Rapport de projet TP AP4A
Système IoT

Pourcine Mattéo Groupe 1

October 22, 2023

Contents

1 Explication du projet

Le projet a pour but d'implémenter un système IoT composé de 4 capteurs (température, humidité, son et lumière), d'un serveur et d'un scheduler.

Le serveur recevra les données des capteurs par l'intermédiaire du scheduler pour les écrire dans des fichiers de log ou directement les écrire dans la sortie standard. Le scheduler lui recevra les données des capteurs et les enverra au serveur à des intervalles de temps différents. Ces intervalles seront choisit par l'utilisateur du programme.

Ce projet sera réaliser à l'aide du language C++ et donc en utilisant les concepts de la programmation orientée objet. Vous pourrez trouver en haut à droite de chaque page un lien menant au répertoire git du code source du projet. Différentes sources ont été utiles à la réalisation de celui-ci, principalement les TD et CM d'AP4A mais aussi la documentation microsoft sur le C++.

Ci-dessous le diagramme UML de mon implémentation pour une visualisation du squelette du projet.

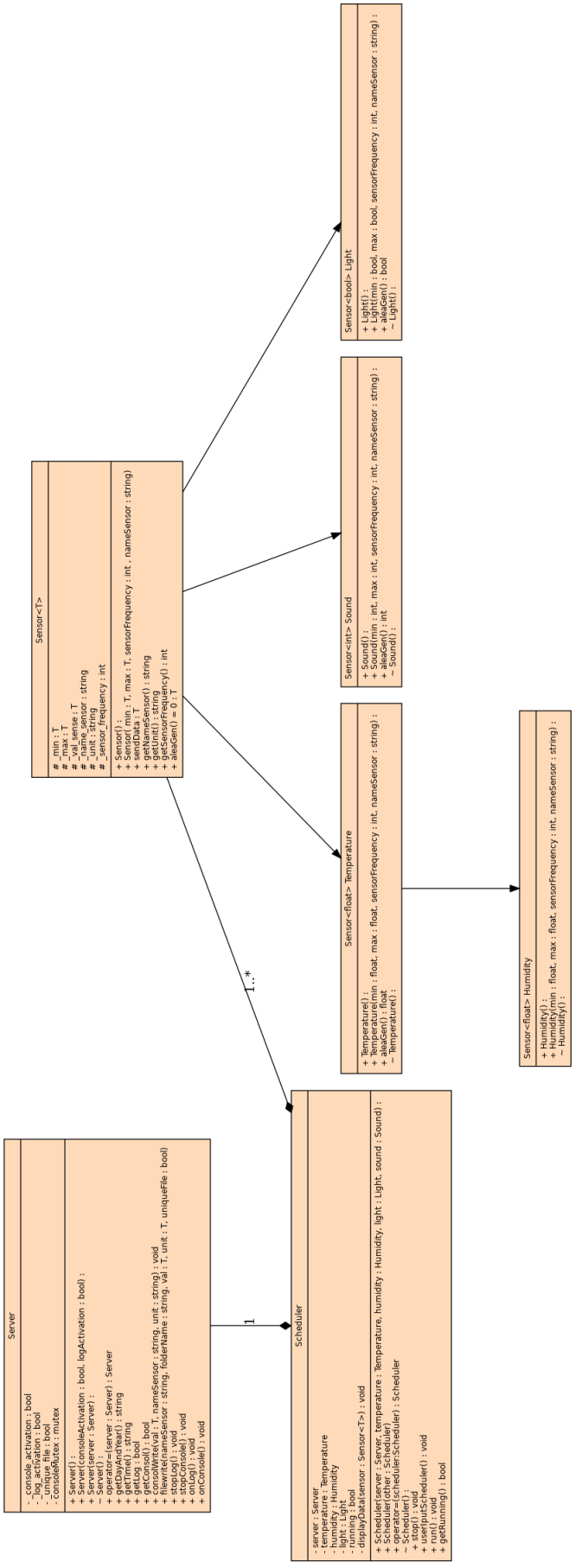

Figure 1: Diagramme UML

2 Choix et implémentation

J'ai choisi de garder la ligne directrice du diagramme UML fourni en TP concernant l'implémentation. Cependant de nouvelles fonctions ont été ajouté et d'autres ont été supprimé pour des choix pratiques. De plus, certains héritages ont été revu comme par exemple celui de "Humidity".

En effet dans mon implémentation, "Humidity" hérite de "Température" avec un héritage de spécialisation donc public. Je vais essayer d'apporter quelques précisions concernant ce choix.

La classe mère est une classe "template" virtuelle pour définir une base commune. La méthode AleaGen est "virtual pure". Ainsi je n'ai plus qu'à redéfinir cette méthode dans les classes filles. Tous les capteurs héritent des attributs "protected" de la classe mère. Ainsi le constructeur spécifique implémenté est commun à toutes les classes.

Le choix de faire hérité "Humidity" de "Température" vient tout simplement du fait que dans mon implémentation, ces deux capteurs sont du même type et ont les mêmes attributs. J'ai donc trouvé cette méthode judicieuse vis-à-vis du contexte.

La classe Server permet quand à elle de gérer la gestion des écritures dans les logs, la console et d'avoir accès au temps du système. Pourquoi avoir choisi d'implémenter une méthode qui retourne la date, l'heure et l'année ?

Quand l'utilisateur choisis l'écriture dans un fichier .txt, le nommage de ce fichier ce fait avec le nom du capteur choisit par l'utilisateur et la date, comme ceci "nom-du-capteur-date.txt". Ainsi si on imagine que ce système s'exécute 24 heures sur 24, chaque capteur possède un fichier de log par jour.

Si on imagine maintenant que l'utilisateur veut écrire toutes les données dans un seul fichier. Il peut activer le booléen "uniqueFile". Les données sont séparées par des virgules. Il pourra extraire les données et les utiliser à sa guise.

La classe "Scheduler" fait le lien entre entre le serveur et les capteurs en utilisant la composition. La partie la plus technique du projet est la gestion de la fréquence à laquelle les données des capteurs sont écrites dans les logs et la console.

La solution utilisée pour répondre à ce problème est l'utilisation du multithreading. Chaque capteur possède un thread.

Les threads sont évidemment créés dans le Scheduler. Celui ci reçoit les valeurs de chaque capteur par des getter implémenter dans la classe "Sensor". Ensuite le scheduler envoit ces valeurs au serveur qui à son tour les écris dans la console ou les logs. J'expliquerais en détails le coeur du projet dans la partie **2.1**.

Passons maintenant à l'interface utilisateur. J'ai voulu ici laisser un maximum de choix concernant l'activation de l'écriture console ou log, le choix de la présence ou non des capteurs, l'écriture dans un fichier unique pour les 4 capteurs ou tout simplement de choisir un fichier par capteur.

Grâce à l'utilisation des threads l'utilisateur peut, pendant l'exécution du programme, activer ou désactiver l'écriture dans la console et/ou dans les logs ou quitter le programme. Lorsque l'utilisateur décide d'arrêter les deux écritures, il a le choix entre les réactiver ou tout simplement quitter le programme.

2.1 Explication des fonctions principales

Passons maintenant à la partie crucial du projet. Après avoir coder et tester les classes "Server", "Sensor" et une partie de la classe "Scheduler", il fallait implémenter 'X' méthodes qui allait faire le lien entre "Sensor" et "Server". Ce 'X' est égal à deux. Il s'agit de "displayData()" et de "run()".

2.1.1 Explication de displayData()

Cette méthode template "private" de Scheduler est destiné à être appelée dans un thread. Elle prend en paramètre une référence sur un Sensor<T>et ne retourne rien.

Son fonctionnement est assez simple. Elle vérifie premièrement que le capteur passé en paramètre n'est pas un capteur inconnu. Si c'est le cas, on entame une vérification sur le système en lui même. Si le scheduler est désactivé, on ne prend pas la peine d'aller plus loin.

Ensuite on vérifie si l'utilisateur à activité l'écriture console. Si c'est le cas on appel la méthode consoleWrite de Server en passant en paramètre la valeur du capteur en question, son nom et son unité. On fait de même avec l'activation ou non des logs par l'utilisateur.

Pour finir, en ayant en tête la méthode run() et l'utilisation des threads, on veille à utiliser la méthode "sleep for" de la classe thread en passant en paramètre la fréquence d'envoie du capteur rentrée par l'utilisateur.

Remarque : L'accès la console étant concurrent, c'est à dire que si plusieurs threads tentent d'y accéder en écriture, l'affichage risque d'être defectueux et illisible pour l'utilisateur. Pour résoudre ce problème j'ai ajouté un attibut de type mutex dans la classe Server qui définit la méthode consoleWrite pour lock l'accès à la console lorsque le thread écrit les données des capteurs dans celle-ci. Ainsi à la fin de la fonction le thread unlock l'accès et un autre thread peut venir effectuer une tâche sans être gêner par d'autres threads.

2.1.2 Explication de run()

Cette méthode est une méthode publique de Scheduler qui ne renvoit rien et qui ne prend rien en paramètre.

Son but est simple, tant que le scheduler est actif, des threads sont créés pour chaque capteur et appellent la méthode displayData avec en paramètre un des quatres capteurs attributs de scheduler. Ainsi les threads effectuent leur tâche puis se terminent. Si le système n'est pas désactivé, on recommence.

On remarque que ces deux méthodes précédemment expliquées et les trois grandes classes définient dans le projet sont intrinsèquelent liées.

2.1.3 Création du main

Le main sert à initialiser les capteurs et le serveur en fonction des choix de l'utilisateur. Pour rendre possible le choix d'arrêter le programme ou l'écriture dans les logs/console, un thread intervient encore une fois

3 Améliorations possibles