E-AGLE Trento
Racing Team

Telemetry

Luca Martinelli
luca.martinelli@studenti.unitn.it

Ivan Martini
ivan.martini@studenti.unitn.it

Eugenio Berretta
eugenio.berretta@studenti.unitn.it

Academic year 2019/2020

Contents

1 Concept

After managing to build an electric car we needed a telemetry. It was essential to have something that joined all the sensors and gave us an easy and unique way to understand what happened in the car. As almost every team, we could have bought an already-made solution and plugged it to the canbus. But that was not the philosophy of ours. We built a telemetry from scratch, projecting and programming it.

1.1 Requirements

All the sensors, the steering wheel and the ecu are plugged to the canbus. They send and receive all the messages through that. The simplest debugging method could be done is save a can log, task that is actually currently done by the steering wheel by using can-utils (https://github.com/linux-can/can-utils). But having a raw can log is of course not enough and it is also uncomfortable. So we decided to create a telemetry that read all the messages from the canbus and parsed them in a more readable and handable way. After some time, when we added a new base-rover GPS, we decided to plug the rover GPS directly on the telemetry and read its messages through its serial port.

1.2 Purpose

The purpose of the telemetry is making the teams know in a comfortable and easy way what happened in the car. It joins all the messages from the canbus and the GPS in one place and handles them in a desidered way. It stores the data in a local and easily exportable database and forwards the data via mqtt so that some pseudo-real-time applications can be ran. The data can then be managed as wanted, mostly used by the analysis team.

2 Decision

2.1 The raspberry

The telemetry needed to read the messages of the can, to have enough computation power to parsing all of them, saving them and sending them via mqtt and it needed also enough memory to store large amounts of data.

2.2 The operative system

All the members of the team use Linux. Raspberry runs almost always with Linux. All the libraries and programs that we wanted to use ran on Linux. Using Linux for the telemetry was certainly not a difficult choice. We decided though not to use Raspbian but to use Ubuntu (arm) instead. This is because Ubuntu's support for what we needed was greater and most of the members of the team use it in their own pc. Using Ubuntu was also a good idea in order to be able to develope locally in our pc's the telemetry and having not too much truble in porting it to the telemetry

2.3 The programming language

We used C as the programming language of the telemetry because we needed the best performances that we could have achieved. The telemetry needed to read all the can messages, parse most of them and store them in a data structure. Then, every a certain amount of time, it would have discarded the data structure before saving the data in the database and created a new one. Only with a low-level language we managed to reach the performances we needed. We did not use C++ because we thought that C provided us all what we needed. Furthermore, without using classes of more "high-level" libraries, we were more sure that we would have not lost performances.

2.4 The database

We used mongodb as database. A big part of the team members knew it and this is one of the reasons because we chose mongodb, among the fact that it has good performances. But the main reason for our choice was that it is flexible and versatile. Using a relational database, we would have been forced to choose a static schema for the data and changing it in the future would have been a hell. The structure would have surely changed because it was probable that new sensors or messages would have been added after the developement of the telemetry. With mongodb this problem just disappeard.

2.5 The sending protocol

We wanted be able to display the sensors data also while the car was running. We chose to use the mqtt protocol because it was one of the most popular and easy to implement. With mqtt we can have a pseudo-realtime and show data for instance in a web frontend that uses mqtt over websocket while the car is running.

2.6 The activation

During the first tests we noticed two things. The first is that it was very annoying to connect every time via ssh to the raspberry in order to start or stop the telemetry. The second was that for big amounts of time the car was still and there was no reason to add rumor and megabytes for unneeded data. Our solution was making the telemetry comunicating with te steering wheel via the canbus. The steering wheel was able to start and stop the telemetry, while the telemetry responded with its status, that was displayed on the wheel monitor. The telemetry is a program that is always running, but the steering wheel can make it idle or active. When it is idle it saves nothing in the database, but continues to send the data via mqtt because it does not occupy space.

2.7 WIFI and autorun

To send data via mqtt and be reachable via ssh, the telemetry needs to be connected to a net. We have bought a soap of the Huawei and we use it as a portable WIFI hotspot that stays always in the car. The raspberry automatically connects to the soap on boot-time. To avoid loosing time connecting via ssh and starting the telemetry program, we used systemctl to authomatize it and make it a service that is always running.

2.8 The exporter

Even if the telemetry was an always-running service, we needed to connecto to it via ssh every time we needed to export the data. And it was a very long and tedious thing. We started a side project that eventually evolved in a webapp exporter, with a Vue.js frontend and a nodejs backend that is served by the raspberry itself. The served backend is also an always-running service and the only task needed to connect to it is to connect to the Huawei soap and open a browser on the telemetry IP.

2.9 Timestamp

The telemetry saves the timestamp of the messages when they are received and also the timestamp of the BSON document when it is saved. We wanted to have the epoch unix timestamp in milliseconds. The problem was that the soap had (and still does not have) a sim to connect to internet. Connecting to ssh and updating the data-time every time we turned on the car was a hell, so we decided to develope a provvisory solution. The solution is another always running service, it is a nodejs service that listens to the mqtt and when it receives a particular message, it changes the date-time with the one provided. This temporary solution was also used to simulate the steering wheel when it was not still ready to send the start and idle signals to the telemetry. More information and the code can be found in this repo: https://github.com/eagletrt/eagletrt-telemetria-controller

2.10 The sessions

In order to have more meaningful data, we decided to group it into sessions. A session is a period of time when the car is driven always by the same pilot, doing always the same type of race in the same ciruit, stopping only for short amounts of time. This gives us more organized and handable data and helps a lot the analysis team. A session has as references the pilot, the type of race and the timestamp when the telemetry started saving the data. When the steering wheel actives the telemetry, it specifies also the pilot and the type of race, that are selected by the pilot.

2.11 The developement envinronment

We did not write the telemetry directly in the raspberry. That is also why we chose Ubuntu arm as the operative system of the raspberry, because it was the same (but arm) of the one in our machines. To emulate the canbus we mout a virtual can in our pc and to emulate the can messages we use canplayer (https://github.com/linux-can/can-utils) with a real can log of the machine. Since the gps is connected with a serial port, we managed to write a simulator that fakes that serial port and takes an ubx gps log as input. We made the simulator as a side-project and this is the repo (https://github.com/FilippoGas/eagletrt-telemetry-simulator).

2.12 The compilation

We do not cross compile the code of the telemetry. It does not change so frequently and we just put the new code (or clone from github) and compile it directly in the raspberry through ssh. By doing this, we can also modify the code during tests and recompile it if needed.

2.13 The configuration

2.13.1 Purpose

We made a json file config that is read when the telemetry starts. So if we have to change for instance the mongodb uri, we do not have to recompile the code, but just to modify the json file and to restart the program.

2.13.2 Details

The config resides in the config.json file. The options are:

- mqtt: The options regarding the mqtt connection
  - host: The host of the mqtt connection
  - port: The port of the mqtt connection
  - data_topic: The topic where the bson data is sent
  - log_topic: The topic where the telemetry log is sent
- mongodb: The options regarding the mongodb connection
  - host: The host of the mongodb connection
  - port: The port of the mongodb connection
  - db: The db where the data will be saved
  - collection: The collection where the data will be saved
- gps: The options regarding the rover gps plugged to the telemetry
  - plugged: If the gps is plugged to the telemetry. If the gps is not plugged but this option is set to 1, the telemetry will try to read the gps serial port and will crush.
  - simulated: If the gps is simulated and not real.

  - interface: The interface name of the gps serial port

- pilots: The array containing the possible pilots who drive the car. Every time the steering wheel enables the telemetry, it specifies also the index of the pilot who drives the car. The pilots name will be added to the current session, to the data saved in the database
- races: The array containing the possible races that the car can perform. Every time the steering wheel enables the telemetry, it specifies also the index of the race that the car is performing. The race name will be added to the current session, to the data saved in the database
- circuit: The array containing the possible circuits where the car is running. Every time the steering wheel enables the telemetry, it specifies also the index of the circuit where the car is running. The circuit name will be added to the current session, to the data saved in the database. NB: currently not implemented
- can_interface: The interface of the canbus
- sending_rate: The number of milliseconds every which the telemetry saves the accumulated data, before emptying it and repeating the cycle
- verbose: If also the debug messages will be logged in the console

2.13.3 Example

This is an example of config.json

```
{
  "mqtt": {
    "host": "localhost",
    "port": 1883,
    "data\_topic": "telemetria",
    "log\_topic": "telemetria\_log"
  },

  "mongodb": {
    "host": "localhost",
    "port": 27017,
    "db": "eagle\_test",
    "collection": "scimmera"
  },

  "gps": {
    "plugged": 1,
    "simulated": 1,
    "interface": "/dev/pts/4"
  },

  "pilots": [
    "default",
```

```
    "Ivan",
    "Filippo",
    "Mirco",
    "Nicola",
    "Davide"
  ],

  "races": [
      "default",
      "Autocross",
      "Skidpad",
      "Endurance",
      "Acceleration"
  ],

  "circuits": [
    "default",
    "Vadena",
    "Varano",
    "Povo"
  ],

  "can\_interface": "can0",
  "sending\_rate": 500,
  "verbose": 0
}
```

2.14 The cycle

The telemetry accumulates all the messages in a data structure and saves it every a certain amount of milliseconds. This way the data is more standardized and by grouping it we limit the overhead of the BSON format and decrease the size of the saved and sent data.

2.15 The data structure

A big part of the code is related to the telemetry data structure. It is the model of the data, that specifies what and how is stored in the database. Here are written all the considered messages, the type of their values and all the rest.

2.15.1 The C struct

This things have to be put in a C variable in some way, wo there is a struct that reflects that model and has to be changed every time the data structure changes, for instance when new messages are added.

2.15.2 The structure.json

There is a json file, the structure.json file, that is a sort of ”manifest” file and reflects the telemetry data structure. Every time something changes in the structure it is updated.

2.15.3 The structure of the structure

The structure of the gathered data is saved in the structure.json file and is based on the possible messages:

- The structure is a json object, but every primitive key contains the value type instead of the value itself
- For each message there is an array of objects
- Each array contains only two elements: the message object and the message max count
- Each message object represents the model of the parsed message. It has the timestamp of the receive time and a value containing the value of the message (a value or an object of values)
- Each message max count represents the maximum number of messages that can be saved in a single document
- Each array can be saved directly in the "root" of the structure object, or nested in other objects, in order better to organize the structure. For instance, all the bms message arrays are nested in the bms object
- There is an id key, containing a progressive id of the document starting by 1
- There is a timestamp key, containing the timestamp when the document is saved in the db, or sent via mqtt
- There is a sessionName key, that refers to the session of the document itself

Each time the telemetry enters in the ENABLED state, the id number is reset and a new session is created. A session is based on the timestamp when the telemetry was enabled and the parameters (pilot, race) passed by the steering wheel. A session document containing these parameters is created and inserted in the database.

2.15.4 Example of structure.json

```
{
    "id": "int",
    "timestamp": "long",
    "sessionName": "char*",
    "throttle": [
        {
            "timestamp": "long",
            "value": "double"
        }, 200
    ],
    "brake": [
        {
            "timestamp": "long",
            "value": "double"
```

```
        }, 200
    ],
    "bms\_hv": {
        "temperature": [
            {
                "timestamp": "long",
                "value": {
                    "max": "double",
                    "min": "double",
                    "average": "double"
                }
            }, 500
        ],
        "voltage": [
            {
                "timestamp": "long",
                "value": {
                    "max": "double",
                    "min": "double",
                    "total": "double"
                }
            }, 500
        ]
    }
}
```

2.15.5 Example of session document

This is an example of the session document:

```
{
    "sessionName": "20200309\_175011\_default\_default",
    "timestamp": 1583772611,
    "formatted\_timestamp": "20200309\_175011",
    "pilot": "default",
    "race": "default"
}
```

2.15.6 Example of data document

This is an example of the data document:

```
{
    "id": 23,
    "timestamp": 10483862400000,
    "sessionName": "2020\_04\_23\_\_12\_00\_00\_\_pilot\_race",
    "throttle": [
        {
```

```
            "timestamp": 10483862400001,
            "value": 0
        },
        {
            "timestamp": 10483862400002,
            "value": 5
        },
        {
            "timestamp": 10483862400003,
            "value": 6
        }
    ],
    "brake": [
         {
            "timestamp": 10483862400001,
            "value": 0
        },
        {
            "timestamp": 10483862400004,
            "value": 100
        }
    ],
    "bms\_hv": {
        "temperature": [
            {
                "timestamp": 10483862400000,
                "value": {
                    "max": 28,
                    "min": 22,
                    "average": 25
                }
            }
        ],
        "voltage": [
            {
                "timestamp": 10483862400000,
                "value": {
                    "max": 312,
                    "min": 200,
                    "total": 250
                }
            },
            {
                "timestamp": 10483862400007,
                "value": {
                    "max": 312,
                    "min": 200,
```

```
                "total": 250
            }
        }
    ]
  }
}
```

2.16 The code generation

2.16.1 Mantainibility problem

Here comes a big problem. C is a statically typed language. This means that it is not suited to handle json object like the structure. A structure like the one in the example above, but much longer in the reality, should be used in C code. The result is a structure definition of 350 lines of code, an instance allocation of 50 lines of code, a bson conversion of 400 lines of code and an instance deallocation of 30 lines of code. Taking in account that the structure changed frequently, because the new sensors or can messages were added, mantaining and keeping updated that code was worse than the hell. If only a message changed, we would have to change four different parts of the code and some of them were between 400 lines of code that seem always the same.

2.16.2 The solution

Here is the purpose of the structure.json. We developed a Typescript program that takes as in put the structure.json and generates for us the part of the code that depends on the structure. If the structure changes, instead of changing some C code in four different parts finding the right line between thousands, we only need to change the structure.json and run a program that does the task for us. The task that before occupied the 85% of our time can be now achived in a matter of seconds.

2.17 The mqtt log

During debugging we could simply look at our terminal to see the logs and to understand what was happening during the execution of the telemetry. During the tests, these logs are still available using journalctl, being the telemetry a service. The problem is that this is uncomfortable during the execution, hence we decided that part of the log will be sent via mqtt on a different topic from the one of the data, so that we can see the logs from another device.

2.18 The states

2.18.1 List of the states

The behaviour of the telemetry can be described with four states:

- INIT: When the telemetry is starting, it reads the config file and sets up everything (canbus interface, gps serial port, mqtt connection, mongodb connection). After being executed, it will pass to the IDLE state.
- IDLE: The telemetry is actually running. There are two threads that read all the can messages and gps messages and save them in a structure. Every 500ms this structure is converted to bson and sent via mqtt. After being executed, it will repeat itself unless a can message to enable the telemetry is received.

- ENABLED: The telemetry does the same thing of the IDLE state, but it also saves the data in mongodb. After being executed, it will repeat itself unless a can message to disable the telemetry is received.
- EXIT: The telemetry tries to gently deallocate all the data and close all connections before exiting. It is usually reached when an error occurs.

2.18.2 Purpose of the states

Representing the telemetry as a state machine is a powerful strategy to start to implement it in a low-level language such as C. Our C implementation of the state machine changed from a simple switch to a matrix of transition functions and a unique global variable called condition.

3 Implementation

The code of the telemetry is totally implemented in C. There are some json files for the configuration and the code generator and an sh file for the compilation.

3.1 Structure

The project has these files:

- package.json: It can seem strange to find a package.json file in a project totally written in C. We added it because it contains the dependency of the eagletrt-code-generator (https://www.npmjs.com/pac code-generator). Since we added it, we added also some npm scripts that organized the bash commands that we otherwise would have to run
- structure.json: It defines the telemetry data structure and is the input provided to the code generator
- config.json: It contains the configuration of the program, it can be changed without recompiling the program and has already been described
- compile.sh: It is a bash script that compiles the program
- idle_signal.sh and enable_signal.sh: These are used only during developement, to simulate the steering wheel can messages that start and stop the telemetry
- main.c: It is the root of the program
- /state_machine: In this folder there is the code reguarding the state machine
- /services: In this folder there are modules of C code divided by type of service (mongodb, mqtt, ...) that are used by the state function of the state machine
- /utils: In this folder there are modules of C code that provide more general purpose functions

3.2 The npm scripts

The npm scripts simplify the life of who uses the project. They are:

- "transpile": The executed command is "npx eagle generate" and it executes the code generator
- "compile": The executed command is "./compile.sh" and it compiles the C code
- "start": The executed command is "sudo ./sender.out config.json" and it starts the program
- "serve": The executed command is "npm run compile && npm run start" and it compiles and start the program
- "enable": The executed command is "./enable_signal.sh" and it simulates (debug during developement) the steering wheel can message that starts the telemetry
- "idle": The executed command is "./idle_signal.sh" and it simulates (debug during developement) the steering wheel can messages that stops the telemetry

3.3 The state machine

The code is based on a state machine implemented with a matrix of transition functions. There is a unique mega-global variable called "condition", that is indeed the condition of the state machine. Initially the code was full of global variables, but that made it very messy. Removing completely the global variables would have put a lot of overhead in the functions parameters and creating the condition variable seemed to be the right way to organize the code.

3.3.1 The states

The states are:

- INIT: When the telemetry is starting, it reads the config file and sets up everything (canbus interface, gps serial port, mqtt connection, mongodb connection). After being executed, it will pass to the IDLE state.
- IDLE: The telemetry is actually running. There are two threads that read all the can messages and gps messages and save them in a structure. Every 500ms this structure is converted to bson and sent via mqtt. After being executed, it will repeat itself unless a can message to enable the telemetry is received.
- ENABLED: The telemetry does the same thing of the IDLE state, but it also saves the data in mongodb. After being executed, it will repeat itself unless a can message to disable the telemetry is received.
- EXIT: The telemetry tries to gently deallocate all the data and close all connections before exiting. It is usually reached when an error occurs.

3.3.2 The c code organization

The C code organization is divided in four parts:

- main.c: It is the root of the project and simply runs the state machine
- /state_machine: It is the state machine and contains its definition of the state machine and the functions for each state. It uses as dependencies the /utils and /services libraries. Also, it provides to them the definition of the condition struct.
- /services: It contains various modules of C code, separated by purpose (mongodb, mqtt, ...). It has in the header the extern of the condition struct, so its function are aware of that variable and use it.
- /utils: It contains various modules of C code, separated by purpose (mongodb, mqtt, ...). It has not in the header the extern of the condition struct, so its function are not aware of that variable and do not use it. These are functions mostly used by the /services libraries and are more general purpose.

4 Usage

4.1 Prerequisites to run on local machine

1. Use a Linux operative system
2. Install gcc to compile c programs, on ubuntu sudo apt install build-essential
3. Install nodejs to execute js scripts, on ubuntu follow this post (https://linuxize.com/post/how-to-install-node-js-on-ubuntu-18.04/)
4. Install mosquitto to host an mqtt broker, sudo apt install mosquitto && sudo apt install mosquitto-clients
5. Install canutils to connect to canbus, sudo apt install can-utils
6. Install mongodb to have the local database, on ubuntu follow this guid (https://docs.mongodb.com/man mongodb-on-ubuntu/)
7. Install mongodriver for c to use mongodb from c, on ubuntu sudo apt install libmongoc-dev && sudo apt install libbson-dev
8. Install mqttdriver for c to use mqtt from c, on ubuntu sudo apt install libmosquitto-dev

4.2 Before running the telemetry

1. Make sure mongod service is active (check that the command "mongo" works)
2. Simulate the canbus and the gps serial port (check this repo https://github.com/FilippoGas/eagletrt-telemetry-simulator) This is needed when debugging the telemetry in a local pc

4.3 To run the telemetry

1. Clone this repo https://github.com/eagletrt/fenice-telemetria-sender
2. Execute npm i to install the nodejs dependencies
3. Execute npm run transpile or npx eagle generate to execute the js script and generate the c code
4. Execute npm run compile or ./compile.sh to compile the c code
5. Execute npm run start or sudo ./sender.out config.json to start the telemetry (executing npm run serve does all the last three points with an only command)

4.4 To change the telemetry status

1. Execute npm run enable or ./enable.sh to enable the telemetry and make it saving data on the db
2. Execute npm run idle or ./idle.sh to disable the telemetry and make it stopping saving data on the db This is usually useful when debbugging the application on a local computer, because simulates what does the steering wheel of the car

4.5 To check everything is going well

1. Before starting the telemetry execute mosquitto_sub -t telemetria_log. It should show the log of the telemetry.

2. Execute mosquitto_sub -t telemetria, it should show the data sent by the telemetry via mqtt (it is sent even when the telemetry is idle).

3. Open mongo compass and check the data is saved on mogodb

5 Versions

Many changes have been done to the telemetry.

5.1 Mongodb organisation

5.1.1 What is a session

Every time there was a test or race, multiple "sessions" were saved. A session is a period of time were the car is driven by the same pilot, doing the same type of test/race, in the same circuit, stopping only for short amounts of time.

5.1.2 First mongodb organisation

Initially, before changing structure, the mongodb was organized in different databases, such as one for each car (chimera and fenice). Each collection in a database corresponded to a session. The name of a collection was decided dinamically by the telemetry and consisted in the name of the pilot, the type of race and the time when the session started. Inside the collection there were only the data-documents of the session it corresponded.

5.1.3 Current mongodb organisation

At some point the structure of the mongodb database changed and we needed to devlope version 2.0 of the exporter. The current mongodb database structure has only a fixed database "ealetrt". It contains various collections with arbitrary names. The collections have the same functions that the databases of the old structure had. They are currently four (chimera, fenice, chimera_test, fenice_test). Each collection contains two type of documents. The session-documents describe a session with pilot name, type of race, timestamp and a key. The data-documents are the same of the documents in the old structure, with the difference that now they contains a reference to the key of their session document. So, now, a collection corresponds no more to a single session, but can contain more than one.

5.2 Code organisation

5.2.1 During the first test

During the first test on the machine, the code pieces (can, mongodb, mqtt, messages parsing) were mostly done. The problem was that the code was mostly in an unique main.c file and was very difficult to understand and mantain. For a bug that was difficult to find in the main.c we managed to write a javascript version in a day and used it during the very first telemetry test. (Of course it was nowhere near as performant as c)

5.2.2 The first reorganisation

After the first test, we begun to split the c code in more modules. The biggest problem was the c code related to the telemetry data structure.

5.2.3 The code generator

We came up with the javascript code generator and used it to generate the c code related to the telemetry data structure

5.2.4 Another reorganisation

Before another test we reorganised the code again. We made the state machine as it is now, implemented with a matrix of transition functions (before it was only a switch). We added also the /utils and the /services folders.

5.2.5 Branching the generator

After seeing the generator was stable and we did not need to modify it frequently, we decided to rewrite it totally in Typescript and in a clearer code and to publish it as an npm module.

5.2.6 GPS

We added the new GPS with the reading of its serial port and the addition of its messages to the structure.

5.2.7 Threads

During the last test, we noticed that the addition of the GPS made the telemetry too slow. Only when the car was in run state, the number of can messages increased significantly and added to the GPS serial port reading overhead it started loose messages and data. We solved this problem by making two threads: one that read the canbus and another that read the GPS serial port.

6 Conclusion

6.1 Summary

Instead of buying a pre-made telemetry we managed to write one from scratch.

6.2 Future

There are still many changes that are coming for the telemetry. With the new car, fenice, the structure of the can messages will totally change. Firstly, there will be two canbus. So the telemetry will have to support two of them. Then, the can messages id's will be generated automatically and we have already the plan (and a prototype) of the c code generator that will generate automatically also the parsing c code of the messages. Last but not least, an idea could be the one of keeping only a unique telemetry for fenice and the old chimera, making it retrocompatible by properly modifying the config.json.