universidade de aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Gestão Integrada de Redes e Sistemas

Projeto Funkwhale

Relatório 2

2020/2021

Alunos:

Rodrigo Santos, 89180
Daniel Lopes, 87881

Introdução

Neste relatório encontram-se alguns aspetos relativos à segunda parte do projeto de implementação da aplicação Funkwhale num cluster de Kubernetes. Este projeto é do âmbito da cadeira de Gestão Integrada de Redes e Sistemas, do curso MIECT da Universidade de Aveiro.
No primeiro relatório já foi abordada a estratégia de implementação aplicada bem como a de divisão dos componentes pelo cluster.
Será abordado neste segundo relatório a nossa estratégia de implementação de redundância e dos mecanismos usados para balanceamento de carga, bem como a monitorização dos recursos computacionais, automação do aprovisionamento, escalabilidade e elasticidade horizontal, interação com o Load Balancer e sistema de monitorização, e resistência à falha dos componentes previamente instanciados no cluster de Kubernetes.

Redundância dos componentes de software

Para apresentar uma redundância mínima de 1+1 em todos os componentes de software, é preciso replicar todos os componentes de software de modo a que cada um possua no mínimo duas réplicas. No entanto, para certos componentes não é suficiente aumentar o número de réplicas do deployment para se obter redundância. A nossa estratégia foi então a seguinte:

- **PostgreSQL** : 1+2 - (1 Master, 2 Slaves), 1 *master* com permissões de *escrita* e *leitura*, e 2 réplicas *slaves* com permissões para execução apenas de queries de *leitura*. As réplicas *slave* estão em *hot-standby* para caso o master falhe por algum motivo, uma destas assume esse papel.

- **Redis** : 1+2 para redundância, balanceamento de carga e acessos mais rápidos à informação.

- **Celery Worker** : 1+1 réplicas para redundância, e balanceamento de carga, 1 para cada unidade de *Django API.*

- **Celery Beat** : 1+1 réplicas para redundância, e balanceamento de carga, 1 para cada unidade de *Django API.*

- **Django API** : 1+1 réplicas para redundância, e balanceamento de carga.

- **Nginx** : 1+1 réplicas para redundância, e balanceamento de carga.

```yaml
#Create API deployment
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
  namespace: funkwhale
spec:
  replicas: 2
```

Figura 1: Exemplo de replicação de componentes do Deployment da Django API do projeto

Assim, com esta arquitetura, podemos em conjunto com as capacidades do Kubernetes de alocar pods automaticamente de acordo com as necessidades, assegurar um bom desempenho da aplicação para uma comunidade de pequena-média dimensão, tal como a prevista pela documentação da aplicação ***Funkwhale***.

Mecanismos para balanceamento de carga

Num serviço é natural existirem momentos nos quais a quantidade de pedidos e acessos é superior a outros, alturas do ano específicas, ou simplesmente diferentes momentos do dia. Nestas situações é preciso garantir a eficácia e o correto funcionamento do sistema.

Uma vez que todo o sistema tem um mínimo de redundância 1+1 para cada componente garantimos o mínimo de redundância para todo o sistema, bem como algum balanceamento de carga automático levado a cabo por parte dos Serviços criados para cada componente que fazem a distribuição dos pedidos entre as diferentes réplicas de um componente com base num algoritmo de *round robin*.

Contudo, para componentes como a base de dados que opera com 1 master e 2 slaves em hot-standby, a replicação simples do componente não é suficiente, para tal é utilizado uma instância de PgPool-II, que para além de gerir a replicação entre as instâncias da base de dados, faz também a distribuição das leituras entre as várias réplicas da base de dados.

Outro aspecto relevante para o balanceamento de carga, seria fazer balanceamento de carga com base em feedback produzido pelos componentes. Sendo que isto pode ser feito, através de um mecanismo de Third Party Feedback, onde as métricas produzidas pelos componentes são recolhidas, analisadas e normalizadas num dispositivo à parte, dedicado a esse serviço.

Monitorização dos recursos computacionais

Para escalar e providenciar uma aplicação de confiança, é necessário compreender como é que ela se comporta após a sua implementação. Para tal, é possível analisar os containers, pods, serviços, ingressos, etc, existentes num cluster de Kubernetes tal como o que foi usado para implementar a aplicação Funkwhale. O Kubernetes fornece informação detalhada sobre a utilização de recursos na sua Web Dashboard, sobre cada um dos componentes mencionados.

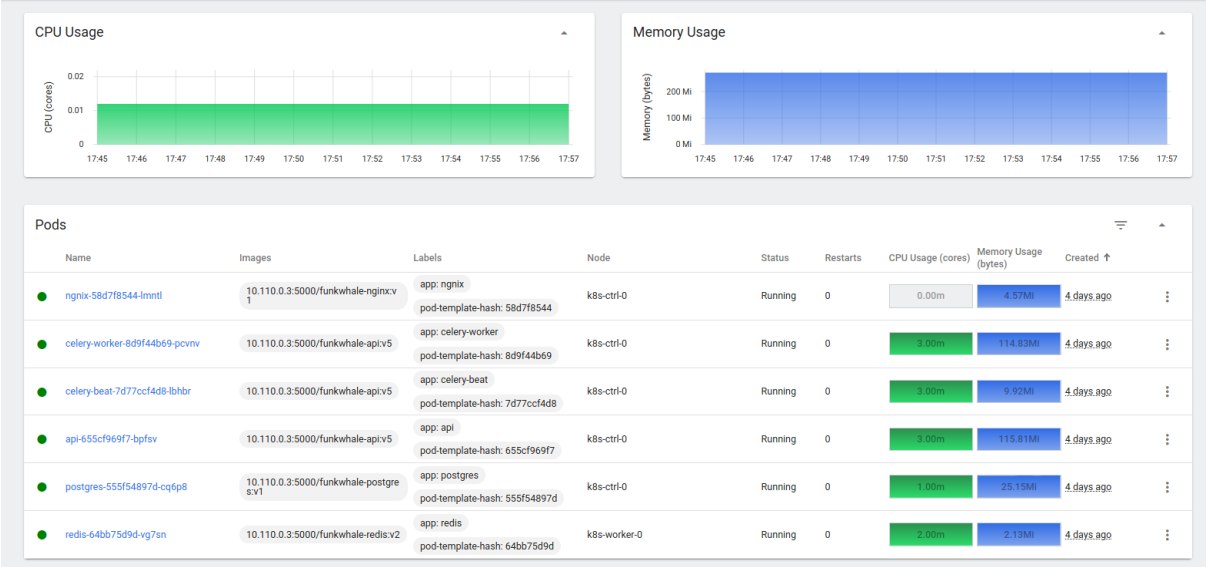

Figura 2: Exemplo de monitorização de recursos Kubernetes (Pods)

Sendo esta uma forma de monitorização passiva do nosso sistema. Contudo, este tipo de monitorização é insuficiente, sendo necessário monitorar de forma mais eficiente e automática.

No Kubernetes é possível utilizar dois pipelines distintos de recolha de dados estatísticos de monitorização, *resource metrics* e *full metrics*, para o nosso projeto pretendemos utilizar um pipeline *full metrics* utilizando **Prometheus**. Permitindo assim, automatizar a resposta do cluster de Kubernetes a métricas consideradas de risco, utilizando ferramentas tais como o Horizontal Pod Autoscaler, mencionado na seção "Automatização do aprovisionamento, escalabilidade e elasticidade horizontal" seguinte.
Com o fim de monitorar os recursos computacionais, foi instanciado um deployment de Prometheus no nosso namespace para recolher as métricas dos componentes, para tal é necessário instanciar também *exporters* para fazer chegar essa informação ao Prometheus. No nosso sistema, tentamos instalar o **PgPool-II** para não só gerir a replicação das instâncias da base de dados, mas também para exportar as métricas necessárias, para o Prometheus. Existindo então dois serviços distintos no nosso deployment de PgPool-II.

```
#Create PgPool deployment
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pgpool
  namespace: funkwhale
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pgpool
  template:
    metadata:
      labels:
```

```
        app: pgpool
    spec:
      containers:
      - name: pgpool
        image: pgpool/pgpool:4.2.2
        ...
      - name: pgpool-stats
        image: pgpool/pgpool2_exporter:1.0
        ...
      volumes:
      - name: pgpool-config
        configMap:
          name: pgpool-config
#Creation of PgPool / PgPool-Stats services
---
apiVersion: v1
kind: Service
metadata:
  name: pgpool
  namespace: funkwhale
spec:
  selector:
    app: pgpool
  ports:
  - name: pgpool-port
    protocol: TCP
    port: 9999
    targetPort: 9999
---
apiVersion: v1
kind: Service
metadata:
  name: pgpool-stats
  namespace: funkwhale
  labels:
    app: pgpool-stats
  annotations:
    prometheus.io/path: /metrics
    prometheus.io/port: "9719"
    prometheus.io/scrape: "true"
spec:
  selector:
    app: pgpool
  ports:
  - name: pgpool-stats-port
    protocol: TCP
    port: 9719
    targetPort: 9719
```

Como é possível verificar no excerto do ficheiro de deployment do PgPool-II apresentado acima, foi instanciado um deployment com 2 imagens a correr, *pgpool:4.2.2* e *pgpool2_exporter:1.0*, inicializadas com a configuração presente no ConfigMap pgpool-config. São também criados 2 serviços, pgpool e pgpool-stats, no qual o primeiro é o serviço para o pgpool, e o segundo é o serviço que ficará responsável de fornecer as métricas ao Prometheus, nos portos definidos.

Automação do aprovisionamento, escalabilidade e elasticidade horizontal

Utilizadores da aplicação esperam sempre que a mesma esteja pronta a ser utilizada quando quiserem várias vezes ao dia, o problema é que a aplicação a determinados momentos necessita de ser atualizada, por forma a corrigir certos problemas, tais como segurança ou performance por exemplo, e isto poderá levar a que a aplicação fique indisponível por determinados momentos, o que não seria o ideal para um sistema, em que haja grande interação por parte dos utilizadores. Para tal, qualquer serviço que queira garantir um acesso confiável e com o máximo de disponibilidade possível, tem de adotar estratégias de automação do aprovisionamento de novos componentes bem como de escalabilidade e elasticidade horizontal.

Para este problema o Kubernetes oferece algumas soluções, das quais nós pretendíamos utilizar o Horizontal Pod Autoscaler, no qual, é possível definir limites de utilização e/ou recursos para que quando um pod atinja esses limites o Kubernetes automaticamente lance novos pods para diminuir a carga sobre os restantes. Deste modo, o Kubernetes, de seguida, automaticamente faz uso das réplicas existentes de cada componente para fazer a distribuição de carga entre eles num modo *round robin*.

```
1 apiVersion: autoscaling/v2beta2
2 kind: HorizontalPodAutoscaler
3 metadata:
4   name: fw-nginx-autoscaler
5   namespace: funkwhale
6 spec:
7   scaleTargetRef:
8     apiVersion: apps/v1
9     kind: Deployment
10    name: nginx
11  minReplicas: 2
12  maxReplicas: 4
13  metrics:
14  - type: Resource
15    resource:
16      name: cpu
17      target:
18        type: Utilization
19        averageUtilization: 50
```

Figura 3: HorizontalPodAutoscaler para o Nginx

Na configuração do *HorizontalPodAutoscaler* acima apresentada, é possível ver que são garantidas no mínimo 2 réplicas (Pods) do deployment de Nginx do nosso sistema, sendo que podem ser criadas até

um máximo de mais 2 outras réplicas, para um total de 4 réplicas, com base na utilização do CPU das réplicas iniciais, caso esta ultrapasse 50%, é instanciada outra réplica.

Outro objetivo do aprovisionamento será arranjar tal forma que permita que a aplicação seja atualizada sem que o sistema fique em baixo, nem que seja por breves momentos. Para isso, será utilizado no Kubernetes o que chamamos de ***"Rolling updates"*** . Como o nosso sistema está distribuído por vários pods contendo e consequentemente esses pods têm várias réplicas, a ideia seria utilizar os ***"Rolling updates"*** no qual as réplicas irão ser atualizadas individualmente uma a uma. Isto faz com que as réplicas com a versão mais antiga continuem a servir os clientes, enquanto as outras vão sendo atualizadas. Quando cada réplica é atualizada, é feito um processo de verificação, que verifica se essa mesma está a correr normalmente e caso sim, já poderá servir os clientes, e o processo será assim, até todas as réplicas serem atualizadas. Com este processo o sistema nunca deixará de funcionar, pois cada réplica é atualizada individualmente, podendo as outras continuarem a servir os clientes. Uma vez que o pod réplica é atualizado, a versão antiga é eliminada, sendo o tráfego reencaminhado para a nova versão.

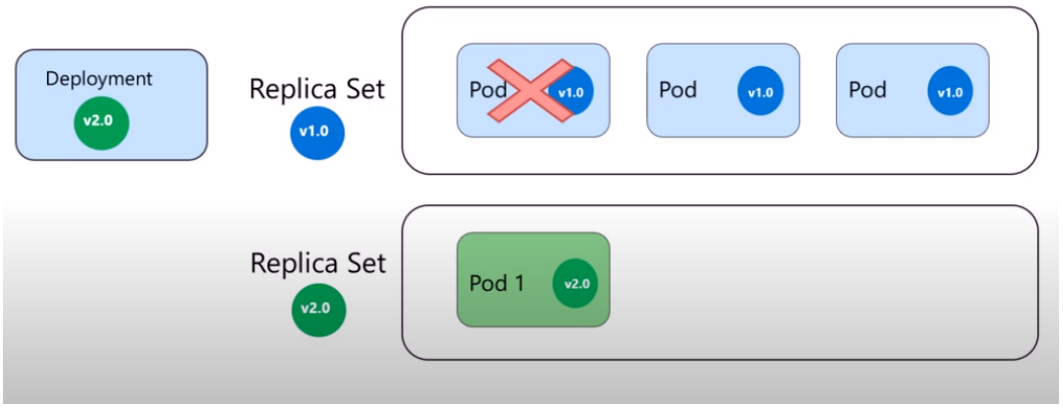

Figura 3: Exemplo gráfico da atualização e a respetiva remoção da versão antiga de uma réplica

Interação com o Load Balancer e sistema de monitorização

Tal como descrito anteriormente o próprio Kubernetes utiliza o balanceamento de carga se o tráfego para um contentor for elevado, ou seja, no nosso projeto, existem várias réplicas para um determinado Pod e caso o tráfego para esse Pod seja elevado, o Kubernetes trata de distribuir a carga para as mesmas réplicas, por forma a reduzir a carga.
No sistema de monitorização usamos a ferramenta Prometheus que é uma ferramenta de monitorização criada para monitorizar ambientes de containers altamente dinâmicos, como por exemplo o Kubernetes.
O nosso sistema implementado no Cluster é composto por vários Pods, sendo que para cada um destes Pods teremos ainda várias réplicas, o que no final de contas teremos uma infraestrutura algo que complexa, portanto, para se monitorizar manualmente, seria complexo, daí a necessidade de se utilizar uma ferramenta de monitorização como o Prometheus.
O prometheus monitoriza constantemente Pod a Pod e caso algum Pod deixe de funcionar ou esteja próximo do máximo dos seus recursos, quer seja, excesso de memória ou por exemplo excesso de uso do processador, o Prometheus tratará de notificar através de alarmes que indicam os problemas e onde

esses problemas ocorrem, por forma a se poder corrigir esses problemas e evitar assim latências nesses Pods.
Por fim, para apresentação das métricas recolhidas e dos dados estatísticos acerca dos componentes integrantes do nosso sistema, poderia ter sido utilizado o **Grafana** que é uma ferramenta de display open-source para monitorização e observação de dados temporais, dados que são apresentados numa dashboard de fácil construção. O Grafana seria uma boa opção por permitir a utilização do Prometheus como datasource, desta maneira, seria possível analisar as métricas dos componentes em tempo real.

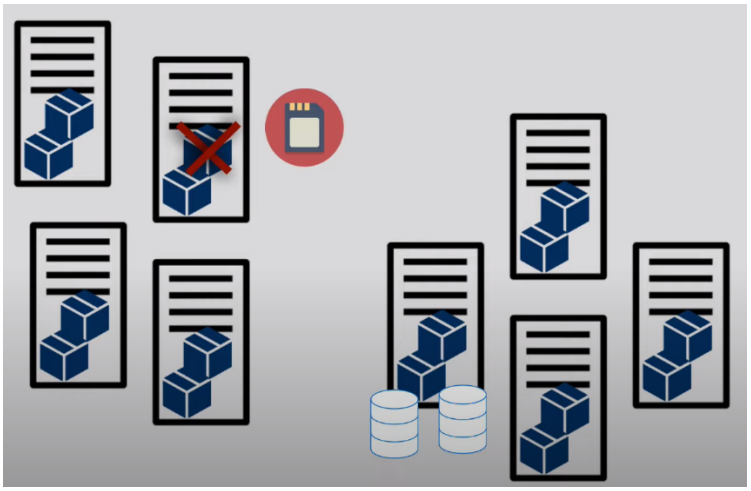

Figura 4: Exemplo gráfico de um container apresentar um erro num determinado pod

Resistência à falha

Para garantir que o serviço implementado é confiável e tem o desempenho desejado, é necessário garantir que há resistência à falha dos seus diversos componentes.
Num cluster de Kubernetes, isto é possível garantir, em parte, automaticamente, uma vez que num cluster Kubernetes os containers/pods podem ser configurados com a funcionalidade de restart, no caso de falha.

```
219        restartPolicy: Always
```

Figura 5: Configuração de restart automático

Com esta configuração assim que detectada a baixa de um pod, este será reiniciado, minimizando assim o tempo no qual se encontra fora de serviço.
Combinando esta funcionalidade com a redundância dos componentes de software implementada, já obtemos um certo nível de resistência à falha.

É também de extrema importância garantir a persistência de informação importante, como no caso da nossa aplicação, utilizadores, passwords e também ficheiros de música adicionados pelos utilizadores. Para tal, foram criados *PersistentVolumes* e respetivos *PersistentVolumeClaims* para os componentes PostgreSQL e Redis presentes na nossa aplicação, de modo, a que mesmo que esses componentes falhem, a informação possa ser recuperada com o mínimo possível de perdas e o normal funcionamento da aplicação restaurado o mais rapidamente possível.

De seguida são mostradas as configurações necessárias para criar estes volumes bem como os seus claims.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: fw-postgres-pv
  namespace: funkwhale
  labels:
    app: postgres
spec:
  storageClassName: local-path
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/srv/funkwhale/data/postgres:/var/lib/postgresql/data"
  claimRef:
    kind: PersistentVolumeClaim
    namespace: funkwhale
    name: fw-postgres-pv-claim
```

Figura 6: Ficheiro yaml de descrição do PersistentVolume para os dados do PostgreSql

É possível observar na imagem, a criação de um *PersistentVolume* de nome fw-postgres-pv, para o nosso namespace "funkwhale", com a capacidade de 1Gib, no modo de acesso *ReadWriteMany* configurado, por fim é feita a reserva do *PersistentVolumeClaim* respetivo.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: fw-postgres-pv-claim
  namespace: funkwhale
  labels:
    app: postgres
spec:
  storageClassName: local-path
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Figura 7: Ficheiro yaml de descrição do PersistentVolumeClaim

Nesta imagem está apresentada a criação de um *PersistentVolumeClaim* para o noso *PersistentVolume* de PostgreSql, de nome fw-postgres-pv-claim, para o nosso namespace, que faz o pedido de 1Gib ao *PersistentVolume*, ou seja, a totalidade dos recursos de armazenamento do volume.
Algo que também apresenta uma importância extrema no que toca a resistência à falha é o backup de dados, que para os dados da base de dados é fundamental. Backups de dados são absolutamente

necessários, e é necessário consoante o tipo de dados e a sua utilização decidir entre backups físicos, ao nível do disco, ou backups numa localização remota.

Conclusão

Apesar de todos os pontos neste relatório terem sido abordados, a nível prático não foi possível implementá-los a todos com sucesso. Contudo, foram criados e alguns mecanismos de balanceamento de carga, monitorização e resistência a falhas, ainda que incompletos por vezes.
Foram encontradas algumas dificuldades na implementação da aplicação no ambiente do cluster de Kubernetes, o que levou a uma instalação incompleta da aplicação a fornecer.
Sob o ponto de vista académico, este projeto foi bastante útil, uma vez que nos proporcionou um contacto com o Kubernetes, dando-nos a possibilidade de aprender e experimentar à medida que o trabalho era desenvolvido.