ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

IoT Cloud
TP2

TP2 – Stockage

Partie 1: Construire et déployer son service

1.1 Introduction

Dans ce TP, vous allez devoir mettre en place une architecture de stockage de données en utilisant une base de données *NoSQL* [16] et en particulier *MongoDB* [8].

Comme vu en cours, une telle architecture doit être adaptée à la situation et une multitude de possibilités d'architectures correctes existe.

Votre rôle, en tant qu'ingénieur, est de savoir concevoir une architecture adaptée à la situation. Vous devez, évidemment, aussi savoir déployer une telle architecture concrètement.

Ce TP a pour objectif de vous faire réfléchir sur une bonne solution d'architecture à mettre en place pour répondre au *Fil Rouge* des TPs, puis de la déployer et de jouer avec.

Pour rappel, vous devez rendre un rapport individuel à la fin du semestre. Vous devez y mettre toutes les informations qui vous semblent pertinentes. Notez que les réponses aux questions encadrées en gris seraient à considérer comme un minimum à faire apparaitre dans ce rapport.

1.2 Définition de l'architecture

Dans un premier temps, vous devez :

1. Définir et expliquer l'architecture de stockage utilisant MongoDB (au niveau matériel et logiciel) *Databases* / *Collections* / *Serveurs* qui vous suivra durant les TPs pour répondre au *Fil Rouge*. – Comment allez-vous stocker ?
2. Définir et expliquer la structure des données (modèle du document stocké sur MongoDB) qui vous suivra durant les TPs pour répondre au *Fil Rouge*. – Qu'est-ce que vous allez stocker ? [6]
3. Définir et expliquer les *indexes* que vous allez utiliser sur votre base de données MongoDB afin de pouvoir accéder au mieux à vos données. – Comment allez-vous y accéder ? [7]

Pour vous aider, quelques questions à vous poser :

- Est-ce que les données d'un client doivent être cloisonnées de celles des autres clients ?
- Est-ce qu'il faut une *collection* par capteur, par maison, par client, par quartier… ?
- Quel impact cela va-t-il avoir sur la façon d'utiliser, combiner et retrouver mes données ?
- Comment gérer les droits d'accès aux *databases*/*collections* ?
- Est-ce que, si les messages se retrouvent dans la même *collection*, je pourrais les ré-identifier facilement ?
- Du point de vue des consommateurs de données, sera-t-il plus simple et/ou plus sécurisé d'avoir une ou plusieurs *databases*/*collections* de messages ?
- Quel sera l'impact si je perds complètement un *nœud* de mon architecture matérielle ?
- Est-ce qu'il faut que je mette toutes les *databases* sur la même grappe de serveurs, ou il me faut plusieurs grappes de serveurs ?
- Si, demain, j'ai vraiment beaucoup de clients, est-ce que mon architecture sera *scalable* [18] ? Est-ce qu'elle *passera à l'échelle* [17] ? Est-ce que mes données seront bien réparties ou est-ce que je risque de créer des nœuds plus remplis que d'autres ?
- Si, demain, j'ai des clients internationaux qui ne veulent pas que leurs données sortent de leur pays, est-ce que mon architecture globale reste viable ?
- Et s'ils veulent héberger cette partie de ma solution directement chez eux pour contrôler plus finement ce qui sort de chez eux ?
- Est-ce que je n'ai besoin que du nom du capteur, de sa valeur et de la date de relevé de la valeur ? *Quid* de l'unité ?
- Que ce passe-t-il si un capteur doit être remplacé ? Est-ce que je garderais le même nom de capteur ? Et s'il ne produit pas les données avec la même unité ?
- Quelles seront les requêtes les plus courantes pour accéder à mes données ? Est-ce qu'il faut que j'ajoute un *index* sur les dates ? Sur les noms de capteurs ? Sur les valeurs ? Sur chaque paramètre ?

Notez que votre architecture pourra être amenée à évoluer au fur et à mesure des TPs. Mais, si vous réfléchissez bien dès maintenant à un ensemble de problématiques possibles, vous gagnerez du temps et n'aurez peut-être pas à faire (trop) évoluer votre architecture…

1.3 Déployer votre architecture

Vous pouvez maintenant :

4. Déployer un serveur MongoDB à partir du *docker-compose* proposé à la Figure 1.1.
5. Implémenter votre architecture de stockage de données à partir de l'interface web proposée par Mongo-Express (mais si, vous allez la trouver…). Vous pouvez sinon utiliser MongoDB Compass comme interface de gestion [5].

1.4 Découvrir l'envoi et la réception de données

Super, on y met quelque chose maintenant ? Essayez de :

6. Créer un script simple (avec le langage de programmation que vous voulez) permettant d'envoyer des données pour 1 capteur sur une *collection*.
7. Créer un script simple (avec le langage de programmation que vous voulez) permettant de lire les données d'une *collection* et de les afficher sur la console.
8. Créer des scripts plus avancés permettant de :
   (a) récupérer la dernière valeur connue d'un capteur.
   (b) calculer la moyenne des valeurs d'un capteur entre deux dates fixées.
   (c) récupérer la valeur minimale d'un capteur entre deux dates fixées.

Pour créer vos scripts, vous pouvez aller voir du côté des tutoriels "MongoDB Drivers and ODM" de MongDB ici : `https://docs.mongodb.com/ecosystem/drivers/`. Personnellement j'ai trouvé l'exemple en Python avec la librairie `PyMongo` [12] assez simple, mais vous avez le choix des langages sur la page du tutoriel principal. D'autres sources peuvent aussi être utiles [13].

```yaml
version: '3.1'

services:

  mongo:
    image: mongo:4.2.0-bionic
    hostname: "mongo"
    restart: always
    labels:
      NAME: "mongo"
    networks:
      - iot-labs
    ports:
      - 27017:27017

  mongo-express:
    image: mongo-express:0.49.0
    hostname: "mongo_express"
    restart: always
    ports:
      - 8081:8081
    environment:
      ME_CONFIG_MONGODB_SERVER: mongo
    labels:
      NAME: "mongo_express"
    networks:
      - iot-labs

networks:
  iot-labs:
    external: true
```

Figure 1.1 – Exemple de *docker-compose* permettant de déployer un serveur MongoDB et une interface de gestion Mongo-Express. Voir [3] et [2] pour plus de détails.

1.5 Automatisation du déploiement

Vous devez maintenant :

9. Créer un script (avec le langage de programmation que vous voulez) permettant de déployer automatiquement l'architecture définie en Section 1.2. Votre script doit répondre aux deux situations suivantes (qui sont peut être, suivant votre architecture, la même situation pour vous) :
   - Le monde vient de s'écrouler (pour vous, cela veut dire que vous avez perdu tous vos serveurs) et vous devez redéployer l'architecture de base permettant d'ajouter des clients à l'avenir.
   - Le monde s'ouvre à vous, un nouveau client arrive ! Vous devez déployer le nécessaire pour commencer à recevoir les messages des objets connectés de ce client.

1.6 Génération automatique de messages

Reprenons le générateur de *mock data* [15] et votre RabbitMQ du TP précédent. Un projet de Marcel Maatkamp [4], basé sur une librairie NodeJS de Ruquay K Calloway [1] devrait vous permettre de connecter une *queue* de messages à une *collection*. Un exemple vous est donné en Figure 1.2, n'hésitez pas à l'adapter avec la documentation.

10. Utiliser le code proposé en Figure 1.2 pour envoyer des données cohérentes générées précédemment sur une *queue* RabbitMQ vers une *collection* MongoDB pour quelques capteurs du *Fil Rouge*.

```yaml
#https://github.com/marcelmaatkamp/docker-rabbitmq-mongodb
version: '3.7'
services:

  amqp2mongo1:
    image: "marcelmaatkamp/rabbitmq-mongodb"
    hostname: "amqp2mongo1"
    environment:
      AMQPHOST: 'amqp://<RABBITMQ_HOSTNAME>'
      MONGODB: 'mongodb://<MONGODB_HOSTNAME>/<MONGODB_DATABASE>'
      MONGOCOLLECTION: '<MONGODB_COLLECTION>'
      TRANSLATECONTENT: 'true'
    command: '<RABBITMQ_QUEUE>'
    tty: true
    labels:
      NAME: "amqp2mongo1"
    networks:
      - iot-labs
    restart: always

networks:
  iot-labs:
    external: true
```

Figure 1.2 – Exemple de *docker-compose* permettant de déployer un service de récupération de données sur une *queue* spécifique d'un *broker* AMQP vers une *collection* spécifique d'une *database* MongoDB. Voir [4] pour plus de détails.

1.7 Optionnel – Trouver la bonne clé de *sharding*

Quand vous déployez un service MongoDB avec un ensemble de serveurs, c'est-à-dire différents *shards* [11] composés chacun de différents *replica* [9], une question devient essentielle :

Est-ce que mes données seront bien réparties entre les *shards* ?

Le résultat dépend en fait de la *shard key* utilisée [10].

L'idée de cette partie, en se basant notamment sur cette référence [10], est de :

1. Déployer en local un *cluster* MongoDB.
   Pour cela, j'ai légèrement modifié un projet de Rafał Warzycha [14] pour qu'il soit utilisable avec votre environnement. Vous pouvez le cloner à partir de GitHub `https://github.com/pcourbin/mongo-cluster-docker` puis utiliser le script `run_mongo_cluster.sh` pour déployer les serveurs MongoDB en local.
2. Créer une *database* et une *collection*.
3. Créer un *index* sur la *collection*, c'est lui qui sera utilisé pour la *shard key*.
4. Activer le *sharding* en spécifiant la *shard key*.
5. Envoyer des données en quantité suffisante pour voir comment elles se répartissent.
6. Discuter du choix de la *shard key* et tester d'autres possibilités.

Partie 2: Bibliographie

[1] Ruquay K Calloway. *Librairie NodeJS AMQP to MongoDB*. URL: https://www.npmjs.com/package/amqp-to-mongo.

[2] Docker. *Image Docker officielle de Mongo-Express*. URL: https://docs.docker.com/samples/library/mongo-express/.

[3] Docker. *Image Docker officielle de MongoDB*. URL: https://docs.docker.com/samples/library/mongo/.

[4] Marcel Maatkamp. *Docker RabbitMQ to MongoDB*. URL: https://hub.docker.com/r/marcelmaatkamp/rabbitmq-mongodb.

[5] MongoDB. *Compass*. URL: https://www.mongodb.com/download-center/compass.

[6] MongoDB. *Data Modeling*. URL: https://docs.mongodb.com/manual/data-modeling/.

[7] MongoDB. *Indexes*. URL: https://docs.mongodb.com/manual/indexes/.

[8] *MongoDB*. URL: https://www.mongodb.com/.

[9] MongoDB. *Replication*. URL: https://docs.mongodb.com/manual/replication/.

[10] MongoDB. *Shard Keys*. URL: https://docs.mongodb.com/manual/core/sharding-shard-key/.

[11] MongoDB. *Sharding*. URL: https://docs.mongodb.com/manual/sharding/.

[12] PyMongo. *Tutorial for PyMongo*. URL: https://api.mongodb.com/python/current/tutorial.html.

[13] Robert Walters. *Getting Started with Python and MongoDB*. URL: https://www.mongodb.com/blog/post/getting-started-with-python-and-mongodb.

[14] Rafał Warzycha. *mongo-cluster-docker*. URL: https://github.com/senssei/mongo-cluster-docker.

[15] Wikipedia. *Mock object*. URL: https://en.wikipedia.org/wiki/Mock_object.

[16] Wikipedia. *NoSQL*. URL: https://fr.wikipedia.org/wiki/NoSQL.

[17] Wikipedia. *Passage à l'échelle d'une application web*. URL: https://fr.wikipedia.org/wiki/Passage_%C3%A0_l%27%C3%A9chelle_d%27une_application_web.

[18] Wikipedia. *Scalability*. URL: https://fr.wikipedia.org/wiki/Scalability.