IoT Cloud
TP1

TP1 – Ingestion

Partie 1: Construire et déployer son service

1.1 Introduction

Dans ce TP, vous allez devoir mettre en place une architecture d'ingestion de données en utilisant un *Message Broker* [8] et en particulier *RabbitMQ* [6].

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 *Exchanges*/*Queues* qui vous suivra durant les TPs pour répondre au *Fil Rouge* (réception des données).

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 un *exchange* par capteur, par maison, par client, par quartier... ?
- Quel impact cela va-t-il avoir sur la configuration de mes capteurs / objets connectés ?
- Comment gérer les droits d'accès aux *queues* ?
- Est-ce que, si les messages se retrouvent dans la même *queue* de messages, je pourrai 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 *queues* de messages ?
- Quel sera l'impact si je perds complètement une *queue* / un *noeud* de mon architecture ?
- Est-ce qu'il faut que je mette toutes les *queue* de messages 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* [11] ? Est-ce qu'elle *passera à l'échelle* [10] ?
- 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 ?

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 évoluer votre architecture...

Dans un second temps, imaginez que nous souhaitions utiliser un *Message Broker* pour envoyer des ordres à des objets chez le client. Vous devez donc :

2. Définir et expliquer l'architecture *Exchanges*/*Queues* qui vous servirait dans ce cas à "redescendre" des informations vers des objets chez le client.

Vous n'aurez pas à implémenter cette architecture par la suite.

1.3 Déployer votre architecture

Vous pouvez maintenant :

3. Déployer un serveur RabbitMQ à partir du *docker-compose* proposé à la Figure 1.1.
4. Implémenter votre architecture de remontée de données à partir de l'interface web proposée par RabbitMQ (mais si, vous allez la trouver...)

Vous noterez peut être que le réseau `iot-labs` est indiqué comme `external`. Cela permettra à l'avenir d'y connecter de nouveaux services simplement (votre base de données etc). Mais il faut donc créer ce réseau à l'avance... En suivant cette documentation [4], vous pourriez notamment lancer la commande présentée :

```
docker network create --attachable iot-labs
```

(Aller, voyons le nombre d'entre vous qui me demanderont de leur montrer cette ligne...)

1.4 Découvrir l'envoi et la réception de messages

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

5. Jouer avec votre architecture en générant des données à la main à partir de l'interface web. Testez par exemple en générant quelques données pour 2 capteurs différents de 2 pièces différentes pour chacun des clients. (au début, la question était formulée : "*jouer avec une queue pour générer des messages à la main*" mais, vous connaissant un peu, il a semblé plus sûr de reformuler la demande.)
6. Créer un script simple (avec le langage de programmation que vous voulez) permettant de lire les messages d'une *queue de messages* et de les afficher sur la console.
7. Créer un script simple (avec le langage de programmation que vous voulez) permettant de générer des données pour 1 capteur.

Pour créer vos scripts, vous pouvez aller voir du côté des tutoriels "Hello World!" de RabbitMQ ici : `https://www.rabbitmq.com/getstarted.html`. Personnellement j'ai trouvé l'exemple en Python avec la librairie `pika` [5] assez simple, mais vous avez le choix des langages sur la page du tutoriel principal. D'autres sources peuvent aussi être utiles [2].

```yaml
version: '3.7'
services:

  rabbitmq1:
    image: "rabbitmq:3-management"
    hostname: "rabbitmq1"
    environment:
      RABBITMQ_ERLANG_COOKIE: "SWQOKODSQALRPCLNMEQG"
      RABBITMQ_DEFAULT_USER: "rabbitmq"
      RABBITMQ_DEFAULT_PASS: "rabbitmq"
      RABBITMQ_DEFAULT_VHOST: "/"
    ports:
      - "15672:15672"
      - "5672:5672"
    networks:
      - iot-labs
    labels:
      NAME: "rabbitmq1"

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

Figure 1.1 – Exemple de *docker-compose* permettant de déployer un serveur RabbitMQ. Voir [3] pour plus de détails.

1.5 Automatisation du déploiement

Vous devez maintenant :

8. 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

Un peu d'aide pour simplifier les prochains TPs... générons des *mock data* [9]. Un projet de Senzing [7] a été étendu [1] pour générer des données de capteurs avec valeur entière ou flottante. Un exemple vous est donné en Figure 1.2, n'hésitez pas à l'adapter avec la documentation [1].

9. Utiliser le code proposé en Figure 1.2 pour générer des données cohérentes pour tous les capteurs du *Fil Rouge*.

```yaml
version: '3.7'
services:

  moke1:
    image: "pcourbin/mock-data-generator:latest"
    hostname: "moke1"
    environment:
      SENZING_SUBCOMMAND: random-to-rabbitmq
      SENZING_RANDOM_SEED: 1
      SENZING_RECORD_MIN: 1
      SENZING_RECORD_MAX: 100
      SENZING_RECORDS_PER_SECOND: 1
      SENZING_RABBITMQ_HOST: rabbitmq1
      SENZING_RABBITMQ_PASSWORD: rabbitmq
      SENZING_RABBITMQ_USERNAME: rabbitmq
      SENZING_RABBITMQ_QUEUE: client1
      MIN_VALUE: 500
      MAX_VALUE: 700
      SENZING_DATA_TEMPLATE: '{"SENSOR":"Temp1","DATE":"date_now", "VALUE":"float"}'
    tty: true
    labels:
      NAME: "moke1"
    networks:
      - iot-labs

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

Figure 1.2 – Exemple de *docker-compose* permettant de déployer un service de génération de données fictives vers un RabbitMQ. Voir [1] pour plus de détails.

Partie 2: Bibliographie

[1] Pierre COURBIN. *Extension du projet de Senzing pour générer des données de capteurs*. URL: https://github.com/pcourbin/mock-data-generator.

[2] CloudAMQP. *Part 2.3: Getting started with RabbitMQ and Python*. URL: https://www.cloudamqp.com/blog/2015-05-21-part2-3-rabbitmq-for-beginners_example-and-sample-code-python.html.

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

[4] Docker. *Network Create*. URL: https://docs.docker.com/engine/reference/commandline/network_create/.

[5] Pika. *Documentation de la librairie Pika pour connexion à RabbitMQ*. URL: https://pika.readthedocs.io.

[6] *RabbitMQ*. URL: https://www.rabbitmq.com/.

[7] Senzing. *Projet de génération de données fictives*. URL: https://github.com/Senzing/mock-data-generator.

[8] Wikipedia. *Message broker*. URL: https://en.wikipedia.org/wiki/Message_broker.

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

[10] 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.

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