar a arquitetura que iremos implementar.

Apesar de não ser âmbito deste trabalho, é relevante acrescentar que o ACI, à data de hoje, também já integra com a cloud pública da Amazon.

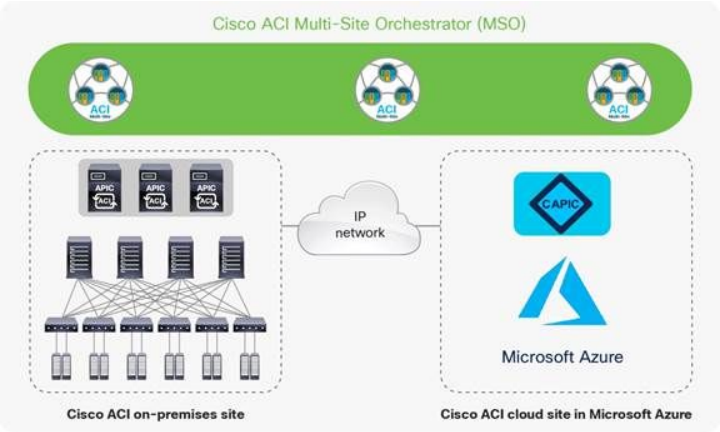


Figura 97 - Representação da arquitetura ACI mais Azure (retirado de [51])

A Figura 97 permite constatar que esta integração obriga à adição de três novas componentes à arquitetura, nomeadamente:

- *Multi-site Orchestrator (MSO)*
- *Cloud APIC (CAPIC)*
- Conectividade entre o centro de dados e o Azure.

O MSO é precisamente o ponto central de gestão que nos vai permitir obter uma visão unificada, e um único ponto de controlo de toda a infraestrutura. Será no MSO que iremos definir a política de segurança a aplicar no controlador do centro de dados local, o APIC, ou no controlador da rede do Azure, o CAPIC. Será também responsável por orquestrar toda a componente de interligação entre os dois locais.

O CAPIC, conforme referimos anteriormente, é a componente que vai gerir toda a política de rede da cloud, recorrendo ao modelo de políticas criado no MSO. A principal diferença entre o APIC instalado localmente no centro de dados e o CAPIC do Azure, é que as políticas enviadas para o APIC são instaladas diretamente nos switchs do centro de dados, ao passo que essas mesmas políticas no

CAPIC têm de ser traduzidas para serem posteriormente aplicadas na infraestrutura de rede do Azure. A aplicação propriamente dita das políticas é através da utilização das API's que a cloud Azure disponibiliza para automatização da sua própria infraestrutura. Resumindo, o CAPIC é responsável por:

- Disponibilizar uma interface "north-bound" para o ambiente cloud
- Interpreta o modelo de políticas do ACI, traduzindo-o para os conceitos Azure
- Responsável pela descoberta de novos dispositivos na cloud
- Configuração do *control-plane* dos routers da cloud
- Configura o caminho que a informação utiliza entre o ACI e a cloud

4.3.1. Ligação entre o centro de dados local e a cloud

Conforme referimos anteriormente o CAPIC tem a responsabilidade de orquestrar a configuração das diferentes componentes que asseguram a conectividade entre os dois pontos, no entanto, apesar de não ser necessário efetuar qualquer configuração é de extrema importância conhecer a arquitetura no sentido de identificar eventuais limitações à arquitetura global que pretendemos implementar.

Componente *Underlay* da rede de interligação

Toda a conectividade é suportada por dois routers Cisco virtuais do modelo CSR1000V que são instalados em Azure e configurados para estabelecer um túnel *IPSec* para um equipamento no centro de dados local. Estes routers são colocados numa VNET denominada *Infra*, que desempenha exatamente o mesmo papel que a VRF *Infra* existente no centro de dados local. A Figura 98 pretende representar este primeiro passo.

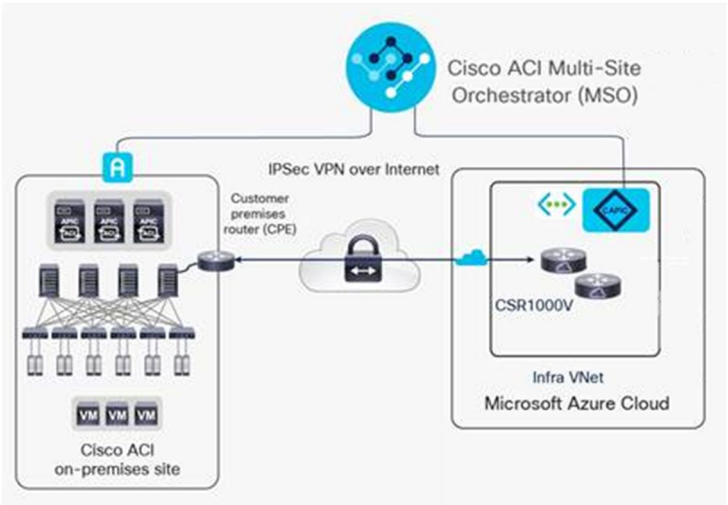


Figura 98 - Componente underlay da arquitetura ACI mais Azure (adaptado de [51])

Estes routers virtuais além de assegurarem a conectividade entre os sites, vão também ser uma peça fundamental para o dinamismo que se pretende obter relativamente ao endereçamento IP,

isto é, será necessário garantir que redes criadas dentro do Azure são anunciadas para o centro de dados, o que tendo em conta que não conseguimos estabelecer adjacências de *routing* diretamente com o Azure, obriga-nos a recorrer ao objeto *Virtual Network Gateway* (VNG), que é precisamente o objeto que nos vai possibilitar este dinamismo.

Tendo em conta esta característica, a arquitetura recomendada dentro da cloud Azure é precisamente uma abordagem *hub&spoke*, isto é, consideramos a VNET Infra como o centro da conectividade com o centro de dados local, e todas as VNET's ligam-se através de um túnel IPSec entre o seu VNG e os routers Cisco. Esta arquitetura confere-nos uma enorme agilidade, podendo inclusivamente ter VNET's em diferentes regiões. A Figura 99 representa esta componente da arquitetura.

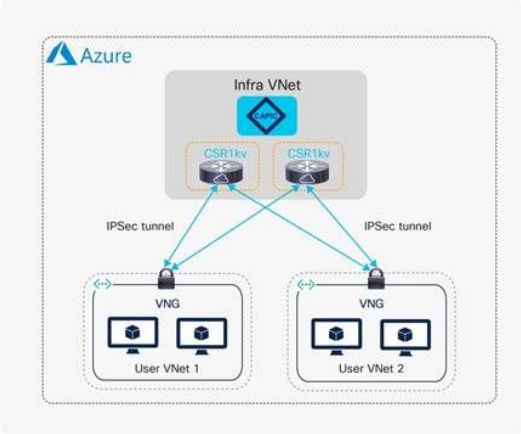


Figura 99 - Conectividade dentro do Azure (retirado de [51])

Componente *Overlay* da rede de interligação

De forma a permitir a correta integração entre os dois mundos, é criado um overlay que utiliza BGP-EVPN para a componente de controlo (*control plane*) e utiliza um encapsulamento VXLAN para a componente de transporte (*data plane*).

Ao selecionar VXLAN para o encapsulamento, que facilmente se entende por ser considerado por muitos como o *standard* para arquiteturas híbridas, é necessário colmatar algumas limitações, daí a adição de BGP-EVPN. As principais limitações do VXLAN no nosso cenário são:

- Uso ineficiente da largura de banda na aprendizagem e envio de tráfego do tipo BUM[7] - replica o tráfego para todos os VTEP
- Tabelas MAC muito extensas, levando à necessidade de recursos adicionais

[7] *Broadcast, Unknown unicast e Multicast*

- Gateways centralizados – obrigando a movimentações de tráfego não-ótimas.
- Falta de autenticação nos dispositivos VXLAN (VTEP's)

Em contrapartida, a adição de BGP-EVPN, confere precisamente as seguintes vantagens [52]:

- Capacidade de anunciar, via BGP, rotas individuais (*host*) ou de redes, permitindo uma melhor visibilidade sobre a conectividade entre dispositivos.
- Autenticação através das funcionalidades nativas do BGP
- Utilização de *Anycast Gateway* distribuído, otimizando fluxos e eliminando os gateways centralizados.
- Supressão de pedidos de ARP, substituindo por operações BGP, melhorando a utilização de largura de banda e reduzindo a dimensão das tabelas MAC.
- Replicação de tráfego *ingress* de forma dinâmica, para conseguirmos disponibilizar uma alternativa *unicast* ao *multicast* utilizado no *underlay*.

Na Figura 100 conseguimos ver uma representação completa das diferentes componentes que asseguram a conectividade entre o ACI do centro de dados e a rede da cloud Azure.

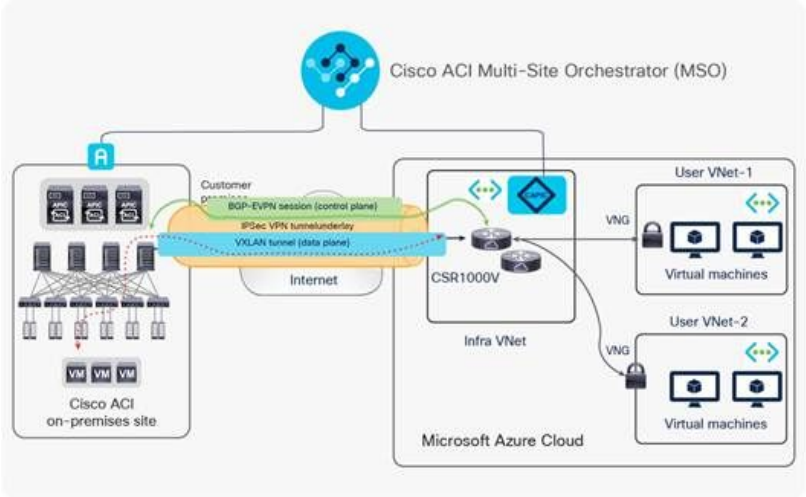


Figura 100 - Arquitetura de rede da ligação do ACI ao Azure (retirado de [51])

4.3.2. Tradução do modelo de politicas do ACI para o modelo do Azure

Conforme vimos anteriormente, quando descrevemos a solução ACI, todas as definições de rede são efetuadas tendo por base objetos específicos da solução, isto é, operamos *Bridge Domains (BD), bd-subnets, Endpoint Groups (EPG) e Contracts.* Em Azure encontramos outros objetos, desta vez, específicos desta plataforma, como é o caso dos *Resource Groups, Virtual Networks, Application Security Groups (ASG), Network Security Groups (NSG)* e filtros específicos para o tráfego que entra ou sai dos objetos.

Uma política de ACI será sempre baseada em EPG's e Contract's como podemos ver na Figura 101.



Figura 101 - Política de segurança em ACI (retirado de [51])

Em contrapartida, em Azure temos de utilizar ASG's, NSG's e filtros de entrada de saída. Na Figura 102 vemos um exemplo em Azure.

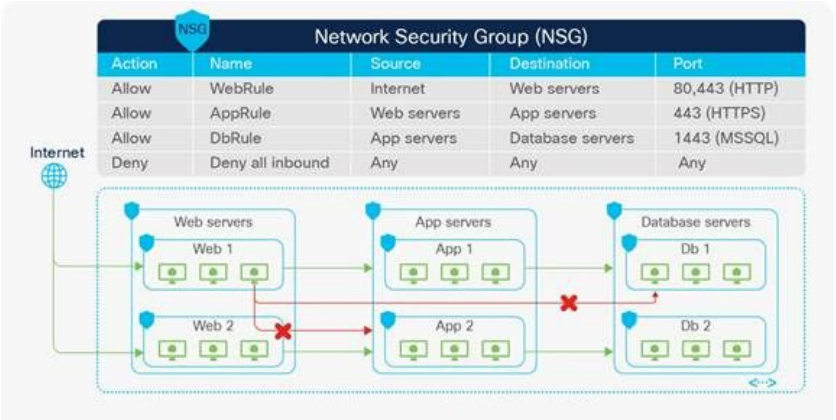


Figura 102 - Política de segurança em Azure (adaptado de [51])

Uma vez que o pretendido é estender a abordagem ACI para a cloud Azure, é necessário efetuar um mapeamento entre todos estes objetos, isto é, ao criar uma politica de ACI para ser aplicada em Azure, o CAPIC utiliza a correspondência visível na Figura 103 para criar os objetos necessários.

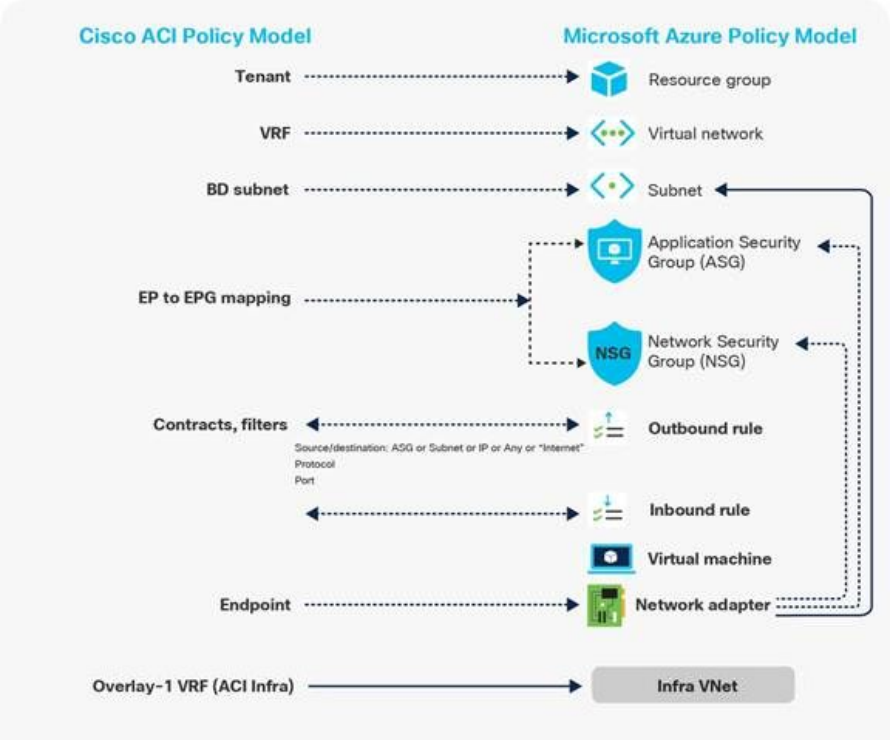


Figura 103 - Correspondência entres os objetos ACI e Azure (adaptado de [51])

5. TESTES E RESULTADOS

5.1. Validação da inter-operabilidade entre o ACI e o Kubernetes

5.1.1. Sincronismo e visibilidade (aplicação *guest-book)*

O primeiro teste que pretendemos realizar é precisamente a instalação de uma aplicação que contenha mais do que um POD e confirmar a sincronização dessa informação para o ACI.

Para isso iremos utilizar um exemplo disponibilizado no site oficial do Kubernetes[8], que consiste numa aplicação composta por uma camada para *front end* composta por três POD's, uma Redis Cache composta por um POD para *master* e dois para *slave*, conforme vemos na Figura 104.

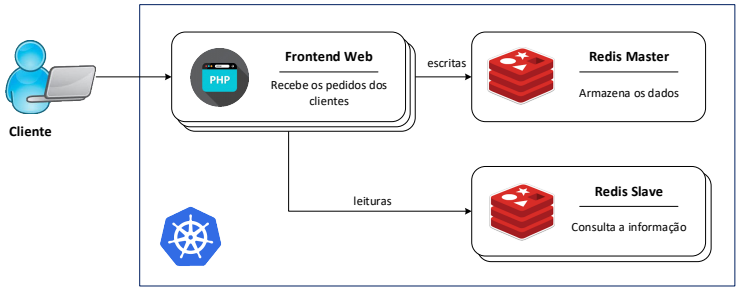


Figura 104 - Arquitetura da aplicação de teste guestbook

Apos executar o comando `kubectl apply` utilizando o descritivo em YAML correspondente, é possível verificar diretamente no Kubernetes os POD's que compõem a solução e o endereço externo que permite a sua utilização, conforme vemos na Figura 105.

```
[developer@sbx16devbox guestbook]$ kubectl get pods
NAME                              READY     STATUS    RESTARTS   AGE
frontend-1768566195-851gj         1/1       Running   0          1h
frontend-1768566195-k0nbv         1/1       Running   0          1h
frontend-1768566195-n9151         1/1       Running   0          1h
redis-master-2365125485-vth7q     1/1       Running   0          1h
redis-slave-3837281623-hv8w0      1/1       Running   0          1h
redis-slave-3837281623-mf4k8      1/1       Running   0          1h

[developer@sbx16devbox guestbook]$ kubectl get services
NAME           CLUSTER-IP       EXTERNAL-IP     PORT(S)          AGE
frontend       192.168.216.76   172.20.16.194   80:32599/TCP     1h
kubernetes     192.168.216.1    <none>          443/TCP          20h
redis-master   192.168.216.78   <none>          6379/TCP         1h
redis-slave    192.168.216.178  <none>          6379/TCP         1h
```

Figura 105 - Criação da aplicação de teste guestbook

[8] https://kubernetes.io/docs/tutorials/stateless-application/guestbook/

Se acederemos ao EXTERNAL-IP atribuído na porta 80 constatamos, conforme vemos na Figura 106 que a aplicação está disponível e que é possível efetuar escritas no *guestbook.*

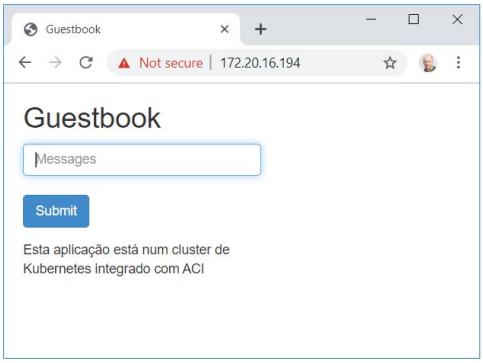


Figura 106 - Aceder à aplicação guest-book

Do lado do APIC é também possível verificar todos os objetos de Kubernetes criados nomeadamente os POD's propriamente ditos, mas também os serviços que permitem a interação entre eles, conforme vemos na Figura 107.

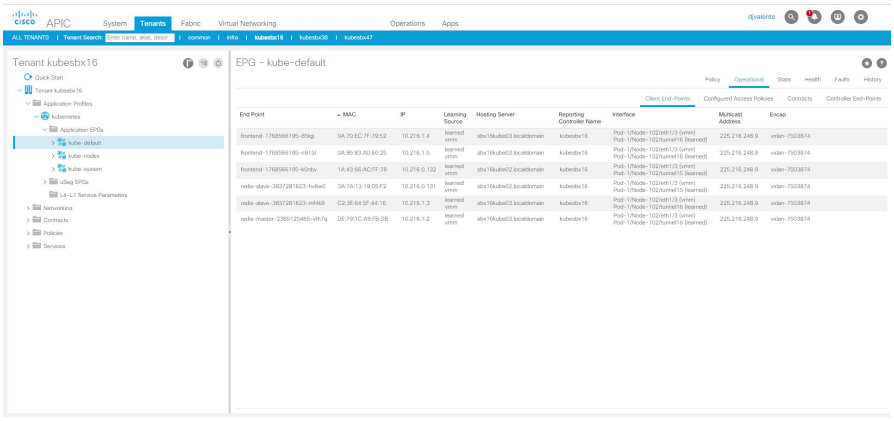


Figura 107 - Validação no ACI da aplicação de teste guest-book

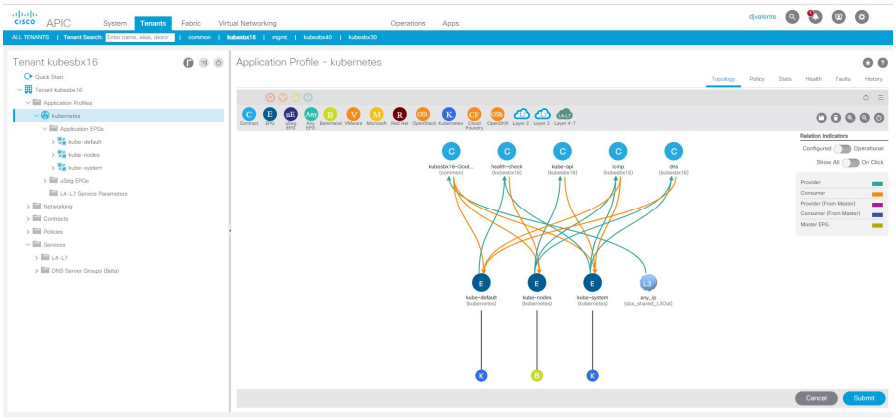


Figura 108 - Validação no ACI da aplicação de teste guestbook (2)

Com este nível de integração os containers são vistos pelo ACI como *endpoints* independentes, na prática, como se tratassem de máquinas virtuais ou servidores físicos. Desta forma eliminamos uma camada de abstração de rede, que nada acrescenta à operação/gestão das soluções. Alem disso,

podemos tirar partido das funcionalidades nativas do ACI para resolução de problemas e visibilidade aplicacional.

5.1.2. Balanceamento de tráfego por hardware

Tirando partido das funcionalidades descritas acima, é possível aproveitar os próprios equipamentos que compõem a rede do datacenter (switchs Cisco Nexus), para balancear tráfego diretamente para os POD's. Conforme já vimos anteriormente, o balanceamento é um aspeto absolutamente critico de uma solução de containers, pelo que esta funcionalidade, em ambientes mais simples, poderá eliminar a necessidade de adicionar uma solução especifica para balancear tráfego.

Para verificarmos como o ACI endereça esta necessidade vamos recorrer à aplicação de teste *my_hero*[9], que apresenta uma arquitetura tradicional a três camadas (frontend, backend, database), conforme vemos na Figura 109.

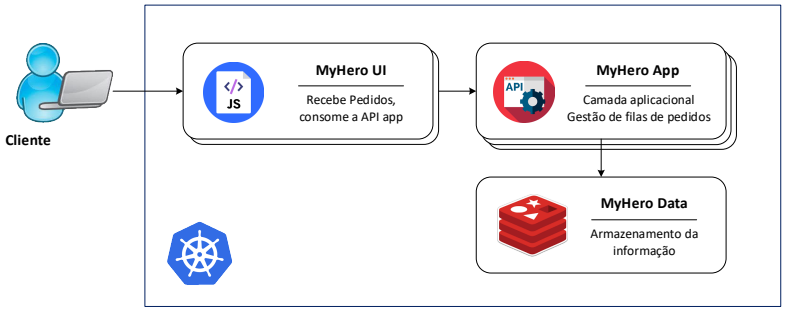


Figura 109 - Arquitetura da aplicação de teste MyHero

A instalação da aplicação será através do comando `kubectl`, passando como parâmetro os ficheiros YAML disponibilizados no github, dando origem no Kubernetes aos POD's visíveis na Figura 110.

```
[developer@sbx16devbox cluster-isolation]$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
myhero-app-1608251026-0d5zf         1/1       Running   0          9m
myhero-app-1608251026-640kg         1/1       Running   0          9m
myhero-app-1608251026-16r1m         1/1       Running   1          9m
myhero-data-2347389596-cxx3g        1/1       Running   0          9m
myhero-ernst-3425573530-cksrp       1/1       Running   0          9m
myhero-mosca-793212554-gfgwk        1/1       Running   0          9m
myhero-ui-3593555378-21pnj          1/1       Running   0          4m
myhero-ui-3593555378-2vk4b          1/1       Running   0          4m
[developer@sbx16devbox cluster-isolation]$
```

Figura 110 - POD's criados para a aplicação MyHero

- 2x myhero-ui – front-end web da aplicação
- 3x myhero-app – camada aplicacional (API)

[9] https://github.com/hpreston/myhero_demo.

- 1x myhero-data – camada de base de dados
- 1x myhero-mosca – fila de mensagens (apoio à componente aplicacional)
- 1x myhero-ernst – serviço que gere a entrada de votos (apoio à componente aplicacional)

A aplicação estará disponível no EXTERNAL-IP do serviço associado ao myhero-ui.

```
[developer@sbx16devbox cluster-isolation]$ kubectl get services
NAME           CLUSTER-IP       EXTERNAL-IP     PORT(S)          AGE
kubernetes     192.168.216.1    <none>          443/TCP          1d
myhero-app     192.168.216.3    172.20.16.195   80:30490/TCP     17m
myhero-data    192.168.216.61   <nodes>         80:30389/TCP     18m
myhero-mosca   192.168.216.154  <nodes>         1883:32333/TCP   18m
myhero-ui      192.168.216.40   172.20.16.196   80:31381/TCP     13m
[developer@sbx16devbox cluster-isolation]$
```

Figura 111 - Acesso à aplicação MyHero

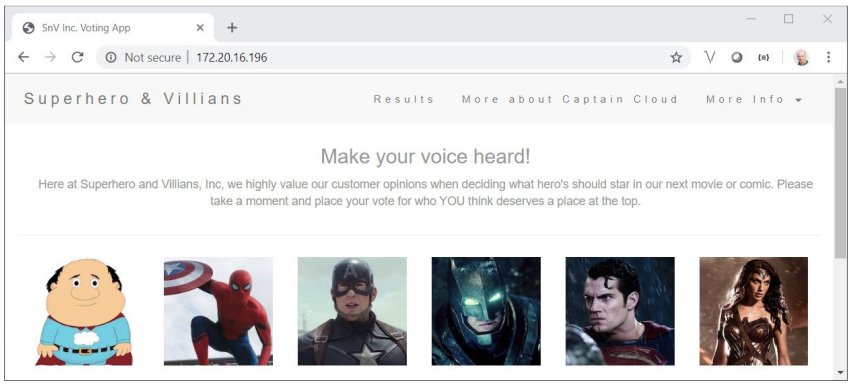


Figura 112 - Aplicação MyHero

Do lado do ACI é necessário analisar de que forma é efetuado o "mapeamento" entre o endereço onde a aplicação está disponibilizada (172.20.16.196) e os endereços dos POD's onde o código está em execução.

Sem qualquer intervenção manual no ACI é possível constatar que a criação de um serviço do tipo LOAD-BALANCER no Kubernetes, despoleta a criação de uma entrada no ACI que permite a entrada de tráfego no *fabric* para os endereços associados ao EXTERNAL-IP.

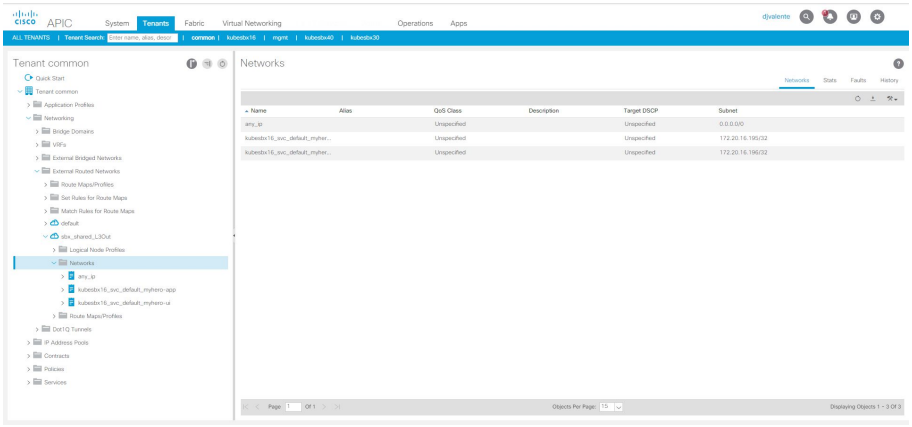


Figura 113 - Adição de entrada no L3Out (common) para a aplicação MyHero

De forma a que o tráfego seja balanceado, é também automaticamente criado um objeto do tipo *Service Graph* que é associado ao contrato que permite a comunicação entre os EPG's. Esta configuração define o tipo de tráfego que é permitido (*filter*), e também uma política de redireccionamento de tráfego (*policy based routing / load balancing)*. Na Figura 114 podemos ver uma representação do contrato, com a referência ao *service graph* e ao *filter*. Na Figura 115 são visíveis as configurações do *service graph*, e finalmente na Figura 116 temos a configuração dos destinos do balanceamento de carga.

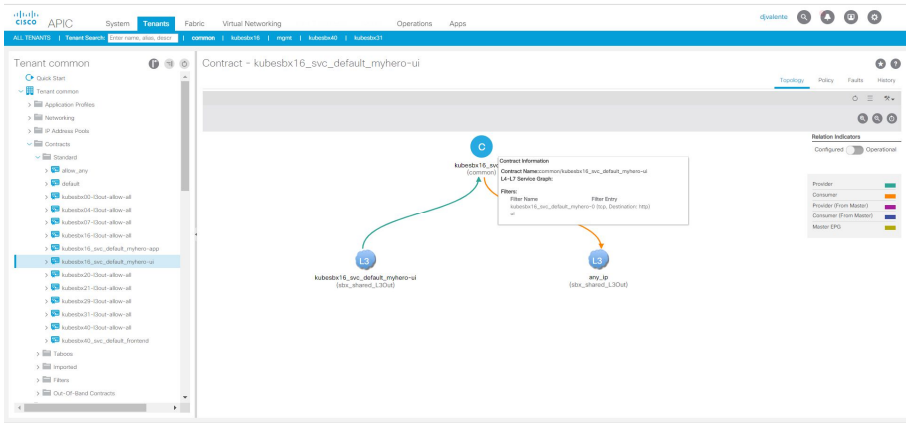


Figura 114 - Contrato no ACI

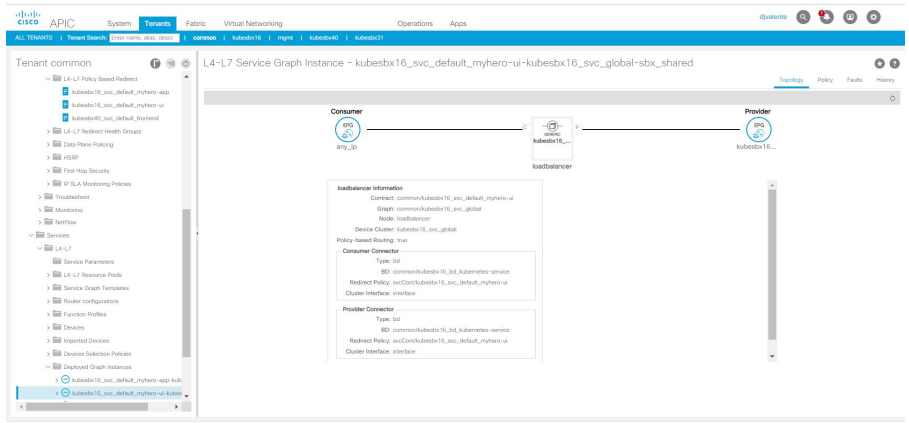


Figura 115 – Service Graph no ACI

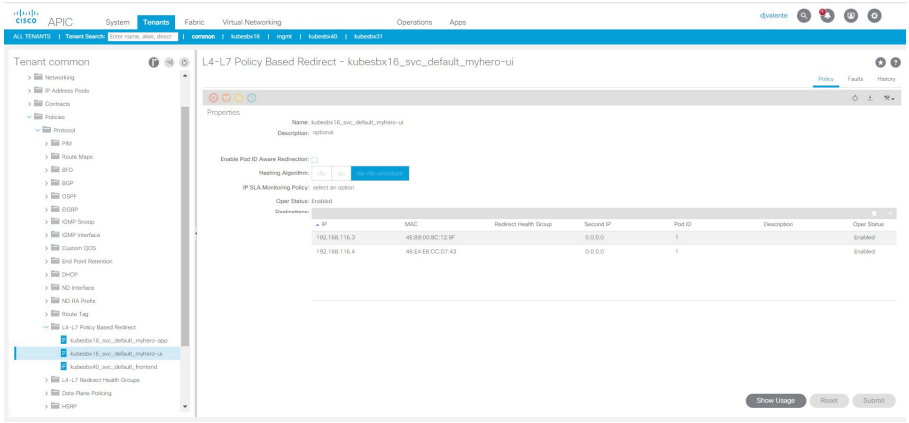


Figura 116 - Política de encaminhamento de tráfego no ACI

A Figura 117 pretende resumir todo o processo de entrada do tráfego no cluster de Kubernetes quando usamos o CNI-ACI.

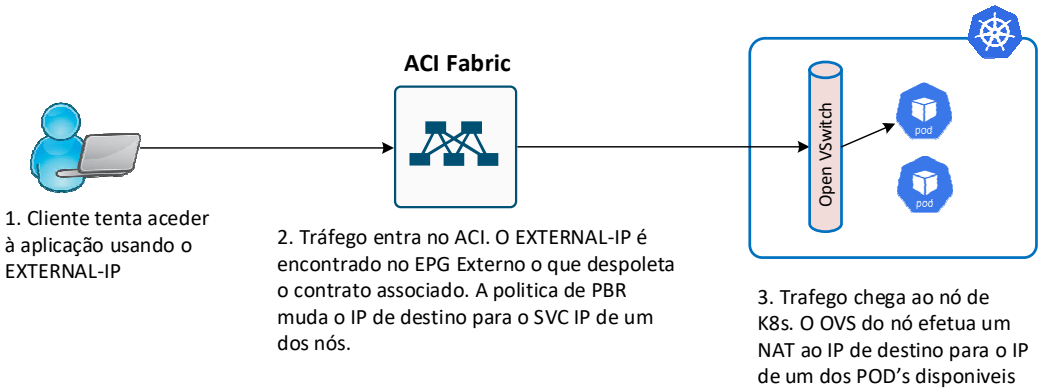


Figura 117 - Representação da entrada de tráfego na solução K8s+ACI

5.1.3. Politica de controlo de acessos (aplicação MyHero)

A integração do ACI com o Kubernetes disponibiliza três diferentes abordagens no que diz respetivo à segmentação e controlo dos acessos entre diferentes POD's:

- Isolamento ao cluster – ao optarmos por esta abordagem o que temos do lado do ACI é apenas um único EPG para o todo o cluster de Kubernetes. Todos os POD's criado serão associados a este EPG, terão a mesma rede, a mesma política de segurança, e a mesma política operacional. Com um isolamento ao cluster, não é necessário criar qualquer contrato para os POD's falarem entre si. Este é precisamente o modo de operação por defeito, conforme vimos no teste anterior os POD's criados foram colocado no EPG `kube-default` e não houve qualquer dificuldade na comunicação entre eles. Optar por este modo tem a vantagem de que não é necessário efetuar qualquer alteração às aplicações, no entanto o ACI não efetuará um controlo específico da comunicação entre POD's.
- Isolamento por *Namespace* – ao isolarmos por *Namespace*, o ACI associa um EPG específico a cada *Namespace*. Neste caso conseguimos que todas as aplicações que pertençam ao mesmo Namespace/EPG falem ente si, sem necessitar de qualquer contrato ou filtro adicional no ACI, mas se as aplicações estiverem em *Namespaces* distintos necessitarem de trocar informação, aí sim, teremos de criar essa permissão de forma explicita no ACI. Esta solução obriga à criação de um EPG para cada *Namespace*, e também à adição de uma *annotation* à definição do *Namespace* no Kubernetes.
- Isolamento por *Deployment* – ao segmentar por Deployment o ACI associará um EPG específico para cada Deployment. Um Deployment em Kubernetes, como já vimos

anteriormente, representa a definição de um microserviço, ou componente aplicacional. Vamos necessitar de contratos para que estas diferentes componentes consigam comunicar entre si. Esta abordagem é a que possibilita um controlo muito mais robusto, no entanto vai obrigar a que no ACI sejam criados *application profiles, EPG's* e *contracts,* mapeando a arquitetura aplicacional e permitindo a comunicação entre os diferentes microserviços. À semelhança do modelo de *Namespace*, será igualmente necessário adicionar *annotations* à definição do deployment para permitir o respetivo mapeamento com o EPG do ACI.

A Figura 118 representa os três tipos de isolamento descritos anteriormente.

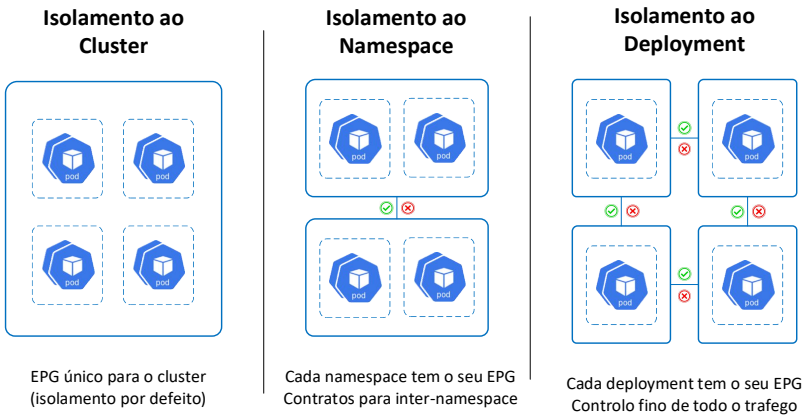


Figura 118 - Representação dos diferentes tipos de isolamento

Se considerarmos o exemplo que descrevemos para validar a componente de balanceamento de carga, em que todos os POD's que compõem a aplicação são colocados no mesmo EPG, neste caso o EPG `kube-default`, verificamos o funcionamento da opção isolamento por cluster. Neste caso não é necessário efetuar no ACI qualquer configuração para que os POD's consigam falar entre si. Vamos então descrever os restantes modos de isolamento.

Controlo de acesso baseado em Namespace

O controlo baseado em *Namespace*, pressupõe a criação de um novo EPG no ACI que posteriormente será associado a um *Namespace* do Kubernetes, permitindo desta forma que a política seja gerida diretamente no APIC. Para o nosso teste criamos um EPG denominado `ns-myhero` e um *Namespace* denominado `myhero`, como podemos verificar na Figura 119 e na Figura 120.

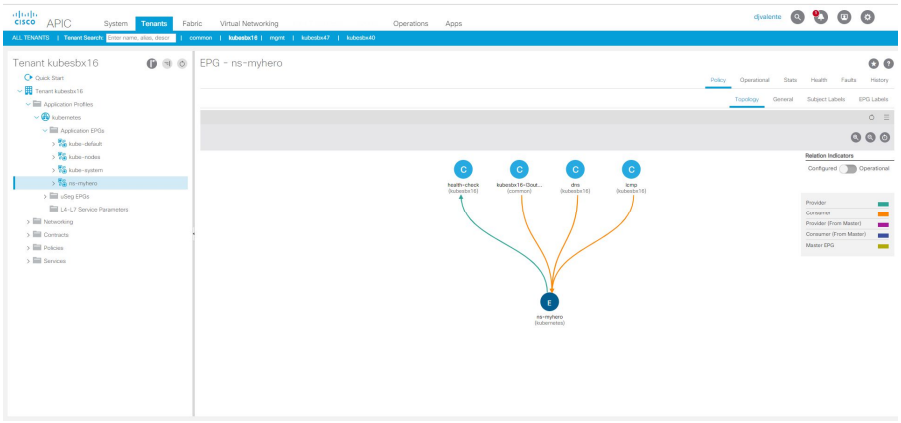


Figura 119 - Novo EPG para isolamento baseado em Namespace

```
[developer@sbx16devbox ~]$
[developer@sbx16devbox ~]$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    4d
kube-public   Active    4d
kube-system   Active    4d
[developer@sbx16devbox ~]$
[developer@sbx16devbox ~]$  kubectl create namespace myhero
namespace "myhero" created
[developer@sbx16devbox ~]$
[developer@sbx16devbox ~]$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    4d
kube-public   Active    4d
kube-system   Active    4d
myhero        Active    15s
[developer@sbx16devbox ~]$
[developer@sbx16devbox ~]$
```

Figura 120 - Criação de um novo Namespace

A forma como associamos o Namespace do Kubernetes com o EPG do ACI é através da utilização de *Annotations.* O comando abaixo adiciona uma *annotation* de ACI ao *Namespace* que criamos.

```
kubectl annotate namespace myhero opflex.cisco.com/endpoint-
group='{"tenant": "kubesbx16", "app-profile": "kubernetes",
"name": "ns-myhero"}' -overwrite
```

Na Figura 121 é possível validar os detalhes do Namespace, incluindo a *annotation*.

```
[developer@sbx16devbox ~]$
[developer@sbx16devbox ~]$ kubectl describe namespace myhero
Name:           myhero
Labels:         <none>
Annotations:    opflex.cisco.com/endpoint-group={"tenant": "kubesbx16", "app-profile": "kubernetes", "name": "ns-myhero"}
Status:         Active

No resource quota.

No resource limits.
[developer@sbx16devbox ~]$
```

Figura 121 - Adicionar annotation ao namespace

A partir deste momento, qualquer objeto que seja criado no namespace `myhero` será colocado no EPG `ns-myhero`.

Vamos utilizar novamente a aplicação MyHero descrita anteriormente, no entanto ao executarmos os comandos `kubectl` que instalam a aplicação vamos adicionar o parâmetro -n `myhero` que irá colocar todos os POD's no namespace `myhero` e consequentemente no EPG `ns-myhero`.

```
[developer@sbx16devbox namespace-isolation]$
 kubectl -n myhero apply -f myhero_data.yaml
 kubectl -n myhero apply -f myhero_mosca.yaml
 kubectl -n myhero apply -f myhero_ernst.yaml
 kubectl -n myhero apply -f myhero_app.yaml
 kubectl -n myhero apply -f myhero_ui_ready.yaml
[developer@sbx16devbox namespace-isolation]$
```

Figura 122 - Criação da aplicação com isolamento namespace

Se criarmos um POD no namespace `default`, o facto de estarem em EPG's que não têm qualquer contrato entre eles, garante que não terão qualquer conectividade.

Na Figura 123 confirmamos que o POD `trojanapp` se encontra no EPG `kube-default`, e na Figura 124 vemos as tentativas de acesso a partir deste POD, ao POD da camada de base de dados da aplicação alojada no EPG `ns-myhero`.

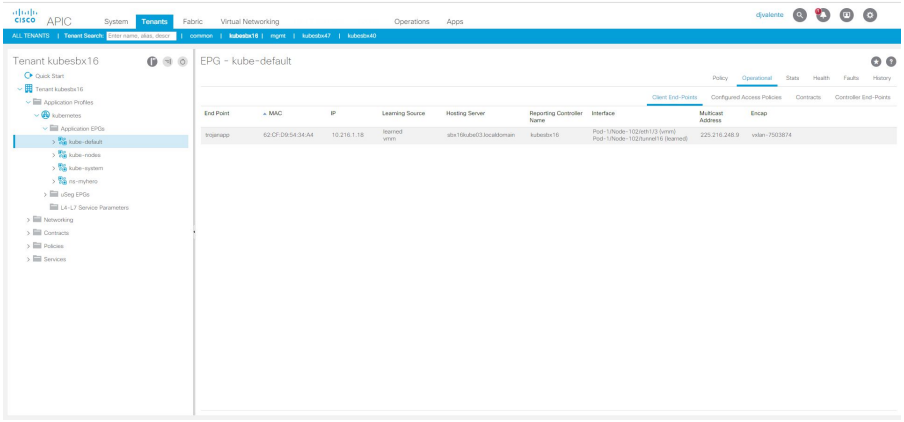


Figura 123 - POD trojanapp no EPG kube-default

```
[root@trojanapp code]#
[root@trojanapp code]# curl myhero-data/options
curl: (6) Could not resolve host: myhero-data; Unknown error
[root@trojanapp code]#
[root@trojanapp code]#
[root@trojanapp code]# curl 10.216.1.14/options
curl: (7) Failed connect to 10.216.1.14:80; Connection timed out
[root@trojanapp code]#
[root@trojanapp code]#
[root@trojanapp code]# curl 10.216.1.14:5000/options
curl: (7) Failed connect to 10.216.1.14:5000; Connection timed out
[root@trojanapp code]#
[root@trojanapp code]#
[root@trojanapp code]# ping 10.216.1.14
PING 10.216.1.14 (10.216.1.14) 56(84) bytes of data.
^C
--- 10.216.1.14 ping statistics ---
130 packets transmitted, 0 received, 100% packet loss, time 129001ms

[root@trojanapp code]#
```

Figura 124 - Validação de acessos no isolamento baseado em namespace

Se repetirmos os testes anteriores, mas agora colocando o POD no namespace `myhero`, verificamos que POD's dentro do mesmo namespace, não têm qualquer controlo ao nível da conectividade.

A Figura 125 permite confirmar que existe mais um POD no EPG `ns-myhero`, e a Figura 126 demonstra que existe total conectividade entre este POD e os restantes POD's contidos no namespace.

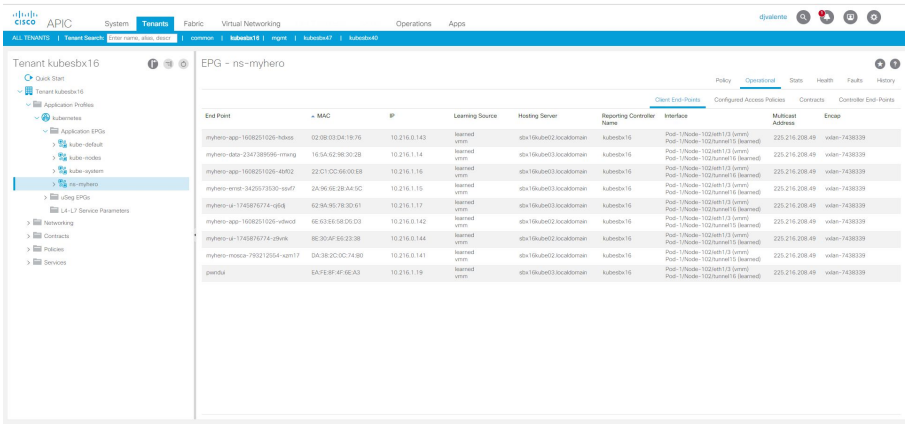


Figura 125 - POD pwndui no EPG ns-myhero

```
[root@pwndui code]#
[root@pwndui code]# curl -H "key: SecureData" myhero-data/options
{
    "options": [
        "Captain Cloud",
        "Spider-Man",
        "Captain America",
        "Batman",
        "Superman",
        "Wonder Woman",
        "Deadpool",
        "Black Widow",
        "Iron Man",
        "Star-Lord",
        "Scarlet Witch",
        "Gamora"
    ]
}[root@pwndui code]#
[root@pwndui code]#
[root@pwndui code]# ping 10.216.1.14
PING 10.216.1.14 (10.216.1.14) 56(84) bytes of data.
64 bytes from 10.216.1.14: icmp_seq=1 ttl=64 time=0.453 ms
64 bytes from 10.216.1.14: icmp_seq=2 ttl=64 time=0.090 ms
64 bytes from 10.216.1.14: icmp_seq=3 ttl=64 time=0.095 ms
64 bytes from 10.216.1.14: icmp_seq=4 ttl=64 time=0.046 ms
^C
--- 10.216.1.14 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.046/0.171/0.453/0.163 ms
[root@pwndui code]#
[root@pwndui code]#
[root@pwndui code]# curl -H "key: SecureData" 10.216.1.14:5000/options
{
    "options": [
        "Captain Cloud",
        "Spider-Man",
        "Captain America",
        "Batman",
        "Superman",
        "Wonder Woman",
        "Deadpool",
        "Black Widow",
        "Iron Man",
        "Star-Lord",
        "Scarlet Witch",
        "Gamora"
    ]
}[root@pwndui code]#
```

Figura 126- Validação de acessos no isolamento baseado em namespace (2)

Controlo de acesso baseado em Deployments

Este modelo do controlo de tráfego é o que permite uma especificação mais granular do tráfego permitido entre micro serviços. É também aquele que vai obrigar a uma configuração do lado do ACI um pouco mais complexa, isto é, necessitaremos de criar um EPG para cada componente aplicacional e os respetivos contratos que vão permitir a comunicação. À semelhança do método anterior vamos utilizar a aplicação MyHero, pelo que na Figura 127 podemos ver o *Application Profile* criado para o efeito.

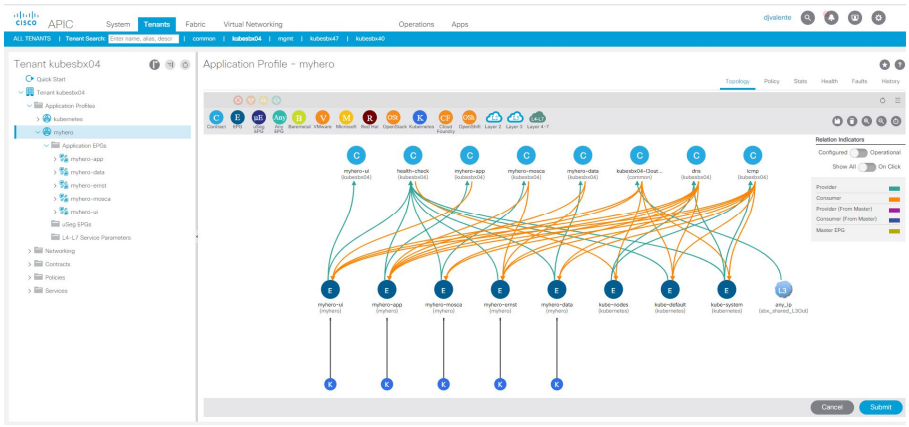


Figura 127 - Application Profile da aplicação MyHero no ACI

De forma a permitir uma melhor leitura dos fluxos aplicacionais permitidos pela política criada, vamos focar-nos apenas no EPG `myhero-data`, que teoricamente seria onde estariam armazenados os dados da aplicação, e daí ser um local onde deveríamos ter uma maior atenção relativamente a acessos não autorizados. Na Figura 128 é possível visualizar o detalhe referente ao *Applicacion Profile* para o EPG em questão.

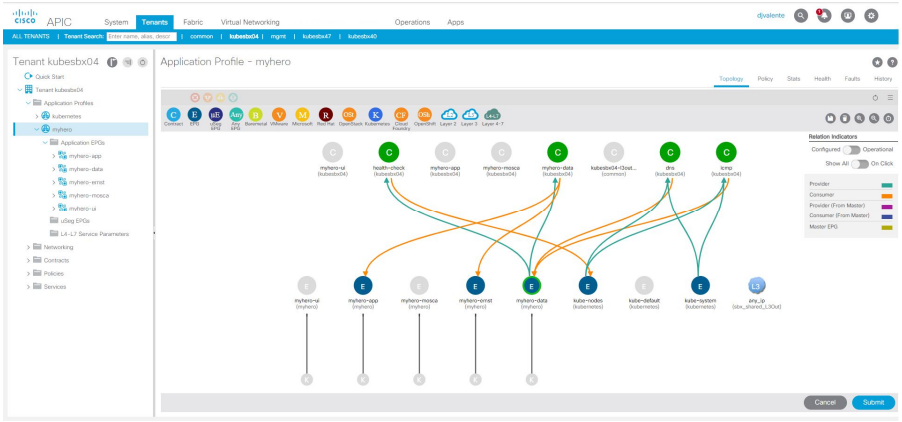


Figura 128 - Detalhe referente aos acessos ao EPG myhero-data

Importante salientar que este EPG apenas tem contrato com quatro EPG's, o que na prática significa que apenas pode enviar ou receber tráfego, de ou para estes quatro EPG's. O tipo de tráfego que é permitido é definido no filtro que está associado a cada contrato. No nosso exemplo criamos

apenas um filtro a permitir tráfego IP, e associamo-lo a todos os contratos. É também relevante mencionar que não existe nenhuma relação entre o `myhero-ui` ou o `L3-Out`, o que significa que a camada de apresentação e o ambiente exterior não terão conectividade direta com a componente de base de dados da aplicação, desta forma reduzimos a probabilidade de comprometer os dados da aplicação, em casos em que apenas é comprometida a camada de apresentação.

Após preparar o ACI para receber a aplicação é necessário descrever o que temos de assegurar do lado do Kubernetes para que cada *Deployment* seja mapeado com o EPG correto. À semelhança do método baseado em *Namespaces*, neste caso também vamos necessitar de recorrer a *annotations*, mas agora associadas aos próprios deployment e não ao *Namespace* como um todo. Na Figura 128, a título de exemplo, vemos um excerto do ficheiro YAML de criação do *Deployment*, podendo constatar que foi adicionada uma linha com a respetiva *annotation*, neste caso informando o ACI que este deployment deverá ser associado ao EPG `myhero-data`.

```
---
apiVersion: v1
kind: Service
metadata:
  name: myhero-data
  labels:
    app: myhero
    tier: data
spec:
  ports:
  - port: 80
    targetPort: 5000
  selector:
    app: myhero
    tier: data
  type: NodePort
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: myhero-data
  annotations:
    opflex.cisco.com/endpoint-group: '{"tenant":"kubesbx04","app-profile":"myhero","name":"myhero-data"}'
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: myhero
        tier: data
```

Figura 129 - Annotation adicionado ao Deployment do myhero-data

Acedendo ao APIC é possível confirmar que os POD's foram colocados nos EPG's definidos nas *annotations* dos respetivos *deployments*. A título de exemplo vemos na Figura 130 os POD's associados ao EPG `myhero-ui`.

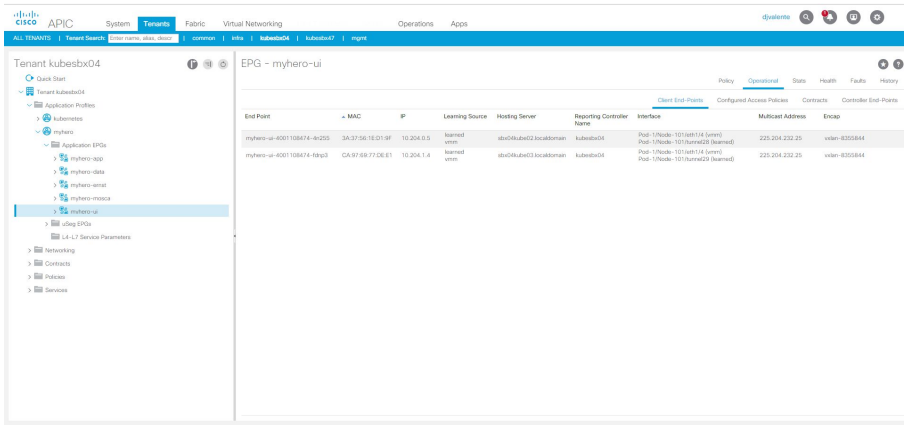


Figura 130 - POD's do EPG myhero-ui com isolamento por deployment

Conforme já vimos anteriormente, um dos pontos interessantes desta integração é a visibilidade que temos diretamente sobre os objetos do Kubernetes, que como vemos na Figura 131 vai bastante alem dos POD's.

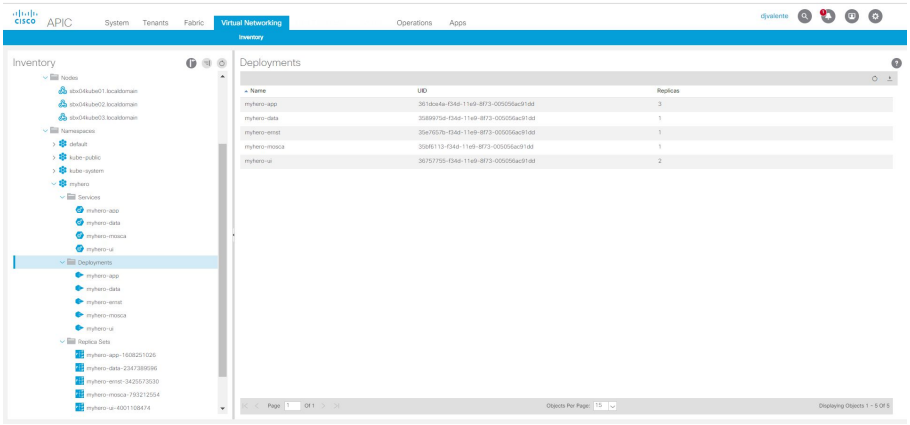


Figura 131 - Visibilidade do ACI sobre os objetos do Kubernetes

De forma a permitir a confirmação de que a política definida inicialmente no APIC está a ser corretamente aplicada à comunicação entre containers, vamos criar um novo container no EPG myhero-ui e daí tentar aceder às diferentes camadas aplicacionais.

Para criar o container de teste utilizamos o comando:

```
kubectl -n myhero run -i --tty pwndui --image=hpreston/devbox \
--overrides='{"apiVersion": "v1","metadata": {
        "annotations": {
            "opflex.cisco.com/endpoint-group":
            "{\"tenant\":\"kubesbxXX\",\"app-
            profile\":\"myhero\",\"name\":\"myhero-ui\"}"
        }
    }
}' --restart=Never --rm -- /bin/bash
```

Na Figura 132 confirmamos que o POD está efetivamente no EPG `myhero-ui`, e na Figura 133 temos a evidencia de que o tráfego direcionado para o POD `myhero-app` é permitido, mas se o destino for o POD `myhero-data` o tráfego é barrado.

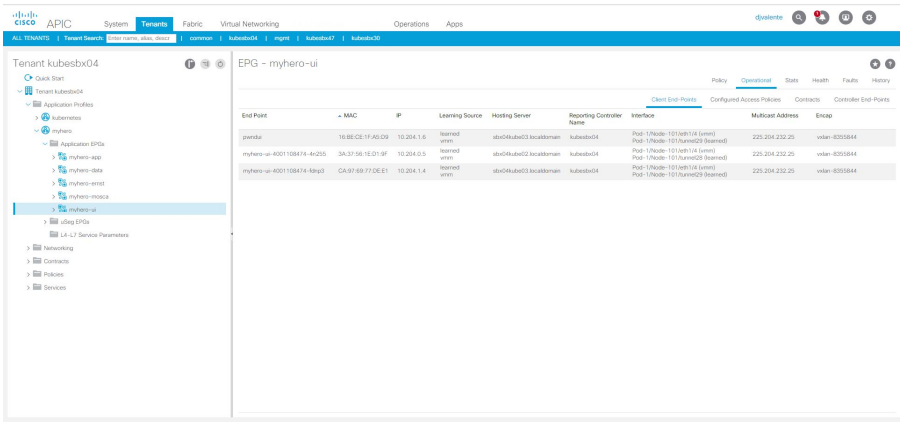


Figura 132 - Criação do container de teste da política baseada em deployment

```
[root@pwndui code]#
[root@pwndui code]#
[root@pwndui code]# curl -H "key: SecureApp" myhero-app/options
{
    "options": [
        "Captain Cloud",
        "Spider-Man",
        "Captain America",
        "Batman",
        "Superman",
        "Wonder Woman",
        "Deadpool",
        "Black Widow",
        "Iron Man",
        "Star-Lord",
        "Scarlet Witch",
        "Gamora"
    ]
}[root@pwndui code]#
[root@pwndui code]#
[root@pwndui code]# curl -H "key: SecureData" myhero-data/options
curl: (7) Failed connect to myhero-data:80; Connection timed out
[root@pwndui code]#
[root@pwndui code]#
```

Figura 133 - Validação da política de controlo por deployments

5.2. Validação da integração entre o ACI e a Cloud Azure

Conforme referimos no capítulo 4.3 a integração do ACI com a cloud Azure necessita de três componentes, conectividade entre os dois locais, um controlador para operar a componente Azure (cAPIC – Cloud APIC) e um orquestrador central (MSO – Multi-site Orchestrator)

5.2.1. Conectividade entre o centro de dados e o Azure.

No sentido de garantir que a conectividade está realmente funcional, podemos verificar no router localizado no datacenter que existe dois interfaces do tipo *tunnel* operacionais e que existem três adjacências de OSPF. A primeira, estabelecida através de uma interface local é precisamente com o segundo router que temos no centro de dados local, e as duas estabelecidas através das interfaces *tunnel* são adjacências com os routers virtuais criado em Azure.

```
CSRv1#sh ip int bri
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet1       10.240.21.73    YES NVRAM  up                    up
GigabitEthernet2       10.240.254.2    YES NVRAM  up                    up
GigabitEthernet3       192.168.3.1     YES NVRAM  up                    up
Loopback10             unassigned      YES unset  up                    up
Loopback101            unassigned      YES unset  up                    up
Loopback1000           88.157.227.103  YES NVRAM  up                    up
Loopback1094           111.111.111.100 YES NVRAM  up                    up
Tunnel1                33.3.1.2        YES NVRAM  up                    up
Tunnel2                33.3.1.6        YES NVRAM  up                    up
CSRv1#sh ip ospf neighbor

Neighbor ID     Pri   State           Dead Time   Address         Interface
1.1.1.1           0   FULL/  -        00:00:37    192.168.3.2     GigabitEthernet3
13.74.181.86      0   FULL/  -        00:00:36    33.3.1.5        Tunnel2
168.63.46.2       0   FULL/  -        00:00:36    33.3.1.1        Tunnel1
CSRv1#
```

Figura 134 - Validação do OSPF entre CSR's físicos e virtuais

No entanto a conectividade entre os dois mundos não é efetuada apenas com base nestes tuneis, pelo que é necessário garantir, conforme vimos na Figura 100, que o *Overlay* BGP-EVPN está operacional. Uma vez que o referido *Overlay*, do lado do centro de dados local termina já no ACI, é precisamente diretamente numa das suas *spine's* que conseguimos verificar o seu estado. A Figura 135 mostra o output do comando `sh bgp l2vpn evpn summary vrf all` executado na *spine* 2 do centro de dados 1 do laboratório.

```
SPINE201# show bgp l2vpn evpn summary  vrf all
BGP summary information for VRF overlay-1, address family L2VPN EVPN
BGP router identifier 192.168.1.101, local AS number 65500
BGP table version is 10318, L2VPN EVPN config peers 5, capable peers 3
151 network entries and 172 paths using 30808 bytes of memory
BGP attribute entries [14/2072], BGP AS path entries [0/0]
BGP community entries [0/0], BGP clusterlist entries [0/0]

Neighbor        V    AS MsgRcvd MsgSent   TblVer  InQ OutQ Up/Down  State/PfxRcd
22.2.0.1        4 65498       0       0        0    0    0     2w2d Idle
22.2.0.2        4 65498       0       0        0    0    0     2w2d Active
33.3.0.1        4 65499      12      12    10318    0    0 00:06:05 5
33.3.0.2        4 65499      12      12    10318    0    0 00:06:15 5
192.168.2.102   4 65500   23277   28808    10318    0    0     2w2d 26
SPINE201#
```

Figura 135 - Validação do Overlay BGP-EVPN

5.2.1. Cloud APIC (CAPIC)

O cAPIC é a componente responsável pela tradução entre ACI e o Azure. Na prática é uma appliance virtual com permissões de administração em toda a infraestrutura virtual integrada com o centro de dados local, permitindo que politicas criadas no MSO sejam traduzidas e aplicadas aos recursos de Azure. Conforme podemos verificar na Figura 136 através do cAPIC é possível efetuar várias ações, mas apenas de visualização e eventualmente resolução de problemas, toda a configuração será efetuada no MSO.

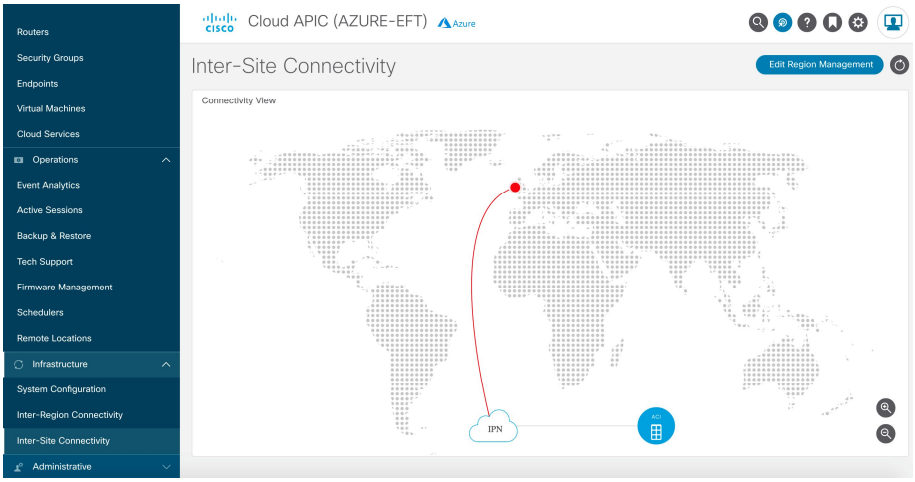


Figura 136 - Interface gráfico do cAPIC

5.2.2. Multi-site Orchestrator (MSO)

À semelhança das restantes componentes, o MSO disponibiliza igualmente uma interface gráfica que permitirá efetuar todas as configurações. A Figura 137 representa o painel de entrada do MSO.

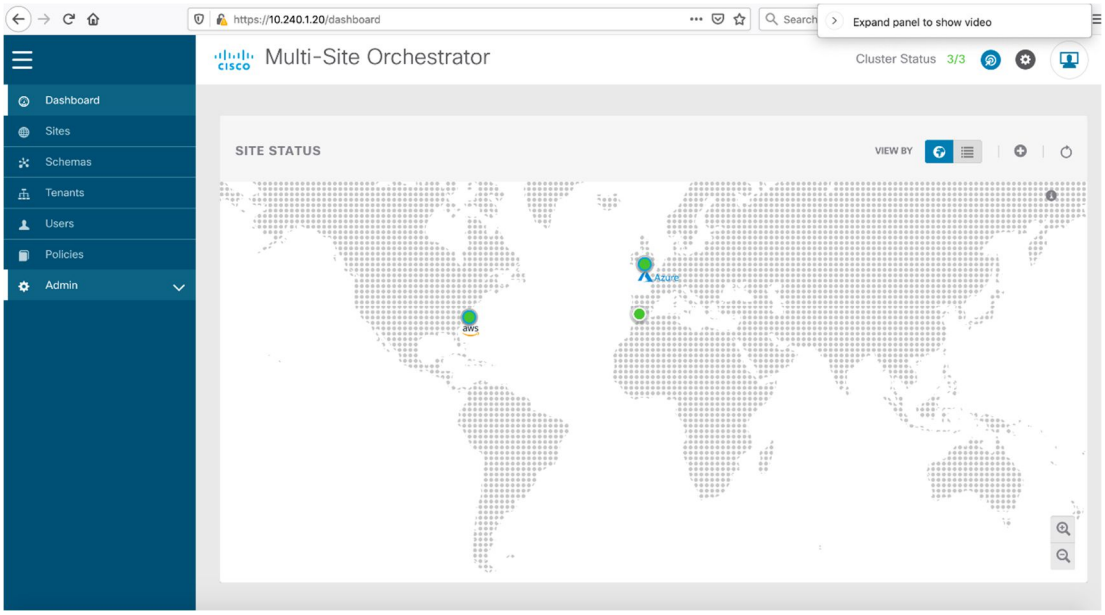


Figura 137 - Dashboard inicial do MSO

Para validarmos se o ACI, através do MSO consegue ter visibilidade sobre o que acontece em Azure, e consequentemente definir e aplicar políticas de acesso, criamos via MSO um EPG denominado SHARED_TEST2, associando-lhe vários *Selectors*, como vemos na Figura 138. Os *Selectors* são condições que vão ser usadas para mapear objetos de Azure aos EPG's do ACI. Importante referir que o nosso EPG, na configuração do MSO, está associado ao Azure.

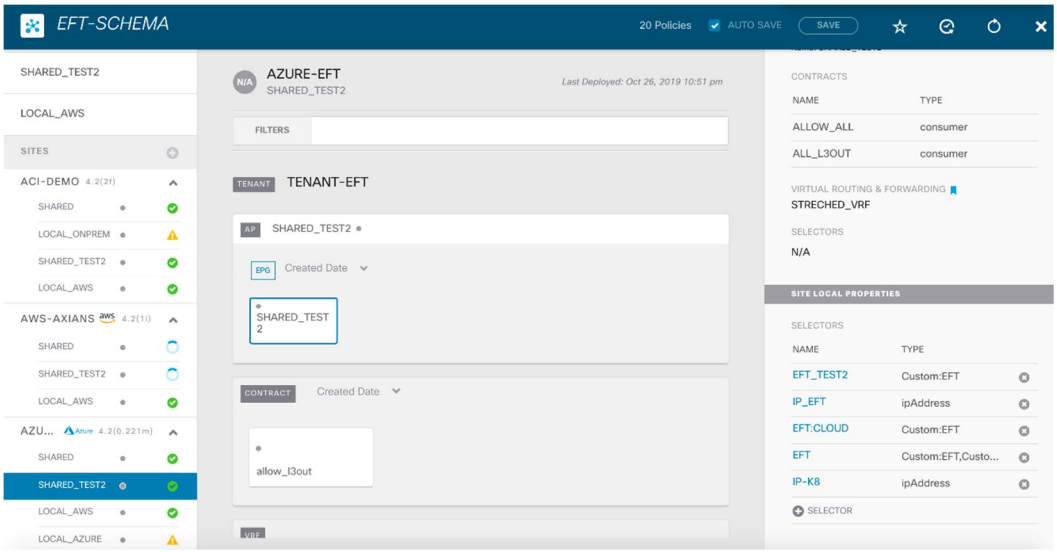


Figura 138 - MSO Criação de EPG

Ao fazermos esta configuração o ACI no centro de dados recebe via BGP a rede IP do lado do Azure, como vemos na Figura 139, garantindo desta forma a conectividade pretendida.

```
LEAF101# show ip route vrf TENANT-EFT:STRECHED_VRF
IP Route Table for VRF "TENANT-EFT:STRECHED_VRF"
'*' denotes best ucast next-hop
'**' denotes best mcast next-hop
'[x/y]' denotes [preference/metric]
'%<string>' in via output denotes VRF <string>

1.1.1.1/32, ubest/mbest: 2/0, attached, direct
    *via 1.1.1.1, lo5, [0/0], 02w01d, local, local
    *via 1.1.1.1, lo5, [0/0], 02w01d, direct
10.33.0.0/16, ubest/mbest: 2/0
    *via 33.3.0.2%overlay-1, [200/0], 01:17:18, bgp-65500, internal, tag 65499
    *via 33.3.0.1%overlay-1, [200/0], 01:17:18, bgp-65500, internal, tag 65499
10.33.1.4/32, ubest/mbest: 1/0
    *via 33.3.0.1%overlay-1, [200/0], 00:56:59, bgp-65500, internal, tag 65499
10.33.1.5/32, ubest/mbest: 1/0
    *via 33.3.0.2%overlay-1, [200/0], 00:02:21, bgp-65500, internal, tag 65499
10.44.44.0/24, ubest/mbest: 1/0, attached, direct, pervasive
    *via 10.1.160.66%overlay-1, [1/0], 02w02d, static
10.44.44.254/32, ubest/mbest: 1/0, attached, pervasive
    *via 10.44.44.254, vlan44, [0/0], 02w02d, local, local
192.168.100.0/24, ubest/mbest: 2/0
    *via 33.3.0.2%overlay-1, [200/0], 01:17:18, bgp-65500, internal, tag 65499
    *via 33.3.0.1%overlay-1, [200/0], 01:17:18, bgp-65500, internal, tag 65499
196.254.0.4/32, ubest/mbest: 1/0
    *via 33.3.0.2%overlay-1, [200/0], 00:02:21, bgp-65500, internal, tag 65499
196.254.0.8/32, ubest/mbest: 1/0
    *via 33.3.0.1%overlay-1, [200/0], 00:56:59, bgp-65500, internal, tag 65499
LEAF101#
```

Figura 139 - Rede Azure aprendida via BGP

O esperado é que ao associar ama uma máquina virtual em Azure uma marcação (*tag*) que corresponda a um dos *Selectors* associado a este EPG, o MSO consideraria essa máquina como fazendo parte deste mesmo EGP aplicando-lhe as políticas eventualmente definidas, que no caso de Azure se traduziriam em *Application Security Groups* e *Network Security Groups*. No entanto, apesar de termos verificado esse comportamento para máquinas virtuais, quando se trata de *containers*

criados em AKS, não é possível adicionar *tag's* diretamente aos *containers* o que complica a integração pretendida.

Com vista a contornar esta limitação, adicionamos ao EPG um novo *Selector* que associa ao EPG qualquer objeto de Azure baseado no seu IP. Ao colocarmos o IP propriamente dito do *container* criado dentro do AKS, como vemos na Figura 140, foi possível comunicar com o *container* (10.33.2.5) tendo como origem uma máquina alojada no centro de dados local (10.44.44.10).

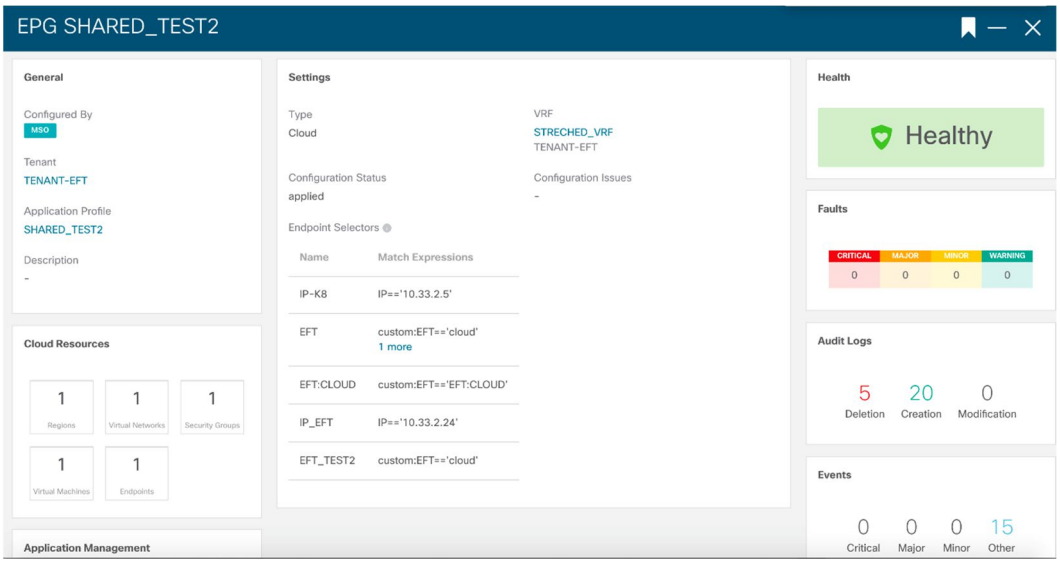


Figura 140 - Selectors criados no EPG

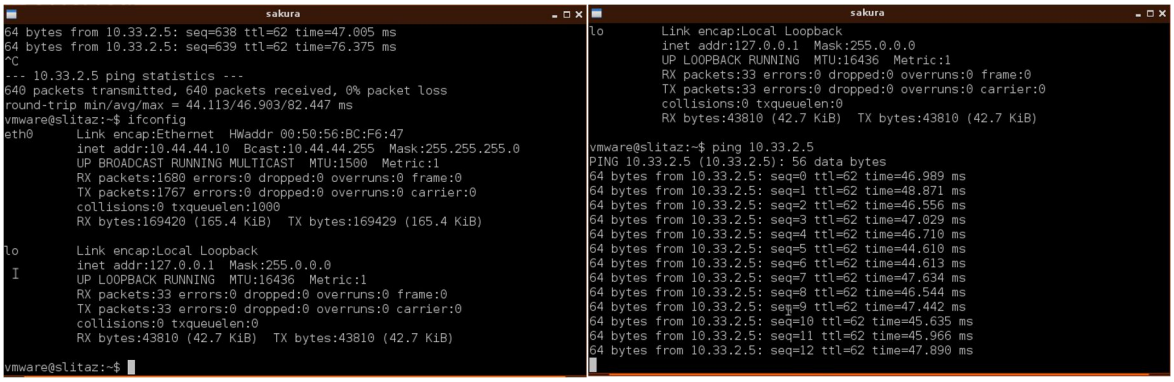


Figura 141 - Validação de conectividade

Analisando a informação no APIC do centro de dados local, verificamos por um lado que a nossa VM de testes pertence ao EPG pretendido (Figura 142) mas mais relevante é possível verificar que o endereço IP do *container* foi automaticamente adicionado ao EPG SHARED_TEST2 conforme pretendíamos (Figura 143), permitindo desta forma a comunicação entre os dois mundos, independentemente de se tratar de um máquina virtual ou de um *container*.

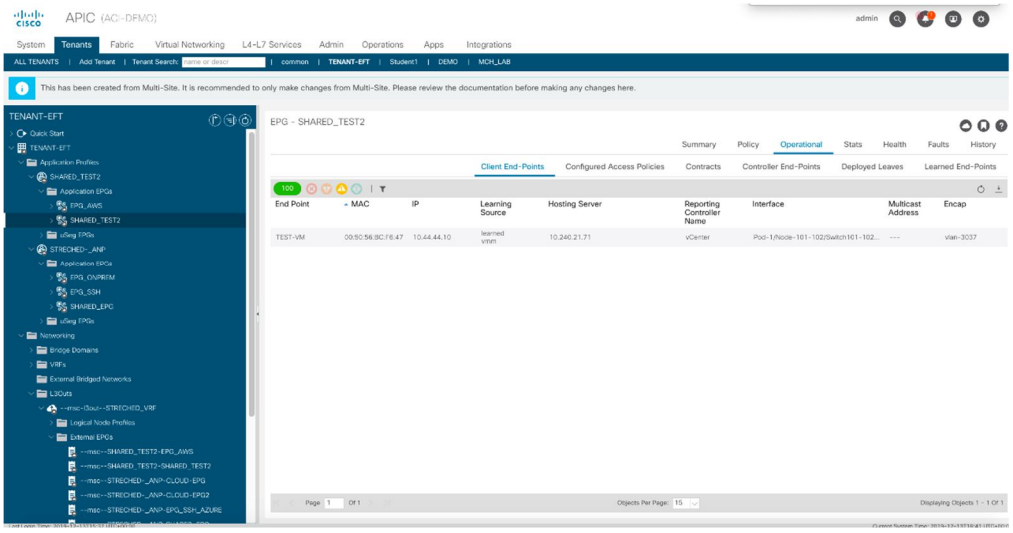


Figura 142 - Máquina virtual no EPG de SHARED_TEST2

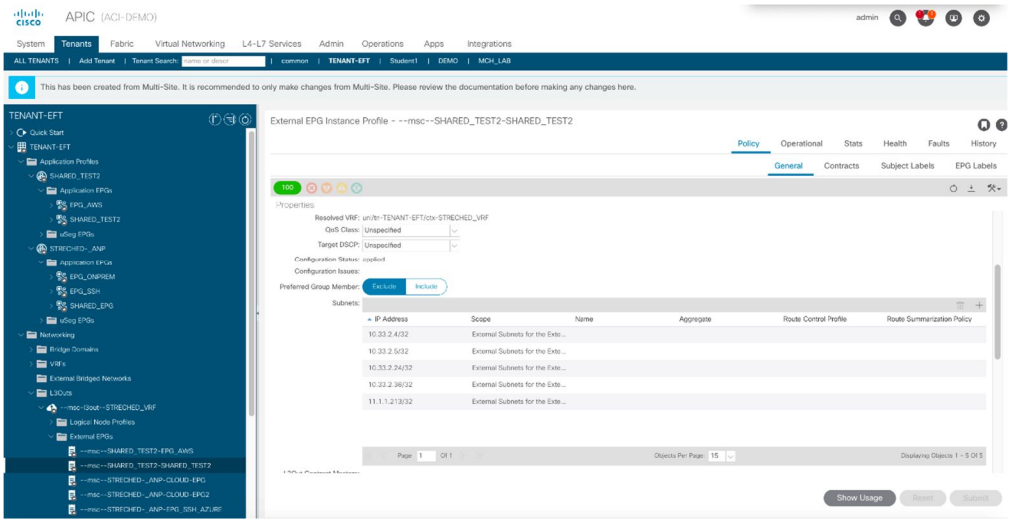


Figura 143 - Container no EPG SHARED_TEST2

O facto de verificarmos que é possível associar containers de AKS a EPG's específicos do ACI, confirma que será possível termos *containers* num cluster de Kubernetes local, a utilizar o ACI como balanceador de tráfego, e quando pretendido criar um novo *container* em AKS e associa-lo a qualquer componente da aplicação, visto que o endereçamento que o cliente final conhece é agnóstico da localização real dos *containers.*

6. CONCLUSÕES

Este capítulo apresenta um resumo do trabalho desenvolvido. Introduz o problema que pretendemos abordar, explicando a arquitetura implementada bem como os resultados obtidos em laboratório. Termina com a identificação de algumas questões que necessitam de maior desenvolvimento.

6.1. Resumo do trabalho

Por forma a conseguir desenvolver um trabalho relacionado com *containers*, foi necessário estudar a razão para a sua utilização e como se enquadram nas atuais metodologias de desenvolvimento aplicacional, nomeadamente a sua relação direta com arquiteturas baseadas em microserviços e arquiteturas nativas para cloud. Considerando que o desafio pressupunha a necessidade de executar *containers* num centro de dados local e também numa cloud pública, foi também necessário explorar os métodos existentes na cloud pública para executar *containers*, bem como a utilização e orquestração dos serviços adjacentes, nomeadamente armazenamento e balanceamento. Nas cloud's públicas, ao optarmos pelo método baseado no orquestrador Kubernetes, constatamos que toda a integração está já assegurada pelo provedor da cloud, no entanto no centro de dados local é necessário integrar com as tecnologias existentes.

Considerando que o ambiente da empresa proponente, e consequentemente a nossa bancada de testes, é composta por uma infraestrutura de rede Cisco ACI (SDN) e por balanceadores de carga F5, antes de validarmos a integração do centro de dados local com a cloud pública, foi necessário integrar a rede e os balanceadores no próprio Kubernetes local. Para isso foi necessário explorar os conceitos associados às SDN, mais especificamente soluções para centros de dados culminando no estudo da solução SDN do fabricante Cisco.

Após concluir a integração de todas as componentes locais, foi necessário interligar o centro de dados com a cloud Azure.

Nos capítulos iniciais apresentamos uma pequena descrição das principais soluções para a construção de arquiteturas híbridas, e na descrição da bancada de testes, descrevemos a solução adotada.

No capítulo referente aos testes é possível constatar a correta integração do Kubernetes com o balanceador de tráfego proposto, é também possível observar que o ACI consegue ter total visibilidade sobre os *containers* conseguindo estender a sua metodologia/politica para este novo tipo de *workload*.

Na componente de integração com a cloud Azure, apesar de verificarmos que a integração não é tão transparente para *containers* como para máquinas virtuais, é também possível estender os conceitos e a política de controlo de acessos criada no ACI para a cloud Azure, inclusivamente para *containers* criado em clusters geridos pelo próprio provedor de cloud.

6.2. Principais contribuições

A principal contribuição desta dissertação é a especificação e desenvolvimento de uma solução que permita dar resposta aos desafios que a empresa proponente identificou como referência para o sucesso do trabalho, permitindo aumentar a flexibilidade da solução de apoio ao processo de venda, e demonstrando que a conversão das aplicações para *containers* facilitará imenso a aceleração da movimentação de aplicações para a cloud.

De salientar que o trabalho desenvolvido de integração do ACI com clusters de Kubernetes no centro de dados local e consequente integração com a infraestrutura de um provedor de cloud pública, permitindo que de um único ponto se tenha visibilidade sobre todos os *workloads* e se consiga gerir e aplicar politicas independentemente do local ou do formato em que a aplicação é construída, irá também facilitar, do ponto de vista operacional, a adoção destes novas formas de "entregar" infraestrutura, uma vez que tendo conhecimento de gestão da solução Cisco ACI, é possível gerir ambientes de rede em Azure e ambientes de rede de Kubernetes, evitando que as equipas de infraestrutura necessitem de adquirir conhecimento profundo/especifico nas várias tecnologias que a organização decida adotar.

6.3. Trabalho Futuro

Apesar da empresa proponente planear para o ano de 2020 implementar todas as integrações descritas ao longo desta dissertação, ficam ainda alguns pontos por esclarecer, nomeadamente a integração com múltiplos fabricantes de *storage* e também a utilização da integração do F5 no modo *Cluster* quando o cluster já está integrado com o ACI, uma vez que são recomendados CNI's distintos.

6.3.1. Integração de storage através de CSI (Container Storage Instance)

Todas as soluções apontadas no capitulo 4.1.2 são soluções disponibilizadas diretamente pelo Kubernetes, isto é, são desenvolvidas, compiladas e distribuídas com os próprios binários do Kubernetes, o que condiciona eventuais adições de novas soluções, uma vez que obrigará a integrar o novo desenvolvimento no código base do Kubernetes.

Para agilizar este processo, adotou-se para a componente de armazenamento a mesma abordagem que já é utilizada para a componente de rede, tendo sido criados *plugins* que permitam

aos fabricantes desenvolver as suas próprias integrações, facilitando a adição de novas funcionalidades.

Atualmente é possível utilizar dois *plugins*:

- Container Storage Interface (CSI)
- Flexvolume

6.3.2. Integração de um cluster Kubernetes em simultaneo com F5 e ACI

Conforme vimos anteriormente a integração com ACI obriga à utilização do CNI específico da solução SDN, o ACI-CNI. A utilização deste CNI é um requisito para a integração da rede com o cluster. Por outro lado, a integração em modo *Cluster,* que permite um melhor balanceamento da carga, recomenda o CNI *Flannel* que apesar de ser baseado em tuneis VXLAN, capacidade que o ACI também tem, não existe à data de hoje uma abordagem conjunta igualmente suportada por ambos os fabricantes. Esta limitação, vai impedir a utilização do F5 como controlador do tráfego de *ingress* no cluster alojado no centro de dados local.

BIBLIOGRAFIA

[1] N. Dragoni, S. Giallorenzo e A. L. Lafuente, Microservices: Yesterday, Today, and Tomorrow., Springer, Cham, 2017.

[2] A. Balalaie, A. Heydarnoori e P. Jamshidi, Migrating to Cloud-Native Architectures Using Microservices: An Experience Report, Springer, Cham, 2016.

[3] T. Giovanni, B. Sandro , B. Martin , D. Florian e E. Andrew , "An architecture for self-managing microservices," *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud,* pp. 19-24, 2015.

[4] K. Hyojoon e F. Nick, "Improving network management with software defined networking," *IEEE Communications Magazine,* vol. V, nº 2, 2013.

[5] A. Lucien e P. Maurizio, The Policy Driven Data Center with ACI: Architecture, Concepts, and Methodology, Cisco Press, 2014.

[6] A. Saeed e H. Shahram, "Optimization of SDN Flow Operations in Multi-Failure Restoration Scenarios," *IEEE Transactions on Network and Service Management,* vol. 13, nº 3, 2016.

[7] M. Víctor, R. Omer, B. José e A. Unai, "Modelling Performance & Resource Management in Kubernetes," *IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC),* 2016.

[8] "addons," [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/addons/.

[9] "ACI Launch date," [Online]. Available: https://blogs.cisco.com/perspectives/happy-birthday-cisco-application-centric-infrastructure.

[10] J. v. d. Merwe, S. Rooney e S. Crosby, "The Tempest-a practical framework for network programmability," *IEEE Network,* vol. 12, nº 3, 1998.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson e J. Rexford, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Computer Communication Review,* vol. 38, nº 2, 2008.

[12] "TR10: Software-Defined Networkinge," [Online]. Available: https://www.technologyreview.com/s/412289/tr10-software-defined-networking/.

[13] "Nuage Networks recognized for Data Center Innovation of the Year in Telecom Asia Readers' Choice & Innovation Awards," [Online]. Available: https://www.nuagenetworks.net/newsroom/press-releases/year/2013/.

[14] "The History of NSX and the Future of Network Virtualization," [Online]. Available:

https://www.vmware.com/radius/history-nsx-future-network-virtualization/.

[15] B. Pfaff, J. Pettit e T. Koponen, Extending Networking into the Virtualization Layer, Duke University, 2014.

[16] “Open Network Operating System,” [Online]. Available: https://www.slideshare.net/umeshkrishnaswamy/open-network-operating-system.

[17] S. Venticinque, L. Tasquier e B. Martino, “A restfull interface for scalable agents based cloud services,” *International Journal of Ad Hoc and Ubiquitous Computing,* vol. 16, nº 4, 2014.

[18] A. Abdul e S. Kumar, “Study of Converged Infrastructure & Hyper Converge Infrastructre As Future of Data Centre,” *International Journal of Advanced Research in Computer Science,* vol. 8, nº 5, pp. 900-903, 2017.

[19] S. Weil, S. Brandt, E. Miller, D. Long e C. Maltzahn, “Ceph: a scalable, high-performance distributed file system,” *OSDI Proceedings of the 7th symposium on Operating systems design and implementation,* pp. 307-320, 2006.

[20] Nishtha e M. Sood, “Software defined network — Architectures,” International Conference on Parallel, Distributed and Grid Computing, 2014.

[21] “ACI Launch date,” [Online]. Available: https://blogs.cisco.com/perspectives/happy-birthday-cisco-application-centric-infrastructure .

[22] “OpFlex: An Open Policy Protocol White Paper,” [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731302.html .

[23] “Cisco Application Policy Infrastructure Controller, Release 1.0,” [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/1-x/release/notes/apic_rn_101.html .

[24] “Application Centric Infrastructure (ACI), the policy driven data centre,” [Online]. Available: https://pt.slideshare.net/CiscoCanada/application-centric-infrastructure-aci-the-policy-driven-data-centre.

[25] “Refreshing my memory on cisco ACI,” [Online]. Available: http://sateeshkolagani.com/?p=245 .

[26] Firefly, ACI Test Drive, Firefly, 2015.

[27] “Connect an on-premises network to Azure using a VPN gateway,” [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/hybrid-networking/vpn.

[28] “Hybrid cloud solutions,” [Online]. Available: https://azure.microsoft.com/en-in/solutions/hybrid-cloud-app/ .

[29] “Desenvolvimento ágil de software,” [Online]. Available: https://pt.wikipedia.org/wiki/Desenvolvimento_%C3%A1gil_de_software.

[30] “Measurement and Evaluation for Docker Container Networking,” International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2017.

[31] “Open Source Container Networking,” [Online]. Available: https://contiv.io .

[32] “ACI CNI,” [Online]. Available: https://www.cisco.com/c/en/us/td/docs/net_mgmt/cisco_container_platform/1-0/Installation_Guide/CCP-Installation-Guide-01/CCP-Installation-Guide-01_chapter_00.html#id_70491 .

[33] “Production-Grade Container Orchestration,” [Online]. Available: http://kubernetes.io.

[34] “O que é PaaS?,” [Online]. Available: https://azure.microsoft.com/pt-pt/overview/what-is-paas/.

[35] “O que é IaaS?,” [Online]. Available: https://azure.microsoft.com/pt-pt/overview/what-is-iaas/.

[36] Kubernetes, “Kubernetes Network Plugin,” [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.

[37] “Install kubectl on Windows,” [Online]. Available: https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-on-windows.

[38] “Storage Classes,” [Online]. Available: https://kubernetes.io/docs/concepts/storage/storage-classes/.

[39] “Dynamically create and use a persistent volume with Azure disks in Azure Kubernetes Service (AKS),” [Online]. Available: https://docs.microsoft.com/en-us/azure/aks/azure-disks-dynamic-pv.

[40] “K8S Getting started,” [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/.

[41] “Creating Highly Available clusters with kubeadm,” [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/.

[42] “Kubernetes Architecture,” [Online]. Available: https://mapr.com/blog/kubernetes-kafka-event-sourcing-architecture-patterns-and-use-case-examples/.

[43] “etcd,” [Online]. Available: https://etcd.io.

[44] “Weave Net,” [Online]. Available: https://www.weave.works/oss/net/.

[45] “BIG-IP Controller Modes,” [Online]. Available: https://clouddocs.f5.com/containers/v2/kubernetes/kctlr-modes.html.

[46] CoreOS, “Flannel,” [Online]. Available: https://coreos.com/flannel/docs/latest/.

[47] F5, “BIG-IP and flannel VXLAN Integration,” [Online]. Available: https://clouddocs.f5.com/containers/v2/kubernetes/flannel-bigip-info.html.

[48] “K8s Volumes,” [Online]. Available: https://kubernetes.io/docs/concepts/storage/volumes/.

[49] “Kubernetes CIFS Volume Driver,” [Online]. Available: https://github.com/juliohm1978/kubernetes-cifs-volumedriver.

[50] Cisco, “Virtual Machine Manager Domains,” [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/1-x/aci-fundamentals/b_ACI-Fundamentals/b_ACI-Fundamentals_chapter_01011.html.

[51] Cisco, “Cisco Cloud ACI on Microsoft Azure White Paper,” [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-742844.html.

[52] Cisco, “Implementing VXLAN in a Data Center,” [Online]. Available: https://www.ciscolive.com/c/dam/r/ciscolive/emea/docs/2016/pdf/LTRDCT-2223.pdf.

[53] “Repositories/Ubuntu,” [Online]. Available: https://help.ubuntu.com/community/Repositories/Ubuntu.

[54] “Install and Set Up kubectl,” [Online]. Available: https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl.

[55] “Cisco ACI Multi-Site Architecture White Paper,” [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-739609.html .

[56] “Cisco Application Centric Infrastructure Multipod Configuration White Paper,” [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-739714.html .

[57] Y. Rekhter, “RFC1918,” Cisco Systems, February 1996. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1918.txt.

[58] J. Case, “RFC1098 - A Simple Network Management Protocol (SNMP),” University of Tennessee at Knoxville, April 1989. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1098.txt.

[59] E. R. Enns, “RFC6241 - Network Configuration Protocol (NETCONF),” Juniper Networks,

June 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6241.txt.

[60] M. Mahalingam, “RFc7348 - rtual eXtensible Local Area Network (VXLAN),” Cumulus Networks, August 2014. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7348.txt.

7. ANEXOS

7.1. Instalar um cluster K8S local com um nó usando o Minikube

A forma mais fácil e mais rápida de termos um cluster Kubernetes operacional é utilizando o Minikube. O Minikube é uma ferramenta que representa um cluster de apenas um nó, que na prática é excelente para desenvolvimento e/ou testes.

7.1.1. Instalar o Minikube

A melhor forma de instalar o Minikube é descarregar uma máquina virtual que vai funcionar como um nó do cluster. Como pretendemos um ambiente de testes controlado já estamos a utilizar uma máquina virtual com o sistema operativo Ubuntu, pelo que teremos de ativar a possibilidade de virtualização dentro desta máquina. Só assim conseguiremos ter o Minikube a correr através do VirtualBox dentro da nossa máquina virtual com o sistema operativo Ubuntu.

Instalar o Virtualbox

Para instalar a última versão do Virtualbox através da linha de comandos, necessitamos de ativar o repositório *Multiverse*, de forma a permitir instalar aplicativos sujeitos a direitos de utilização ou restrições legais [53], para isso usamos o seguinte comando:

```
$ sudo add-apt-repository multiverse
```

Com este repositório ativo, já é possível instalar o Virtualbox normalmente:

```
$ sudo apt install virtualbox virtualbox-ext-pack
```

Ativar a virtualização na máquina virtual

Tendo em conta que estamos a utilizar uma máquina virtual baseada em *VMware Workstaion*, temos de assegurar que a funcionalidade *Virtualize Intel V-x/EPT* se encontra ativa nas definições da máquina. É importante também garantir que temos os recursos necessário, uma vez que o minikube vai "lançar" uma máquina virtual com dois CPU's e 2048M de memória pelo que no nosso caso parametrizamos a nossa máquina de base para o dobro destes recursos.

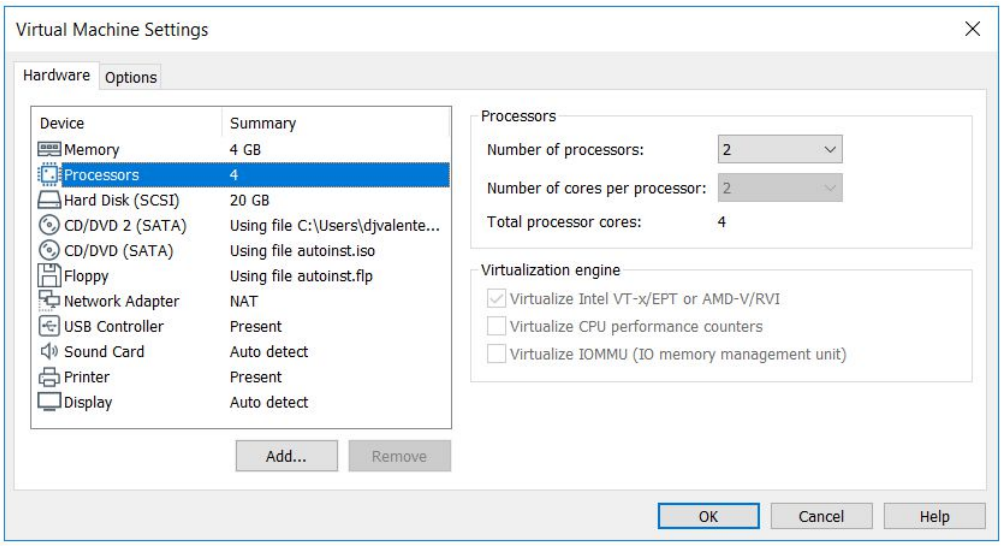


Figura 144 - Definições de uma máquina virtual

Instalar o Kubectl

O *kubectl* é uma ferramenta baseada em linha de comandos que permite gerir e instalar aplicações em clusters Kubernetes. A sua instalação necessita apenas de três passos [54]:

- Validar o certificado SSL da origem da ferramenta

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg
| sudo apt-key add -
```

- Adicionar o repositório correto

```
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" |
sudo tee -a /etc/apt/sources.list.d/kubernetes.list
```

- Finalmente instalar através do comando apt-get

```
sudo apt-get update
sudo apt-get install -y kubectl
```

Descarregar e arrancar o Minikube

O Minikube tem de ser descarregado e colocado numa pasta que permita a sua execução de qualquer diretório, pelo que teremos que o mover para a `$PATH`.

```
$ wget
https://storage.googleapis.com/minikube/releases/latest/minikub
e-linux-amd64
$ chmod +x minikube-linux-amd64
$ sudo mv minikube-linux-amd64 /usr/local/bin/minikube
```

Finalmente corremos o comando `minikube start`. Este comando vai despoletar o *download* da máquina virtual que posteriormente será o nó do cluster de Kubernetes.

```
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$ minikube status

!   Error getting host status: state: machine does not exist

*   Sorry that minikube crashed. If this was unexpected, we would love to hear from you:
-   https://github.com/kubernetes/minikube/issues/new
kubernetes@ubuntukubprtscr:~$ minikube delete
x   Deleting "minikube" from virtualbox ...
-   The "minikube" cluster has been deleted.
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$ minikube start
o   minikube v1.0.0 on linux (amd64)
$   Downloading Kubernetes v1.14.0 images in the background ...
>   Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
-   "minikube" IP address is 192.168.99.100
-   Configuring Docker as the container runtime ...
-   Version of container runtime is 18.06.2-ce
:   Waiting for image downloads to complete ...
-   Preparing Kubernetes environment ...
@   Downloading kubeadm v1.14.0
@   Downloading kubelet v1.14.0
-   Pulling images required by Kubernetes v1.14.0 ...
-   Launching Kubernetes v1.14.0 using kubeadm ...
:   Waiting for pods: apiserver proxy etcd scheduler controller dns
-   Configuring cluster permissions ...
-   Verifying component health .....
+   kubectl is now configured to use "minikube"
=   Done! Thank you for using minikube!
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$
```

Figura 145 - Informação referente ao arranque do Minikube

Como podemos ver na Figura 145, após conclusão do processo podemos confirmar a disponibilidade do cluster com o comando `kubectl cluster-info`. Este comando exibe também os URL's de alguns componentes do cluster, nomeadamente o da API do Master.

Parametrizações Linux adicionais

Tendo em conta que a grande maioria das ações que vamos necessitar de executar, serão exatamente com o comando kubectl, podemos configurar as seguintes facilidades:

- Adicionar um alias – editar o ficheiro `~/.bashrc` e adicionar a linha `alias k=kubectl`
- Adicionar a possibilidade de completar comandos e nome de objetos com a tecla <TAB> - adicionar ao mesmo ficheiro a linha `source <(kubectl completion bash)`
- Para podermos combinar as duas facilidades mencionadas acima, isto é, utilizar o alias "k" e o usar o <TAB> para completar comandos temos de alterar o output do comando `kubectl completion`, adicionando também a linha `source <(kubectl completion bash | sed s/kubectl/k/g)`.

Para que estas alterações surtam efeito temos de executar o comando `source ~/.bashrc`.

7.1.2. Lançar uma aplicação no cluster através de comando Imperativos

O primeiro modo de gerir objetos num cluster que vamos descrever é o modo imperativo, que na prática é um modo que através de comandos `kubectl` operamos diretamente sobre os objetos. Este modo tem a vantagem de ser mais simples e de efetuar mudanças imediatas aos objetos do

cluster. Como desvantagens este modo não tem qualquer registo sobre as alterações, o que inviabiliza qualquer controlo de versões ou auditoria.

Para criarmos um container com a imagem que criamos, e guardamos no DockerHub, basta executar o comando:

```
kubectl run djv --image=djvalente/testes --port=88 --
generator=run-pod/v1
```

Este comando cria um container com base na imagem `djvalente/testes`, informando o Kubernetes de que a aplicação está à espera de pedidos na porta 88.

Como existe um *Overlay* de rede para os containers, é necessário configurar um ponto de ligação entre a rede IP que conhecemos e o próprio container, esta operação é conhecida como expor o container e consegue-se através do comando:

```
kubectl expose pod djv --type=NodePort --name djv-http
```

Para vermos o resultado dos comandos anteriores podemos executar o comando o `kubectl get services` que lista o detalhe dos serviços expostos no cluster.

```
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$ kubectl get services
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
djv-http     NodePort    10.108.215.255   <none>        88:30035/TCP   23h
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP        25h
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$
```

Figura 146 - Listagem de serviços expostos num cluster Kubernetes

Como podemos ver na Figura 146, o serviço que expusemos `djv-http` ainda não recebeu um endereço IP externo, esta situação deve-se ao facto de estarmos a utilizar o minikube, num cluster normal o serviço só estaria disponível assim que fosse possível ver o endereço neste output.

No minikube temos um outro comando que nos permite "descobrir" o endereço que podemos utilizar para testar a nossa aplicação `minikube service djv-http --url`.

```
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$ minikube service djv-http --url
http://192.168.99.100:30035
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$
kubernetes@ubuntukubprtscr:~$ ifconfig vboxnet0
vboxnet0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.99.1  netmask 255.255.255.0  broadcast 192.168.99.255
        inet6 fe80::800:27ff:fe00:0  prefixlen 64  scopeid 0x20<link>
        ether 0a:00:27:00:00:00  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 439  bytes 188146 (188.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

kubernetes@ubuntukubprtscr:~$
```

Figura 147 - Endereço externo do container e endereço de rede

Na Figura 147 além de podermos ver o resultado do comando mencionado acima, é também possível validar a que rede IP pertence o endereço.

Na prática o que está a acontecer é um mapeamento entre um endereço IP da rede do Virtualbox (192.168.99.100) e um outro endereço interno do cluster (10.108.215.255).

Verificar os Pod's

Como já descrevemos anteriormente, em Kubernetes não trabalhamos diretamente com containers, pelo que não é possível listar os containers existentes num cluster, é, no entanto, possível listar os pod's com o comando `kubectl get pods`.



Figura 148 - Output do comando kubectl get pods

Para obter mais informações utilizamos o comando `kubectl describe pod`.

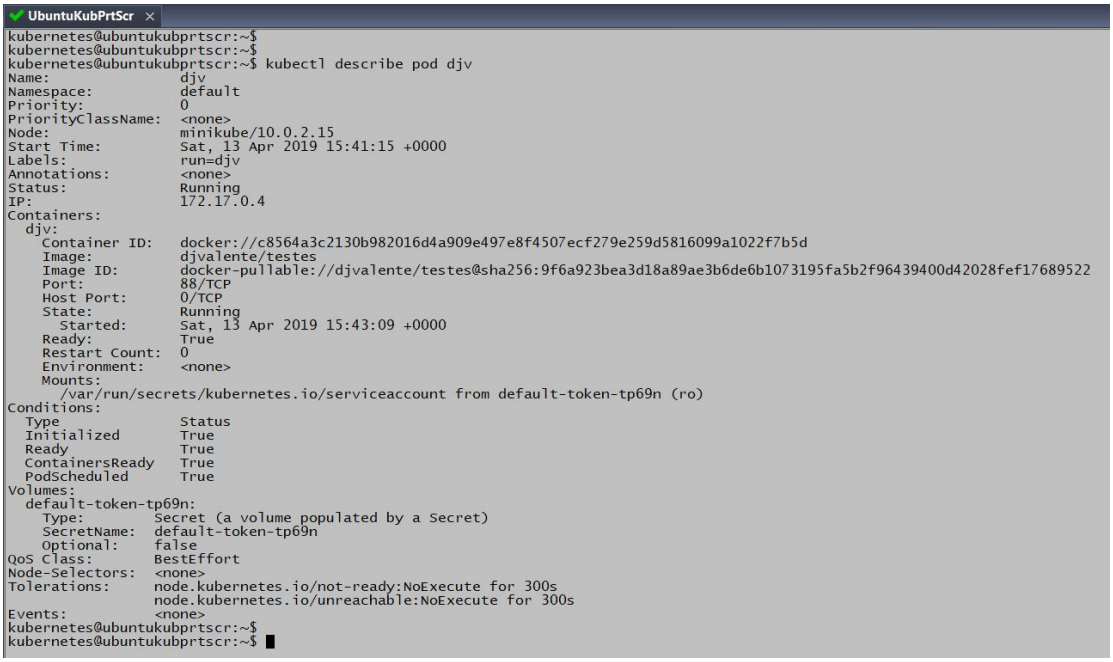


Figura 149 - Output do comando kubectl describe pod

7.1.3. Executar uma aplicação no cluster através de objetos declarativos

No modo de configuração através de objetos imperativos usamos igualmente a ferramenta `kubectl` com o parâmetro `create`, `replace`, `delete`, etc., passando como parâmetro pelo menos um ficheiro YAML (Yet Another Markup Language) ou JSON (JavaScript Object Notation). Este ficheiro terá de conter a definição completa dos objetos que pretendemos criar.

Como principal vantagem quando comparado com o método anterior temos, a possibilidade de recorrer a ferramentas de controlo de versões (Git) para gerir as configurações, podemos integrar com processos de revisão das configurações antes da aplicação em ambientes produtivos e podemos utilizar *templates*.

O formato mais utilizado em Kubernetes é o YAML razão pela qual é exatamente o que adotamos neste exemplo. Recorrendo a este método, a criação do container de testes passa apenas pela criação de um ficheiro denominada `djv1.yaml` com o conteúdo visível na Figura 150 e pela execução do comando `kubectl create -f djv1.yaml`.

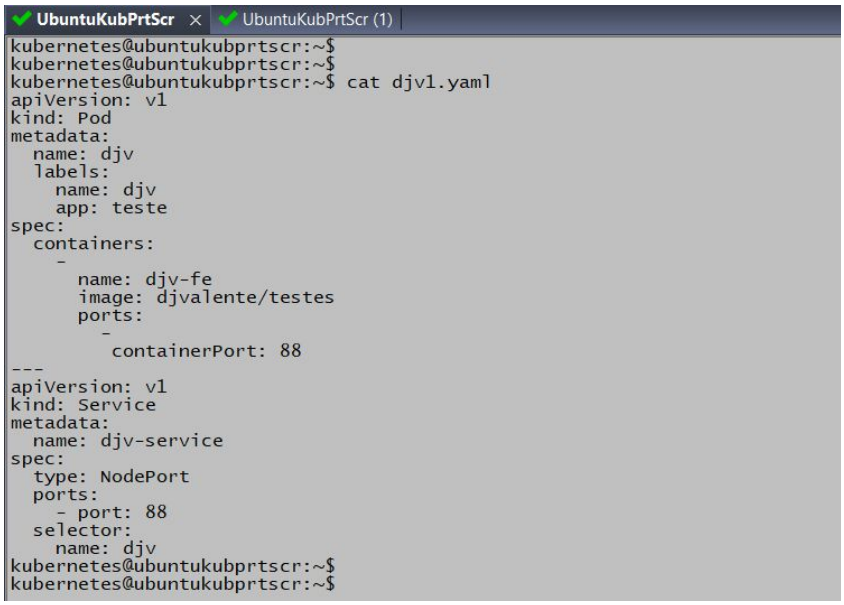


Figura 150 - Exemplo 1 de um ficheiro YAML

Ao analisarmos o descritivo criado, facilmente reparamos que é composto por duas partes distintas, a primeira descreve a informação necessária à criação do *pod* propriamente dito, e a segunda às definições relativas à exposição da aplicação para fora do cluster.

Conforme vemos na Figura 150 definimos a porta em que o serviço é disponibilizado tanto no container (`ContainerPort`) como no serviço (`port`). Incluir a porta na definição do container é meramente informacional, se removermos esta definição do ficheiro YAML não causa qualquer impacto, no entanto é considerado uma boa prática este registo, uma vez que facilita posteriores analises, dando uma visibilidade clara de quais as portas que o POD tem expostas.

7.1.4. Escalar uma aplicação

Uma das grandes vantagens de trabalharmos com containers e Kubernetes é a facilidade com que conseguimos escalar a nossa solução horizontalmente, isto é, conseguimos definir o número de replicas que pretendemos que determinado container tenha dentro do cluster. Ao expor o endereço do "deployment" e não do nó onde a aplicação se encontra conseguimos não só balancear o tráfego entre

todas as replicas, como conseguir uma abstração relativa ao nó físico onde a aplicação se encontra. Esta funcionalidade é particularmente útil em casos de falha de hardware. Na prática o Kubernetes vai garantir a existência de um determinado número de replicas do container, independentemente do número de nós físicos disponíveis no cluster.

Conforme podemos ver na Figura 151 na componente de criação do container adicionamos a informação `spec.replicas: 2` definindo que pretendemos dois pod's distintos a executar o container, e na criação do serviço através da definição `spec.selector.app: testes` expomos o *deployment*.

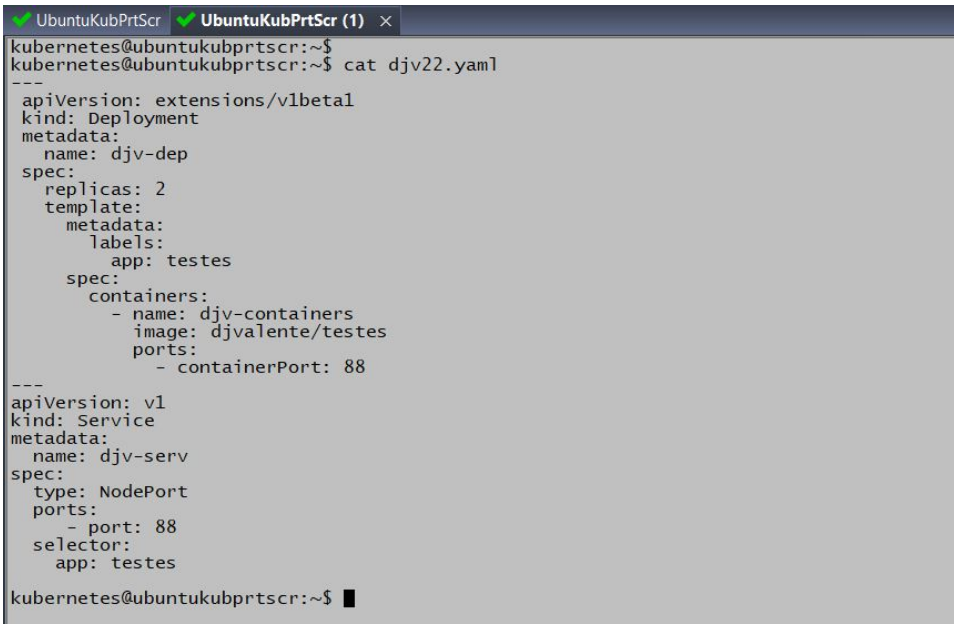


Figura 151 - Ficheiro YAML de uma solução com replicas

Importante salientar que esta configuração permite um balanceamento de carga automático, conforme vemos na Figura 152.



Figura 152 - Demonstração do balanceamento automático