

Interaction Distribuée

Physically Extended Interactive Technologies

Remi LABORIE
Kahina CHALABI
Fatima-Zohra EL HANTATI

2022/2023

Sommaire

Travail de spécification	2
Représentation formelle	2
Architecture générale	2
Preuve de concept	3
Service opérationnel	4
Explication de la méthode développée	4
Introduction	4
Envoi des données capteurs	4
Serveur	5
Client	6
Conclusion	6

Travail de spécification

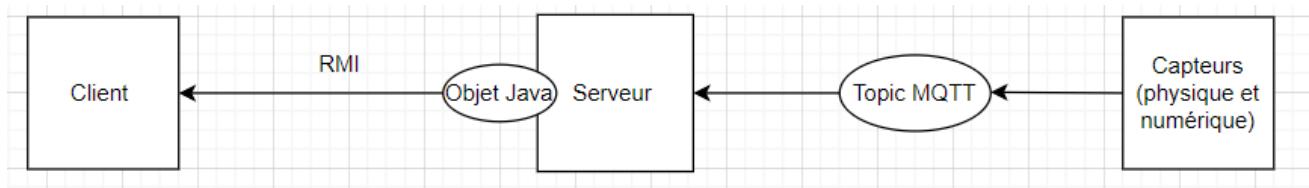
Représentation formelle

Nous pouvons représenter un capteur par plusieurs informations:

- Un nom
- URL si c'est une API
- Port de communication, comment on récupère la / les données du capteur
- Le format des données récupérées
- La fréquence maximale d'envoi des données

Architecture générale

La figure suivante montre l'architecture utilisée :



Des capteurs publient leurs données sur un topic MQTT grâce au protocole MQTT qui assure la communication.

Le serveur récupère donc les données de tous les capteurs qui ont été publiées sur le topic, les répartit dans les ensembles de capteurs et fait les traitements voulus. Il met ensuite à disposition une interface Java avec RMI qui peut être utilisée par des clients afin de récupérer les données souhaitées.

Pourquoi MQTT ? Le protocole MQTT se démarque des autres par sa simplicité de mise en œuvre et l'assurance d'une transmission de données bidirectionnelle. Cependant, il nécessite de créer un serveur MQTT pour cela, utilisant ainsi une nouvelle machine. Comme nous faisons tout en local nous n'avons pas ce problème là, nous utilisons shiftr.io pour créer le serveur en local.

Pourquoi RMI ? Permet à un ou plusieurs clients d'appeler des méthodes sur un objet stocké à distance sur un serveur. Il nécessite d'avoir la même interface des deux côtés, ceci est généralement un inconvénient. Notamment car cela ne permet de faire que des clients en Java. Ce qui ne nous pose pas de problème car nous travaillons quasiment uniquement en Java.

Preuve de concept

Définition des capteurs que nous utilisons :

- Capteur météo en ligne (OpenWeatherMap)
 - URL pour récupérer la météo de Toulouse :
<https://api.openweathermap.org/data/2.5/weather?q=Toulouse&appid=8739106db67b65d2ee403c932278b5cd&lang=fr&units=metric>
 - Port de communication : API REST
 - Format de données : fichier Json
 - La fréquence maximale dépend du prix auquel la clé de l'API a été achetée. Avec la clé gratuite, nous sommes limités à 60 appels par minute soit 1 par seconde.
- Capteur de température
 - Port de communication : USB
 - Format de données : des entiers à convertir en température
 - Fréquence maximale : /
- Capteur de distance à ultrason
 - Port de communication : USB
 - Format de données : le capteur utilise le principe de l'écho, un court signal sonore est envoyé, ce dernier sera réfléchi par une surface et renvoyé vers le capteur.
 - Fréquence maximale : 40kHz

Capteurs physiques : l'avantage de ces capteurs c'est qu'on sait ce qu'on manipule, on maîtrise la donnée, néanmoins, c'est difficile d'avoir de bonnes valeurs (gestion des erreurs de mesure, etc.)

Capteurs API : Facile à utiliser (requêtes web pour récupérer les données traitées) mais on ne maîtrise pas ce qu'on récupère, on doit faire avec ce qu'on reçoit comme donnée, nous n'avons aucun contrôle dessus.

Grâce à ces différents capteurs, nous pouvons créer plusieurs ensembles de capteurs pour différentes utilisés. Par exemple un ensemble contenant le capteur météo en ligne et le capteur de température physique pour en faire une moyenne.

Service opérationnel

Nous avons choisi de développer un Bot discord. Ce Bot peut envoyer des messages d'alertes s'il y a un problème. Il peut être paramétré par les utilisateurs grâce à des commandes permettant de créer, modifier ou supprimer des ensembles de capteurs; définir des ensembles de valeurs dans lesquelles les capteurs doivent rester, dans ce cas, le Bot enverra des alertes si les valeurs sont en dehors de ces limites, etc.

Explication de la méthode développée

Introduction

Nous n'avons pas développé beaucoup de fonctionnalités mais assez pour permettre de continuer le développement et en ajouter de nouvelles.

Nous avons développé tout en local mais, dans l'idéal (pour une utilisation réelle), il faudrait une machine pour chaque programme et remplacer les *localhost* par l'adresse permettant d'accéder à la machine distante.

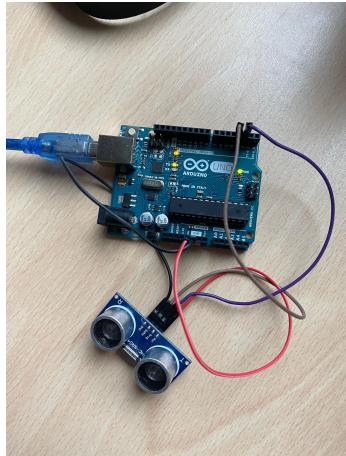
Envoi des données capteurs

En suivant notre architecture, nous avons développé un programme pour chaque capteur, adapté à chacun pour récupérer les données souhaitées et les envoyer sur le broker MQTT.

Nous avons un programme Java pour l'API de météo en ligne et un programme Java pour un capteur physique (un capteur ultrason) qui sont respectivement : *MeteoSender* et *DistanceSender*.

MeteoSender; récupère d'abord les données de l'API openweathermap puis envoie la température de Toulouse grâce à MQTT (on pourrait envoyer plusieurs données différentes).

DistanceSender; récupère les données sur le port série envoyées par Arduino toutes les 300ms. Nous avons choisi une vitesse assez rapide afin de pouvoir envoyer des données toutes les secondes avec les Sender (la fréquence d'envoi des données sur MQTT est modifiable).



Capteur Ultrason contrôlé via Arduino

Note : Nous avons choisi de récupérer les données du capteur physique en port série car nous avons trouvé que c'était plus simple, mais on aurait pu le faire avec MQTT directement, ce qui demande un peu plus de connaissance sur Arduino.

Serveur

CapteursReveiver (qui aurait dû s'appeler *CapteursReceiver* ou bien *Server*) est le serveur que nous avons développé. Il récupère toutes les données envoyées par les capteurs (physique et numérique) et va les mettre dans les ensembles qui utilisent ces capteurs.

Il y a une interface *Data* qui contient les attributs définissant un capteur (représentation formelle), que nous n'avons pas renseigné, ainsi que les fonctions disponibles pour une utilisation côté serveur. Pour chaque capteur, il existe donc une classe implémentant cette interface pour enregistrer les données récupérées. Dans notre cas, nous avons une donnée par capteur mais nous pouvons ajouter autant de données que l'on souhaite.

Il y a une classe de listener par capteur : *MeteoListener* et *DistanceListener* qui écoutent le / les topics MQTT qui les intéressent. Ils récupèrent les données envoyées par les senders et les stockent dans les *Data*.

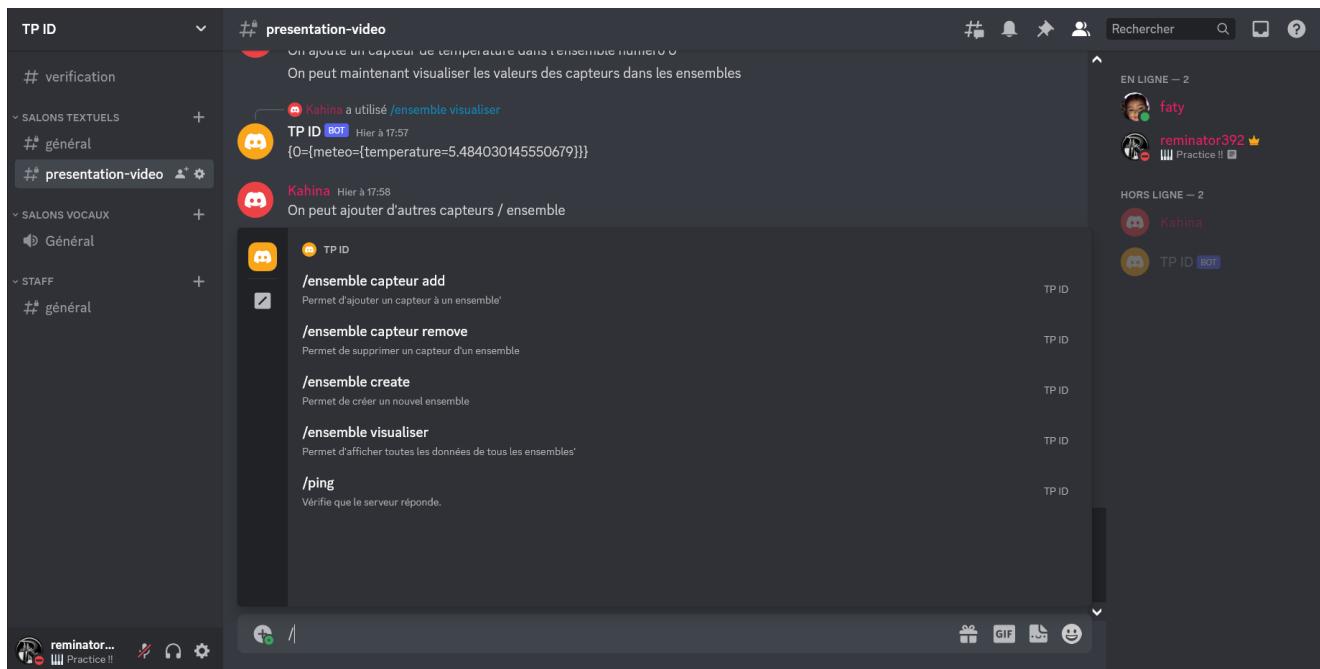
Le serveur met à disposition un Objet Java grâce à RMI qui implémente l'interface *EnsembleInterface*.

Cette interface contient toutes les fonctions utilisables par un client qui devra avoir la même interface. Nous avons développé un début de gestion d'ensemble pour montrer comment cela fonctionne. Il est possible de créer des ensembles, ajouter des capteurs à ces ensembles et récupérer les données d'un ou de tous les ensembles.

Client

Un exemple simple de client utilisant notre serveur est disponible : *ClientCapteur*
 Mais notre principal client est *ClientCapteursDiscord* qui va lancer un bot discord pour que les utilisateurs de discord puissent exécuter les commandes directement depuis cette application. Comme dit précédemment, nous n'avons développé que quelques fonctionnalités ; créer des ensembles, ajouter des capteurs dans les ensembles et visualiser les données de ces capteurs.

Nous pouvons voir sur la figure suivante le serveur dédié pour tester notre client Bot discord ainsi que les commandes disponibles. Une vidéo de démonstration est disponible sur le github.



Conclusion

Pour conclure, ce projet nous a permis d'avoir des connaissances assez solides sur la communication et l'échange de données via des topic et des protocoles de communications, nous nous sommes basés évidemment sur les concepts vus en cours.