Rapport Projet AP4A :

Simulateur IoT pour la surveillance de la qualité de l'air en espace de travail

UZUN Enes-Alperen

Automne 2024

C++

Responsable UV : Franck Gechter

Introduction

Ce projet a pour objectif de développer un simulateur IoT en C++ qui modélise un écosystème connecté permettant de surveiller la qualité de l'air. Le simulateur inclut plusieurs capteurs (température, humidité, qualité de l'air, son) simulés, chacun ayant un comportement propre, qui envoient des données vers un serveur central. Ce serveur reçoit, analyse et stocke ces données, offrant ainsi un aperçu en temps réel des conditions environnementales dans un espace de travail simulé. Tout cela est géré par un ordonnanceur qui dicte le début et la fin des différents processus.

La réalisation de ce projet a nécessité l'utilisation de concepts avancés en programmation orientée objet, ainsi que la gestion de la communication et du traitement des données en temps réel. Ce document présente en détail le processus de conception, les choix techniques et les résultats obtenus.

Choix d'implémentation

Pour se projet, nous avons décidé d'utiliser 3 classes.

Une classe Server qui va permettre de traiter et stocker les données.

Une classe Sensor qui représente un capteur générique et sert de classe mère pour les différents types de capteurs spécifiques.

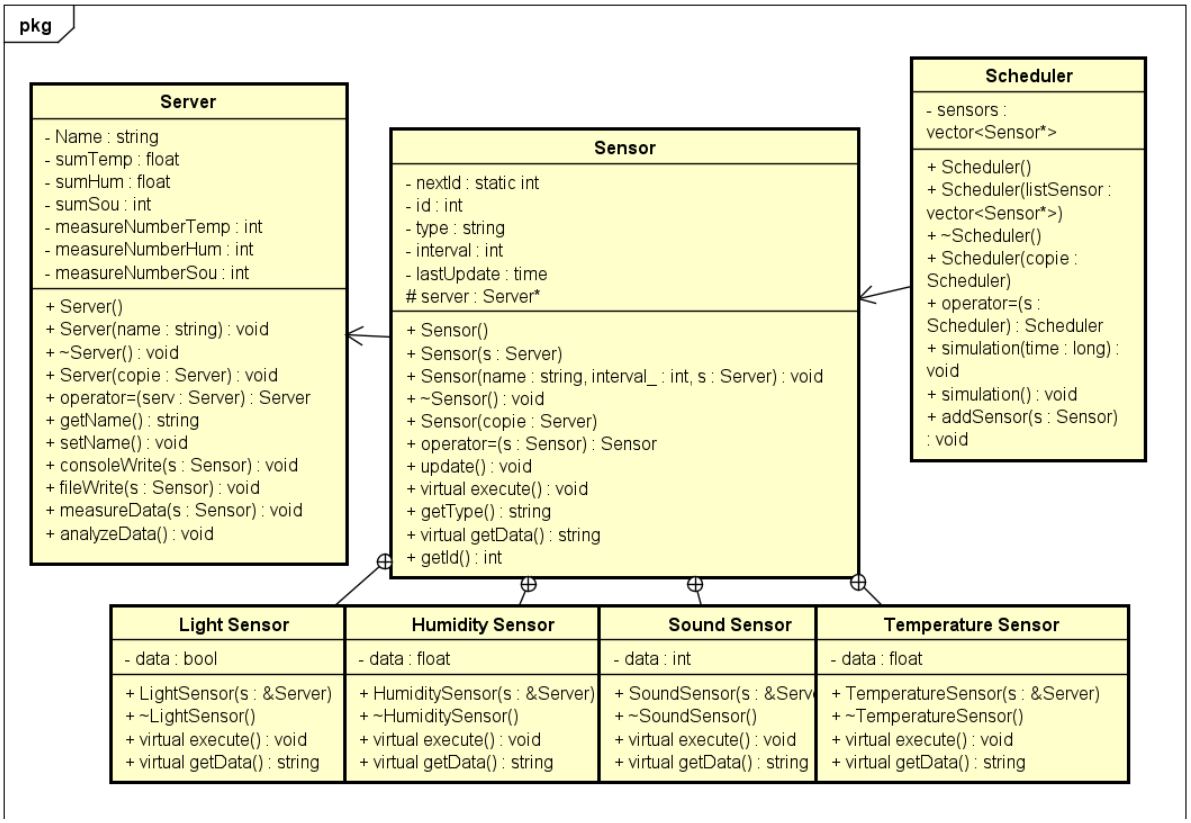
Une classe Scheduler qui gère l'activation et la simulation des capteurs dans le temps.

La classe Sensor a besoin d'un Server pour pouvoir lui envoyer les données qu'elle génère.

La classe Server a besoin d'un Sensor pour pouvoir obtenir des données et les traiter par la suite.

La classe Scheduler a besoin d'une liste de Sensor pour pouvoir les ordonner.

On peut représenter cela par le diagramme UML suivant :



Classe Server

Constructeurs et Opérateurs

- La classe `Server` possède un constructeur par défaut, un constructeur avec paramètre, un destructeur par défaut, un constructeur de copie, et un opérateur

d'affectation. Ces éléments assurent une gestion propre des objets `Server` et permettent d'instancier un serveur avec un nom unique, ou de copier ses attributs d'un serveur à un autre.

Gestion du nom du serveur

- La classe inclut des méthodes `getName()` et `setName()` qui permettent d'accéder au nom du serveur et de le modifier. Le nom du serveur est essentiel pour identifier les fichiers de log qu'il génère.

Méthodes d'écriture des données

- **`consoleWrite(const Sensor& s) const`** : Cette méthode affiche dans la console des informations sur les données reçues, telles que le nom du serveur, l'identifiant du capteur, le type de capteur et les données générées. Cela permet une visualisation en temps réel des informations reçues.
- **`fileWrite(const Sensor& sensor) const`** : Cette méthode enregistre les données des capteurs dans un fichier CSV spécifique au type de capteur. Elle vérifie d'abord si le fichier existe et crée, si nécessaire, les en-têtes pour les nouveaux fichiers. Ensuite, elle enregistre les données au format `DATE,ID,DATA`, en incluant la date et l'heure actuelles, l'identifiant du capteur, et la mesure obtenue. Cette méthode permet de conserver un historique des données reçues et de faciliter leur analyse.

Traitement des données

- **`measureData(const Sensor& sensor)`** : Cette méthode est utilisée pour accumuler les mesures provenant des capteurs de type température, humidité, et son. Pour chaque type de capteur, elle additionne les valeurs reçues et incrémente le compteur correspondant. Ces données accumulées seront ensuite utilisées pour calculer des moyennes durant la simulation.

Analyse des données

- **`analyzeData() const`** : Cette méthode affiche les moyennes des différentes mesures (température, son, humidité) obtenues durant la simulation. En utilisant les données cumulées par `measureData`, elle fournit des informations résumées sur les conditions environnementales surveillées par le système.

Classe Sensor

Constructeurs et Opérateurs

- La classe `Sensor` dispose d'un constructeur par défaut, d'un constructeur prenant un serveur en paramètre, d'un constructeur paramétré qui initialise le type de capteur et l'intervalle de mise à jour, ainsi que d'un constructeur de copie et d'un opérateur d'affectation. Ces constructeurs facilitent la création et la gestion de différents objets `Sensor` en fournissant des moyens flexibles pour initialiser les capteurs et gérer la copie d'objets.
- Chaque capteur possède un identifiant unique (`id`), généré automatiquement à l'aide de l'attribut statique `nextId`, qui incrémente l'identifiant à chaque nouvelle instance de `Sensor`.

Attributs principaux

- **`type`** : Représente le type de capteur (par exemple, température, humidité, son). Cet attribut permet d'identifier la nature des données générées par chaque capteur.
- **`interval`** : Indique la fréquence en secondes avec laquelle le capteur doit générer et envoyer des données au serveur. Cette valeur est utilisée pour déterminer si le capteur doit s'activer et transmettre de nouvelles informations.
- **`server`** : Pointeur vers un objet `Server`, nécessaire pour transmettre les données générées. Cette relation permet au capteur d'interagir directement avec le serveur.

Méthodes d'accès

- **`getType()`** et **`getId()`** : Ces méthodes permettent d'obtenir le type de capteur et son identifiant unique, facilitant l'identification de chaque capteur dans le système.

Méthode `update()`

- Cette méthode est essentielle pour gérer l'intervalle de mise à jour du capteur. Elle vérifie si le temps écoulé depuis la dernière mise à jour dépasse l'intervalle spécifié, et si c'est le cas, elle exécute la méthode `execute()`. Cela permet de contrôler la fréquence d'envoi des données en simulant un environnement réel où les capteurs s'activent périodiquement.

Attribut statique `nextId`

- L'attribut `nextId` est un compteur statique utilisé pour générer un identifiant unique pour chaque capteur créé. Il est incrémenté chaque fois qu'un nouveau capteur est instancié, garantissant ainsi une identification unique au sein du système.

Classes heritantes de Sensor

`LightSensor,HumiditySensor,TemperatureSensor et SoundSensor`

Les classes héritantes de `Sensor` apportent des spécificités qui permettent de simuler différents types de capteurs dans le système. Ces classes dérivées adaptent les fonctionnalités de base de `Sensor` et ajoutent des attributs et méthodes spécifiques pour chaque type de mesure.

Attribut spécifique `data`

- Chaque classe héritante possède un attribut `data` adapter au type de mesure qu'elle effectue, comme un `float` pour des valeurs d'humidité, un `int` pour des niveaux sonores, ou un `bool` pour indiquer la présence de lumière.
- Cet attribut permet de stocker la donnée générée par le capteur et sera utilisé lors de l'envoi vers le serveur.

Constructeur

- Le constructeur de chaque classe héritante initialise le type de capteur et son intervalle de mise à jour. C'est donc dans la classe qu'on définit l'intervalle, pour l'instant celui-ci est choisi de manière arbitraire. En appelant le constructeur de `Sensor`, on s'assure que l'objet est correctement lié à un serveur, auquel il transmettra ses données.

Surcharge de la méthode `execute()`

- La méthode `execute()` est redéfinie dans chaque classe dérivée pour générer une valeur unique au capteur. La valeur générée sera du même type que l'attribut `data` de la classe.
- Une fois la donnée générée, le capteur interagit avec le serveur en appelant plusieurs de ses méthodes :
    - `consoleWrite()` pour afficher la mesure en temps réel,
    - `fileWrite()` pour enregistrer la mesure dans un fichier,
    - `measureData()` pour ajouter cette mesure à l'analyse globale.

Surcharge de `getData()`

- La méthode `getData()` est redéfinie dans chaque classe héritante pour renvoyer la mesure sous forme d'une chaine de caractère afin d'être utilisable dans la classe Server.

Ainsi, chaque classe dérivée de `Sensor` représente un type de capteur unique et est optimisée pour ses propres mesures, tout en utilisant l'infrastructure commune définie dans la classe `Sensor` pour s'intégrer harmonieusement dans le système IoT. Cette architecture rend l'ajout de nouveaux capteurs facile et flexible, en permettant de définir des comportements personnalisés pour chaque nouveau type de capteur.

Classe Scheduler

Gestion des capteurs (`sensors`)

- La classe `Scheduler` maintient une liste de pointeurs vers des objets `Sensor`, permettant ainsi de manipuler un ensemble de capteurs de manière uniforme, quels que soient leurs types spécifiques.
- Les capteurs peuvent être ajoutés à cette liste via la méthode `addSensor`, permettant au `Scheduler` de gérer dynamiquement les capteurs.

Constructeurs et Opérateurs

- `Scheduler` propose un constructeur par défaut et un constructeur paramétré prenant une liste de capteurs. Un constructeur par copie et un opérateur d'affectation permettent de créer des copies de l'objet `Scheduler`, avec duplication des pointeurs de capteurs.
- Le destructeur garantit la libération de la mémoire allouée pour chaque capteur dans la liste, évitant ainsi les fuites de mémoire.

Méthodes de Simulation (`simulation`)

- La méthode principale, `simulation`, permet d'exécuter la simulation en activant les capteurs à des intervalles réguliers pendant une durée donnée.
- Deux versions de la méthode `simulation` existent :
    - **`simulation(long long t)`** : Exécute la simulation pendant une durée de `t` secondes, en vérifiant périodiquement le temps écoulé pour décider de continuer ou non.
    - **`simulation()`** : Exécute la simulation pendant 30 secondes par défaut, gérant le déroulement des mises à jour des capteurs de manière similaire.
- Dans chaque itération de simulation, le `Scheduler` appelle la méthode `update` de chaque capteur. Cette méthode gère la fréquence de mise à jour propre à chaque capteur, permettant ainsi une simulation fidèle de leur comportement réel.

Coordination des mises à jour

- Le `Scheduler` utilise la bibliothèque `chrono` pour gérer la durée et le timing de la simulation, contrôlant ainsi précisément la fréquence d'activation des capteurs.
- En simulant une boucle de mise à jour régulière, `Scheduler` reproduit le comportement d'un système IoT en temps réel, où les capteurs collectent et envoient des données de manière coordonnée.

Conclusion

Ce projet a été une excellente opportunité pour approfondir ma compréhension de la programmation orientée objet. J'ai pu mettre en pratique différentes notions théoriques vues en cours, telles que l'encapsulation, l'héritage et le polymorphisme. La structuration du projet en classes distinctes et interconnectées, comme Server, Sensor, et Scheduler, m'a permis de comprendre l'importance de concevoir un code modulaire et réutilisable.

L'utilisation de l'encapsulation pour protéger les données internes des classes, l'application de l'héritage pour créer des capteurs spécifiques à partir d'une classe mère, et le polymorphisme pour permettre à ces capteurs de se comporter de manière unique, m'ont aidé à maîtriser les fondements de la programmation orientée objet. Ce projet m'a non seulement permis de renforcer mes compétences techniques, mais aussi de mieux apprécier la logique et la rigueur nécessaires pour concevoir un système complexe et évolutif.