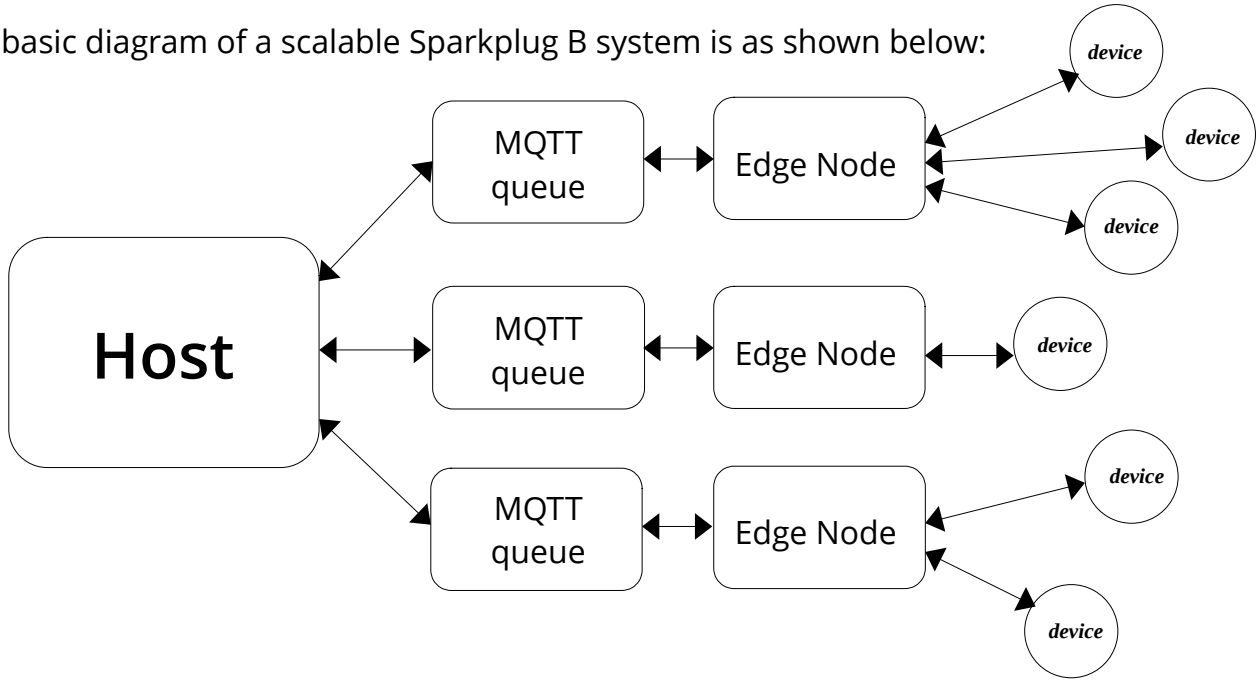
Introduction

Sparkplug B is a communication protocol developed for IOT devices. The protocol itself builds on top of MQTT and Google's Protobuf protocols. Sparkplug uses MQTT to transport the messages between multiple devices in a Sparkplug network. All of the data communicated between the devices, except for a few status messages, are encoded in a few specific Protobuf formats.

The basic diagram of a scalable Sparkplug B system is as shown below:



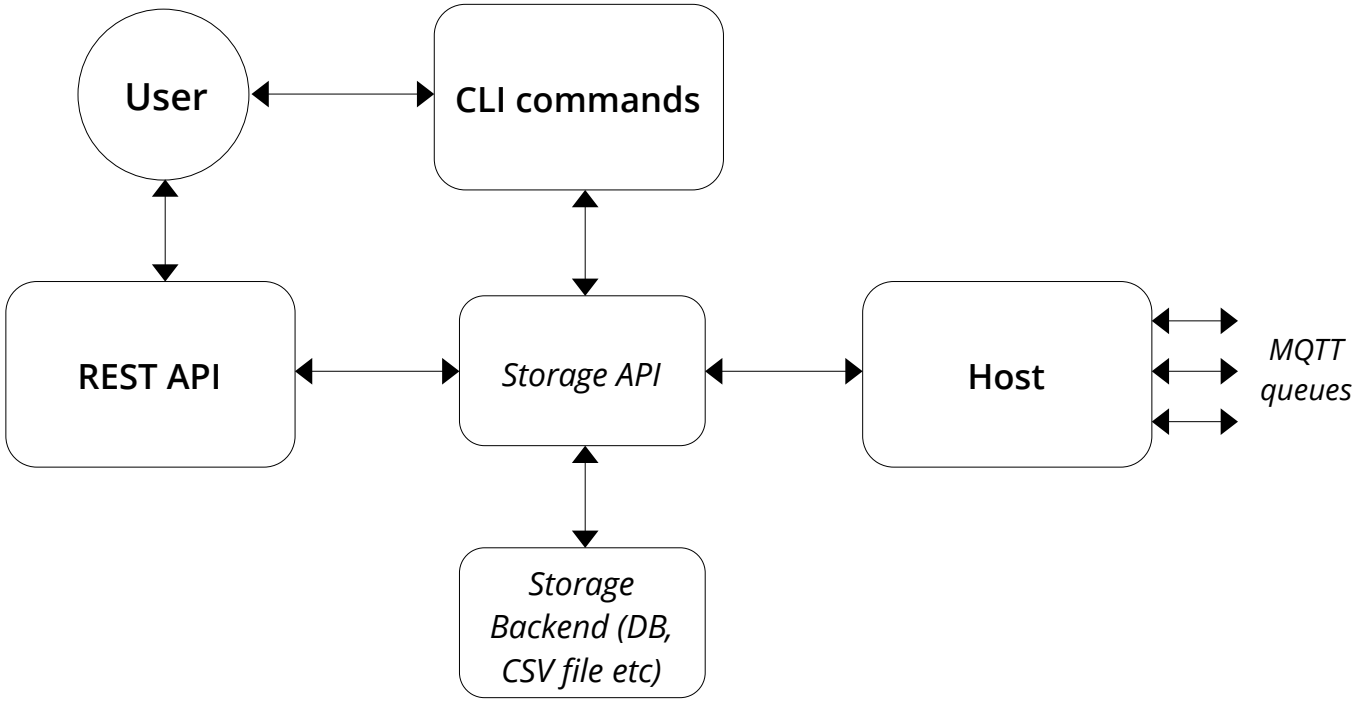
The system consists of a few edge nodes, one or more MQTT queues, and a Sparkplug host. Each of the IOT devices interfaces with one of the edge nodes in the system. It is the job of the edge node to gather raw data from the devices, encode it in the specified Protobuf format, and then put it in the MQTT queue for the host to consume as per the Sparkplug B specification. To ensure a scalable system, there can be multiple MQTT queues where the edge nodes can put their messages on. However, each edge node must connect to a single MQTT queue.

The Sparkplug host consumes the data sent by the edge nodes, and gathers it for further processing by the user. In a Sparkplug environment, the host is the only interface that an user can directly interact with.

The purpose of this challenge was to implement a Sparkplug B compliant host which can collect the data sent by the devices, and provide a few interfaces for an user to query the data. The details of the implementation of the host is specified on the next page.

Implementation

The Sparkplug B host side system implemented in this particular project has the following architecture:



The host system acts a bridge between the MQTT queues and the storage. It listens for various (N/D)BIRTH, (N/D)DEATH, (N/D)DATA publications, and stores the information to the storage backend via the use of the interal storage APIs.

The storage API provides an interface for all other systems to talk to the storage backend, without the knowledge of how the storage is actually implemented. Right now, the storage backend is implemented as a ***sqlite*** DB. The storage APIs implement the actual SQL queries that are needed to be run for any particular action to happen. The rest of the system only calls the functions provided by the API to get their respective data. By implementing the same functions for another backend, the storaging solution can be switched without any change to the host, CLI or the REST modules.

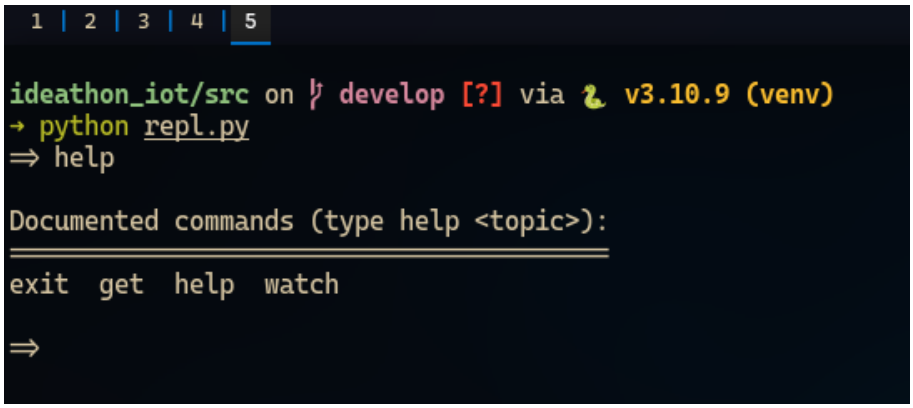
This project provides two ways of interaction to the storage by the user - via the use of a command line REPL (Read – Eval – Print Loop), or via the use of a REST API. All three of the host system, CLI and REST interfaces are independent modules and can be run without any dependency on the other. The next section will show how the interaction using the various interfaces take place in this particular implementation.

Interactions

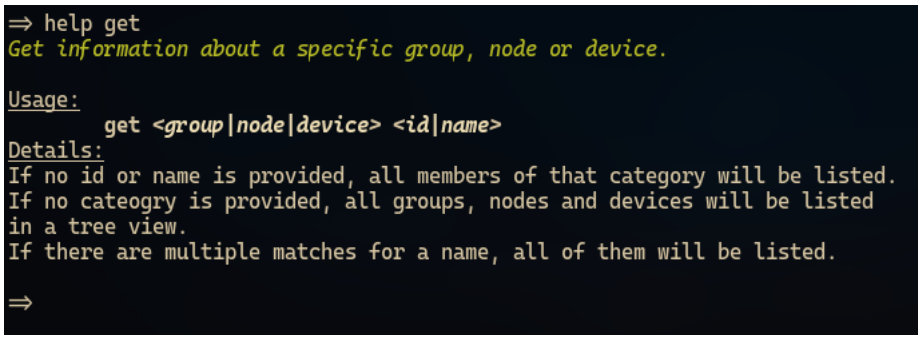
1. Command Line

The system provides a command line interface for querying the data, and in future sending commands to the nodes and/or devices.
Launching the CLI gives an user the following prompt:

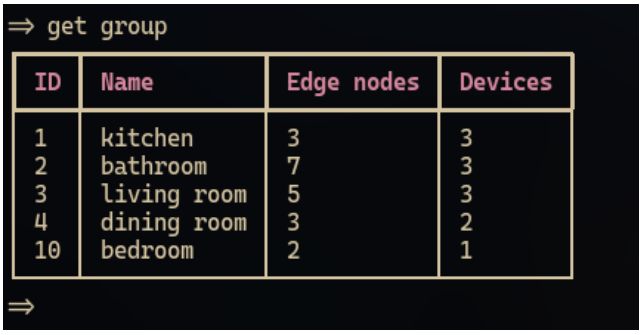
To get a list of commands to run, we start with *help*:



To view help for a particular command, we use *help <command>*. Let's start with *help get*.



As per the instructions, *get* retrieves details for a particular group/node/device. Let's try *get group*.



To get the list of all nodes, we can use *get node*.

```
⇒ get node
```

ID	Name	Group	Devices	Status
1	node0	kitchen	3	ONLINE
2	node1	bathroom	3	ONLINE
3	node2	living room	3	ONLINE
4	node3	dining room	2	ONLINE
5	node4	dining room	3	ONLINE
6	node5	bathroom	1	ONLINE
7	node6	kitchen	3	ONLINE
8	node7	bathroom	2	ONLINE
9	node8	bathroom	3	ONLINE
10	node9	bedroom	1	ONLINE
11	node10	living room	3	ONLINE
12	node11	bathroom	5	ONLINE
13	node12	bathroom	5	ONLINE
14	node13	bathroom	2	ONLINE
15	node14	bedroom	3	ONLINE
16	node15	kitchen	2	ONLINE
17	node16	living room	4	ONLINE
18	node17	living room	2	ONLINE
19	node18	dining room	5	ONLINE
20	node19	living room	2	ONLINE

```
⇒
```

To get the details of a specific device, *get node_name/device_name* can be used like the following.

```
⇒ get device node19/device0
```

ID	Name	Group	Node	Status	Metrics (Name, Value, Last Updated)
56	device0	living room	node19	ONLINE	state 'OFF' 1 second brightness 87 1 second

```
⇒
```

To get the complete topology of the system, we can run *get* without any arguments:

```
1 | 2 | 3 | 4 | 5                                                    8%  64%
⇒ get
kitchen
├─ node0
│  ├─ device0 (id=1)
│  │  ├─ temperature (type=float, value=74.36222839355469, timestamp=1678094930)
│  │  └─ unit (type=string, value=F, timestamp=1678094930)
│  ├─ device1 (id=2)
│  │  ├─ state (type=string, value=ON, timestamp=1678094930)
│  │  └─ speed (type=int, value=61, timestamp=1678094930)
│  └─ device2 (id=3)
│     └─ motion (type=boolean, value=0, timestamp=1678094930)
├─ node6
│  ├─ device0 (id=16)
│  │  └─ humidity (type=int, value=45, timestamp=1678094930)
│  ├─ device1 (id=17)
│  │  ├─ state (type=string, value=OFF, timestamp=1678094930)
│  │  └─ brightness (type=int, value=50, timestamp=1678094930)
│  └─ device2 (id=18)
│     └─ motion (type=boolean, value=0, timestamp=1678094930)
└─ node15
   ├─ device0 (id=43)
```

The list will contain all the groups, nodes and devices in the system in a tree view.
As per the target of this challenge, one other command is implemented in this REPL, named *watch*.

```
⇒ help watch
Watch a specific device for live changes.

Usage:
        watch <device_name>
Details:
If multiple devices match the name, all of them will be watched.
Press Ctrl+C to exit the live view.

⇒
```

Running *watch node19/device0* gives the following display:

```
⇒ watch node19/device0
```

ID	Name	Group	Node	Status	Metrics (Name, Value, Last Updated)
56	device0	living room	node19	ONLINE	state 'OFF' 2 seconds brightness 92 2 seconds

The view gets automatically updated after 1 second, notice the *Metrics* column.

```
⇒ watch node19/device0
```

ID	Name	Group	Node	Status	Metrics (Name, Value, Last Updated)
56	device0	living room	node19	ONLINE	state 'ON' 2 seconds brightness 47 2 seconds

For the purpose of a minimum viable product, only these two commands are implemented. But the CLI can be easily extended to add new commands, and can even have commands that processes data on the fly, using a Python like scripting language.

Another way of interacting to the system is via the REST APIs, which will be shown in the next section.

Interactions

2. REST API

The system provides a REST API based on FastAPI which provides several endpoints for querying data.
The server can be started like the following, by using *uvicorn* as an example,

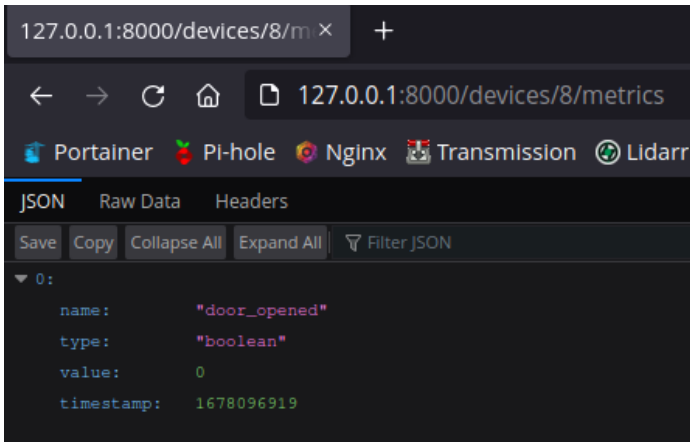
```
ideathon_iot/src on  develop [?] via  v3.10.9 (venv) took 51m29s
→ uvicorn api:app --reload
INFO:     Will watch for changes in these directories: ['/mnt/Programs/TCS/ideathon_iot/src']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [47486] using StatReload
INFO:     Started server process [47488]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

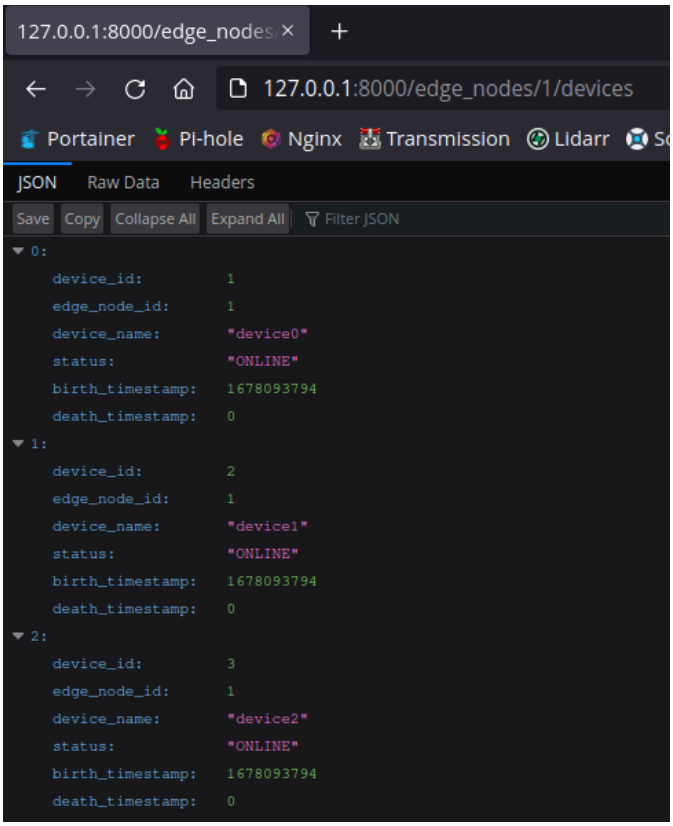
The following endpoints are provided by the API:

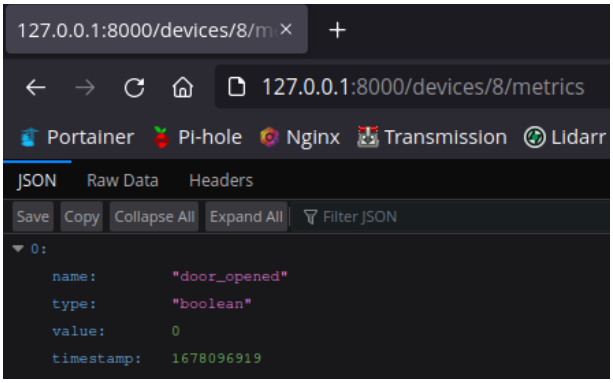
GET	`/groups`	Get Groups
GET	`/edge_nodes`	Get Edge Nodes
GET	`/devices`	Get Devices
GET	`/groups/{group_id}`	Get Group By Id
GET	`/edge_nodes/{edge_node_id}`	Get Edge Node By Id
GET	`/devices/{device_id}`	Get Device By Id
GET	`/groups/{group_id}/devices`	Get Devices By Group
GET	`/edge_nodes/{edge_node_id}/devices`	Get Devices By Edge Node
GET	`/groups/{group_id}/edge_nodes`	Get Edge Nodes By Group
GET	`/devices/{device_id}/metrics`	Get Metrics By Device

The REST API can be extended indepedently, and can also have methods to send messages to the devices. The proposed scripting language can also be exposed via the REST API, where the processing can be done on the server side, in case of a large, scalable system.

Example of the outputs given by the API endpoints are shown below:







All of the endpoints return JSON responses in the *key:value* format.

Configuration

The application can be configured with a configuration file, which is read by the all the systems in the architecture. The configuration file looks like the following right now:

```
ideathon_iot/src on  develop [!?] via  v3.10.9 (venv)
→ cat config.json
```

File: config.json

```
{
    "id": "sparkplugHost1",
    "mqtt": {
        "host": "localhost",
        "port": 1883,
        "username": "mqtt",
        "password": "mqtt"
    },
    "reordering_timeout": 1000,
    "zones": [
        "bedroom",
        "kitchen",
        "living room",
        "bathroom",
        "dining room"
    ],
    "db": {
        "type": "sqlite",
        "url": "sparkplug.db",
        "username": "sparkplug",
        "password": "sparkplug"
    }
}
```

Right now, the MQTT configuration, zone/group list, and the DB connection details can be specified via the configuration file. In future, it can be extended to control the behaviour of the REST API, or the output of the CLI interface.

Conclusion

The source code for the application can be found in the following GitHub link:
IdeathonIOT on GitHub

It contains a readme which details how to setup and start the system. It also contains a client program which simulates a vast collection of IOT devices communicating over the Sparkplug B protocol.

This has been a great learning experience, and I want to thank Mr. Prateep Misra for giving me the oppurtunity to explore the world of IOT devices.