UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie informatique

RAPPORT APP3

Systèmes répartis
GIF391

Présenté à
Équipe de formateurs de la session S3

Présenté par
Raphael Bouchard – bour0703
Alexis Guérard – guea0902

Sherbrooke – 12 juin 2023

TABLE DES MATIÈRES

LISTE DES FIGURES

1. DESCRIPTION DE LA SOLUTION POUR LES SIX ÉTAPES DE LA PROBLÉMATIQUE

1.1 EXEMPLES D'UTILISATION DES COMMANDES NOUVEAU, RETRAIT ET LISTE

Pour la première étape, nous avons utilisé Docker pour créer un système monolithique minimaliste. À l'aide des commandes Docker appropriées, comme docker ps, docker images, docker run, docker exec, docker stop et docker build, nous avons géré un conteneur basé sur l'image construite "arret01" dans la machine virtuelle à l'aide de la commande « docker build -t arret01 . » . Nous avons vérifié le système en faisant la commande « docker run -it arret01 ». Les commandes "nouveau", "retrait" et "liste" étaient disponibles pour ajouter, supprimer et afficher les citations dans le serveur. La commande "nouveau" permet d'ajouter une nouvelle citation générée aléatoirement à l'aide de la commande "fortune" dans le serveur monolithique. La commande "retrait" permet de supprimer une citation spécifique. La commande "liste" affiche toutes les citations présentes. La commande "aide" fournit des informations détaillées sur les différentes commandes disponibles.

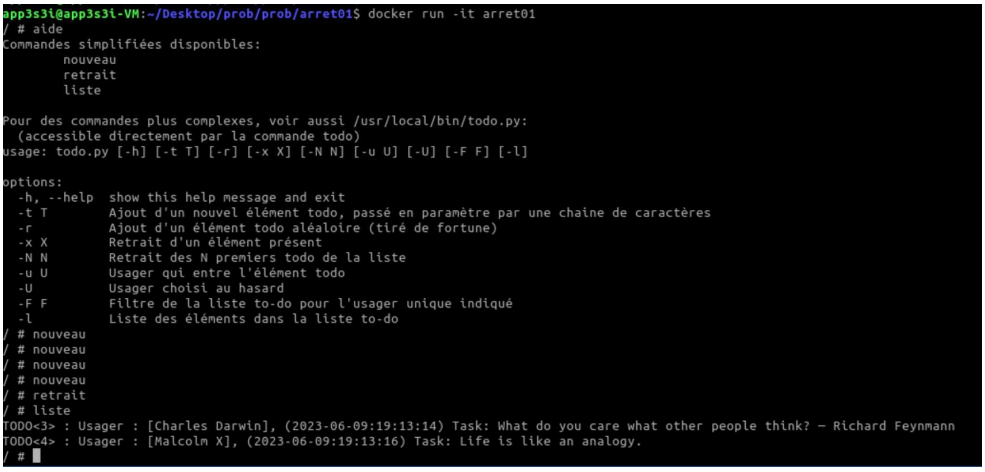

Figure 1 : Exemples d'utilisation des commandes nouveau, retrait et liste

1.2 FICHIER DE CONFIGURATION FONCTIONNEL

Pour la deuxième étape de la problématique, nous avons corrigé le fichier de configuration pour assurer le bon fonctionnement du système. Nous avons ajouté la configuration du réseau pour le serveur, le client et le gestionnaire, ainsi que les environnements nécessaires. De plus, nous avons utilisé un volume pour permettre de voir les modifications dans le fichier "todo.txt". Pour démarrer le système, nous avons construit les trois images à l'aide des Dockerfiles fournis, puis nous avons exécuté la commande "docker-compose up" pour lancer les conteneurs. Ensuite, nous avons exécuté le conteneur du client et utilisé la commande "call" pour créer des citations automatiquement et les ajouter dans notre fichier "todo.txt".

docker-compose.yaml

```yaml
version: '3.2'

# Définition des services docker
services:
    # Ces services comprennent un serveur todo, un client, et un superviseur qui observe les tâches
    c1:
      # Client - Ajoute et retire des tâches automatiquement
      container_name: arret02-c1
      image: arret02-client
      tty: true
      networks:
        - arret02-todo-net
      environment:
        - MONSERVEUR='s1'
        - MONNOM='c1'

    s1:
      # Serveur - Traite les demandes todo
      container_name: arret02-s1
      image: arret02-serveur
      tty: true
      volumes:
        - "./tmp_todo1:/todo"
      networks:
        - arret02-todo-net


    gestionnaire:
      # Permet d'observer les tâches en cours
      container_name: arret02-gestionnaire
      image: arret02-gestionnaire
      tty: true
      environment:
        - MONSERVEUR='s1'
        - MONNOM='c1'
      networks:
        - arret02-todo-net

# Réseau par défaut
networks:
  arret02-todo-net:
```

Figure 2 : Fichier de configuration

1.3 FICHIER YML UTILISÉ POUR DOCKER-COMPOSE

Pour la troisième étape de la problématique, nous avons modifié le fichier .yml pour docker-compose afin de gérer la configuration du système avec trois paires client-serveur distinctes et une troisième image pour le gestionnaire qui se connecte aux trois serveurs. Dans le fichier .yml, nous avons défini les services pour chaque client et serveur en utilisant les fichiers Dockerfiles. Chaque client a été lié à un serveur spécifique en spécifiant les conteneurs correspondants dans la configuration du réseau. Nous avons également utilisé des volumes pour chaque serveur afin de voir les modifications apportées aux listes "todo" des clients. Cela a permis aux clients d'utiliser le système individuellement, tout en permettant au gestionnaire d'accéder en lecture aux listes "todo" des trois clients.

```yaml
version: '3.2'

# Définition des services docker
services:
    # Ces services comprennent trois
    c1:
      # Premier client - Ajoute et r
      container_name: arret03-c1
      image: arret03-client1
      tty: true
      networks:
        - arret03-todo-net
      depends_on:
        - s1
      environment:
        - MONSERVEUR=s1
        - MONNOM=c1

    c2:
      container_name: arret03-c2
      image: arret03-client2
      tty: true
      networks:
        - arret03-todo-net
      depends_on:
        - s2
      environment:
        - MONSERVEUR=s2
        - MONNOM=c2

    c3:
      container_name: arret03-c3
      image: arret03-client3
      tty: true
      networks:
        - arret03-todo-net
      depends_on:
        - s3
      environment:
        - MONSERVEUR=s3
        - MONNOM=c3

    s1:
      # Premier serveur - Traite les demande
      container_name: arret03-s1
      image: arret03-serveur1
      tty: true
      volumes:
        - "./tmp_todo1:/todo"
      networks:
        - arret03-todo-net

    s2:
      container_name: arret03-s2
      image: arret03-serveur2
      volumes:
        - "./tmp_todo2:/todo"
      tty: true
      networks:
        - arret03-todo-net


    s3:
      container_name: arret03-s3
      image: arret03-serveur3
      volumes:
        - "./tmp_todo3:/todo"
      tty: true
      networks:
        - arret03-todo-net

    gestionnaire:
      container_name: arret03-gestionnaire
      image: arret03-gestionnaire
      tty: true
      environment:
        - MONSERVEUR='s1,s2,s3'
        - MONNOM='c1,c2,c3'
      networks:
        - arret03-todo-net

# Réseau par défaut
networks:
  arret03-todo-net:
```

Figure 3 : Fichier YML de l'arret03

1.4 EXPLICATION DES PROBLÈMES INHÉRENTS À CETTE ÉTAPE

À cette quatrième étape, il fallait faire fonctionner un seul serveur pour les trois clients, avec également un gestionnaire. Cependant, le serveur ne garantit pas l'atomicité des opérations dans le fichier todo.txt. Cela soulève des problèmes potentiels, car plusieurs clients pourraient tenter d'accéder et de modifier simultanément le fichier, ce qui peut entraîner des conflits, des incohérences ou des pertes de données.

La problématique principale réside dans la synchronisation des opérations effectuées par les différents développeurs sur le fichier partagé. En l'absence de garantie d'atomicité, il peut y avoir des situations où deux clients tentent de modifier le fichier en même temps, ce qui peut entraîner des conflits de données ou des écritures incomplètes. Il peut également y avoir des problèmes de cohérence lorsque des opérations simultanées entraînent des incohérences dans le contenu du fichier.

Un autre défi est de gérer efficacement les accès concurrents au fichier partagé. Si plusieurs développeurs tentent de lire, ajouter ou supprimer des éléments simultanément, il peut y avoir des problèmes de concurrence qui peuvent compromettre l'intégrité et la cohérence des données.

Il faudrait donc mettre en place des mécanismes de synchronisation et de verrouillage appropriés pour garantir la cohérence des opérations effectuées.

1.5 FICHIER YML UTILISÉ POUR DOCKER-COMPOSE

Au cinquième arrêt, nous avions mis en place une solution où le serveur principal (todo) avait été amélioré pour fonctionner avec une instance de l'image de la base de données PostgreSQL (todo-bd). Cette base de données assurait l'atomicité des requêtes, garantissant ainsi la cohérence des opérations effectuées sur les données. L'image PostgreSQL était tirée du registre par défaut de Docker, ce qui facilitait son intégration dans le système. Nous avions donc modifié le fichier docker-compose.yml qui regroupait les trois clients, le gestionnaire, le serveur todo-bd et la base de données PostgreSQL, permettant ainsi le déploiement et l'exécution de l'ensemble du système avec Docker Compose.

```yaml
version: '3.2'

services:
  # Premier client - Ajoute et retire des tâches
  c1:
    container_name: arret05-c1
    image: a5-client1
    build:
      context: .
      dockerfile: Dockerfile.client
    depends_on:
      - s1
    environment:
      - MONSERVEUR=s1
      - MONNOM=c1
    tty: true

  # Deuxième client - Ajoute et retire des tâche
  c2:
    container_name: arret05-c2
    image: a5-client2
    build:
      context: .
      dockerfile: Dockerfile.client
    depends_on:
      - s1
    environment:
      - MONSERVEUR=s1
      - MONNOM=c2
    tty: true

  # Troisième client - Ajoute et retire des tâch
  c3:
    container_name: arret05-c3
    image: a5-client3
    build:
      context: .
      dockerfile: Dockerfile.client
    depends_on:
      - s1
    environment:
      - MONSERVEUR=s1
      - MONNOM=c3
    tty: true

  # Premier serveur - Traite les demandes todo
  s1:
    container_name: arret05-s1
    image: a5-serveur
    build:
      context: .
      dockerfile: Dockerfile.serveur
    tty: true
    depends_on:
      - gestionnaire
    environment:
      - "DB_HOST=db"


  # Permet d'observer les tâches en cours
  gestionnaire:
      container_name: arret05-gestionnaire
      image: a5-gestionnaire
      build:
        context: .
        dockerfile: Dockerfile.gestionnaire
      tty: true
      environment:
        - MONSERVEUR='s1'
        - MONNOM='c1,c2,c3'

  # Permet d'enregistrer les tâches
  db:
    container_name: arret05-db
    image: postgres:15
    environment:
      - "POSTGRES_PASSWORD=postgres"
      - "POSTGRES_USER=postgres"
      - "POSTGRES_DB=postgres"
    volumes:
      - todo-db:/var/lib/postgresql/datum
      - ./init_db/sql:/docker-entrypoint-initdb.d/

volumes:
  todo-db:
```

Figure 4 : Fichier YML de l'arret05

1.6 FICHIER YML UTILISÉ POUR L'ORCHESTRATION AVEC KUBERNETES

Au sixième arrêt, nous avons évolué notre système pour le rendre utilisable à grande échelle. Pour éviter les problèmes de congestion sur une seule base de données, nous avons dupliquer la base de données. Pour cela, nous avons utilisé Kubernetes pour répartir les bases de données dans plusieurs pods, permettant ainsi le déploiement sur différentes machines physiques. En utilisant des fichiers yml adaptés à Kubernetes, nous avons pu développer et faire fonctionner efficacement le système à grande échelle, en exploitant les fonctionnalités de réplication offertes par Kubernetes.

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: a6-client
  labels:
    app: todo
spec:
  replicas: 10
  selector:
    matchLabels:
      app: todo
  template:
    metadata:
      labels:
        app: todo
    spec:
      containers:
        - name: c1
          image: a6-client
          tty: true
          imagePullPolicy: Never
          env:
          - name: MONSERVEUR
            value: 's1' # Must match the service name in the server yaml
```

Figure 5 : client_pod.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: db-connect
spec:
  selector:
    name: db-todo
  ports:
  - name: db-port
    protocol: TCP
    port: 5432
    targetPort: db-port
---

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: a5-db
  labels:
    app: todo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: todo
  template:
    metadata:
      labels:
        app: todo
        name: db-todo
    spec:
      containers:
        - name: db
          image: postgres:15
          imagePullPolicy: "IfNotPresent"
          ports:
            - containerPort: 5432
              name: db-port
          envFrom:
            - secretRef:
                name: db-secrets
          volumeMounts:
            - name: mysql-initdb
              mountPath: /docker-entrypoint-initdb.d
            - name: postgredb
              mountPath: /var/lib/postgresql/data

      volumes:
        - name: postgredb
          persistentVolumeClaim:
            claimName: postgres-pv-claim
        - name: mysql-initdb
          configMap:
            name: mysql-initdb-config
```

Figure 6 : db_pod.yaml

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: a6-gestionnaire
  labels:
    app: todo
spec:
  replicas: 10
  selector:
    matchLabels:
      app: todo
  template:
    metadata:
      labels:
        app: todo
    spec:
      containers:
        - name: gest
          image: a6-gestionnaire
          tty: true
          imagePullPolicy: Never
          env:
          - name: MONSERVEUR
            value: 's1'  # Must match the service name in the server yaml
```

Figure 7 : gest_pod.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: s1
spec:
  selector:
    name: a6-serveur
  ports:
  - name: ssh-port
    protocol: TCP
    port: 22
    targetPort: ssh-connect
---

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: a6-serveur
  labels:
    app: todo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: todo
  template:
    metadata:
      labels:
        app: todo
        name: a6-serveur
    spec:
      containers:
        - name: s1
          image: a6-serveur
          imagePullPolicy: Never
          tty: true
          ports:
            - name: ssh-connect
              containerPort: 22
          env:
            - name: MONSERVEUR
              value: s1
          volumeMounts:
            - name: config
              mountPath: "/tmp/conf"
              readOnly: true
            - name: creds
              mountPath: "/tmp/creds"
              readOnly: true

      volumes:
        - name: creds
          secret:
            secretName: db-secrets
            optional: false
            items:
            - key: "db-creds"
              path: "creds.properties"
        - name: config
          configMap:
            name: server-config-env
            items:
            - key: "configs.conf"
              path: "config.properties"
```

Figure 8 : serveur_pod.yaml

2. DISCUSSION DE LA STRUCTURE, DES AVANTAGES ET DES INCONVÉNIENTS DE LA VERSION ULTIME DU SYSTÈME

2.1 TECHNOLOGIES LINUX SOUS-JACENTES UTILISÉES

Les technologies Linux sous-jacentes utilisées pour permettre la mise en œuvre des conteneurs sont les Cgroups (Control Groups), les Namespaces et les systèmes de fichiers à union. Les Cgroups permettent de définir et de limiter les ressources utilisées par les conteneurs, garantissant une utilisation équilibrée des ressources système. Les Namespaces fournissent un environnement isolé pour chaque conteneur, séparant les processus, les utilisateurs, les réseaux et les systèmes de fichiers. Les systèmes de fichiers à union superposent des vues cohérentes de plusieurs systèmes de fichiers, permettant aux conteneurs d'avoir leur propre espace de stockage tout en partageant une base commune. Ces technologies Linux sont essentielles pour l'isolation, la gestion des ressources et la gestion des systèmes de fichiers nécessaires au bon fonctionnement des conteneurs et à la création d'environnements conteneurisés performants et sécurisés.

2.2 IDENTIFICATION DES RESSOURCES

L'identification des ressources se fait grâce aux Cgroups (Control Groups). Les Cgroups permettent de définir et contrôler les ressources allouées à chaque conteneur, tel que le CPU, la mémoire, les E/S (entrées/sorties) et le réseau. Ils assurent une gestion fine des ressources en attribuant des limites, des priorités et des quotas à chaque groupe de conteneurs. Ainsi, les ressources sont efficacement identifiées et isolées, ce qui permet de prévenir les conflits et d'optimiser l'utilisation des ressources disponibles dans le système.

2.3 CONTRÔLE DE L'ACCÈS AUX RESSOURCES

Le contrôle de l'accès aux ressources est assuré par les mécanismes de sécurité et de gestion des autorisations du système d'exploitation Linux. Chaque conteneur fonctionne dans son propre environnement isolé grâce à l'utilisation des Namespaces. Les Namespaces garantissent que les ressources attribuées à un conteneur sont uniquement accessibles par celui-ci et ne peuvent pas être utilisées par d'autres conteneurs ou par le système hôte. De plus, les politiques de contrôle

d'accès au niveau du système d'exploitation, telles que les permissions de fichiers, les utilisateurs et les groupes, sont appliquées pour garantir que seules les ressources autorisées peuvent être accédées par les conteneurs et les processus qui y sont exécutés. Cela permet de maintenir l'isolation et la sécurité du système tout en contrôlant l'accès aux ressources.

2.4 GESTION DES ACCÈS AUX FICHIERS UTILISÉS

2.4.1 PILOTES UTILISABLES POUR LA PERSISTANCE DU SYSTÈME DE FICHIERS À UNION

Dans le cadre de l'arrêt 6 où le système doit être déployé à grande échelle, plusieurs pilotes peuvent être utilisés pour la persistance du système de fichiers à union. Parmi les options courantes figurent OverlayFS, AUFS, Device Mapper, et ZFS. Ces pilotes permettent de superposer les systèmes de fichiers, fournissant une abstraction qui combine les fichiers et les répertoires provenant de différentes sources en une seule vue cohérente.

2.4.2 PILOTE DE PERSISTANCE A ÉTÉ UTILISÉ ET POURQUOI ?

Pour répondre aux besoins de la problématique présentée à l'arrêt 6, le pilote de persistance OverlayFS aurait pu être utilisé. OverlayFS est un pilote léger et efficace, intégré nativement au noyau Linux. Il offre une bonne performance, une faible surcharge et une facilité d'utilisation. De plus, il prend en charge la superposition de plusieurs systèmes de fichiers en utilisant des couches, ce qui facilite la gestion des fichiers et des répertoires dans un environnement distribué. En utilisant OverlayFS, il est possible de mettre en place une solution de persistance du système de fichiers à union pour les conteneurs dans Kubernetes, permettant ainsi la gestion efficace des données partagées entre les conteneurs tout en minimisant la complexité et les problèmes de performance.

2.5 DISCUSSION DE LA CONFIGURATION RÉSEAU PERMETTANT AUX DIFFÉRENTS CONTENEURS D'INTERAGIR

Pour permettre l'interaction entre les conteneurs dans Kubernetes, une configuration réseau appropriée est mise en place. Chaque conteneur se voit attribuer une adresse IP unique à l'intérieur d'un réseau virtuel. Les conteneurs sont regroupés en pods, et la communication à l'intérieur d'un pod se fait en utilisant des adresses IP locales. Pour permettre l'interaction entre les pods, Kubernetes utilise des services qui fournissent des adresses IP stables et des noms DNS. Cela permet une communication fluide entre les conteneurs et services dans un cluster Kubernetes.

2.6 DISCUSSION DE LA DUPLICATION MISE EN PLACE POUR LES RESSOURCES

2.6.1 QUELLES RESSOURCES DOIVENT ÊTRE DUPLIQUÉES, ET POURQUOI ?

Dans la version ultime du système, où celui-ci doit être utilisé à grande échelle par mille développeurs, la ressource clé à dupliquer est la base de données. Étant donné qu'un grand nombre d'utilisateurs interagiront avec le système et effectueront des opérations sur la base de données, la duplication est nécessaire pour répartir la charge et éviter la congestion sur une seule base de données. En associant un maximum de cent développeurs par instance de base de données, chaque instance peut gérer un sous-ensemble de requêtes et garantir des performances optimales.

2.6.2 DANS QUEL CAS LA DUPLICATION PEUT SE FAIRE SUR UNE MACHINE RÉELLE UNIQUE, DANS QUEL CAS ELLE DOIT ÊTRE DISTRIBUÉE SUR PLUSIEURS MACHINES RÉELLES ?

Dans l'optique d'une utilisation à grande échelle avec mille développeurs, il est préférable de distribuer les bases de données sur plusieurs machines réelles. Cela permet de bénéficier de la puissance de calcul et de la mémoire de plusieurs machines, ainsi que de répartir la charge de manière équilibrée. En utilisant Kubernetes et en configurant des pods pour chaque instance de base de données, il est possible de déployer les bases de données sur un ensemble de machines distinctes. Cette distribution garantit une meilleure extensibilité, une résilience accrue et une tolérance aux pannes, assurant ainsi la continuité du service même en cas de défaillance matérielle d'une machine.