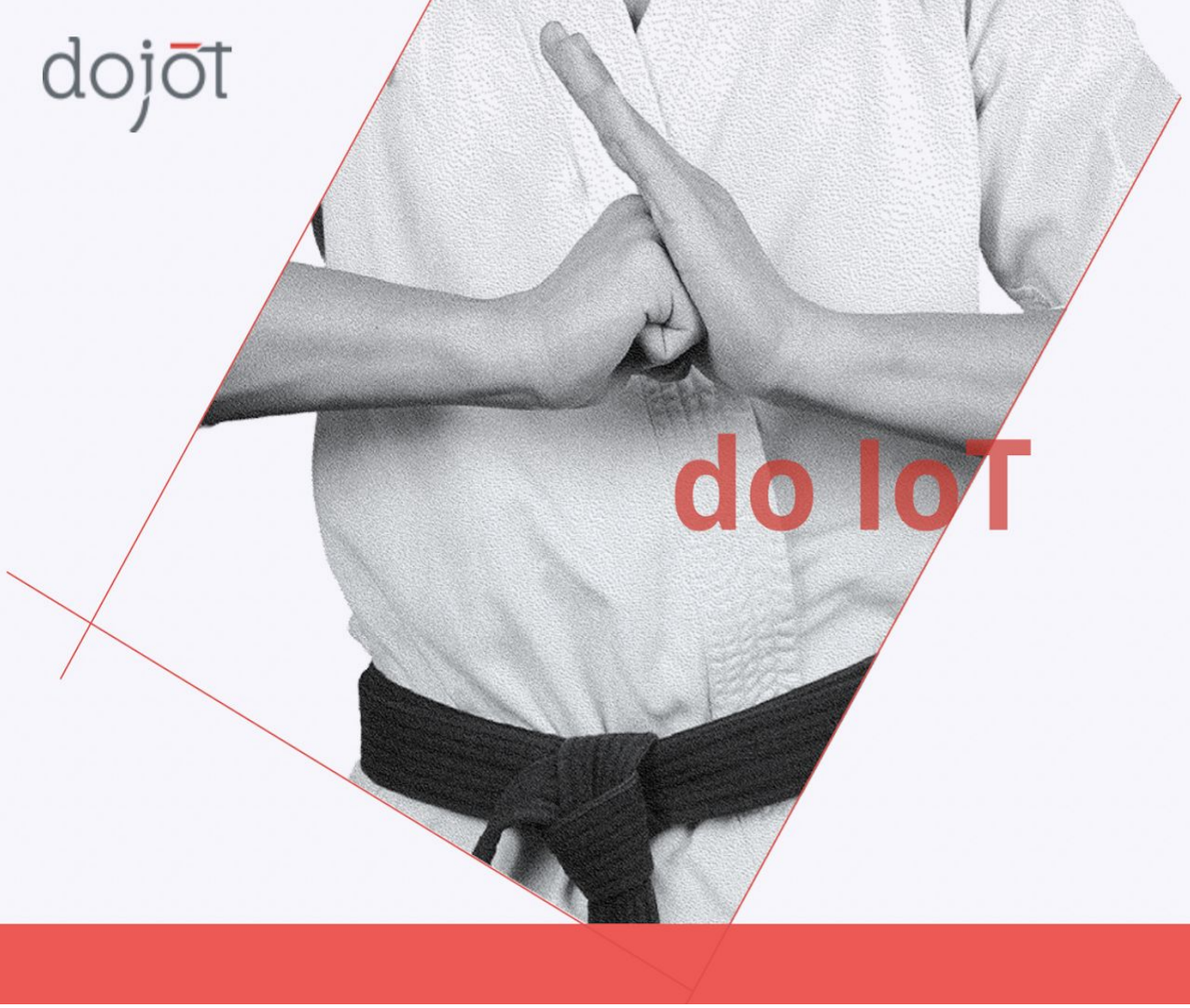


Dojot v0.5.0

Guia de Instalação

Instalando a dojot no Kubernetes

Última modificação: 24/09/2020

Introdução

Este documento aborda a configuração da dojot v0.5.0 no Kubernetes utilizando um balanceador de carga Layer 4. Para utilizar a dojot em um ambiente que suporte até 100.000 dispositivos conectados via MQTT, utilizamos o HAProxy como balanceador de carga.

Para executar a dojot em um ambiente que suporte poucos dispositivos ou LWM2M, utilizamos o NGINX como balanceador de carga. Atualmente não é possível executar a Dojot em um ambiente que suporte até 100.000 de dispositivos utilizando o NGINX, assim como não é possível utilizar dispositivos LWM2M utilizando o HAProxy.

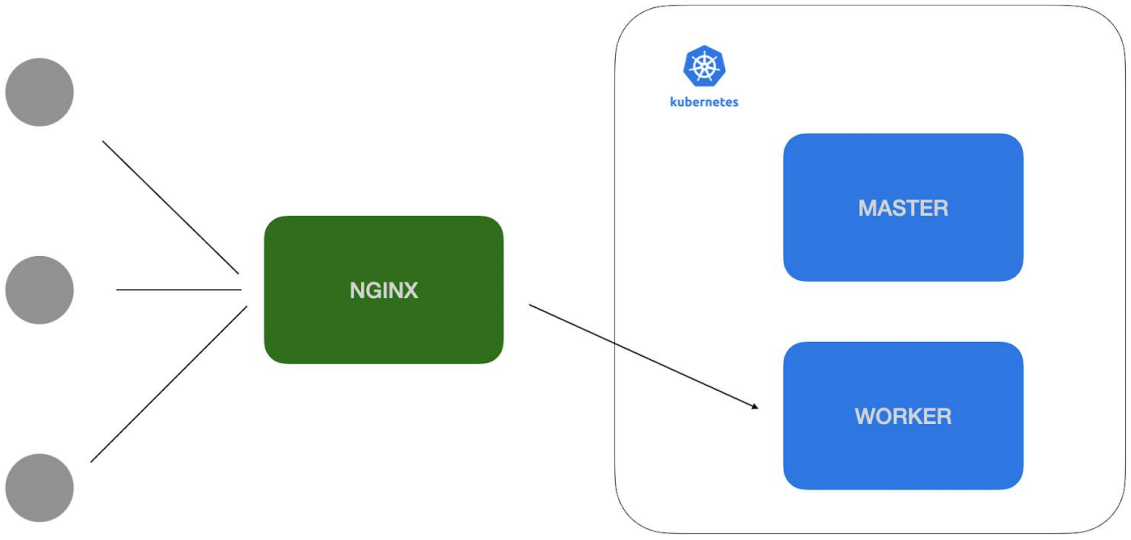
No ambiente que suporta poucos dispositivos, realizamos testes simulando 500 dispositivos enviando mensagens de 100B para dojot a cada 15 segundos. Sendo assim, essa é a definição de “poucos dispositivos” citada no documento.

No ambiente que suporta até 100.000 dispositivos conectados, a dojot não processa e nem armazena as mensagens enviadas pelos dispositivos, já esse é um ambiente utilizado somente para testes de carga e nem todos componentes da dojot são disponibilizados. Já em um ambiente com poucos dispositivos, todos os componentes da dojot são disponibilizados e as mensagens são processadas e armazenadas.

Ambiente

Para um ambiente com poucos dispositivos

Para utilizar a Dojot com poucos dispositivos conectados, podemos levar em conta o seguinte ambiente:



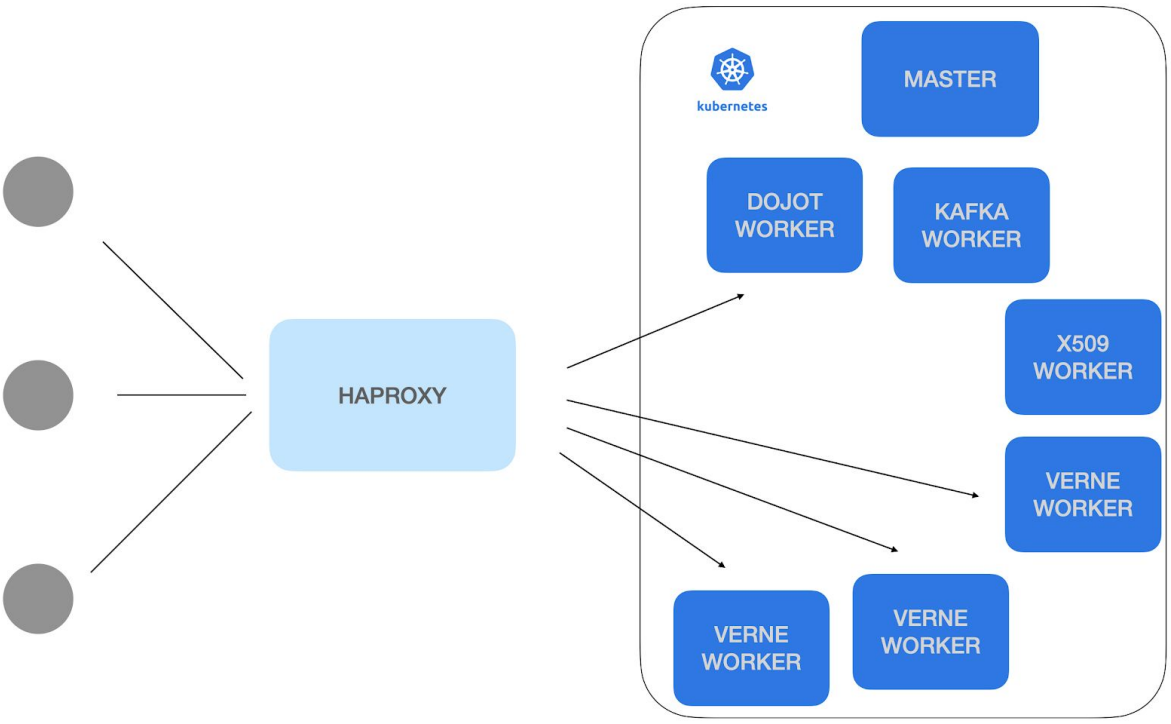
Na imagem acima vemos 3 nós, que podem ser máquinas virtuais ou físicas:

- **NGINX**: Load balancer layer 4. Todas as requisições TCP e UDP passam por ele e são redirecionadas para os workers do kubernetes. Nesse nó temos somente o NGINX instalado. Ele possui regras de balanceamento TCP e UDP e todos os dispositivos e clientes devem se conectar ao NGINX para acessar a dojot.
- **K8s Master**: Nó master do Kubernetes. Nele temos o *control plane* que administra o nosso cluster kubernetes. Não temos nenhum serviço da dojot sendo executado nesse nó. Sua única responsabilidade é administrar o cluster kubernetes .
- **K8s Worker**: Nó worker do kubernetes. Todos os serviços da dojot são executados nesse nó. Como veremos mais adiante, para um ambiente onde há uma carga maior de dispositivos, é necessária a utilização de mais nós no cluster k8s. Para uma maior disponibilidade, também é recomendado no mínimo 2 workers.

É importante lembrar que mesmo para um ambiente com poucos dispositivos, é de extrema importância o mapeamento de volumes no cluster em um ambiente de produção, para que, em caso de falhas, o sistema consiga manter o estado e os dados dos contêineres.

Para um ambiente com até 100.000 dispositivos

Para utilizar a Dojot com até 100.000 dispositivos conectados, podemos levar em conta o seguinte ambiente:



Na imagem acima vemos 8 nós, que podem ser máquinas virtuais ou físicas:

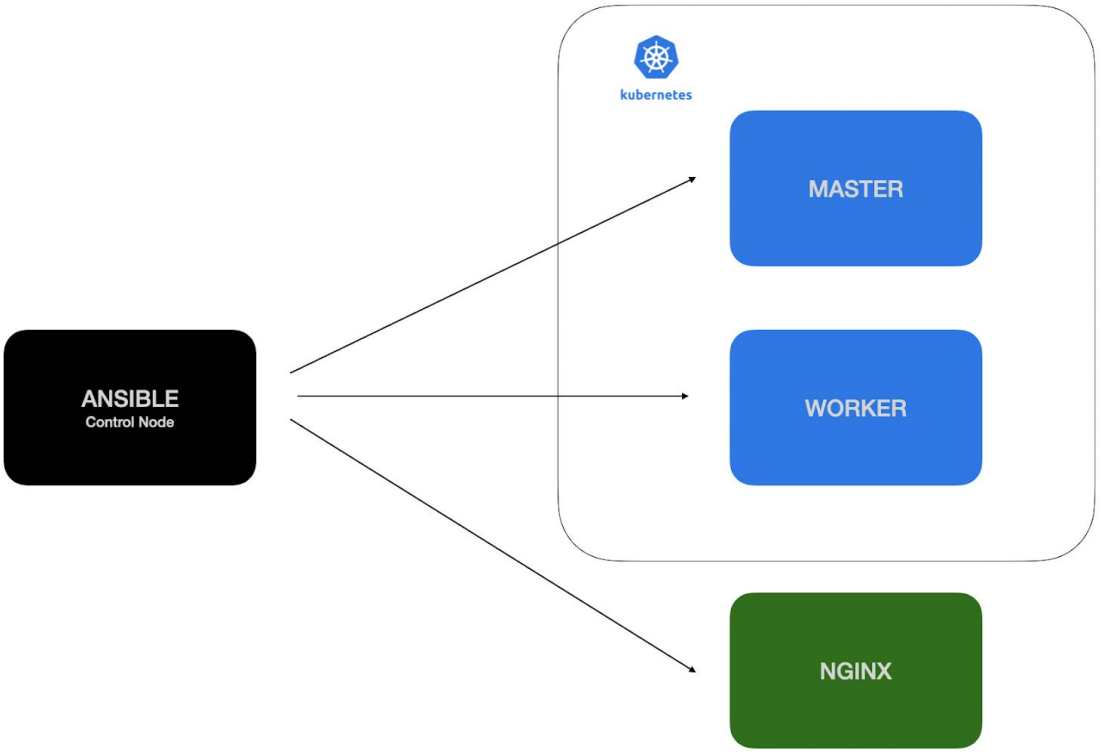
- **HAPROXY**: Load balancer layer 4. Todas as requisições TCP passam por ele e são redirecionadas para os workers do kubernetes. Nesse nó temos somente o HAPROXY instalado. Ele possui regras de balanceamento TCP e todos os dispositivos e clientes devem se conectar ao HAPROXY para acessar a dojot.
- **K8s Master**: Nó master do Kubernetes. Nele temos o *control plane* que administra o nosso cluster kubernetes. Não temos nenhum serviço da dojot sendo executado nesse nó. Sua única responsabilidade é administrar o cluster kubernetes .
- **Dojot Worker**: Nó worker do kubernetes onde alguns serviços da dojot são executados.
- **Kafka Worker**: Nó worker onde o Apache Kafka e os componentes relacionados com esse serviço são executados. Ele fica em um nó separado para que tenha o uso exclusivo de recursos desse nó, otimizando a performance. Assim é possível utilizar um hardware com IO otimizado para esse nó.
- **X509 Worker**: Nó worker onde o componente X509 será executado. Ele fica em um nó separado para que tenha o uso exclusivo de recursos do host, otimizando a performance.
- **Verne Workers:** Nós workers onde os Iot Agents com VerneMQ são executados. Eles também ficam em nós separados para que possam ter um uso exclusivo de recursos, otimizando a performance. O tráfego de rede, uso de CPU e memória desses nós é intenso. Além disso, outras configurações de redirecionamentos de pacotes de rede são utilizadas nesses nós visando otimizar a performance.

Considerações gerais

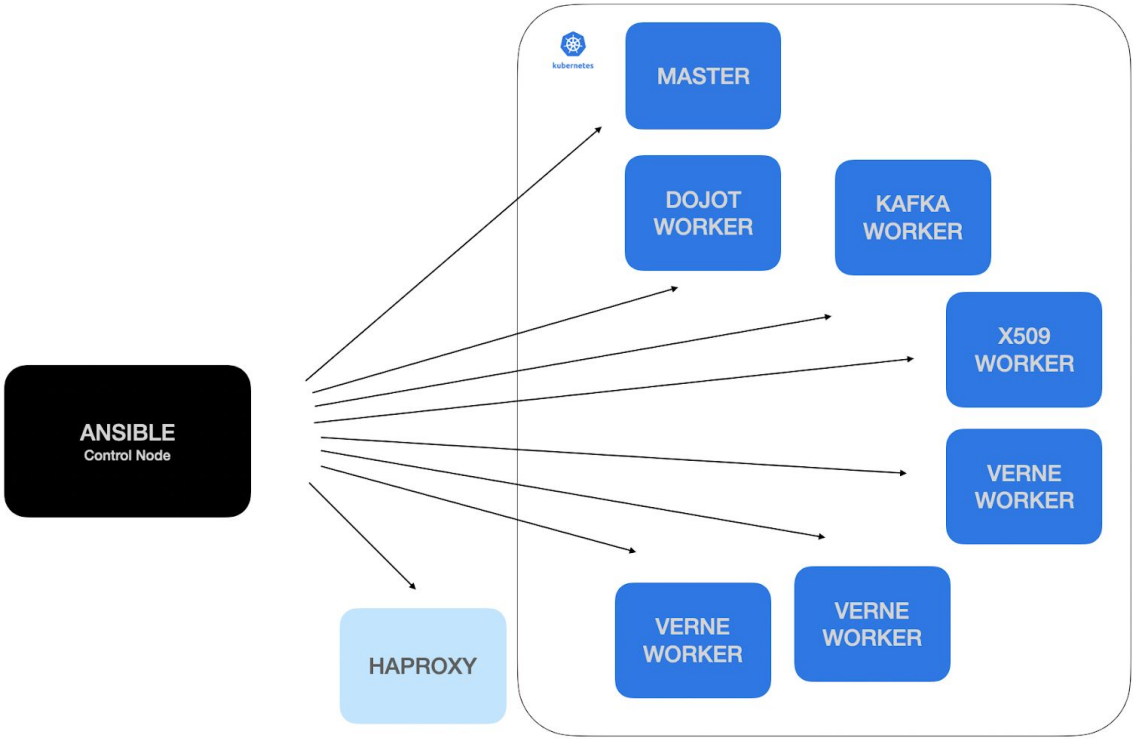
Atualmente temos playbooks do Ansible para automatizar a configuração dos ambientes. Para isso assumimos ter os requisitos de hardware e software atendidos. As máquinas precisam ser acessíveis por SSH para que os comandos sejam executados pelo Ansible.

Mesmo sendo possível executar os playbooks de dentro do cluster k8s, vamos abordar nesse documento a execução de fora do cluster.

Abaixo um exemplo para o ambiente que suporta poucos dispositivos:



E um ambiente com suporte para até 100.000 dispositivos conectados:



Considerando os modelos acima, temos uma máquina com o Ansible instalado, chamada "control node", que podemos utilizar para realizar as instalações e configurações dos nós *(atualmente não é possível utilizar o MS Windows como control node)*. O ansible então, através dos playbooks criados pela equipe da dojot, poderá realizar a instalação e configuração do NGINX/HAProxy e k8s, assim como o deploy completo da dojot. Os playbooks se encontram no repositório [dojot/ansible-dojot](). Esse repositório é versionado e temos os playbooks adequados para cada versão da dojot.

Requisitos de hardware e software

Para um ambiente com poucos dispositivos

Todos os testes foram executados no Ubuntu 18.04 (bionic 64).

Software:

- Instalado previamente pelo usuário (Control Node):
  - Ubuntu 18.04 LTS
  - Ansible 2.9.6 (Recomendamos a instalação com pip3);
  - sshpass;
  - GIT.

- Instalado juntamente com o deploy da Dojot:
  - Kubernetes: Kubeadm 1.17;

    - Docker: Docker CE 19.03.4;
    - Balanceador de carga L4: NGINX Open Source 1.8.

Hardware:

- NGINX: 1 core de processamento (2.2GHZ), 1GB de RAM e 20GB de disco;
- Master Node: 2 cores de processamento (2.2GHZ), 2GB de RAM e 30GB de disco;
- Worker Node: 4 cores de processamento (2.2GHZ), 6GB de RAM e 40GB de disco;

Para um ambiente com até 100.000 dispositivos

Todos os testes foram executados no Ubuntu 18.04 (bionic 64).
Software:

- Instalado previamente pelo usuário (Control Node):
    - Ubuntu 18.04 LTS;
    - Ansible 2.9.6 (Recomendamos a instalação com pip3);
    - sshpass;
    - GIT.

- Instalado juntamente com o deploy da Dojot:
    - Kubernetes: Kubeadm 1.17 com **Service Topology** ativado;
    - Docker: Docker CE 19.03.4;
    - Balanceador de carga L4: HAProxy Community 2.0;

Hardware:

- **HAProxy:** 4 core de processamento (2.2GHZ), 4GB de RAM e 20GB de disco;
- **Master Node:** 2 cores de processamento (2.2GHZ), 2GB de RAM e 20GB de disco;
- **Dojot Node:** 4 cores de processamento (2.2GHZ), 4GB de RAM e 30GB de disco;
- **kafka Node:** 6 cores de processamento (2.2GHZ), 6GB de RAM e 30GB de disco;
- **X509 Node:** 4 cores de processamento (2.2GHZ), 4GB de RAM e 20GB de disco;
- **Verne Nodes** (para cada nó): 8 cores de processamento (2.2GHZ), 8GB de RAM e 30GB de disco;

Download do repositório

Para iniciar a instalação, precisamos baixar o repositório com os playbooks do ansible. O seguinte comando é executado no control node do ansible para clonar o repositório “ansible-dojot”:

$ git clone https://github.com/dojot/ansible-dojot.git

Será criado um diretório com o nome "ansible-dojot", devemos então acessar esse diretório e mudar para a TAG da versão da dojot que queremos utilizar. Lembrando que a branch master sempre aponta para a última TAG (versão) da dojot:

```
$ cd ansible-dojot
$ git checkout v0.5.0 -b v0.5.0
```

Nesse exemplo, mudamos a tag para a versão v0.5.0 da dojot. Todos os playbooks e scripts nessa TAG estão adaptados para essa versão específica da dojot, ou seja, para cada versão temos uma TAG específica.

Configuração do cluster kubernetes

Para um ambiente com poucos dispositivos

Se já temos um cluster kubernetes na versão 1.17, podemos pular essa etapa. Mas se não temos um cluster ainda, precisamos criar um. Para isso, precisamos de no mínimo 2 máquinas físicas ou virtuais com os requisitos de hardware descritos acima. Através do control node do Ansible, executamos o playbook para a instalação e configuração do kubernetes que será utilizado para orquestrar os containers Docker da dojot.

É interessante alterar o hostname das máquinas que farão parte do cluster kubernetes para facilitar a visualização dos nós quando necessário. No Ubuntu 18.04 utilizamos o comando "hostnamectl". Para alterar o hostname da máquina que será o master node do kubernetes utilizamos o seguinte comando:

$ sudo hostnamectl set-hostname k8s-master

E para alterar o hostname da máquina que será o worker node usamos:

$ sudo hostnamectl set-hostname k8s-worker

Para executar o playbook do ansible com sucesso, precisamos informar os hosts onde o k8s precisa ser instalado e configurado. Configuramos isso no arquivo "*inventories/example_local/hosts.yaml*".

```
---
all:
  hosts:
    ...
    master_host:
      ansible_host: MASTER_HOST_ADDRESS #change to the host of master node on k8s
    worker_host:
      ansible_host: WORKER_HOST_ADDRESS #change to the host of worker node on k8s
  children:
    ...
    k8s-nodes:
      children:
        master_nodes:
          hosts:
            master_host:
        worker_nodes:
          hosts:
            worker_host:
  ...
```

Ocultamos o restante do conteúdo do arquivo com "..." para dar mais foco no que vamos utilizar agora.

Nesse arquivo, configuramos os hosts que receberão a instalação do k8s. Como exemplo, considerando que o "master" node tem o ip "192.168.0.10" e o "worker" node tem o ip 192.168.0.11. O arquivo hosts.yaml ficaria da seguinte forma:

```
---
all:
  hosts:
    ...
    master_host:
      ansible_host: 192.168.0.10
    worker_host:
      ansible_host: 192.168.0.11
  children:
    ...
    k8s-nodes:
      children:
        master_nodes:
          hosts:
            master_host:
        worker_nodes:
          hosts:
            worker_host:
  ...
```

Podemos também ter mais de um worker node, como no exemplo abaixo:

```
---
all:
  hosts:
    ...
    master_host:
      ansible_host: 192.168.0.10
    worker_host:
      ansible_host: 192.168.0.11
    worker_host_two:
      ansible_host: 192.168.0.12
  children:
    ...
    k8s-nodes:
      children:
        master_nodes:
          hosts:
            master_host:
        worker_nodes:
          hosts:
            worker_host:
            worker_host_two:
  ...
```

É importante saber que o playbook do ansible para instalação e configuração do kubernetes utiliza os grupos "**k8s-nodes**", "**master_nodes**" e "**worker_nodes**", como destacado abaixo. Sendo assim, todos os hosts abaixo desses grupos serão afetados e adicionados ao cluster. Os hosts que estão abaixo de "*master_nodes*" serão configurados como master node no kubernetes, e os hosts abaixo do grupo "worker_nodes" serão configurados como worker nodes.

```
...
 k8s-nodes:
   children:
     master_nodes:
       hosts:
         master_host:
     worker_nodes:
       hosts:
         worker_host:
         worker_host_two:
...
```

Com nosso arquivo "hosts.yaml" devidamente configurados, podemos executar o playbook do ansible para configurar o nosso cluster k8s:

```
$ ansible-playbook -K -k -u dojot -i inventories/example_local k8s.yaml
```

No exemplo acima, o ansible irá conectar nas máquinas remotamente via SSH utilizando o usuário "dojot". O ansible também vai perguntar o password do usuário “dojot” no terminal. Mas outras abordagens podem ser utilizadas para melhorar a segurança.

O ansible vai mostrar no log tudo que está executando e alterando. Todos os comandos na cor "amarela" significam que algo foi alterado na máquina remota. A cor verde significa que nada foi alterado e a cor vermelha significa erros. Não devemos utilizar o cluster caso algum erro seja acusado no log.

No final da execução do playbook, podemos acessar o nó master do kubernetes e executar o comando "*kubectl get nodes*". Se a saída for parecida com essa, significa que o cluster foi criado corretamente:

```
NAME          STATUS   ROLES    AGE   VERSION
k8s-master    Ready    master   0d    v1.17.3
k8s-worker    Ready    <none>   0d    v1.17.3
```

Lembrando que o "NAME" do nó no kubernetes é o hostname da máquina onde o kubernetes foi instalado.

Em um ambiente de produção, onde é necessário que o estado da aplicação seja restaurado caso algum container ou algum nó do cluster apresente algum problema, precisamos configurar volumes para que os dados de alguns serviços sejam persistidos.

Existem diversas maneiras e abordagens para configurar volumes no kubernetes. Nesse documento, vamos abordar a criação de volumes locais.

Os volumes locais utilizam o disco dos nodes do cluster como storage. Sendo assim, precisamos criar diretórios no cluster para que os volumes sejam mapeados corretamente. Precisamos então acessar os nós por SSH e criar os diretórios localmente. O administrador do cluster pode criar os diretórios onde achar mais interessante, mas os exemplos que estão no diretório *"local_storage_example/volumes", do repositório "ansible-dojot" baixado anteriormente no control node,* apontam para os seguintes locais dentro dos nós:

Node	**Diretório**
dojot	/mnt/data/kong
dojot	/mnt/data/minio
dojot	/mnt/data/mongodb
dojot	/mnt/data/postgres
dojot	/mnt/data/prometheus
dojot	/mnt/data/kafka_ws
kafka	/mnt/data/kafka
kafka	/mnt/data/zookeeper/data
kafka	/mnt/data/zookeeper/log
x509	/mnt/data/ejbca

Em um ambiente que suporte poucos dispositivos, que possui somente um nó worker no cluster, a configuração deve ser realizada para que fique da seguinte forma:

Node	**Diretório**
dojot	/mnt/data/kong
dojot	/mnt/data/minio
dojot	/mnt/data/mongodb
dojot	/mnt/data/postgres
dojot	/mnt/data/prometheus

dojot	/mnt/data/kafka_ws
dojot	/mnt/data/kafka
dojot	/mnt/data/zookeeper/data
dojot	/mnt/data/zookeeper/log
dojot	/mnt/data/ejbca

No exemplo acima, para um ambiente com poucos dispositivos, assumimos que o único nó worker do kubernetes tem o label "***dojot.components/group=dojot***", pois sem isso o volume não poderá ser criado e configurado de acordo com o yaml de criação do volume. Para adicionar o label no nó worker do kubernetes podemos utilizar o seguinte comando:

$ kubectl label nodes k8s-worker ***dojot.components/group=dojot***

Para que os scripts de criação de volumes de exemplo possam ser utilizados, é necessário a criação dos diretórios nos locais apontados acima. Caso contrário, será necessário entrar nos arquivos do diretório *"local_storage_example/volumes"* e alterar o parâmetro path dos volumes como no exemplo abaixo, onde o diretório de volume do PostgreSQL foi alterado para *"/home/anderson/postgres"*:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv-postgres
  labels:
    type: local
    app: dojot
    db: postgres
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  local:
    path: /home/anderson/postgres
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: dojot.components/group
          operator: In
          values:
          - dojot
```

Para um ambiente que suporte poucos dispositivos, também precisamos alterar a seção “nodeAffinity” dos arquivos para que fiquem todos da seguinte forma:

```
nodeAffinity:
  required:
    nodeSelectorTerms:
    - matchExpressions:
      - key: dojot.components/group
        operator: In
        values:
        - dojot
```

Depois de ajustar os arquivos corretamente, vamos agora copiá-los para o nó master do kubernetes para que possamos aplicá-los com o kubectl. Podemos utilizar o seguinte comando para copiar os arquivos:

$ scp -r local_storage_example/volumes dojot@192.168.0.10:~/

Com esse comando, copiamos o diretório "volumes" que está dentro de "local_storage_example" para dentro do diretório raiz do usuário "dojot" do host master (192.168.0.10).

Com todos os arquivos de criação de volumes copiados, podemos então acessar nó master do kubernetes por SSH e executar o seguinte comando para criar os volumes:

$ ssh dojot@192.168.0.10
$ kubectl apply -f volumes

Todos os arquivos do diretório "volumes" serão aplicados e todos os volumes serão criados. Para visualizar o status dos volumes podemos utilizar o comando *"kubectl get pv*". A seguinte lista será exibida:

```
→  ~ NAME                          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM                                STORAGECLASS    REASON   AGE
local-pv-kafka                2Gi        RWO            Retain           Bound       dojot/kafka-volume-kafka-server-0    local-storage            22d
local-pv-kafka-ws             5Mi        ROX            Retain           Bound       dojot/kafka-ws                       local-storage            22d
local-pv-kong                 5Mi        ROX            Retain           Bound       dojot/kong                           local-storage            22d
local-pv-minio                2Gi        RWO            Retain           Bound       dojot/minio                          local-storage            22d
local-pv-mongo                2Gi        RWO            Retain           Bound       dojot/mongodb-volume-mongodb-0       local-storage            22d
local-pv-postgres             2Gi        RWO            Retain           Bound       dojot/psql-volume-postgres-0         local-storage            22d
local-pv-prometheus           2Gi        RWO            Retain           Bound       dojot/prometheus                     local-storage            22d
local-pv-x509-identity-mgmt   10Mi       RWX            Retain           Bound       dojot/x509-identity-mgmt             local-storage            22d
local-pv-zk-data              1Gi        RWO            Retain           Bound       dojot/zk-volume-data-zookeeper-0     local-storage            22d
local-pv-zk-log               1Gi        RWO            Retain           Bound       dojot/zk-volume-log-zookeeper-0      local-storage            22d
```

Para um ambiente com até 100.000 dispositivos

A abordagem para montar um cluster k8s que suporte 100.000 dispositivos é praticamente a mesma para montar um cluster que suporte poucos dispositivos. Precisamos de mais workers, com configurações diferentes para suportar a carga. Sendo assim, podemos executar os mesmos passos do capítulo anterior, modificando o nosso arquivo hosts.yaml que fica em "*inventories/example_local/hosts.yaml*". Abaixo um exemplo do arquivo modificado:

```
---
all:
  hosts:
    ...
    master_host:
      ansible_host: 192.168.0.10
    dojot_worker_host:
      ansible_host: 192.168.0.11
    kafka_worker_host:
      ansible_host: 192.168.0.12
    x509_worker_host:
      ansible_host: 192.168.0.13
    first_verne_worker_host:
      ansible_host: 192.168.0.14
    second_verne_worker_host:
      ansible_host: 192.168.0.15
    third_verne_worker_host:
      ansible_host: 192.168.0.16
  children:
    ...
    k8s-nodes:
      children:
        master_nodes:
          hosts:
            master_host:
        worker_nodes:
          hosts:
            dojot_worker_host:
            kafka_worker_host:
            x509_worker_host:
            first_verne_worker_host:
            second_verne_worker_host:
            third_verne_worker_host:
  ...
```

Antes de executar nosso playbook, seria interessante também ajustar o hostname de cada um dos nós, para que fique mais fácil sua visualização no k8s. Como exemplo, para configurar o hostname de um dos nós que será utilizado pelo VerneMQ, acessamos o host do nó e executamos o seguinte comando:

```
$ sudo hostnamectl set-hostname verne-1
```

Com nosso arquivo "hosts.yaml" devidamente configurado e cada nó com seu hostname definido, podemos executar o playbook do ansible para configurar o nosso cluster k8s:

```
$ ansible-playbook  -K -k -u dojot -i inventories/example_local k8s.yaml
```

No final da execução do playbook, podemos acessar o nó master do kubernetes e executar o comando "*kubectl get nodes*". Se a saída for parecida com essa, significa que o cluster foi criado corretamente:

```
NAME      STATUS   ROLES    AGE   VERSION
master    Ready    master   0d    v1.17.3
verne-1   Ready    <none>   0d    v1.17.3
verne-2   Ready    <none>   0d    v1.17.3
verne-3   Ready    <none>   0d    v1.17.3
dojot     Ready    <none>   0d    v1.17.3
kafka     Ready    <none>   0d    v1.17.3
x509      Ready    <none>   0d    v1.17.3
```

Agora precisamos adicionar labels nos nós do kubernetes para que possamos distribuir os microsserviços corretamente no cluster. Com isso melhoramos a performance e garantimos o bom funcionamento dos serviços. Como exemplo, o nó que roda o kafka server, não deveria ter outros serviços sendo executados no mesmo nó, para que a performance do kafka não seja comprometida.

Devemos adicionar os seguintes labels em nosso cluster:

Nó k8s	**Label**
verne-1	dojot.components/group=vernemq
verne-2	dojot.components/group=vernemq
verne-3	dojot.components/group=vernemq
dojot	dojot.components/group=dojot
kafka	dojot.components/group=kafka
x509	dojot.components/group=x509

Adicionamos os labels acessando o nó master do k8s. No exemplo abaixo, adicionamos o label no nó kafka:

```
$ kubectl label nodes kafka dojot.components/group=kafka
```

Precisamos adicionar labels em todos os nós conforme tabela acima. O playbook de deploy da dojot irá distribuir os serviços no cluster considerando os labels que foram adicionados. Como exemplo, o serviço kafka será executado no nó que contém o label dojot.components/group=kafka. Por isso a importância de adicionar os labels no serviço.

Em um ambiente de produção, onde é necessário que o estado da aplicação seja restaurado caso algum container ou algum nó do cluster apresente algum problema, precisamos configurar volumes para que os dados de alguns serviços sejam persistidos. Em um ambiente com 100.000 dispositivos conectados, essa configuração também é necessária para obter a performance necessária.

Existem diversas maneiras e abordagens para configurar volumes no kubernetes. Nesse documento, vamos abordar a criação de volumes locais.

Os volumes locais utilizam o disco dos nodes do cluster como storage. Sendo assim, precisamos criar diretórios no cluster para que os volumes sejam mapeados corretamente. Precisamos então acessar os nós por SSH e criar os diretórios localmente. O administrador do cluster pode criar os diretórios onde achar mais interessante, mas os exemplos que estão no diretório *"local_storage_example/volumes", do repositório "ansible-dojot" baixado anteriormente no control node,* apontam para os seguintes locais dentro dos nós:

Node	**Diretório**
dojot	/mnt/data/kong
dojot	/mnt/data/minio
dojot	/mnt/data/mongodb
dojot	/mnt/data/postgres
dojot	/mnt/data/prometheus
dojot	/mnt/data/kafka_ws
kafka	/mnt/data/kafka
kafka	/mnt/data/zookeeper/data
kafka	/mnt/data/zookeeper/log
x509	/mnt/data/ejbca

Para que os scripts de criação de volumes de exemplo possam ser utilizados, é necessário a criação dos diretórios nos locais apontados acima. Caso contrário, será necessário entrar nos arquivos do diretório *"local_storage_example/volumes"* e alterar o parâmetro path dos volumes como no exemplo abaixo, onde o diretório de volume do PostgreSQL foi alterado para *"/home/anderson/postgres"*:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv-postgres
  labels:
    type: local
    app: dojot
    db: postgres
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
```

```
persistentVolumeReclaimPolicy: Retain
storageClassName: local-storage
local:
  path: /home/anderson/postgres
nodeAffinity:
  required:
    nodeSelectorTerms:
    - matchExpressions:
      - key: dojot.components/group
        operator: In
        values:
        - dojot
```

Depois de ajustar os arquivos corretamente, vamos agora copiá-los para o nó master do kubernetes para que possamos aplicá-los com o kubectl. Podemos utilizar o seguinte comando para copiar os arquivos:

$ scp -r local_storage_example/volumes dojot@192.168.0.10:~/

Com esse comando, copiamos o diretório "volumes" que está dentro de "local_storage_example" para dentro do diretório raiz do usuário "dojot" do host master (192.168.0.10).

Com todos os arquivos de criação de volumes copiados, podemos então acessar nó master do kubernetes por SSH e executar o seguinte comando para criar os volumes:

$ ssh dojot@192.168.0.10
$ kubectl apply -f volumes

Todos os arquivos do diretório "volumes" serão aplicados e todos os volumes serão criados. Para visualizar o status dos volumes podemos utilizar o comando *"kubectl get pv*". A seguinte lista será exibida:

```
→  ~ NAME                          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM                                 STORAGECLASS    REASON   AGE
local-pv-kafka               2Gi        RWO            Retain           Bound       dojot/kafka-volume-kafka-server-0   local-storage            22d
local-pv-kafka-ws            5Mi        ROX            Retain           Bound       dojot/kafka-ws                      local-storage            22d
local-pv-kong                5Mi        ROX            Retain           Bound       dojot/kong                          local-storage            22d
local-pv-minio               2Gi        RWO            Retain           Bound       dojot/minio                         local-storage            22d
local-pv-mongo               2Gi        RWO            Retain           Bound       dojot/mongodb-volume-mongodb-0      local-storage            22d
local-pv-postgres            2Gi        RWO            Retain           Bound       dojot/psql-volume-postgres-0        local-storage            22d
local-pv-prometheus          2Gi        RWO            Retain           Bound       dojot/prometheus                    local-storage            22d
local-pv-x509-identity-mgmt  10Mi       RWX            Retain           Bound       dojot/x509-identity-mgmt            local-storage            22d
local-pv-zk-data             1Gi        RWO            Retain           Bound       dojot/zk-volume-data-zookeeper-0    local-storage            22d
local-pv-zk-log              1Gi        RWO            Retain           Bound       dojot/zk-volume-log-zookeeper-0     local-storage            22d
```

Deploy da dojot

Para um ambiente com poucos dispositivos

Para fazer deploy da dojot, precisamos primeiro configurar as variáveis que o playbook irá usar. Nosso playbook toma decisões de configuração com base em variáveis. O arquivo de variáveis que fica em i*nventories/example_local/group_vars/all/dojot.yaml*, tem este conteúdo:

```
dojot_namespace: dojot
dojot_version: v0.5.0
dojot_domain_name: 102.168.0.12
dojot_storage_class_name: "local-storage"
dojot_kubernetes_rbac: true

dojot_zk_persistent_volumes: true
dojot_psql_persistent_volumes: true
dojot_x509_identity_mgmt_persistent_volumes: true
dojot_mongodb_persistent_volumes: true
dojot_minio_persistent_volumes: true
dojot_kafka_persistent_volumes: true
dojot_kafka_ws_persistent_volumes: true
dojot_apigw_persistent_volumes: true

dojot_guiv2_enabled: false
dojot_auth_email_enabled: false
dojot_insecure_mqtt: 'true'
dojot_vernemq_replicas: 1
dojot_bridges_replicas: 1
dojot_x509_identity_mgmt_replicas: 1
dojot_kafka_ws_enable_tls: false
dojot_apigw_enable_mutual_tls: false

dojot_fixed_nodeports_enabled: true
dojot_nodeports:
  apigw:
    http: 30001
    https: 30002
  metrics: 30003
  mqtt: 30004
  mqtts: 30005
  lwm2m:
    coap: 30006
    coaps: 30007
    file_server: 30008
    file_servers: 30009
    http: 30010

dojot_enable_node_affinity: false
dojot_node_label:
    dojot: dojot
    x509: x509
   kafka: kafka
   vernemq: vernemq

dojot_enable_locust_exporter: false
dojot_locust_exporter:
  ip: 127.0.0.1
  port: 9646
```

Vamos agora detalhar as alterações explicando o papel de cadas variável no deployment:

- **dojot_domain_name**: utilizada por alguns serviços internos e principalmente pelo serviço que gera certificados para os dispositivos. Vamos subir um balanceador de carga para receber as conexões dos dispositivos, então nessa variável precisa ter o endereço do host do balanceador de carga, ou o domínio que aponta para o host do balanceador.

- **dojot_storage_class_name**: utilizada para especificar o storage class utilizado pelos serviços da dojot, na verdade quando configuramos e criamos nossos volumes, esse storage class já foi criado com o nome “local-storage”, então só precisamos informar no arquivo de variáveis. No momento do deployment os serviços vão utilizar o storage dessa classe.

- **dojot_*_persistent_volumes**: existe uma variável para cada serviço que pode utilizar persistência de dados em volumes no kubernetes. Se a variável tem o valor “true” então o volume é criado para o serviço.

- **dojot_fixed_nodeports_enabled**: define se os serviços da dojot com NodePort utilizam portas fixas. Para esse ambiente precisamos que as portas sejam fixas, pois precisamos utilizar um balanceador de carga que balanceia as requisições nessas portas. É possível modificar o número dessas portas através da variável “*dojot_nodeports*”.

Também precisamos alterar o arquivo hosts.yaml que fica em "*inventories/example_local/hosts.yaml*", caso ainda não tenha sido alterado. Nesse arquivo temos a configuração do host onde o playbook de deploy será executado. O arquivo deve ficar da seguinte forma:

```
---
all:
  hosts:
    ...
    master_host:
      ansible_host: 192.168.0.10
  children:
    ...
    dojot-k8s:
      hosts:
        master_host:
  ...
```

Precisamos alterar o grupo "dojot-k8s" para o que playbook seja executado no nó master do nosso cluster kubernetes. Outra estratégia seria utilizar um arquivo de configuração local do kubernetes, mas não vamos abordar esse assunto nesse documento.

Com nosso arquivo de variáveis e nosso arquivo de inventário configurados, podemos agora executar o playbook de deployment da dojot utilizando o seguinte comando:

*$ ansible-playbook -K -k -u **dojot** -i inventories/example_local deploy.yaml*

Lembrando que o parâmetro "-u" informa o usuário do host master.

No modo padrão, utilizando esse comando, a dojot será disponibilizada utilizando o VerneMQ como Iot Agent. Temos ainda mais duas opções de Iot Agents que podem ser utilizadas na dojot. O Iot Agent Mosca e o LWM2M.

Utilizando o comando abaixo, podemos fazer deploy da dojot utilizando o VerneMQ e o LWM2M juntos:

*$ ansible-playbook -K -k -u **dojot** -i inventories/example_local deploy.yaml **--tags "vernemq, lwm2m"***

Utilizando esse outro comando, podemos fazer deploy utilizando ao Iot Agent Mosca e o LWM2M:

*$ ansible-playbook -K -k -u **dojot** -i inventories/example_local deploy.yaml **--tags "mosca, lwm2m"***

Podemos também fazer deploy de um Iot Agent:

*$ ansible-playbook -K -k -u **dojot** -i inventories/example_local deploy.yaml **--tags "mosca"***

Ao executar o playbook, o ansible fará o deploy da dojot em nosso cluster. Os logs informam o status de cada task e no final da execução todos os microsserviços da dojot estarão disponíveis no cluster.

Apesar de os serviços estarem disponíveis, precisamos monitorar o status de cada um. O deployment foi realizado, mas os microsserviços possuem dependências entre eles e precisam subir os serviços dentro do container. Podemos acessar o nó master do kubernetes por ssh e executar o seguinte comando para acompanhar o status dos serviços:

$ ssh dojot@192.168.0.10
$ watch kubectl get pods -n dojot

No exemplo acima, acessamos o nó master do kubernetes por ssh e executamos o comando para listar os todos pods da dojot. Para sair do modo "watch" é só usar Ctrl+c.

A saída desse comando é uma lista com todos os microsserviços da dojot e o status de cada um:

```
NAME                                        READY   STATUS      RESTARTS   AGE
auth-57579c6989-49hdx                       2/2     Running     0          5d1h
backstage-59865469d4-k454p                  1/1     Running     2          5d1h
cron-fb9947cf-jkf4k                         1/1     Running     0          5d1h
data-broker-5cfdf576df-tbrsb                1/1     Running     0          5d1h
data-broker-redis-5f967dbcdb-pb79z          1/1     Running     0          5d1h
data-manager-744c6fd9d5-fchjj               1/1     Running     0          5d1h
device-manager-7cfb8755f-mjfnw              1/1     Running     0          5d1h
device-manager-redis-6645bd65b6-5zk7p       1/1     Running     0          5d1h
flowbroker-db77847fd-pwvbh                  3/3     Running     2          5d1h
gui-6f7db8cd58-x6l44                        1/1     Running     0          5d1h
gui-v2-77d8967c5b-cld64                     1/1     Running     0          5d1h
history-6549c8fcc7-mjcc6                    1/1     Running     0          5d1h
image-manager-7975dd669b-9hbxc              1/1     Running     0          5d1h
k2v-bridge-0                                1/1     Running     4          5d1h
kafka-server-0                              1/1     Running     0          5d1h
kafka-ws-7d89697f9f-psnk8                   1/1     Running     0          5d1h
kafka-ws-redis-0                            1/1     Running     0          5d1h
kafka2ftp-7dd99c6c44-fcrb9                  1/1     Running     1          5d1h
kong-c574d7897-j99gr                        1/1     Running     0          5d1h
kong-migrate-s42k4                          0/1     Completed   0          5d1h
kong-migrate-up-wjxgc                       0/1     Completed   0          5d1h
kong-route-config-kbgmh                     0/1     Completed   1          5d1h
minio-0                                     1/1     Running     0          5d1h
mongodb-0                                   1/1     Running     0          5d1h
persister-75664cb8cb-gzx52                  1/1     Running     0          5d1h
postgres-0                                  1/1     Running     0          5d1h
rabbitmq-bff4b85df-xch72                    1/1     Running     0          5d1h
v2k-bridge-0                                1/1     Running     4          5d1h
vernemq-k8s-0                               1/1     Running     5          5d1h
vernemq-k8s-deployment-69d556df7b-h6cqh     1/1     Running     0          5d1h
vmq-operator-6774ff9ff4-nnwng               1/1     Running     0          5d1h
x509-identity-mgmt-7f76fff569-sfs2t         1/1     Running     0          5d1h
zookeeper-0                                 1/1     Running     0          5d1h
```

O status de cada serviço no seu cluster precisa ser o mesmo dos serviços mostrados acima. Isso pode levar alguns minutos para acontecer. Após esse período a dojot já estará pronta para ser utilizada, mas para acessar o sistema precisamos ainda configurar o load balancer.

Para um ambiente com até 100.000 dispositivos

Agora podemos sair do console do nó master do cluster e voltar para o control node do Ansible para fazer deploy da dojot. O ambiente que suporta até 100.000 dispositivos conectados precisa de alguns ajustes de deployment de alguns serviços para que tudo funcione corretamente. Sendo assim, vamos editar o arquivo de variáveis que fica em i*nventories/example_local/group_vars/all/dojot.yaml* . Abaixo destacamos os trechos que foram alterados no arquivo:

```
dojot_namespace: dojot
dojot_version: v0.5.0-alpha.4
dojot_domain_name: dojot.domain

dojot_storage_class_name: "local-storage"

dojot_kubernetes_rbac: true

dojot_zk_persistent_volumes: true
dojot_psql_persistent_volumes: true
dojot_x509_identity_mgmt_persistent_volumes: true
dojot_mongodb_persistent_volumes: true
```

```
dojot_minio_persistent_volumes: true
dojot_kafka_persistent_volumes: true
dojot_kafka_ws_persistent_volumes: true
dojot_apigw_persistent_volumes: true

dojot_guiv2_enabled: false

dojot_auth_email_enabled: false

dojot_insecure_mqtt: 'true'

dojot_vernemq_replicas: 3
dojot_bridges_replicas: 9
dojot_x509_identity_mgmt_replicas: 1

dojot_kafka_ws_enable_tls: false
dojot_apigw_enable_mutual_tls: false

dojot_fixed_nodeports_enabled: true

dojot_nodeports:
  apigw:
    http: 30001
    https: 30002
  metrics: 30003
  mqtt: 30004
  mqtts: 30005
  lwm2m:
    coap: 30006
    coaps: 30007
    file_server: 30008
    file_servers: 30009
    http: 30010

dojot_enable_node_affinity: true
dojot_node_label:
  dojot: dojot
  x509: x509
  kafka: kafka
  vernemq: vernemq

dojot_enable_locust_exporter: false
dojot_locust_exporter:
  ip: 127.0.0.1
  port: 9646
```

Vamos agora detalhar as alterações explicando o papel de cadas variável no deployment:

- **dojot_domain_name**: utilizada por alguns serviços internos e principalmente pelo serviço que gera certificados para os dispositivos. Vamos subir um balanceador de carga para receber as conexões dos dispositivos, então nessa variável precisa ter o

endereço do host do balanceador de carga, ou o domínio que aponta para o host do balanceador.

- **dojot_storage_class_name**: utilizada para especificar o storage class utilizado pelos serviços da dojot, na verdade quando configuramos e criamos nossos volumes, esse storage class já foi criado com o nome “local-storage”, então só precisamos informar no arquivo de variáveis. No momento do deployment os serviços vão utilizar o storage dessa classe.

- **dojot_*_persistent_volumes**: existe uma variável para cada serviço que pode utilizar persistência de dados em volumes no kubernetes. Se a variável tem o valor “true” então o volume é criado para o serviço.

- **dojot_vernemq_replicas**: define a quantidade de réplicas para o Iot Agent VerneMQ. Para suportar até 100.000 dispositivos, é necessário no mínimo 3 réplicas do serviço.

- **dojot_bridges_replicas**: define a quantidade de bridges criadas para o VerneMQ no cluster. Cada instância do VerneMQ precisa de pelo menos 3 bridges para suportar até 100.000 dispositivos, totalizando o total de 9 bridges.
- **dojot_fixed_nodeports_enabled**: define se os serviços da dojot com NodePort utilizam portas fixas. Para esse ambiente precisamos que as portas sejam fixas, pois precisamos utilizar um balanceador de carga que balanceia as requisições nessas portas. É possível modificar o número dessas portas através da variável “*dojot_nodeports*”.

- **dojot_enable_node_affinity**: define se os serviços devem ser distribuídos utilizando os labels que configuramos anteriormente. Se estiver com o valor “true” e se configuramos os labels nos nós corretamente, os serviços serão distribuídos corretamente no cluster possibilitando uma otimização na performance.

Também precisamos alterar o arquivo hosts.yaml que fica em "*inventories/example_local/hosts.yaml*", caso ainda não tenha sido alterado. Nesse arquivo temos a configuração do host onde o playbook de deploy será executado. O arquivo deve ficar da seguinte forma:

```
---
all:
  hosts:
    ...
    master_host:
      ansible_host: 192.168.0.10
  children:
    ...
    dojot-k8s:
      hosts:
        master_host:
```

...

Precisamos alterar o grupo "dojot-k8s" para o que playbook seja executado no nó master do nosso cluster kubernetes. Outra estratégia seria utilizar um arquivo de configuração local do kubernetes, mas não vamos abordar esse assunto nesse documento.

Podemos agora fazer deploy da dojot utilizando o seguinte comando:

*$ ansible-playbook -K -k -u **dojot** -i inventories/example_local deploy.yaml **--tags 100k***

Utilizando a tag “100k”, forçamos o ansible fazer deploy somente dos serviços necessários para o ambiente que suporte até 100.000 dispositivos. Quando utilizamos essa tag, somente o IOT Agent e os serviços necessários para comunicação com o Apache Kafka são disponibilizados. Ou Seja, esse é um ambiente utilizado somente para testes de carga. Nesse ambiente, a dojot não processa e nem armazena as mensagens enviadas pelos dispositivos.

Após o deployment, podemos acessar o nó master do kubernetes e utilizar o seguinte comando para acompanhar o status dos serviços:

$ ssh dojot@192.168.0.10
$ watch kubectl get pods -n dojot

O resultado deve ser algo parecido com a imagem abaixo:

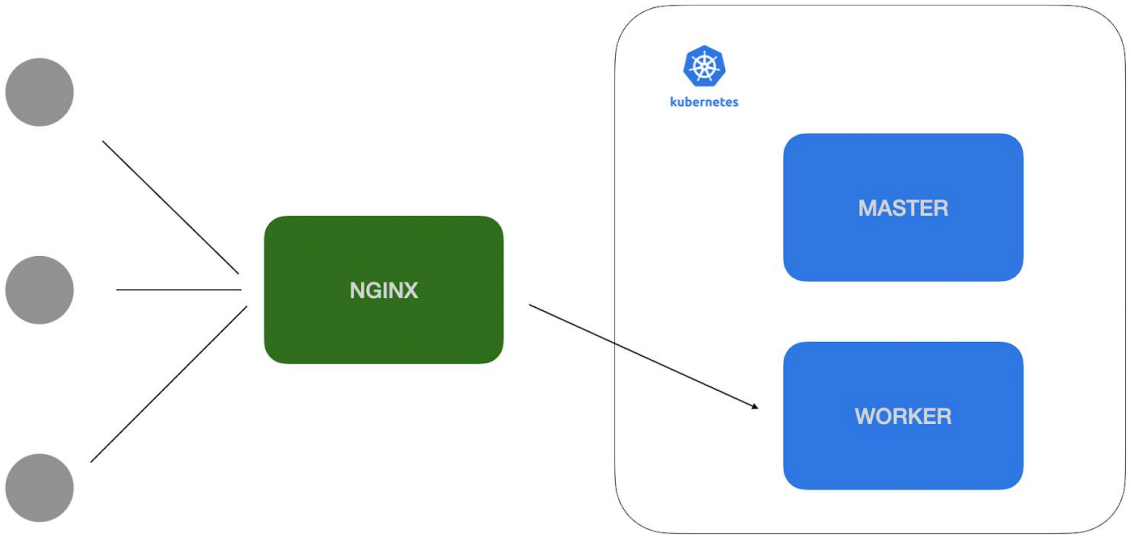
```
NAME                                    READY   STATUS      RESTARTS   AGE
auth-5794cd465-6qw95                    2/2     Running     0          2d5h
data-broker-67ccd47d68-g7d9p            1/1     Running     1          2d5h
data-broker-redis-78656688d5-qhs6h      1/1     Running     0          2d5h
k2v-bridge-0                            1/1     Running     202        2d5h
k2v-bridge-1                            1/1     Running     197        2d5h
k2v-bridge-2                            1/1     Running     211        2d5h
k2v-bridge-3                            1/1     Running     203        2d5h
k2v-bridge-4                            1/1     Running     202        2d5h
k2v-bridge-5                            1/1     Running     208        2d5h
k2v-bridge-6                            1/1     Running     207        2d5h
k2v-bridge-7                            1/1     Running     192        2d5h
k2v-bridge-8                            1/1     Running     206        2d5h
kafka-loopback-0                        1/1     Running     3          2d5h
kafka-loopback-1                        1/1     Running     2          2d5h
kafka-loopback-2                        1/1     Running     2          2d5h
kafka-loopback-3                        1/1     Running     1          2d5h
kafka-loopback-4                        1/1     Running     0          2d5h
kafka-loopback-5                        1/1     Running     0          2d5h
kafka-loopback-6                        1/1     Running     0          2d5h
kafka-loopback-7                        1/1     Running     0          2d5h
kafka-loopback-8                        1/1     Running     0          2d5h
kafka-server-0                          1/1     Running     1          2d5h
kafka-ws-7d999c5ff4-qk49p               1/1     Running     0          2d5h
kafka-ws-redis-0                        1/1     Running     0          2d5h
kong-5975bdddd6-ccxmh                   1/1     Running     0          2d5h
kong-migrate-rb6m9                      0/1     Completed   0          2d5h
kong-migrate-up-plpcx                   0/1     Completed   0          2d5h
kong-route-config-gjm66                 0/1     Completed   0          2d5h
mongodb-0                               1/1     Running     0          2d5h
postgres-0                              1/1     Running     0          2d5h
v2k-bridge-0                            1/1     Running     203        2d5h
v2k-bridge-1                            1/1     Running     205        2d5h
v2k-bridge-2                            1/1     Running     203        2d5h
v2k-bridge-3                            1/1     Running     209        2d5h
v2k-bridge-4                            1/1     Running     211        2d5h
v2k-bridge-5                            1/1     Running     202        2d5h
v2k-bridge-6                            1/1     Running     209        2d5h
v2k-bridge-7                            1/1     Running     209        2d5h
v2k-bridge-8                            1/1     Running     204        2d5h
vernemq-k8s-0                           1/1     Running     135        2d5h
vernemq-k8s-1                           1/1     Running     136        2d5h
vernemq-k8s-2                           1/1     Running     145        2d5h
vernemq-k8s-deployment-69d556df7b-7ppqr 1/1     Running     0          2d5h
vmq-operator-6774ff9ff4-6wtjc           1/1     Running     0          2d5h
x509-identity-mgmt-6f4dcfd68-4xmw8      1/1     Running     0          2d5h
zookeeper-0                             1/1     Running     0          2d5h
```

Com isso finalizamos o deploy da dojot com um ambiente que suporte até 100.000 dispositivos conectados.

Configuração do load balancer

Para um ambiente com poucos dispositivos

Para utilizar a dojot com pouca carga de dispositivos e para que possamos utilizar também o LWM2M, devemos configurar o NGINX como balanceador de carga. Esse balanceador receberá todas as requisições dos clientes e fará o balanceamento de carga entre os nós worker do kubernetes, ou seja, nosso cluster não é acessado diretamente pelo cliente. Como mostrado na imagem abaixo:

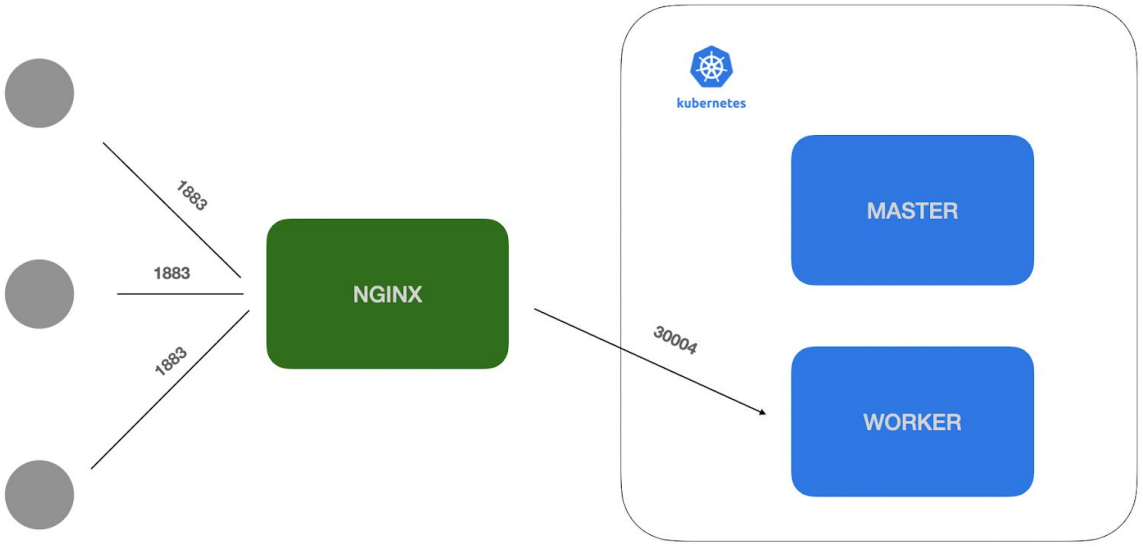


Temos um playbook para instalação e configuração do NGINX. Mas antes de executá-lo, precisamos configurar o nosso arquivo hosts.yaml novamente para informar ao ansible o host da máquina que terá o NGINX instalado e também em qual será o nó worker responsável por hospedar os micro-serviços da dojot. Para isso editamos o nosso arquivo de inventário que está em "*inventories/example_local/hosts.yaml*":

```
---
all:
  hosts:
    ...
    nginx:
      ansible_host: 192.168.0.12
...
children:
...
    apigw_nodes:
      hosts:
        192.168.0.11:
    mqtt_nodes:
      hosts:
        192.168.0.11:
    lwm2m_nodes:
      hosts:
        192.168.0.11:
    metrics_nodes:
      hosts:
        192.168.0.11:
```

No exemplo acima, assumimos que o host do NGINX será "192.168.0.12" e que possuímos apenas um nó worker (192.168.0.11). Caso seu cluster tenha mais de um nó worker você tem a opção de definir em qual nó deseja que cada um desses micro-serviços rode. Essas são as únicas configurações que precisamos fazer.

Lembrando que já alteramos o nosso arquivo de variáveis anteriormente para que o kubernetes utilize portas fixas para os serviços com acesso externo. O playbook do NGINX utiliza essa mesma configuração, ou seja, se o serviço Iot Agent VerneMQ expõe a porta de acesso para MQTT em 30004, o NGINX vai receber a conexão na porta 1883 e redirecionar para a porta 30004 do cluster kubernetes. Como mostrado na imagem abaixo:



Podemos então executar nosso playbook para que as configurações acima sejam aplicadas e para que nosso NGINX seja configurado:

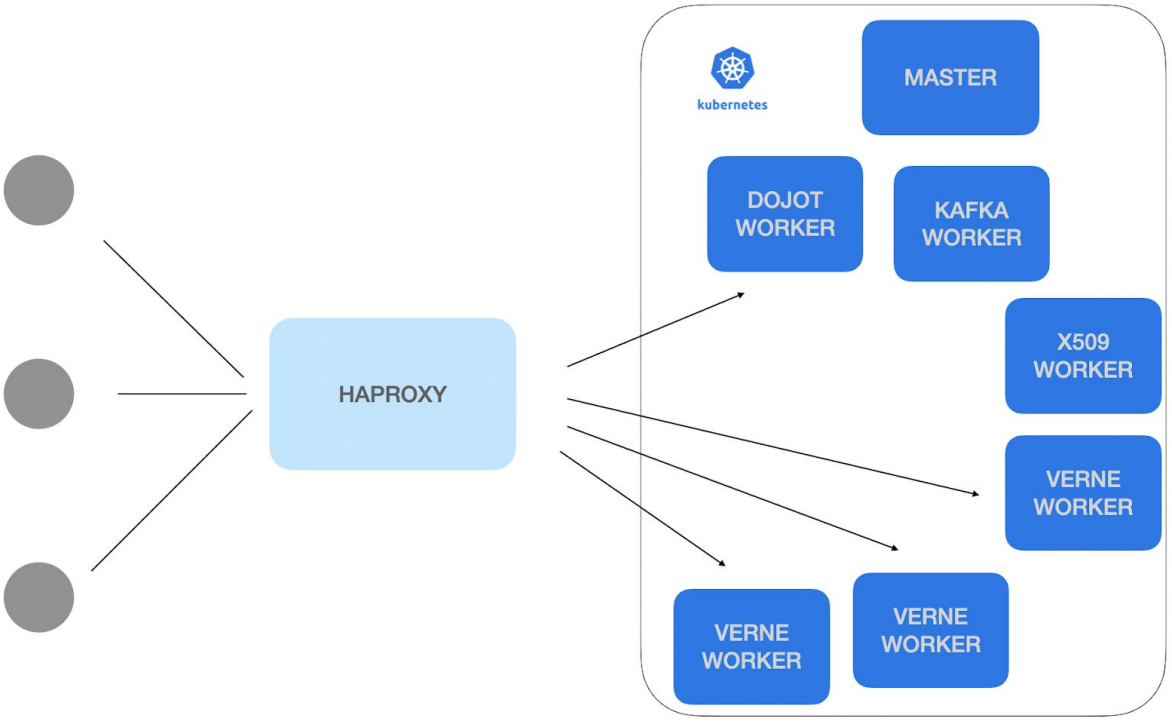
*$ ansible-playbook -K -k **-u dojot** -i inventories/example_local nginx.yaml*

Novamente, podemos acompanhar o resultado da execução do playbook pelos logs. Todos os logs na cor amarela significam que algo foi alterado na máquina remota, a cor verde significa que nada foi alterado e a cor vermelha significa que algo deu erro. Se os logs não demonstrarem nenhum erro na execução, a dojot já estará acessível pelos clientes através do NGINX, ou seja, para acessarmos a interface gráfica da dojot, na nossa configuração de exemplo, utilizaríamos o seguinte endereço no navegador: http://192.168.0.12.

O NGINX redireciona todas as conexões recebidas na porta 80 para a GUI da dojot que está rodando dentro do cluster.

Para um ambiente com até 100.000 dispositivos

Para utilizar a dojot com suporte para até 100.000 dispositivos, devemos configurar o HAProxy como balanceador de carga. Esse balanceador receberá todas as requisições dos clientes e fará o balanceamento de carga entre os nós worker do kubernetes, ou seja, nosso cluster não é acessado diretamente pelo cliente. Como mostrado na imagem abaixo:



Precisamos fazer a configuração do range de portas usadas para tráfego TCP no host do HAProxy, para que possibilite o balanceamento de carga de até 100.000 dispositivos. Para isso, precisamos acessar o host via SSH e editar um arquivo de configuração (No exemplo utilizamos o nano para editar o arquivo, porém você pode usar o editor de texto de sua preferência):

$ ssh dojot@192.168.0.12

$ sudo nano /etc/sysctl.conf

Após acessar o arquivo basta adicionar a seguinte linha de configuração e depois salvar:

net.ipv4.ip_local_port_range = 1024 65535

Por fim, depois que o arquivo já foi modificado e salvo basta rodar o seguinte comando para aplicar as novas configurações:

$ sudo sysctl -p

Para verificar se tudo funcionou basta rodar o seguinte comando:

$ cat /proc/sys/net/ipv4/ip_local_port_range

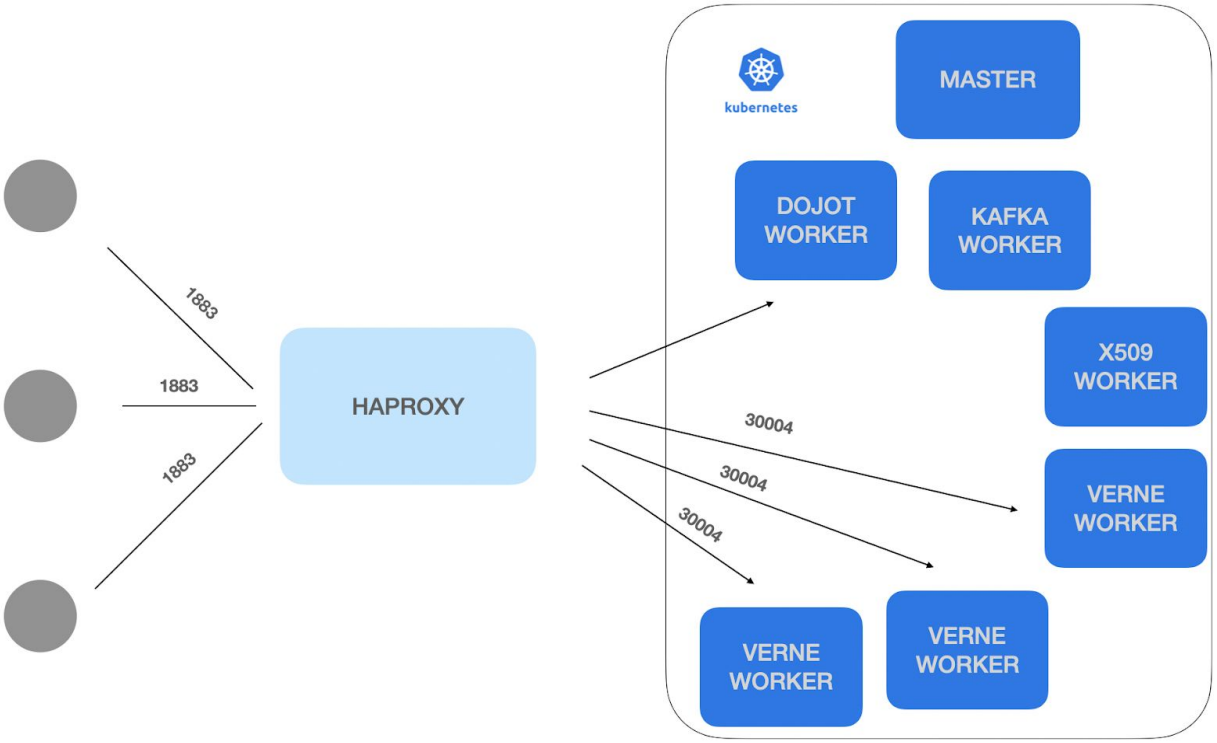
Se a saída do comando for “1024 65535” então tudo funcionou corretamente. Agora já podemos sair do haproxy node.

Voltando para o control node, temos um playbook para instalação e configuração do HAProxy. Mas antes de executá-lo, precisamos configurar o nosso arquivo hosts.yaml novamente para informar ao ansible o host da máquina que terá o HAProxy instalado e também quais são os nós workers, responsáveis por hospedar os micro-serviços utilizados pela dojot. Para isso editamos o nosso arquivo de inventário que está em "*inventories/example_local/hosts.yaml*":

```
---
all:
  hosts:
    ...
    haproxy:
      ansible_host: 192.168.0.12
…
…
children:
…
    apigw_nodes:
      hosts:
        192.168.0.13: #IP do nó dojot-worker
    mqtt_nodes:
      hosts:
        192.168.0.14: #IP do nó verne-worker-1
        192.168.0.15: #IP do nó verne-worker-2
        192.168.0.16: #IP do nó verne-worker-3
    lwm2m_nodes:
      hosts:
        192.168.0.13: #IP do nó dojot-worker
    metrics_nodes:
      hosts:
        192.168.0.13: #IP do nó dojot-worker
```

No exemplo acima, assumimos que o host do HAProxy será o "192.168.0.12", que a máquina dojot-worker tem o IP 192.168.0.13 e que as 3 máquinas verne-worker possuem respectivamente os IPs 192.168.0.14, 192.168.0.15 e 192.168.0.16. Precisamos configurar o arquivo dessa maneira pois os serviços “apigw”, “lwm2m” e “metrics” deverão subir no nó dojot-worker, enquanto o serviço mqtt deve subir nos três nós verne-worker. Essas são as únicas configurações que precisamos fazer.

Lembrando que já alteramos o nosso arquivo de variáveis anteriormente para que o kubernetes utilize portas fixas para os serviços com acesso externo. O playbook do HAProxy utiliza essa mesma configuração, ou seja, se o serviço Iot Agent VerneMQ expõe a porta de acesso para MQTT em 30004, o HAProxy vai receber a conexão na porta 1883 e redirecionar para a porta 30004 do cluster kubernetes. Como mostrado na imagem abaixo:



Podemos então executar nosso playbook para que as configurações acima sejam aplicadas e para que nosso HAProxy seja configurado:

$ ansible-playbook -K -k ***-u dojot*** *-i inventories/example_local haproxy.yaml*

Novamente, podemos acompanhar o resultado da execução do playbook pelos logs. Todos os logs na cor amarela significam que algo foi alterado na máquina remota, a cor verde significa que nada foi alterado e a cor vermelha significa que algo deu erro. Se os logs não demonstrarem nenhum erro na execução, a dojot já estará acessível pelos clientes através do HAProxy.