Sensor Manager Simulator

By: Javad Zandi

This project presents a Sensor Manager simulator, simulating sensor acquisition data with a concurrency feature.

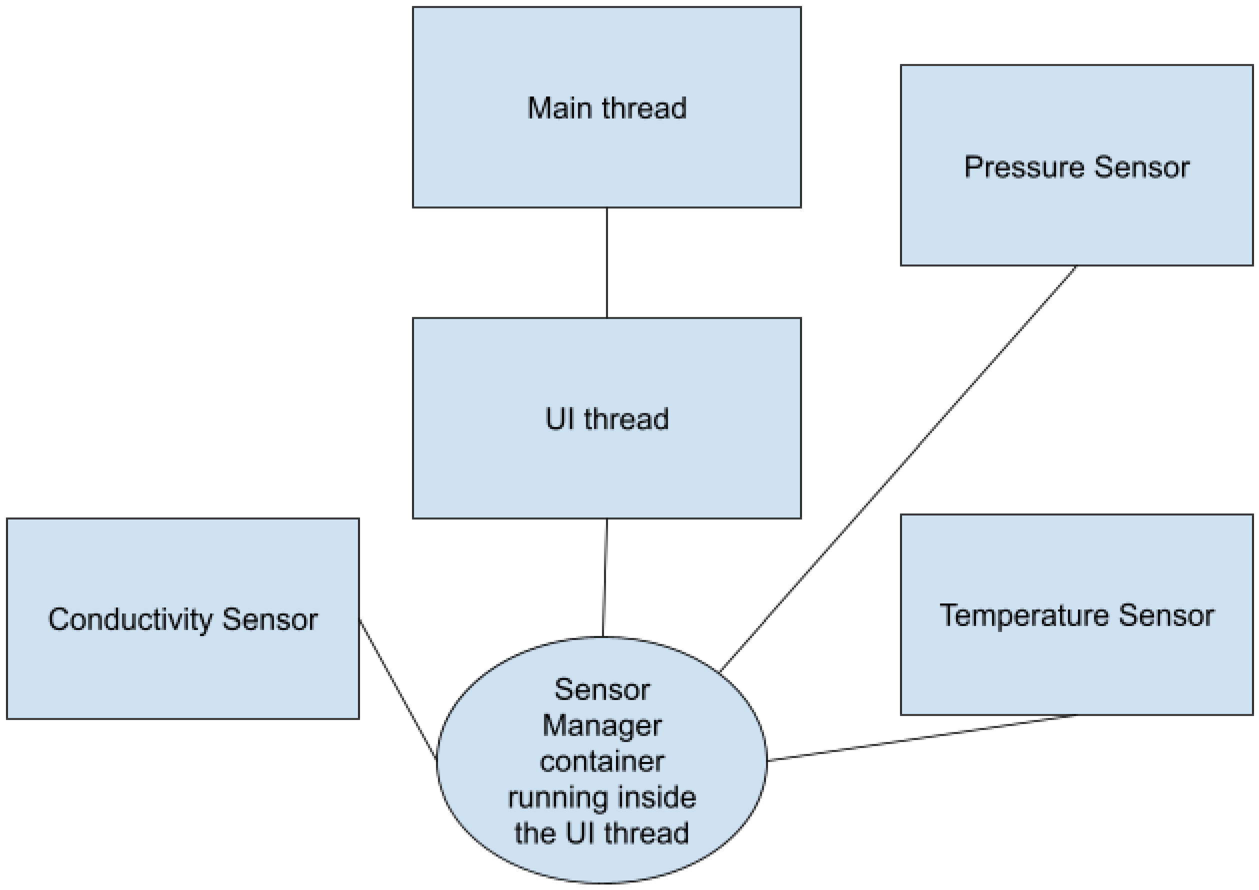
Quality of functional specification

This application simulates sensor data acquisition. SensorManager supports simulating behavior of measuring pressure, temperature, and conductivity with 1s, 8s, and 16s intervals respectively. SensorManager benefits from concurrent programming to acquire sensor data concurrently as much as close to real situations. Essentially, SensorManager is useful when we want to measure, analyze, or test a large number of sensors in a specific architecture for optimizing sensor power consumption or communication when we don't want to invest in making hardware for testing. Moreover, this approach makes development faster because of hardware abstraction. Typically, sensors generate data and print it to the console, representing their measurement value. Upon creating the sensor, it reports its measurement and liveliness immediately on creating and stopping on removing it from the poll. Measurement values are distinguishable by timestamp, showing when the measurement is done, and sensor ID showing which sensor measured that data. All printed values are controlled such that all texts stay readable as it is expected. A command line is provided for interacting with the user. This tool parses the commands and runs a small validation. Supported commands are add, remove, list. These are sample commands for creating, destroying, and listing available sensors inside the simulator:

Example: add p
Output: "[2024-06-07 06:02:12]	Sensor id: 0	data: Pressure	Value: N/A"
This command adds a pressure sensor. You can also use "add pressure"

Sensor types are p|t|c|pressure|temperature|conductivity. Upon running each command, you will see a response from SensorManager corresponding to successful or failure prompts.

Documentation of software architecture



SensorManager benefits from concurrent programming powered by std libs in C++. As you can see, after running the main function of the program, it prints the Usage to advise the user to feed proper values to configure the simulator as close as possible to reality. From there, a UI thread will be created to handle input prompts from the user and interact with the user to receive commands and call proper operation from Sensor Manager. On the other hand, the sensor can be created by a sensor manager with parsed inputs from user input. Finally, sensors start generating data on their proper interval time, pretending real measurements. All rectangles show an independent thread.

Use of object-oriented design principles

SensorManager is implemented in an object-oriented model such that it meets the requirements as well as to be implemented quickly and efficiently. Sensors usually share the same attributes and interface no matter their type. In the presented project, a basic sensor type is designed such that its code can be inherited in different types of sensors. This idea contributes to code reusing and also reduces the cost of implementation as the code base is kept at a minimum, easy to develop test cases or debug the app if it has a problem. In the SensorManager project, temperature, pressure, and conductivity share the same attributes and interface. Because of this, shared attributes and interface moved to the Sensor class and all sensor classes derived from the Sensor class.

Design Steps

Before starting coding, I read the description two times to make sure I understood a clear big picture of the project. Hopefully, the description was clear and understandable. Later on, I created a breakdown of the project for small achievable tasks, easy to follow with a set of defined input and expected behavior in the SensorManager app. From there, I ordered them based on priority determined by the challenge description and deliverable material. A breakdown of these tasks includes:
1- Reading the description two times.
2- Making a bullet point from the expected feature
3- Creating small tasks and features from my big picture
4- Scheduling and ordering tasks based on their priority
5- Initial minimum implementation of the main function and input command parser. This is important as it is needed for testing.
6- Develop a solution for the producer-consumer model as we have the same problem. Sensors are producers and std::cout is the consumer.
7- Checking my tasks list to review what is left.
8- Developing base class for Sensor
9- Developed derived class for three sensors.
10- Integrating codes and compiling, and preparing for testing.
12- 2 synchronous related bugs detected and addressed for smooth interaction with the user.
13- Commiting code to the GitHub repo
14- Writing the documentation for the code.

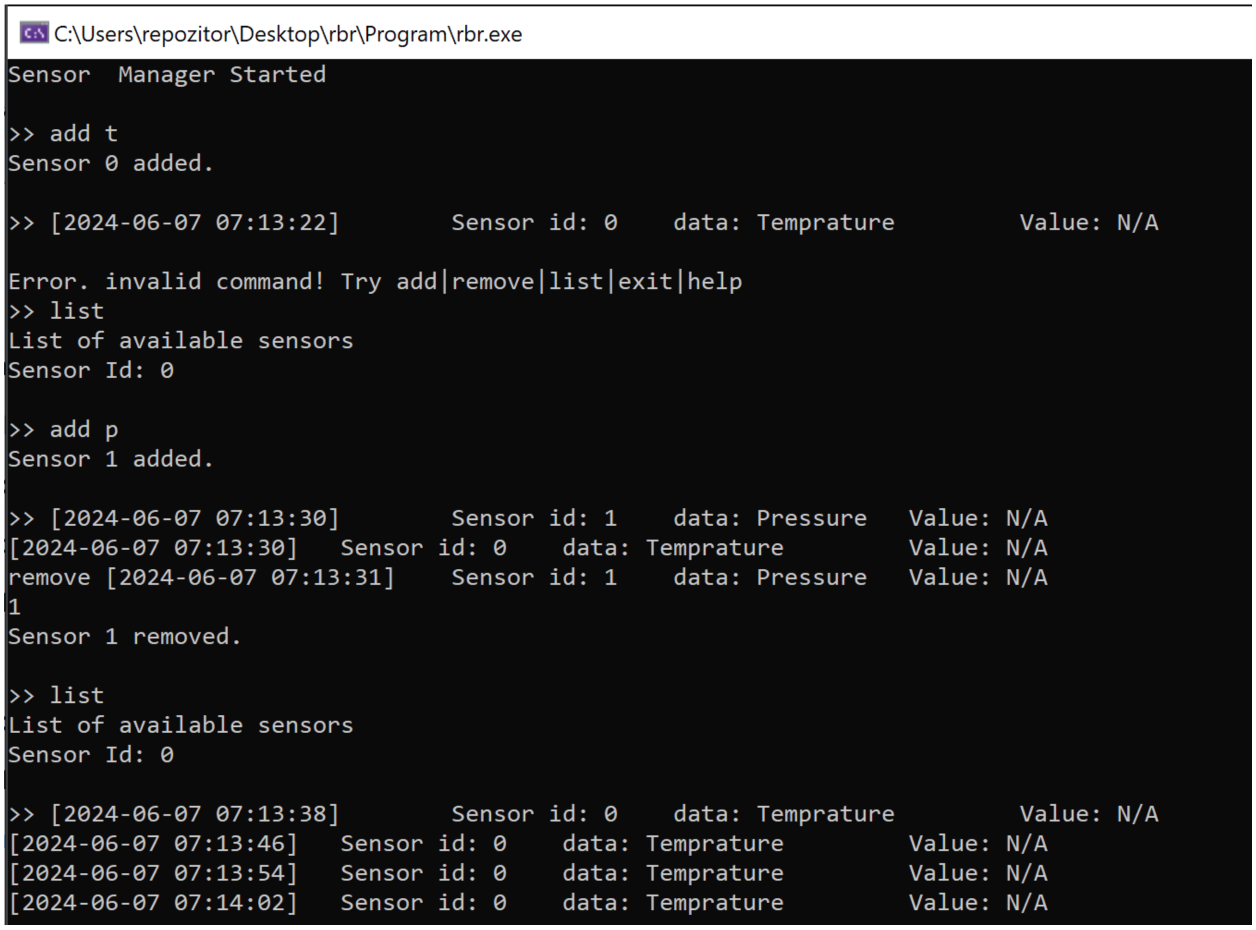
Use of C++-oriented language and library features

All classes in this project are defined in “rbr” namespace to avoid confusion with similar naming in other projects or external libraries. Using std libraries for using smart pointer which is essential for memory management of dynamic containers is another tactic to overcome memory leakage issues. SensorManager has many async operations and std C++ libraries including mutex, conditional_variable are able to synchronize threads and avoid falling into thread contention issues when they want to print a string on the console. Natively, std:cout is not thread-safe and texts printed by different threads interfere with each other. SensorManager benefits from a mutex for managing text distortion. As declaring variables takes time, SensorManager benefits from “auto” for easier and faster implementation.

General use of best practices

Some useful engineering practices were not possible due to the limited time. These practices included developing unit testing, automation testing, and well documentation. However, SensorManager is a solution that has a standard OO architecture, bug-free, deadlock-free, and memory leak-free. Additionally, it interacts with the user-based required description listed. Moreover, A brief README file in the root of the repo developed to show how SensorManager can be compiled.

Presentation of results



C:\Users\repozitor\Desktop\rbr\Program\rbr.exe

```
Sensor  Manager Started

>> add t
Sensor 0 added.

>> [2024-06-07 07:13:22]        Sensor id: 0    data: Temprature         Value: N/A

Error. invalid command! Try add|remove|list|exit|help
>> list
List of available sensors
Sensor Id: 0

>> add p
Sensor 1 added.

>> [2024-06-07 07:13:30]        Sensor id: 1    data: Pressure   Value: N/A
[2024-06-07 07:13:30]   Sensor id: 0    data: Temprature         Value: N/A
remove [2024-06-07 07:13:31]    Sensor id: 1    data: Pressure   Value: N/A
1
Sensor 1 removed.

>> list
List of available sensors
Sensor Id: 0

>> [2024-06-07 07:13:38]        Sensor id: 0    data: Temprature         Value: N/A
[2024-06-07 07:13:46]   Sensor id: 0    data: Temprature         Value: N/A
[2024-06-07 07:13:54]   Sensor id: 0    data: Temprature         Value: N/A
[2024-06-07 07:14:02]   Sensor id: 0    data: Temprature         Value: N/A
```