SIMULATEUR IoT

Enseignants:

Badreddine Chah - Alexandre Brunoud

Par
Michel DIENG

Sommaire:

Présentation du projet

Le TP de AP4A consiste en la réalisation d'un mini-projet individuel visant à appliquer les connaissances acquises lors du cours, offrant un aperçu des attentes du projet Java à rendre en fin de semestre. Le projet implique la création d'un simulateur pour modéliser un écosystème IoT spécialisé dans la surveillance de la qualité de l'air dans un espace de travail. Cet écosystème comprend un serveur interagissant avec divers types de capteurs (température, humidité, lumière, son) répartis dans l'espace de travail. Les données des capteurs peuvent être stockées sous forme de fichiers journaux ou affichées dans une console. Au cours des trois sessions de trois heures allouées, les étudiants mettront en pratique les principes de la programmation, développeront l'architecture du simulateur, écriront le code, et utiliseront des outils de suivi, de versionnage et de travail collaboratif.

Mon premier programme et UML

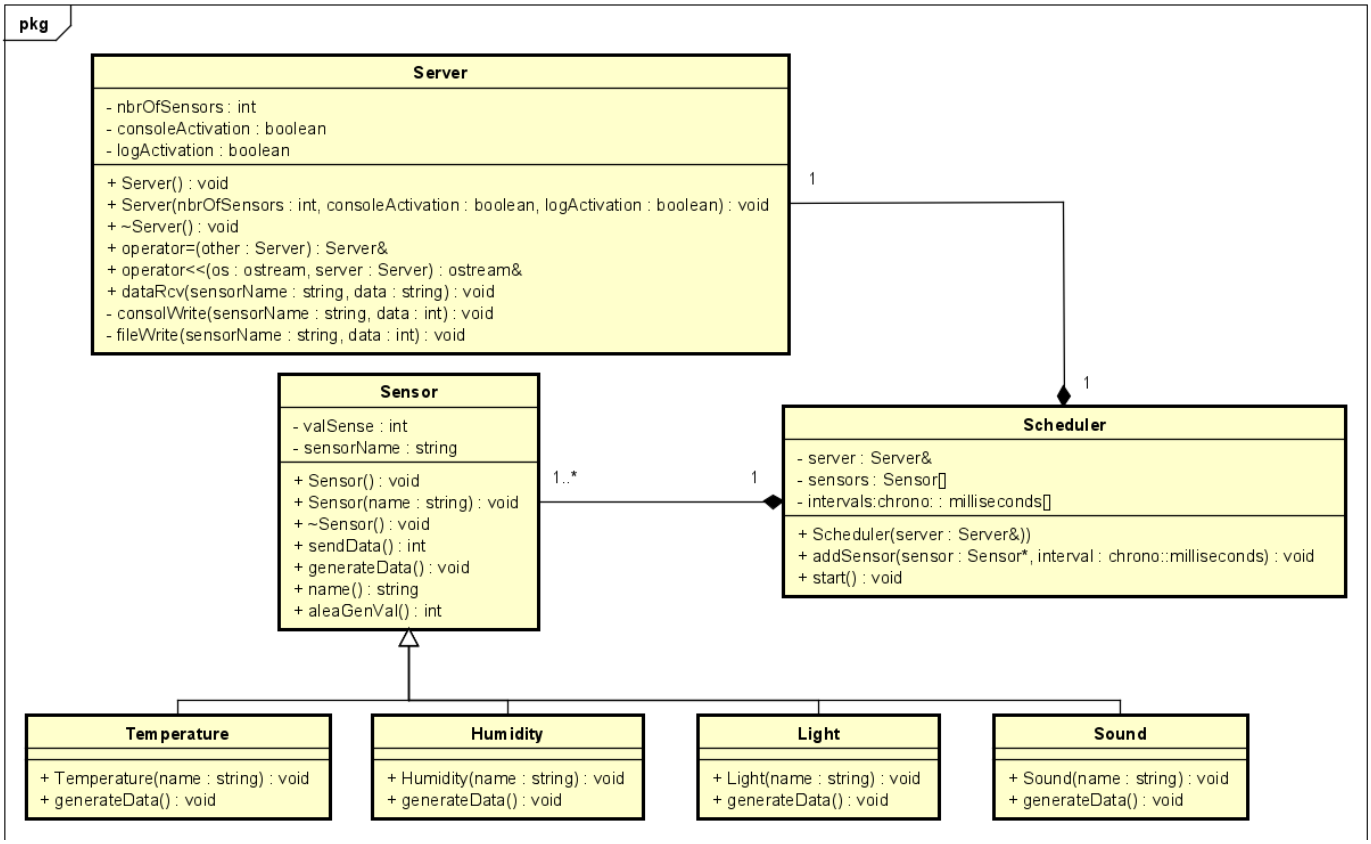
“Hello World !”

Dans ce segment, nous présentons un exemple de programme minimal en langage C++. Le code source consiste en une fonction 'main()' qui affiche le message 'Hello World !' sur la console. Ce programme élémentaire illustre la structure de base d'un programme C++, avec 'std::cout' utilisé pour afficher du texte sur la sortie standard, suivi de 'return 0;' pour indiquer que le programme s'est terminé avec succès. Bien que ce soit un exemple simple, il constitue la fondation de tout programme en C++ et illustre la simplicité et la puissance de ce langage de programmation.

```
PS C:\Users\User> cd C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG
PS C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG> g++ -std=c++17 -Wall -Wextra -We
rror HelloWorld.cpp -o HelloWorld
PS C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG> .\HelloWorld.exe
Hello World !
```

Diagramme UML

Le diagramme UML présenté ci-dessous illustre l'architecture de notre système de gestion de capteurs. Au cœur de ce système se trouve la classe 'Server', qui agit en tant que serveur central pour recevoir, traiter et stocker les données des capteurs. La classe 'Server' est dotée de plusieurs attributs, dont 'nbrOfSensors' pour le nombre de capteurs connectés, 'consoleActivation' et 'logActivation' pour activer ou désactiver la sortie sur la console et la journalisation des données, respectivement.
Les capteurs sont représentés par la classe abstraite 'Sensor', qui contient les attributs de base pour les valeurs mesurées et le nom du capteur. Les classes dérivées, telles que 'Temperature', 'Humidity', 'Light' et 'Sound', héritent de 'Sensor' et spécialisent leur fonctionnalité pour générer des données spécifiques à leur type de mesure.
La classe 'Scheduler' coordonne la génération de données des capteurs et leur transmission au serveur. Elle agrège plusieurs objets 'Sensor' ainsi que les intervalles de mise à jour, permettant une gestion asynchrone de plusieurs capteurs. La classe 'Scheduler' utilise la classe 'Server' pour transmettre les données collectées.
Ce diagramme UML reflète la modularité et la flexibilité de notre système, permettant d'ajouter de nouveaux types de capteurs en héritant de 'Sensor' tout en conservant une structure cohérente. Il illustre également l'agrégation de capteurs dans 'Scheduler' et leur interaction avec le serveur central, constituant ainsi un composant clé de notre système de gestion de capteurs.


pkg
Server
- nbrOfSensors : int
- consoleActivation : boolean
- logActivation : boolean
+ Server() : void
+ Server(nbrOfSensors : int, consoleActivation : boolean, logActivation : boolean) : void
+ ~Server() : void
+ operator=(other : Server) : Server&
+ operator<<(os : ostream, server : Server) : ostream&
+ dataRcv(sensorName : string, data : string) : void
- consolWrite(sensorName : string, data : int) : void
- fileWrite(sensorName : string, data : int) : void
1
1
Sensor
- valSense : int
- sensorName : string
+ Sensor() : void
+ Sensor(name : string) : void
+ ~Sensor() : void
+ sendData() : int
+ generateData() : void
+ name() : string
+ aleaGenVal() : int
1..*
1
Scheduler
- server : Server&
- sensors : Sensor[]
- intervals:chrono: : milliseconds[]
+ Scheduler(server : Server&))
+ addSensor(sensor : Sensor*, interval : chrono::milliseconds) : void
+ start() : void
Temperature
+ Temperature(name : string) : void
+ generateData() : void
Humidity
+ Humidity(name : string) : void
+ generateData() : void
Light
+ Light(name : string) : void
+ generateData() : void
Sound
+ Sound(name : string) : void
+ generateData() : void

Server

Dans le cadre du développement de notre simulateur, nous allons aborder la création de la classe "Server", un composant essentiel de notre système, en cohérence avec ce qui a été précédemment expliqué. Cette classe aura pour responsabilité de recevoir les données émises par l'ensemble des capteurs, puis de les enregistrer dans des fichiers journaux ou de les afficher dans la console. Pour y parvenir, la première étape consistera à mettre en œuvre la forme canonique de Coplien pour cette classe. Une fois cette étape accomplie, deux fonctions spécifiques devront être développées : "consoleWrite" qui permettra la visualisation des données entrantes dans la console, et "fileWrite" pour le stockage des données des capteurs dans des fichiers journaux distincts, chaque capteur ayant son propre fichier dédié.

```
PS C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG> g++ -std=c++17 -Wall -Wextra -We
rror Server.cpp -o Server
PS C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG> ./Server
Données reçues dans la console : 42
Données reçues dans la console : 67
```

Dans cette section, nous avons effectué la surcharge de deux opérateurs, "=" et ">>", chacun ayant des fonctions distinctes. Pour l'opérateur "=", la surcharge a été implémentée pour permettre la copie du contenu d'un objet dans l'objet courant. En ce qui concerne l'opérateur ">>", deux surcharges distinctes ont été réalisées : la première redirection a été configurée pour afficher l'élément désiré dans la console, tandis que la deuxième redirection a été conçue pour acheminer cet élément vers un fichier de journal (log). Dans le cadre de la prochaine session, un rapport concis doit être présenté, détaillant à la fois le rôle et la mise en œuvre de ces opérations susmentionnées.

```
PS C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG> g++ -std=c++17 -Wall -Wextra -We
rror Server_Surcharge.cpp -o Server_Surcharge
PS C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG> ./Server_Surcharge
Nombre de capteurs : 5
Activation de la console : Activée
Activation du log : Activée
```

Sensor

La classe 'Sensor' représente une pierre angulaire de notre système de capteurs, offrant une base flexible et polyvalente pour la collecte de données. Cette classe abstraite définit un ensemble de fonctionnalités essentielles, tout en permettant la création de capteurs spécialisés. Chaque capteur dérivé, tel que 'Temperature', 'Humidity', 'Light' et 'Sound', hérite de 'Sensor' et le personnalise en fonction de son type de mesure. Les constructeurs et méthodes de chaque capteur dérivé sont adaptés pour générer et transmettre des données de manière spécifique, tout en conservant une structure cohérente.
La classe 'Sensor' elle-même inclut des méthodes de base, telles que 'sendData()', qui renvoie la valeur actuelle du capteur, et 'generateData()', une méthode virtuelle pure qui oblige les classes dérivées à mettre en œuvre leur propre logique de génération de données. De plus, 'Sensor' comprend une méthode interne 'aleaGenVal()' pour générer des valeurs aléatoires, simulant ainsi les lectures de capteurs dans un environnement réel.
Cette architecture permet une grande extensibilité et modularité, car de nouveaux types de capteurs peuvent être ajoutés facilement en dérivant de 'Sensor', tout en respectant la structure et les fonctionnalités existantes. Ainsi, notre système est prêt à gérer une variété de capteurs et de types de données pour répondre aux besoins de notre application.

```
PS C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG> g++ -std=c++17 -Wall -Wextra -We
rror Sensor.cpp -o Sensor
PS C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG> ./Sensor
Température : 6 °Celsius
Humidité : 1.3%
Lumière : Off
Son : 46 dB
```

Scheduler

La classe 'Scheduler' est un élément central du système, jouant un rôle crucial dans la gestion des capteurs et la transmission des données vers le serveur. Cette classe prend en charge la planification des mises à jour des capteurs à des intervalles de temps spécifiques, permettant ainsi de collecter des informations précieuses à des moments opportuns. Elle repose sur un ensemble de capteurs et de durées d'intervalle définis par l'utilisateur, garantissant une flexibilité dans la collecte des données. Une fois en marche, le 'Scheduler' démarre des threads dédiés à chaque capteur, permettant la génération de données et leur transmission au serveur de manière asynchrone. Cette approche assure une réactivité optimale, car elle permet à chaque capteur de fonctionner de manière indépendante, tandis que le 'Scheduler' coordonne l'ensemble du processus. En fin de compte, le 'Scheduler' constitue un élément essentiel pour la collecte, la gestion et la transmission efficace de données provenant d'une variété de capteurs vers le serveur central.

```
PS C:\Users\User> cd C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG
PS C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG> g++ -std=c++17 -Wall -Wextra -We
rror Scheduler.cpp -o Scheduler
PS C:\Users\User\OneDrive\Bureau\1_ere_annee_UTBM\AP4A\Code_AP4A_Groupe_B_Michel_DIENG> ./Scheduler
Température : 29 °C
Humidité : 60 %
Température : 17 °C
Lumière : 0 lux
Température : 15 °C
Son : 83 dB
Humidité : 94 %
Température : 12 °C
Lumière : 1 lux
Humidité : 88 %
Température : 55 °C
Lumière : 0 lux
Son : 64 dB
Température : 8 °C
Humidité : 92 %
Température : 68 °C
Humidité : 59 %
Lumière : 0 lux
Température : 49 °C
Son : 86 dB
Température : 20 °C
```

Conclusion

Ce mini-projet au sein du cours AP4A a été une opportunité précieuse pour appliquer nos compétences en programmation. Il nous a permis de créer un simulateur IoT pour surveiller la qualité de l'air dans un espace de travail, avec un serveur interagissant avec divers capteurs (température, humidité, lumière, son). Au cours de trois sessions, nous avons développé une architecture, écrit du code, et utilisé des outils de versionnage.

Ce projet a ainsi renforcé nos compétences en programmation, en nous préparant aux futurs projets IoT et mettant en avant l'importance de la planification et de la conception soignée. Ces compétences seront précieuses dans nos futures entreprises de développement de logiciels et de projets IoT.