Rapport AP4A

BONNET Rémi

AP4A

29/10/2023

Introduction

Dans ce rapport, je vais détailler l'implémentation technique du système de surveillance sensorielle que j'ai développé. Mon objectif initial était de créer un système flexible et facilement extensible pour surveiller divers capteurs tels que l'humidité, le son, la lumière et la température. Nous aborderons dans un premier temps l'implémentation de mes classes, en nous penchant sur la création d'un diagramme de classes UML cohérent avec les besoins du projet. En explorant ces classes, nous comprendrons comment elles interagissent pour garantir un fonctionnement harmonieux du système de surveillance. Par la suite, nous entrerons plus en détails dans l'implémentation en parlant de choix techniques, notamment l'utilisation des threads. Nous examinerons également les améliorations potentielles qui pourraient être implémentées pour rendre le système encore plus performant et polyvalent. Enfin, nous conclurons ce rapport en résumant les principaux points abordés et en mettant en lumière les réussites et les défis rencontrés tout au long du processus de développement.

Conception de l'UML et explications

Dans cet écosystème de surveillance, l'interaction entre trois composants essentiels - le Server, les capteurs et le Scheduler - forme le socle de fonctionnement du système. Chaque élément joue un rôle crucial dans la collecte, le traitement et la présentation des données. UML en annexe.

Server

La classe Server joue un rôle central en traitant les données émises par les capteurs. La gestion des ressources partagées est assurée par les mutex (consoleMutex et fileMutex), garantissant un accès concurrentiel sûr. La méthode consoleWrite permet d'afficher instantanément les données dans la console, fournissant ainsi une interface utilisateur en temps réel. Parallèlement, fileWrite prend en charge l'écriture organisée de ces données dans des fichiers CSV, assurant la persistance et l'analyse ultérieure des données. Ainsi, le Server assure une double fonction : un affichage immédiat des données pour l'utilisateur et leur stockage pour une analyse plus approfondie.

Sensor et ses Classes Dérivées

La classe abstraite Sensor agit comme le modèle générique pour tous les types de capteurs. Elle encapsule des attributs essentiels tels que le type

de capteur (sensorType), la valeur mesurée (valSense) et l'intervalle entre les mesures (intervalle). Les classes dérivées, comme TemperatureSensor, LightSensor, SoundSensor, et HumiditySensor, héritent de cette classe générique. La méthode virtuelle pure aleaGenVal est implémentée individuellement dans chaque classe dérivée, générant ainsi des valeurs aléatoires spécifiques à chaque type de capteur. Cela permet une flexibilité dans la création de capteurs spécialisés tout en maintenant une structure cohérente et uniforme.

Scheduler

La classe Scheduler sert de chef d'orchestre pour le système de surveillance. Elle organise l'exécution des tâches des capteurs en utilisant des threads indépendants pour chaque capteur. La gestion de l'arrêt (stopFlag) et de l'état de chaque thread (stateFlag) est centralisée, assurant ainsi un contrôle efficace sur l'exécution du système. Les capteurs de chaque type sont stockés dans des vecteurs dédiés (temperatureSensors, lightSensors, soundSensors, humiditySensors), permettant une surveillance simultanée et indépendante. Grâce à l'utilisation de threads, le Scheduler garantit une surveillance continue, même pour un grand nombre de capteurs, sans compromettre la performance globale du système.

Interactions entre Classes

L'interaction entre ces classes est orchestrée pour assurer un fonctionnement fluide du système. Lorsque les capteurs génèrent de nouvelles données, celles-ci sont immédiatement transmises au Server. La méthode consoleWrite du Server affiche ces données dans la console, offrant ainsi une interface utilisa-

teur en temps réel. Simultanément, la méthode fileWrite du Server assure l'enregistrement organisé de ces données dans des fichiers CSV, préservant ainsi un historique structuré pour une analyse future. Cette communication entre les classes garantit que le système de surveillance sensorielle fonctionne de manière fiable, avec une réponse immédiate aux changements des capteurs et une persistance assurée des données. Ces interactions entre les classes créent un écosystème cohérent et efficace, formant ainsi la base d'un système de surveillance fiable et adaptable.

Le rôle du multithreading

Une de mes idées centrales à l'origine de ce projet résidait dans la nécessité de gérer un nombre variable de capteurs, sans aucune contrainte quant à leur quantité. Pour concrétiser cette vision, le choix de l'implémentation était crucial. L'utilisation du multithreading a permis un fonctionnement simultané et efficace de l'ensemble des capteurs.

Création des Threads pour Chaque Capteur

Au cœur de notre approche multithread, chaque instance de capteur est associée à un thread dédié. Cela signifie qu'il existe un thread distinct pour chaque capteur, permettant à chaque unité de fonctionner indépendamment et simultanément avec les autres. La création de threads s'effectue lors de l'initialisation de la classe Scheduler. Par exemple, lors de la création d'un capteur de température, un thread spécifique à ce capteur est lancé, exécutant la méthode sensorTask(sensor) ayant pour objectif d'envoyer les données au server pour qu'elles puissent être utilisées.

Contrôle des Threads grâce à des Drapeaux de Contrôle

Pour gérer les threads de manière cohérente et efficace, deux drapeaux de contrôle sont utilisés : stopFlag et stateFlag. Le stopFlag est un indicateur

global qui, lorsqu'il est activé, signale à tous les threads qu'ils doivent se terminer. Le stateFlag, quant à lui, agit au niveau de chaque thread individuel. Lorsque ce drapeau est désactivé (stateFlag = false), le thread est en pause, arrêtant ainsi la génération et la transmission de données au serveur. Lorsque le stateFlag est activé (stateFlag = true), le thread reprend son activité.

Adaptabilité Dynamique du Système grâce aux Threads

Un avantage significatif de l'utilisation des threads réside dans la capacité du système à s'adapter dynamiquement à l'ajout ou au retrait de capteurs. Lorsqu'un nouveau capteur est ajouté, un nouveau thread est créé pour ce capteur spécifique, permettant ainsi son intégration sans perturber le fonctionnement des autres capteurs. De même, lorsqu'un capteur est retiré, son thread associé peut être arrêté de manière ordonnée, assurant ainsi une terminaison propre et évitant toute fuite de ressources.

Communication entre les Threads et le Serveur

Pour transmettre les données des capteurs au serveur, chaque thread exécute la méthode sensorTask spécifique au type de capteur. À l'intérieur de cette méthode, les données du capteur sont générées et ensuite envoyées au serveur via les méthodes consoleWrite et fileWrite de la classe Server. La synchronisation entre ces threads est assurée grâce à l'utilisation de mutex. Les mutex consoleMutex et fileMutex sont verrouillés lors de l'écriture des données dans la console et dans les fichiers CSV respectivement, empêchant ainsi les conflits d'accès simultané. Les mutex sont évidemment placé dans la classe server puisque les fonctions problématiques sont des méthodes de cette classe.

Utilisation d'un thread pour activer le Scheduler

Dans le cœur de notre système de surveillance sensorielle réside une interaction entre les utilisateurs et les capteurs. Le lancement du Scheduler par un thread permet à l'utilisateur de pouvoir interagir avec le programme sans attendre la fin d'exécution des threads sensor. L'utilisateur peut ainsi arrêter l'affichage géré par les threads puis le reprendre ou encore arrêter le programme.

Critique et améliorations envisageables

Interface Utilisateur Intuitive

Pour rendre le système encore plus convivial, l'ajout d'une interface utilisateur graphique intuitive est envisageable. Cette interface permettrait aux utilisateurs de configurer facilement les capteurs et de visualiser les données de manière claire et compréhensible.

Notifications en Temps Réel

L'intégration de notifications en temps réel constituerait une amélioration significative. Ces notifications (envoyées par e-mail par exemple ou en interne), alerteraient instantanément les utilisateurs en cas de dépassement de seuils prédéfinis ou d'événements critiques. Cela garantirait une réactivité accrue face aux situations d'urgence.

Stockage Structuré des Données

L'utilisation de bases de données relationnelles telles que MySQL ou SQLite pourrait optimiser le stockage des données des capteurs. Cela faciliterait une gestion organisée des données, ouvrant la voie à des analyses approfondies.

Réflexion sur les Limitations Actuelles

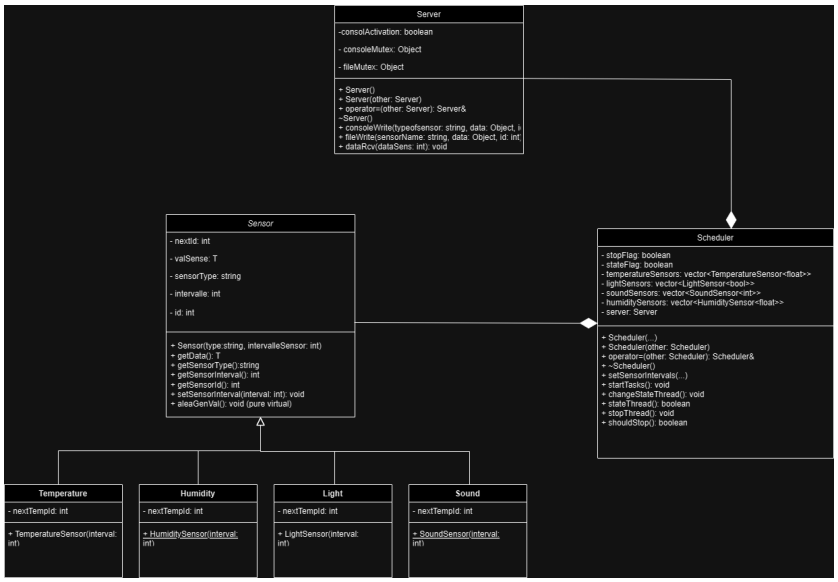
Bien que le système actuel soit fonctionnel, il présente certaines limitations qui méritent d'être soulignées. L'une de ces limitations réside dans le fait que chaque capteur du même type a des paramètres prédéfinis identiques tels que l'intervalle entre les mesures et les seuils. Cette uniformité peut être contraignante dans des environnements où différents capteurs du même type nécessitent des configurations spécifiques en raison de conditions variables. Cette approche standardisée peut entraver l'adaptabilité du système dans des situations complexes où la variabilité des capteurs est essentielle. Par exemple, dans un réseau de capteurs environnementaux, il est courant que les capteurs placés dans des zones différentes aient des seuils et des intervalles de mesure distincts en raison des variations locales des conditions environnementales.

Conclusion

En conclusion, la création du système de surveillance sensorielle fut enrichissante me permettant de découvrir en profondeur les principes de la programmation orientée objet à travers la conception de classes et d'améliorer considérablement mes compétences en multithreading ainsi que la découverte des diagrammes de classes UML. Grâce à l'utilisation ingénieuse des classes Server, Sensor et Scheduler, le système offre une liberté à l'utilisateur et ainsi adapte le programme à ses besoins. L'introduction du multithreading a été cruciale pour permettre une surveillance simultanée et indépendante de divers capteurs. Chaque capteur fonctionne comme une entité autonome, gérée par son propre thread, garantissant ainsi une réactivité instantanée aux changements environnementaux. Cependant, des limitations subsistent, notamment dans la gestion des capteurs de même type. L'uniformité des paramètres prédéfinis peut entraver l'adaptabilité du système dans des environnements complexes. Pour remédier à cela, des améliorations futures pourraient inclure la personnalisation des configurations pour chaque capteur, offrant ainsi une flexibilité accrue dans des contextes variés.

Annexes

UML : diagramme de class



Code des threads

```
void startTasks() {

 std::vector<std::thread> threads;

 for (auto& temperatureSensor : temperatureSensors) {
    threads.emplace_back([&](){this->sensorTask(temperatureSensor);} );
 }

 for (auto& lightSensor : lightSensors) {
    threads.emplace_back([&](){this->sensorTask(lightSensor);} );
 }

 for (auto& soundSensor : soundSensors) {
     threads.emplace_back([&](){this->sensorTask(soundSensor);} );
 }

 for (auto& humiditySensor : humiditySensors) {
     threads.emplace_back([&](){this->sensorTask(humiditySensor);} );
 }

 // Joindre tous les threads
 for (auto& thread : threads) {
     thread.join();
```