[redacted]

BSN EXEMPLAR Group 1_1

Introduction:

The Body Sensor Network (BSN) Exemplar presented in this document is specifically designed for the healthcare domain, aiming to revolutionize patient monitoring through the incorporation of a self-adaptive architecture. Unlike traditional BSNs, this exemplar dynamically adjusts its parameters based on patients' conditions and external factors, overcoming challenges such as the inability to adapt to changing conditions, noise, and uncertainty. This self-adaptive system utilizes feedback mechanisms and machine learning algorithms to optimize performance, ensuring more accurate and responsive patient monitoring.

System Architecture:

The exemplar employs six sensors to monitor the patient's health status, with a central processor analyzing the collected data. Two primary tasks are undertaken to enhance the functionality of the system.

Task 1: Monitoring Sensor Data:

To achieve efficient monitoring of sensor data, a ROS Python node has been implemented. This node subscribes to messages from all sensor data topics, collects the data, and stores it in a database. The genson library is utilized to construct a JSON schema representing the contents of the database. This subscriber node facilitates the monitoring of all patient data, providing a comprehensive overview of the health status.

Task 2: Executing Adaptation:

The publisher.py file publishes data to the enactor endpoints, here implement an http-server to serve as an intermediary between the external data publisher and the ROS subscriber node. This server exposes port 7070, which enables external updates and adaptations to the SensorData. By sending the HTTP POST request to the “/enactor/<topic>” endpoints, we dynamically modify and adapt the sensor data readings.

Implementation Instructions:

To run the system, users are provided with clear instructions:

1. Pull the Docker image.
2. Navigate to the exposed endpoint on the browser.

3. Open the BSN and run the "./run.sh" script in a new terminal.
4. Update the CMakeLists in the modified folder.
5. To monitor the endpoint, run "python listener.py" in the directory of the listener script.
6. To publish new sensor data, run "python publisher.py" in the directory of the publisher script.
7. Monitor the data on the browser at http://localhost:5000/monitor.
8. Execute adaptation by adjusting sensor data at http://localhost:5000/execute/<topic>.

Implementation Details:

/monitor

The /monitor endpoint is implemented using Flask, creating an HTTP server inside the ROS listener node. ROS subscribers continuously receive messages from ROS topics, processed by the ROS Subscriber Class, and stored in the database. The Flask server responds to requests on the /monitor endpoint, providing the current database state in JSON format on port 5000.
We decided to implement the HTTP server and the ROS listener in the same file, as separating the two functionalities into separate files have caused issues. The first approach was to have the listener file send the most recent subscribed data sent to the HTTP server in JSON format.

The server would pick up the sent data and process it into its database, which was a dictionary with keys for each monitorable topic (oximeter, ecg, thermometer, abpd, abps, glucosemeter), and every entry sent by the subscriber the elements. This caused an unknown formatting error, which lead to the HTTP server not being able to depict the information. It is very important to note that the client sends the data with the correct JSON formatting, and the server also receives the JSON data in the correct format, as it could be seen on the print messages.

As a second approach, we tried to keep the dictionary inside the listener, and send the whole data to the HTTP server. Although this approach is less efficient due to the fact that the whole data could grow to be very large based on for how long the machine has been running and collecting information.

Due to the aforementioned problems, we decided to implement the HTTP server and the ROS listener in the same file, as the cause of the trouble seemed to be transferring the data from one file to another. With that change and adaptation of the code, the issues were successfully resolved.

VM had its own issues with Apple silicon chip CPUs and it was really hectic to get it done. Even after getting the cloud image, it was stable enough for us get our work done. Still isn't.

/execute

The execution endpoint initializes an HTTP server on port 5000, listening for incoming HTTP PUT requests to the execute/<topic> endpoints. The publish data function converts data to JSON format and sends an HTTP PUT request to the corresponding endpoint on the local server. This is an example to send a PUT request using curl, containing a JSON payload with the required adaptation:

```
curl -X PUT -H "Content-Type: application/json" -d '{"topic": "/reconfigure_/g3t1_1", "target": "g3t1_1", "action": "freq=0.999"}' localhost:5000/execute
```

Challenges and Experiences:

Several challenges were encountered during the implementation, including updating the CMakeLists.txt file, docker image compatibility issues, and difficulties navigating the ROS documentation. Frequent shutdowns of the Kasm_vu server hosting the VM also posed challenges.

While attempting to implement the http-server with C++, updating the CMakeLists.txt file to include the modifications failed when running the cmake build command.

Numerous challenges were faced during the assignment, primarily related to setting up the environment and working within it. This, in turn, led to more time being allocated to troubleshooting for assignment preparation and less time focusing on the assignment itself.

These problems included issues with downloading the docker container, which would get stuck at 99% multiple times. This was resolved by uninstalling and reinstalling Docker. Despite this, compatibility issues with the docker image persisted. Additionally, frequent shutdowns of the Kasm_vu server hosting the VM made work more interruptive and challenging.

A challenge that remains unresolved is making UPISAS work. The problem initially stemmed from pip not being installed on the machine. It was successfully installed after updating and uninstalling/reinstalling Python. However, when following the UPISAS installation instructions, the process failed because the requirements.txt file attempted to download requests 2.31.0.

Manually installing this package resulted in an error indicating that requests 2.31.0 does not exist. The latest version, 2.27.0, also encountered the same error. To proceed with the assignment, we decided not to invest more time in making UPISAS work.

A minor hurdle was that catkin_make would not work on the docker image, leading us to primarily work on the provided Virtual Machine. Installing rqt_graph on the VM proved troublesome due to invalid GPG keys in the repository. Interestingly, the command rqt_graph worked on the docker image despite these issues.

Work division:

[redacted] - worked on implementation in python

[redacted] - worked on different implementation strategy in cpp(failed during to Cmakelist.txt not updating).

[redacted] - worked on documentation

Link to the forked BSN repository:

[redacted]