Group 5: Report

Mathias Van den Cruijce (r0785409), Jules Verbessem (r0957436), Stijn Hendrix (r0797253), Yurryt Vermeire (r0786618), Arthur Spillebeen ,(r0762529)

Raspberry Pi

The raspberry pi's implementation consists of a python script (see "runHandler.py"), which will start 2 new threads, each of which being responsible for one direction of communication. Before initializing these threads, a secure connection to the MQTT broker is started over TLS with authentication using username/password.

One thread will run the UART_TO_MQTT function, which will read out data from the serial port ttyS0 and validate whether it is correctly formatted (The first 2 bytes represent the group number, the next 2 the voltage and the last 2 the RPM). It will then check if they contain valid values (group number is 5, 0 <= voltage < 3000, 0 <= rpm <= 3000) , after which it will publish a new json formatted message on the "III2024/05/sense" topic on the MQTT broker.

The other thread will run the MQTT_TO_UART function, which upon receiving messages from the MQTT broker on the "III2024/05/control" topic will validate their json formatted content (see if the group number is 5 and check whether the control passed is a valid control name, such as forward/backward/...) . Once validated, it will write the group number (encoded in 1 byte) and the control (mapped to a numerical value in a single byte) to the serial port ttyS0.

MQTT Broker

Our broker is a MQTT mosquito broker, which has been supplied with a self-signed certificate and a couple of different user accounts ("rpi", "frontend" and "tick"). Every connection to this broker is required to be over TLS and authenticated through one of these accounts (through username/password authentication). Each client has a client certificate to connect with, as well as its own account. This provides tamper and eavesdropping prevention as well as authentication.

Frontend

The frontend consists of a flask web server which provides the control UI and endpoints. More specifically, through the UI, POST requests are sent to the "/move" endpoint, which then forwards them to the MQTT broker over MQTT on top of TLS with username/password authentication. Battery updates are sent to the frontend through the "/alert" endpoint, which will propagate them to the browser using websockets.

TICK

InfluxDB process is deployed locally, within its dashboard, we created a bucket, which serves as a NoSQL database. For our project, the bucket is named "robot".

Before running Telegraf, we created a configuration file for InfluxDB. The output and input needs to be configured in this file. The output needs the private token generated by InfluxDB, organization name (ours is "Internet infrastructuur") and the bucket. The input describes how to read the input data that Telegraf receives from the MQTT broker. It needs the servers (["ssl://ip:8883), topics (["III2024/05/sense",]), security items. Telegraf connects to the MQTT broker through MQTT on top of TLS with username/password authentication. See configuration for details.

Chronograf is deployed locally, the dashboard allows you to visualize data stored in a specific bucket of InfluxDB. Visualization is achieved by creating dashboards within Chronograf. For this project, we created a dashboard named "Internet Infrastructuur Dashboard." This dashboard includes two cells, both of which are line graphs: one for RPM and one for voltage.

We created a new alert rule in the Kapacitor. Although Kapacitor successfully reads the data during this step, we encountered an issue where it was unable to send alerts. Due to this problem, we switched to sending alerts directly using InfluxDB tasks instead.

InfluxDB offers its own alert features, which include defining checks, notification endpoints, and notification rules. First, a check is defined to monitor if a specific value from Telegraf exceeds a certain threshold, triggering a state change. The possible states are ok, info, warn, and crit. However, we encountered challenges with this approach. The lack of flexibility in defining states meant we could only send alerts for 4 different percentages. Due to these limitations, we decided to implement an alternative alerting method using InfluxDB tasks.

Our final solution for sending alerts utilizes the task feature of InfluxDB, posting updates every 4 seconds to the endpoint http://127.0.0.1:5000/alert. The task reads voltage values stored in the bucket, selects the most recent value, converts it to a percentage, and sends this as an alert to the endpoint. This custom alert system is implemented using an InfluxDB task.

Embedded

Sensing data format

For sensing, the voltage as well as the RPM of the motors needs to be transmitted. This is done via 6 bytes. The data consists of 3 fields with a length of 2 bytes each. Unint16

was chosen to make it easier to parse and populate buffers as well as to accommodate for the higher values of RPM and voltages (ranges from 0 to 3000).
Dataformat: AA-AA-BB-BB-CC-CC - A Contains the group number, B: Contains the voltage in mV from the level of the capacitors, C: Contains the average RPM from all motors' absolute values in one second.

Freebot controls data format

To control the freeboot, only 2 bytes were needed. These bytes contain the group number and the corresponding action to perform. Uint8 was chosen since this can hold more than enough actions to perform. Dataformat: AA-BB - A: Contains the group number and B: Contains the action needed to perform by the freeboot.

Action mapping (Byte value: action):

- 0: forward
- 1: backward
- 2: right
- 3: left
- 4: side 45°
- 5: side 135°
- 6: side 225°
- 7: side 315°
- 8: clockwise
- 9: counter clockwise
- A: stop

UART

The pins 1.07 and 1.05 are used for receiving and transmitting data on the gateway respectively. The baudrate was also configured to 9600 on both Pi & gateway. The script runHandler.py and the code in central/main.c are the main driving forces for this service.

Bluetooth Low Energy

Freebot peripheral

The freebot contains two threads. The main thread, which is responsible for initializing the freebot, the NUS service, and start advertising over BLE. The freebot only allows connection from the central's address, if the addresses do not match, the connection is closed. It will listen on BLE for messages from the central. The second thread reads the voltage and rpm, averages it, and sends it over BLE using the NUS service to the central in the format found above.

Central

The code of the central gateway, which is mainly based on the sample "BLE UART Central", initializes the UART services to communicate to the pi, and does BLE scanning & GATT discovery for the peripheral. Once a connection is made with the freebot, data is received from the bot every second (rpm and voltage) on BLE via the Nordic UART Service on the RX characteristic and is forwarded to the pi over the UART connection. The central is just a gateway. Both freebot & central contain the code to securely transmit information, however due to some error, this could not be resolved.

MQTT Broker connections