Tolerância a Falhas e Monitoramento com Kubernetes + Prometheus

Objetivo Geral:

Implantar uma aplicação distribuída com Kubernetes em um ambiente local, aplicando técnicas de tolerância a falhas, auto-recuperação, escalonamento horizontal e monitoramento remoto, utilizando dois notebooks interligados.

Descrição da Implantação

- ✅ Notebook A – Cluster Kubernetes
  - Criar um cluster local com Minikube.
  - Implantar uma aplicação com múltiplos pods, iniciando com duas réplicas.
  - **Atenção**: a aplicação não pode ser reutilizada de atividades anteriores da disciplina.
  - Habilitar o mecanismo de auto-healing, garantindo que pods sejam recriados automaticamente em caso de falha.
  - Configurar o Horizontal Pod Autoscaler (HPA) com base no uso de CPU.
- ✅ Notebook B – Prometheus
  - Implantar o Prometheus para monitoramento remoto do cluster Kubernetes no Notebook A.
  - Exibir métricas em tempo real, como:
  - Número de pods ativos
  - Uso de CPU
  - Estado dos pods (Running, Failed, Pending)
  - Ações disparadas pelo HPA

Demonstração de Tolerância a Falhas

Antes de iniciar os testes, mostre que sua aplicação está rodando normalmente com as duas réplicas previstas.

- 1.Deleção Manual de Pod (Auto-Healing)
  - Delete manualmente um dos pods da aplicação.
  - Observe como o controlador do Kubernetes detecta a falha e recria automaticamente um novo pod.
- 📌 O que demonstrar:
  - A recriação rápida de um novo pod após a deleção.
  - O status de “Terminating” do pod anterior e a entrada do novo em “Running”.
  - A atualização das métricas no Prometheus (mudança no número de pods ativos, novo identificador, tempo de reação).
- 2.Sobrecarga de CPU (Escalonamento Horizontal)
  - Gere uma carga de CPU artificial em um ou mais pods da aplicação.
  - Observe como o HPA aumenta automaticamente o número de réplicas para atender à demanda.
- 📌 O que demonstrar:
  - A elevação do consumo de CPU no Prometheus.
  - A criação de novos pods, respeitando o limite configurado no HPA.
  - O tempo de resposta entre o pico de CPU e o escalonamento automático.
  - A redução do número de réplicas após estabilização (se aplicável).

Resultados

Primeiramente, vamos configurar o ambiente, onde recomendamos fortemente que siga a documentação oficial com base em seu sistema operacional.

Usaremos Linux Ubuntu 24.04 LTS neste projeto

Instalando Minikube

Usaremos o comando a baixo para instalar:

```
curl                                                                -LO
https://storage.googleapis.com/minikube/releases/latest/minikube_l
atest_amd64.deb
sudo dpkg -i minikube_latest_amd64.deb
```

Use o comando `minikube start` para checar se o minikube esta funcionando corretamente.

O resultado esperado será semelhante a este:

```
😄  minikube v1.35.0 on Ubuntu 24.04
✨  Automatically selected the docker driver. Other choices: none,
ssh
📌  Using Docker driver with root privileges
👍  Starting "minikube" primary control-plane node in "minikube"
cluster
🚜  Pulling base image v0.0.46 ...
```

Para interagir com o minikube, iremos instalar o kubectl (Linha de Comandos do Kubernets) seguindo a documentação oficial

Comandos usados

```
curl     -LO     "https://dl.k8s.io/release/$(curl     -L     -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
curl     -LO     "https://dl.k8s.io/release/$(curl     -L     -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha2
56"
```

Use este comando para testar se o kubectl foi baixado com sucesso:

```
echo "$(cat kubectl.sha256)  kubectl" | sha256sum --check
```

Resultado esperado:

```
kubectl: OK
```

Agora o instale usando:

```
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Use este comando para testar se a instalação aconteceu corretamente:

```
kubectl version --client
```

Resultado esperado:

```
Client Version: v1.32.3
Kustomize Version: v5.5.0
```

Criar um cluster local com Minikube

Inicialmente vamos criar duas replicas da nossa aplicação mangalivre e uma do nosso banco de dados para centralizar os dados.

Todos os nossos arquivos de configuração do `minikube` devem ficar na pasta k8s para organizar melhor o projeto.

Iremos criar um arquivo chamado `mangalivre-db-deployment.yaml` para o banco de dados com o seguinte conteúdo:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mangalivre-db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mangalivre-db
  template:
    metadata:
      labels:
        app: mangalivre-db
    spec:
      containers:
        - name: mangalivre-db
          image: mangalivre-db
```

```
          imagePullPolicy: Never
          env:
            - name: POSTGRES_USER
              value: "mangalivre"
            - name: POSTGRES_PASSWORD
              value: "mangalivre"
            - name: POSTGRES_DB
              value: "mangalivre"
          ports:
            - containerPort: 5432
          volumeMounts:
            - name: postgres-data
              mountPath: /var/lib/postgresql/data
      volumes:
        - name: postgres-data
          emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: mangalivre-db
spec:
  ports:
    - port: 5432
      targetPort: 5432
  selector:
    app: mangalivre-db
```

Agora vamos criar um arquivo chamado `mangalivre-app-deployment.yaml` para a nossa aplicação:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mangalivre-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: mangalivre-app
  template:
    metadata:
      labels:
        app: mangalivre-app
    spec:
      containers:
        - name: mangalivre-app
          image: mangalivre-app:latest
          imagePullPolicy: Never
          ports:
```

```
            - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: mangalivre-app
spec:
  ports:
    - port: 3000
      targetPort: 3000
  selector:
    app: mangalivre-app
  type: NodePort
```

Com esses arquivos em mãos, iremos usar os arquivos `Dockerfile` preparador anteriormente para iniciar nossa aplicação. Caso tenha curiosidade em saber o conteúdo destes arquivos, clique aqui para acessar o `Dockerfile` do Banco de Dados e clique aqui para acessar o `Dockerfile` da aplicação.

Antes de iniciar, crie um arquivo `.env` dentro da pasta mangalivre com o seguinte conteúdo:

```
DB_USER=mangalivre
DB_PASSWORD=mangalivre
DB_HOST=mangalivre-db
DB_PORT=5432
DB_NAME=mangalivre
```

Inicie o Minikube usando:

```
minikube start
```

Caso queira conferir se ele realmente iniciou corretamente, use:

```
minikube status
```

e caso queira reiniciar, use:

```
minikube stop
minikube start
```

Feito isso, agora vamos criar as imagens do nosso banco de dados e aplicação usando o Docker do Minikube com os seguintes comandos:

```
eval $(minikube docker-env)
docker build -t mangalivre-db:latest ./mangalivre/database/
docker build -t mangalivre-app:latest ./mangalivre/
```

Use o kubectl para aplicar os arquivos de configuração:

```
kubectl apply -f k8s/mangalivre-db-deployment.yaml
kubectl apply -f k8s/mangalivre-app-deployment.yaml
```

Verifique se os pods e serviços foram criados corretamente:

```
kubectl get pods
kubectl get services
```

Caso algum `pod` tenha falhado, tente criar novamente usando:

```
kubectl delete pod -l app=mangalivre-app
kubectl apply -f k8s/mangalivre-app-deployment.yaml
```

Caso precise ver a estrutura e erros, use estes comandos

```
kubectl describe pod NOME_DO_POD
kubectl logs NOME_DO_POD
```

Obs: Lembre de adaptar para o `pod` de sua necessidade

Obtenha o URL do serviço do aplicativo com o comando:

```
minikube service mangalivre-app
```

Isso abrirá a aplicação no navegador no navegador.

Habilitando o mecanismo de auto-healing, garantindo que pods sejam recriados automaticamente em caso de falha

No Kubernetes, o mecanismo de auto-healing já está habilitado por padrão para os pods gerenciados por um Deployment. O controlador do Deployment monitora os pods e recria automaticamente qualquer pod que falhe ou seja excluído.

No entanto, você podemos garantir que o comportamento de auto-healing esteja configurado corretamente e ajustar algumas configurações para melhorar a resiliência.

Ao adicionar um `livenessProbe` Para melhorar o auto-healing, isso permite que o Kubernetes detecte se o contêiner está em um estado inconsistente (por exemplo, travado) e reinicie o pod automaticamente.

Iremos implementar isto em nosso `app` ao modificar o arquivo `mangalivre-app-deployment.yaml` no bloco `containers`:

```
containers:
  - name: mangalivre-app
    image: mangalivre-app:latest
    imagePullPolicy: Never
    ports:
      - containerPort: 3000
    livenessProbe:
      httpGet:
        path: /
        port: 3000
      initialDelaySeconds: 5
      periodSeconds: 10
```

obs:

- **httpGet**: Verifica se o endpoint / na porta 3000 está respondendo.
- **initialDelaySeconds**: Aguarda 5 segundos antes de iniciar as verificações.
- **periodSeconds**: Realiza a verificação a cada 10 segundos.

Podemos adicionar também um `readinessProbe` para garantir que o pod só seja considerado pronto quando estiver realmente funcional. Isso evita que o Kubernetes envie tráfego para um pod que ainda está inicializando.

Iremos implementar isto em nosso `app` ao modificar o arquivo `mangalivre-app-deployment.yaml` no bloco `containers` novamente:

```
    readinessProbe:
      httpGet:
        path: /
        port: 3000
      initialDelaySeconds: 5
      periodSeconds: 10
```

Resultado final para o arquivo `mangalivre-db-deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mangalivre-app
spec:
  replicas: 2
```

```
  selector:
    matchLabels:
      app: mangalivre-app
  template:
    metadata:
      labels:
        app: mangalivre-app
    spec:
      containers:
        - name: mangalivre-app
          image: mangalivre-app:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 3000
          livenessProbe:
            httpGet:
              path: /
              port: 3000
            initialDelaySeconds: 5
            periodSeconds: 10

          readinessProbe:
            httpGet:
              path: /
              port: 3000
            initialDelaySeconds: 5
            periodSeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: mangalivre-app
spec:
  ports:
    - port: 3000
      targetPort: 3000
  selector:
    app: mangalivre-app
  type: NodePort
```

Vamos aplicar as mudanças usando

```
kubectl apply -f k8s/mangalivre-app-deployment.yaml
```

Agora vamos testar se realmente está funcionando.

Veja os pods usando

```
kubectl get pods
```

Seu resultado será proximo deste

```
NAME                                READY   STATUS    RESTARTS   AGE
mangalivre-app-57885677cc-grxnz   1/1     Running   0          89s
mangalivre-app-57885677cc-ps6wn   1/1     Running   0          75s
mangalivre-db-f76d86c6d-h4qth     1/1     Running   0          15m
```

Agora ao deletar um dos pods do app, usando este comando:

```
kubectl delete pod mangalivre-app-57885677cc-grxnz
```

Ao listar novamente usando:

```
kubectl get pods
```

Resultado:

```
NAME                                READY   STATUS    RESTARTS   AGE
mangalivre-app-57885677cc-g9fd8   1/1     Running   0          33s
mangalivre-app-57885677cc-ps6wn       1/1           Running       0
3m28s
mangalivre-db-f76d86c6d-h4qth     1/1     Running   0          17m
```

Agora em caso de algo acontecer com nossa aplicação, os pods serão recriados.

Configurando o Horizontal Pod Autoscaler (HPA) com base no uso de CPU

O HPA depende de métricas para funcionar. No Minikube, precisamos habilitar o `Metrics Server`, que coleta métricas de uso de CPU e memória.

Para habilitar, use este comando:

```
minikube addons enable metrics-server
```

Verifique se o Metrics Server está funcionando usando:

```
kubectl get deployment -n kube-system metrics-server
```

Resultado esperado

```
mauriciobenjamin700@mauriciobenjamin700-Latitude-
5300:~/projects/course/ufpi/minikube-test$ kubectl get deployment
-n kube-system metrics-server
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
metrics-server   1/1     1            0           6s
```

O HPA precisa de limites de CPU (resources.requests.cpu) configurados no Deployment para funcionar. Atualize o arquivo `mangalivre-app-deployment.yaml` para incluir os recursos:

```yaml
spec:
  containers:
    - name: mangalivre-app
      image: mangalivre-app:latest
      imagePullPolicy: Never
      ports:
        - containerPort: 3000
      resources:
        requests:
          cpu: "200m" # 200 milicores (0.2 CPU)
        limits:
          cpu: "500m" # 500 milicores (0.5 CPU)
```

Ao final, seu arquivo `mangalivre-app-deployment.yaml` estará desta forma:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mangalivre-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: mangalivre-app
  template:
    metadata:
      labels:
        app: mangalivre-app
    spec:
      containers:
        - name: mangalivre-app
          image: mangalivre-app:latest
          imagePullPolicy: Never
          ports:
```

```
            - containerPort: 3000
          livenessProbe:
            httpGet:
              path: /
              port: 3000
            initialDelaySeconds: 5
            periodSeconds: 10

          readinessProbe:
            httpGet:
              path: /
              port: 3000
            initialDelaySeconds: 5
            periodSeconds: 10
          resources:
            requests:
              cpu: "200m" # 200 milicores (0.2 CPU)
            limits:
              cpu: "500m" # 500 milicores (0.5 CPU)
---
apiVersion: v1
kind: Service
metadata:
  name: mangalivre-app
spec:
  ports:
    - port: 3000
      targetPort: 3000
  selector:
    app: mangalivre-app
  type: NodePort
```

Aplique as mudanças usando:

```
kubectl apply -f k8s/mangalivre-app-deployment.yaml
```

Agora crie um arquivo chamado `mangalivre-app-hpa.yaml` para configurar o HPA com o seguinte conteúdo:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: mangalivre-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: mangalivre-app
```

```
  minReplicas: 2
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
             averageUtilization: 50 # Escala quando o uso de CPU
ultrapassar 50%
```

Aplique o HPA usando:

```
kubectl apply -f k8s/mangalivre-app-hpa.yaml
```

Resultado esperado:

```
horizontalpodautoscaler.autoscaling/mangalivre-app-hpa created
```

Use o comando abaixo para verificar o status do HPA:

```
kubectl get hpa
```

Você verá algo como:

```
NAME                       REFERENCE                          TARGETS
MINPODS   MAXPODS   REPLICAS   AGE
mangalivre-app-hpa         Deployment/mangalivre-app          cpu:
<unknown>/50%   2         5         2          58s
```

Obs:

- **TARGETS**: Mostra o uso atual de CPU em relação ao alvo configurado (50%).
- **REPLICAS**: Mostra o número atual de réplicas.

Para testar o HPA, podemos gerar uma carga de CPU nos pods do mangalivre-app. Usaremos a ferramenta kubectl exec para executar um script que consome CPU.

Dado nossos pods que podemos escolher usando `kubectl get pods`:

```
NAME                              READY   STATUS    RESTARTS   AGE
mangalivre-app-57885677cc-g9fd8   1/1     Running   0          19m
```

```
mangalivre-app-57885677cc-ps6wn   1/1     Running   0          22m
mangalivre-db-f76d86c6d-h4qth     1/1     Running   0          37m
```

Vamos escolher `mangalivre-app-57885677cc-ps6wn` para o teste.

Execute o script a baixo em outro de seus terminais:

```
kubectl exec -it mangalivre-app-57885677cc-ps6wn -- /bin/sh -c "yes > /dev/null &"
```

Verifique novamente o HPA usando:

```
kubectl get hpa
```

E os pods usando:

```
kubectl top pods
```

Você verá novos pods sendo criados para lidar com a carga.

```
NAME                              CPU(cores)   MEMORY(bytes)
mangalivre-app-57885677cc-g9fd8   1m           86Mi
mangalivre-app-57885677cc-ps6wn   1000m        93Mi
mangalivre-db-f76d86c6d-h4qth     1m           65Mi
mauriciobenjamin700@mauriciobenjamin700-Latitude-
5300:~/projects/course/ufpi/minikube-test$
```

Quando a carga de CPU diminuir, o HPA reduzirá automaticamente o número de réplicas para o valor mínimo configurado (minReplicas).

Notebook B – Prometheus

Instalação e Configuração do Prometheus com Helm + Monitoramento Remoto

Pré-requisitos

- Kubernetes cluster (Minikube, Kind, EKS, etc.)
- `kubectl` instalado e configurado
- `helm` instalado – Instruções oficiais
- Conectividade entre máquinas para monitoramento remoto

Instalação do Prometheus com Helm

1. Adicionar o repositório do Prometheus

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
```

2. Criar um namespace para o Prometheus (opcional)

```
kubectl create namespace monitoring
```

3. Instalar o Prometheus

```
helm install prometheus prometheus-community/prometheus --namespace monitoring
```

4. Verificar os pods

```
kubectl get pods -n monitoring
```

5. Acessar a interface web do Prometheus localmente

```
kubectl port-forward -n monitoring svc/prometheus-server 9090:80
```

Acesse via navegador: http://localhost:9090

Monitorar Outros Clusters na Rede

1. Crie o arquivo `values.yaml` com os scrapes remotos

```
server:
  global:
    scrape_interval: 15s
  extraScrapeConfigs:
```

```
    - job_name: 'remote-cluster-node1'
      static_configs:
        - targets: ['192.168.1.101:9100']
    - job_name: 'remote-cluster-node2'
      static_configs:
        - targets: ['192.168.1.102:9100']
```

2. Instalar (ou atualizar) Prometheus com essa configuração

Nova instalação

```
helm install prometheus prometheus-community/prometheus -f values.yaml --namespace monitoring
```

Atualização

```
helm upgrade prometheus prometheus-community/prometheus -f values.yaml --namespace monitoring
```

3. Rodar Node Exporter nas máquinas remotas

```
docker run -d --name node-exporter -p 9100:9100 --restart=always prom/node-exporter
```

Exemplos de Queries Prometheus para o Pod `mangalivre-app`

Uso de CPU

```
sum(rate(container_cpu_usage_seconds_total{pod="mangalivre-app"}[5m]))
```

```
rate(container_cpu_usage_seconds_total{pod="mangalivre-app"}[5m])
```

Uso de Memória

```
container_memory_usage_bytes{pod="mangalivre-app"}
```

```
container_memory_rss{pod="mangalivre-app"}
```

Disco

```
rate(container_fs_writes_bytes_total{pod="mangalivre-app"}[5m])
```

```
rate(container_fs_reads_bytes_total{pod="mangalivre-app"}[5m])
```

Rede

```
rate(container_network_receive_bytes_total{pod="mangalivre-app"}
[5m])
```

```
rate(container_network_transmit_bytes_total{pod="mangalivre-app"}
[5m])
```

Status do Pod

```
kube_pod_status_phase{pod="mangalivre-app", phase="Running"}
```

Requisições HTTP (se o app expõe essa métrica)

```
rate(http_requests_total{pod="mangalivre-app"}[1m])
```

Consultas com Regex

Todas as métricas com o pod exato

```
{pod="mangalivre-app"}
```

Todas as métricas que começam com `mangalivre` (regex)

```
{pod=~"mangalivre.*"}
```

Com namespace específico

```
{pod=~"mangalivre.*", namespace="default"}
```

Conclusão

Este foi o nosso trabalho sobre Tolerância a Falhas e Monitoramento com Kubernetes + Prometheus. Em caso de dúvidas podem abrir uma issue neste projeto ou entrar em contato com algum dos membros autores a baixo:

- Mauricio Benjamin
- Clistenes Rogder
- Pedro Vital