APPLICATION FLEETMAN

PROJET KUBERNETES

Par :

- Franck MEVENGUE
- Nadia Loukdache
- Cyrine Chammem

NOTRE ÉQUIPE

FRANCK MEVENGUE
Membre 1

NADIA LOUKDACHE
Membre 2

CYRINE CHAMMEM
Membre 3

SOMMAIRE:

INTRODUCTION

L'objectif de ce mini-projet est de déployer une application microservices réelle, Fleetman, sur un cluster Kubernetes composé d'un nœud master et de deux nœuds workers, créés et hébergés sur Hyper-V. L'application Fleetman simule la position de véhicules en temps réel et affiche ces positions sur une interface web. Elle s'appuie sur plusieurs technologies : microservices Spring Boot, MongoDB, ActiveMQ, Nginx et bien sûr Kubernetes pour l'orchestration.
Ce projet a deux objectifs principaux :

Technique : mettre en œuvre un déploiement complet d'une application microservices sur un cluster Kubernetes auto-hébergé (kubeadm), y compris la gestion du stockage, du réseau, des services et des probes de santé.

Pédagogique : comprendre les bonnes pratiques de déploiement (namespace, StatefulSet, Deployments, Services, ConfigMaps), documenter la procédure et être capable de la présenter lors de la soutenance.

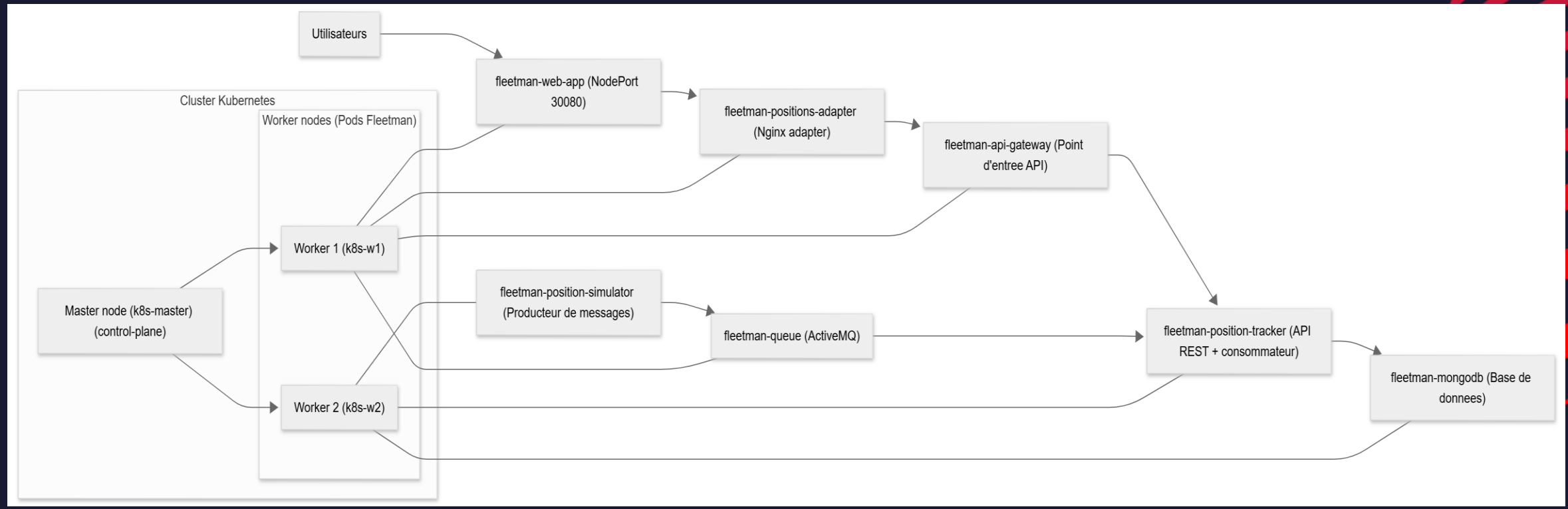

FIGURE 1 : SCHÉMA GLOBAL DU CLUSTER (1 MASTER + 2 WORKERS) ET DE L'APPLICATION FLEETMAN

EXPLICATION : SCHÉMA GLOBAL DU CLUSTER (1 MASTER + 2 WORKERS) ET DE L'APPLICATION FLEETMAN

Le schéma montre d'abord le rôle du **cluster Kubernetes** : le **master (k8s-master)** gère le control-plane, reçoit les manifests, planifie et surveille les pods qui tournent sur **Worker 1 et Worker 2**, où sont déployés tous les services Fleetman. Côté **flux HTTP**, l'utilisateur accède à l'appli via `fleetman-web-app` (NodePort 30080), qui envoie ses requêtes à `fleetman-positions-adapter` (reverse proxy Nginx), puis à `fleetman-api-gateway`, point d'entrée unique qui redirige vers `fleetman-position-tracker` pour récupérer les données. Côté **messages et base de données**, `fleetman-position-simulator` produit des messages de positions dans `fleetman-queue` (ActiveMQ), consommés par `fleetman-position-tracker` qui met à jour `fleetman-mongodb`, ce qui permet ensuite à l'API Gateway et à l'History Service de servir des informations à jour à l'interface utilisateur.

création et gestion des machines virtuelles

système d'exploitation

moteur de conteneurs

kubeadm, kubelet, kubectl pour installer, configurer et administrer Kubernetes

Calico pour la gestion du réseau des pods

FIGURE 1 : LES OUTILS UTILISÉS

II- MISE EN PLACE DE L'INFRASTRUCTURE : HYPER-V ET CLUSTER KUBEADM:

Création des machines virtuelles sur Hyper-V : L'infrastructure repose sur **Hyper-V** installé sur une machine Windows 10/11.

Trois machines virtuelles Ubuntu Server 22.04 ont été créées :

- `k8s-master`
- `k8s-w1` (worker1)
- `k8s-w2` (worker2)

Chaque VM dispose de :

- 2 vCPU
- 4 à 6 Go de RAM pour le master, 3 à 5 Go pour les workers
- 30 Go de disque (40 Go pour le master)
- Connexion réseau via un **vSwitch externe** pour obtenir une adresse IP du réseau local.

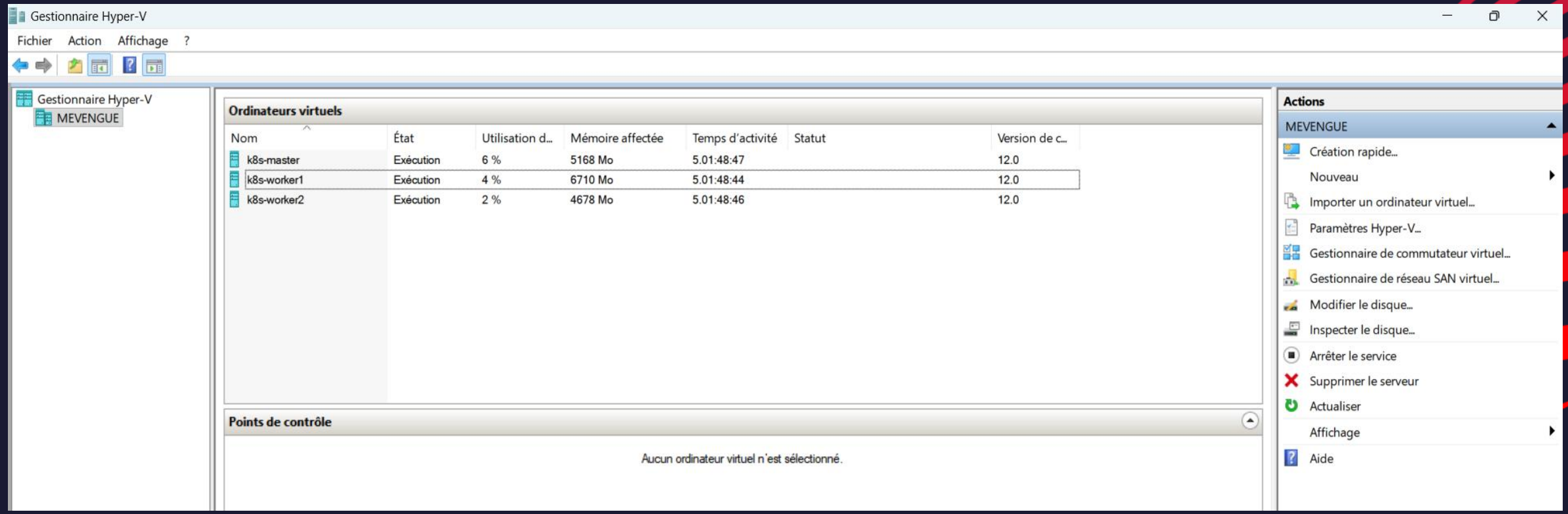

FIGURE 2 : CAPTURE D'ÉCRAN DES VMS HYPER-V (K8S-MASTER, K8S-W1, K8S-W2)

II- MISE EN PLACE DE L'INFRASTRUCTURE : HYPER-V ET CLUSTER KUBEADM:

Préparation système sur chaque nœud : Sur chacun des trois nœuds, la configuration de base a été réalisée :

- Désactivation de **swap** (pré-requis Kubernetes)
- Chargement des modules noyau **overlay** et **br_netfilter**
- Configuration sysctl pour activer le forwarding IP et la prise en compte des règles iptables pour le bridge
- Installation de **containerd** comme runtime de conteneurs
- Installation de **kubeadm, kubelet et kubectl** via les dépôts officiels Kubernetes.

Initialisation du control-plane et ajout des workers : Sur le nœud master, le cluster a été initialisé avec la commande : " sudo kubeadm init --pod-network-cidr=10.244.0.0/16 " . Le paramètre **--pod-network-cidr** prépare l'utilisation du plugin réseau **Flannel**. Une fois l'initialisation terminée, le fichier **admin.conf** a été copié dans **$HOME/.kube/config** afin de pouvoir utiliser **kubectl** depuis le master. Le plugin CNI **Flannel** a ensuite été déployé. Les deux nœuds workers **k8s-w1** et **k8s-w2** ont rejoint le cluster via la commande **kubeadm join** fournie par l'init.

Le cluster est opérationnel et les deux workers sont connectés. Les deux workers, ainsi que le master, sont indiqués comme étant « Ready » :

```
master@k8s-master:~$ kubectl get nodes -o wide
NAME          STATUS   ROLES           AGE     VERSION
k8s-master    Ready    control-plane   139m    v1.29.15
k8s-worker1   Ready    <none>          10m     v1.29.15
k8s-worker2   Ready    <none>          7m37s   v1.29.15
```

FIGURE 3 : SORTIE DE KUBECTL MONTRANT LE MASTER ET LES 2 WORKERS EN READY

III- ARCHITECTURE FONCTIONNELLE DE L'APPLICATION FLEETMAN:

Présentation générale : Fleetman est une application composée de plusieurs **microservices** communicant via HTTP et via une **queue ActiveMQ**. L'objectif est de simuler des véhicules, d'envoyer leurs positions, de les stocker dans MongoDB et de les afficher sur une carte dans une interface web accessible via un NodePort.

Les principaux composants sont :

- **fleetman-web-app** : interface utilisateur (Nginx + frontend) exposée en NodePort (30080).
- **fleetman-api-gateway** : passerelle API unifiée vers les services backend.
- **fleetman-position-tracker** : consomme les messages de la queue, stocke les positions et expose une API REST `/vehicles`.
- **fleetman-position-simulator** : simule les positions de véhicules et publie des messages dans ActiveMQ.
- **fleetman-queue** : broker de messages ActiveMQ.
- **fleetman-mongodb** : base de données (StatefulSet + PersistentVolume).
- **fleetman-history-service** : service Python Flask pour l'historique des positions.
- **fleetman-positions-adapter** : adapter Nginx (proxy supplémentaire optionnel).

Chemin complet des données dans Fleetman :

- Les **utilisateurs** accèdent à l'application via la **fleetman-web-app** exposée en **NodePort 30080**.
- La Web App envoie les requêtes HTTP à **fleetman-positions-adapter**, qui joue le rôle de **proxy Nginx**, puis vers **fleetman-api-gateway**, le **point d'entrée unique** des APIs.
- L'API Gateway appelle soit **fleetman-history-service** (pour lire l'historique), soit **fleetman-position-tracker** (pour les positions courantes).
- En parallèle, **fleetman-position-simulator** produit des messages de position envoyés dans **fleetman-queue (ActiveMQ)**, qui sont consommés par le **position-tracker**.
- Le **position-tracker** écrit les positions dans **fleetman-mongodb** : c'est ainsi que l'interface affiche les positions et l'historique en temps réel.

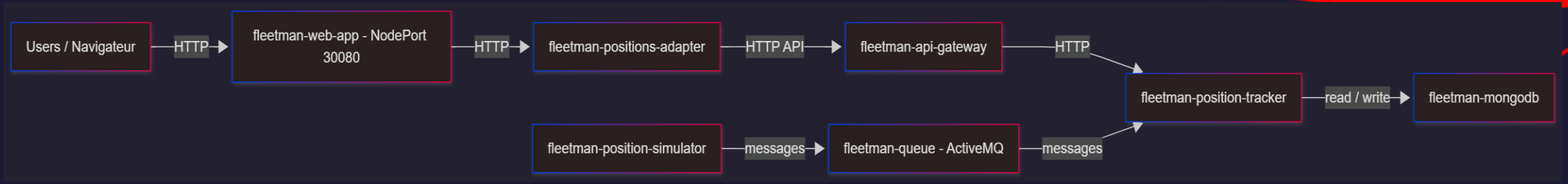

FIGURE 4 : SCHÉMA D'ARCHITECTURE DES MICROSERVICES FLEETMAN ET DE LEURS ÉCHANGES

III- ARCHITECTURE FONCTIONNELLE DE L'APPLICATION FLEETMAN:

Flux de données : Le flux de données principal est le suivant :

- **Position Simulator** génère des positions GPS pour 12 véhicules, toutes les 500 ms, et envoie ces données à **fleetman-queue** (ActiveMQ).
- **Position Tracker** consomme ces messages, les stocke dans **MongoDB** (collection `vehiclePosition`) et expose une API REST `/vehicles`.
- **API Gateway** agrège certaines données et expose une API unifiée `/api/vehicles/**`.
- **History Service** lit directement MongoDB pour fournir l'historique d'un véhicule (`/api/vehicles/{name}/history`).
- **Web App** (Nginx) reçoit les requêtes de l'utilisateur, route `/api/` vers l'API Gateway et l'History Service, puis affiche sur une carte la position des véhicules en temps réel.

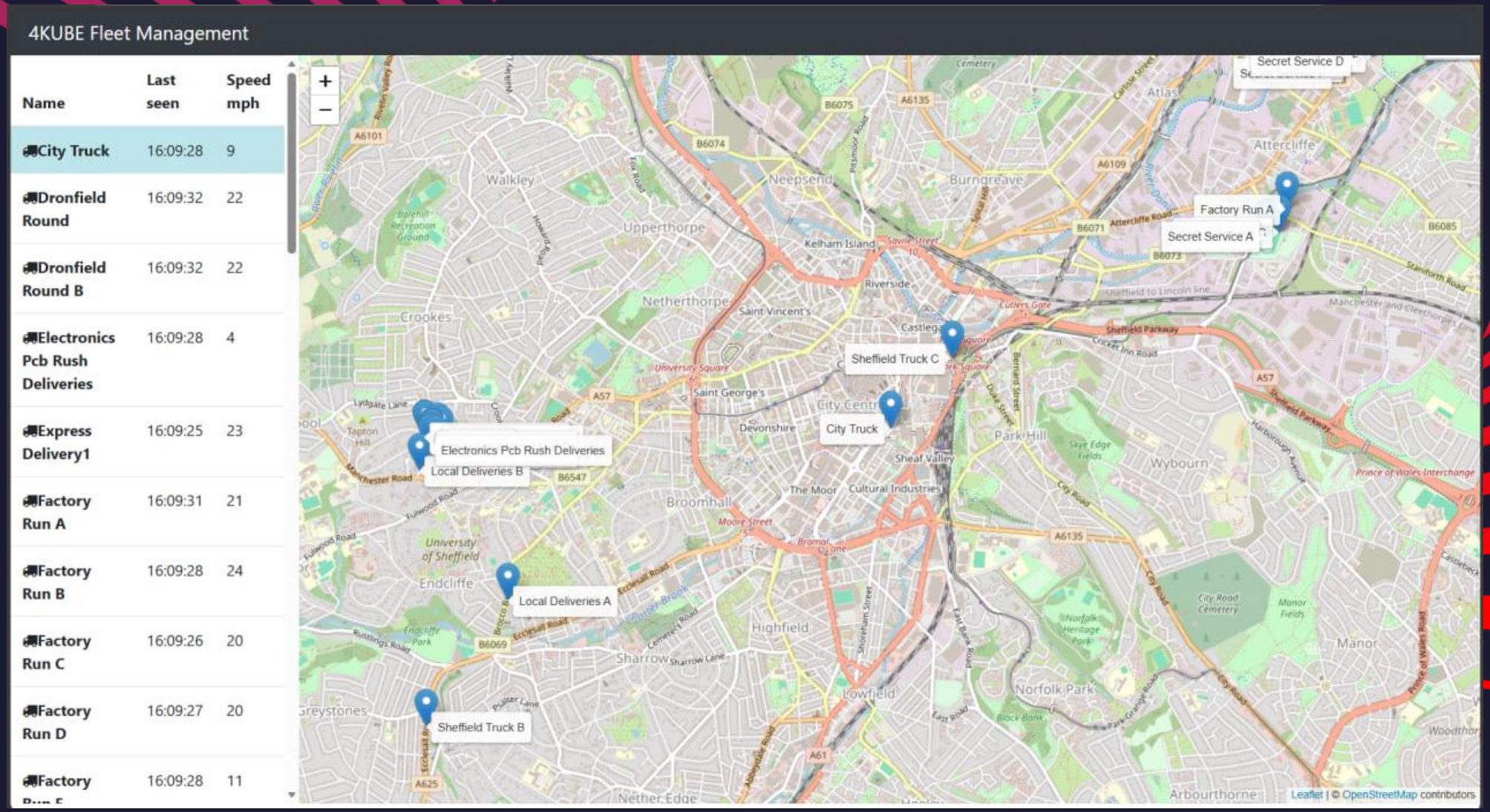

FIGURE 5 : CAPTURE D'ÉCRAN DE L'INTERFACE WEB FLEETMAN AVEC LES VÉHICULES SUR LA CARTE

IV- STRATÉGIE DE DÉPLOIEMENT SUR KUBERNETES:

Organisation des manifests et namespace : Tous les manifests Kubernetes sont regroupés dans le dossier `k8s/`. Le déploiement commence par la création d'un **namespace dédié** : " kubectl apply -f k8s/namespace.yaml " .

Déploiement manuel étape par étape : L'ordre de déploiement est critique pour respecter les dépendances.

1. **MongoDB** (StatefulSet + Service headless)
2. **ActiveMQ Queue** (Deployment + Service ClusterIP)
3. **ConfigMap Nginx** pour la Web App
4. **Position Simulator**
5. **Position Tracker**
6. **API Gateway**
7. **History Service**
8. **Positions Adapter**
9. **Web App** (Deployment + Service NodePort)

FIGURE 6 : EXEMPLE DE COMMANDES

bash

Copier le code

```
kubectl apply -f k8s/fleetman-mongodb.yaml
kubectl apply -f k8s/fleetman-queue.yaml
kubectl apply -f k8s/fleetman-webapp-config.yaml
kubectl apply -f k8s/fleetman-position-simulator.yaml
kubectl apply -f k8s/fleetman-position-tracker.yaml
kubectl apply -f k8s/fleetman-api-gateway.yaml
kubectl apply -f k8s/fleetman-history-service.yaml
kubectl apply -f k8s/fleetman-positions-adapter.yaml
kubectl apply -f k8s/fleetman-web-app.yaml
```

IV- STRATÉGIE DE DÉPLOIEMENT SUR KUBERNETES:

Scripts d'automatisation Linux :

Une deuxième approche consiste à utiliser les scripts bash présents dans le dossier `k8s/deploy` :

- `01-namespace.sh`
- `02-core-services.sh` (MongoDB + ActiveMQ + attente de readiness)
- `03-app-services.sh` (microservices applicatifs + webapp)
- `04-verify.sh` (vérifications automatiques)
- `deploy-all.sh` (enchaîne les quatre étapes).

Ces scripts permettent d'automatiser le déploiement complet et d'ajouter des étapes de **vérification** (état des pods, services, DNS, NodePort, etc.).

V- VÉRIFICATIONS, TESTS ET DÉMONSTRATION

Vérification de l'état des pods et des services : Après le déploiement, plusieurs commandes sont utilisées pour vérifier le bon fonctionnement de l'application :

" kubectl get pods -n fleetman "

" kubectl get svc -n fleetman "

" kubectl get pvc -n fleetman "

" kubectl get statefulset -n fleetman "

Ces commandes confirment que :

- le StatefulSet MongoDB est **Ready** ;
- les Deployments (queue, simulator, tracker, api-gateway, web-app, etc.) ont tous leurs réplicas **AVAILABLE** ;
- le service `fleetman-web-app` est exposé en **NodePort 30080**.

```
master@k8s-master:~$ kubectl get pods -A
NAMESPACE           NAME                                       READY   STATUS    RESTARTS        AGE
fleetman            fleetman-api-gateway-549f6f7854-klwg4      0/1     Running   1 (29s ago)     6m35s
fleetman            fleetman-api-gateway-6bf976d898-z2vmw      0/1     Running   1 (76s ago)     7m19s
fleetman            fleetman-mongodb-0                         1/1     Running   0               76m
fleetman            fleetman-position-simulator-64d7dbb5df-bd4km   1/1     Running   2 (26s ago)     6m32s
fleetman            fleetman-position-tracker-5bc8bd495d-qb7n6     0/1     Running   1 (30s ago)     6m33s
fleetman            fleetman-position-tracker-cbfd847fb-qcr8w      0/1     Running   1 (43s ago)     7m15s
fleetman            fleetman-queue-7867bd86f5-gpv8j            1/1     Running   0               6m35s
fleetman            fleetman-webapp-649d7bf465-gt9k2           1/1     Running   0               6m32s
kube-flannel        kube-flannel-ds-5sdw7                      1/1     Running   0               33m
kube-flannel        kube-flannel-ds-97c9n                      1/1     Running   0               36m
kube-flannel        kube-flannel-ds-pkwxm                      1/1     Running   0               78m
kube-system         coredns-76f75df574-7dwdl                   1/1     Running   0               164m
kube-system         coredns-76f75df574-tdlq5                   1/1     Running   0               164m
kube-system         etcd-k8s-master                            1/1     Running   22              164m
kube-system         kube-apiserver-k8s-master                  1/1     Running   20              164m
kube-system         kube-controller-manager-k8s-master         1/1     Running   23 (145m ago)   164m
kube-system         kube-proxy-62ksr                           1/1     Running   0               33m
kube-system         kube-proxy-hfnr4                           1/1     Running   0               164m
kube-system         kube-proxy-zbz5p                           1/1     Running   0               36m
kube-system         kube-scheduler-k8s-master                  1/1     Running   22 (145m ago)   164m
local-path-storage  local-path-provisioner-6d9d9b57c9-6hhf7    1/1     Running   0               76m
```

FIGURE 7 : VÉRIFICATION DE L'ÉTAT DES PODS ET DES SERVICES

VI- DIFFICULTÉS RENCONTRÉES ET SOLUTIONS APPORTÉES :

Au cours du projet, plusieurs difficultés typiques d'un environnement Kubernetes on-prem ont été rencontrées :

1. Absence de StorageClass par défaut
   + Symptôme : le PVC de MongoDB reste en état `Pending`.
   + Solution : installation de `local-path-provisioner` et définition de la StorageClass `local-path` comme classe par défaut.
2. Pods en CrashLoopBackOff (microservices Spring Boot)
   + Souvent lié au fait que MongoDB ou la queue ActiveMQ n'étaient pas encore prêts.
   + Solution : ajout d'**ordre de déploiement** (MongoDB & queue d'abord), probes de readiness plus adaptées, et parfois redémarrage ciblé avec `kubectl rollout restart`.
3. Problèmes de résolutions DNS internes
   + Diagnostic via `kubectl exec` + `nslookup` / `curl`.
   + Correction des URLs internes pour utiliser les FQDN Kubernetes du type `<service>.<namespace>.svc.cluster.local`.

VI- DIFFICULTÉS RENCONTRÉES ET SOLUTIONS APPORTÉES :

4. **Positions qui n'apparaissent pas dans la webapp :**
   + Cause possible : queue ActiveMQ non fonctionnelle ou simulateur non démarré.
   + Solution : vérification des logs, redémarrage de `fleetman-queue` et `fleetman-position-simulator` via `kubectl rollout restart`.

Ces difficultés ont permis de mieux comprendre l'importance :

- de la **séquence de déploiement**,
- de la **santé des dépendances** (base de données, broker),
- et de la **visibilité** via les logs et les événements Kubernetes (`kubectl get events`).

CONCLUSION :

Ce mini-projet nous a permis de déployer une application microservices réelle (Fleetman) sur un cluster Kubernetes complet (1 master, 2 workers) en partant de la création des machines sur Hyper-V jusqu'à l'accès à l'interface web via un NodePort. Nous avons ainsi consolidé nos compétences sur les objets Kubernetes (Deployments, StatefulSets, Services, ConfigMaps, PVC), la gestion réseau (CNI Flannel) et le stockage, tout en développant de bons réflexes de débogage (logs, probes, DNS, dépendances entre services). Ce travail nous a donné une vision concrète d'un déploiement on-premise en production et ouvre la voie à des améliorations futures comme le monitoring, la haute disponibilité et l'intégration dans une chaîne CI/CD.

MERCI