Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Bernardo Coutinho Barreiros de Freitas

SNMP Agent for On-Board-Units in Vehicular Systems

February 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Bernardo Coutinho Barreiros de Freitas

SNMP Agent for On-Board-Units in Vehicular Systems

Master dissertation
Master Degree in Informatics Engineering

Dissertation supervised by
Bruno Alexandre Fernandes Dias

February 2022

AUTHOR COPYRIGHTS AND TERMS OF USAGE BY THIRD PARTIES

This is an academic work which can be utilized by third parties given that the rules and good practices internationally accepted, regarding author copyrights and related copyrights.

Therefore, the present work can be utilized according to the terms provided in the license bellow.

If the user needs permission to use the work in conditions not foreseen by the licensing indicated, the user should contact the author, through the RepositóriUM of University of Minho.

License provided to the users of this work

Attribution-NonCommercial
CC BY-NC
https://creativecommons.org/licenses/by-nc/4.0/

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

João Bernardo Coutinho Barreiros de Freitas

Assinado por: **JOÃO BERNARDO COUTINHO BARREIROS DE FREITAS**
Num. de Identificação: 15021949
Data: 2022.02.14 13:11:43+00'00'

ACKNOWLEDGEMENTS

Com o fim deste trabalho a minha jornada académica chega ao fim e como tal é um bom momento para refletir e agradecer a toda as pessoas que nestes anos me ajudaram, especialmente no último ano, ano em que desenvolvi o protótipo que irá ser apresentado nesta dissertação.

Porque o desenvolvimento e redação da presente dissertação seria completamente impossível sem o apoio e a ajuda prestados por várias pessoas, desde já, um enorme e especial obrigado. No entanto, antes de começar a exposição, cumpre agora individualizar todos aqueles que contribuíram, de forma notória, para a realização do presente desenvolvimento, fazendo com que este caminho não fosse trilhado solitariamente.

Primeiramente, aos meus pais, Anabela Freitas e Pedro Freitas, e irmão, José Pedro Freitas, por toda a ajuda, paciência, apoio, conforto e carinho incondicional, seria injusto da minha parte não prestar o maior dos agradecimentos.

Em segundo lugar, gostava de agradecer ao resto da minha família e amigos, por serem de e para a vida, que direta ou indiretamente contribuíram para a realização deste trabalho e por todos os bons momentos.

Em terceiro, gostaria de deixar um reconhecimento público de agradecimento ao Professor Doutor Bruno Alexandre Dias pela persistência, rigor, exigência e disponibilidade em ajudar e orientar a presente dissertação, desde o primeiro dia. Sem este contributo este trabalho seria completamente impossível.

Por fim queria agradecer ao resto da equipa docente da escola de Engenharia da Universidade do Minho por terem feito desta jornada na Universidade do Minho os melhores anos da minha vida.

ABSTRACT

On average over 60 Million automobiles are sold every year in the whole world and at one point or another every single one of these vehicles will require some form of maintenance to be performed.

With the ever increasing complexity of these vehicles, any maintenance job has also increased in its difficulty and time required to complete, as such there is a need for a set of fast and reliable diagnostic tools to speed up this process.

Furthermore, with the ever closer introduction of Vehicular ad hoc networks (*VANET*s), there is a need for an application that is able to read sensor data in real time and change the state of actuators in a vehicle with minimum delay, allowing for the introduction of such methods like platooning, which require several vehicles of different types and models to accelerate or brake simultaneously while also allowing a closer headway between vehicles, since the reaction time of such a system would be entirely based on the latency of the communication method/protocol being used and not the capabilities of the human driving the vehicle.

As such. the main objective of this project is to create and test a Management Information Base (*MIB*) specification to be implemented on an Simple Network Management Protocol (*SNMP*) agent inside an On-Board-Unit (*OBU*) that allows a company or individual to quickly and safely access all information gathered from the vehicles' own sensors while also allowing for its configuration and, at the same time, managing errors in the system.

This system will make use of the preexisting Controller Area Network (*CAN*) technology to access and gather data from a vehicles sensors so that it can be accessed in real-time through an application. Such an application will communicate with the vehicles *OBU* using *SNMP*. This solution should be capable of handling more requests for data than already existing standard technologies and protocols, such as On Board Diagnostics (*OBD-II*), while also being faster than them. Additionally a way for users or other entities in a *VANET* to activate/deactivate specific actuators should also be included in this solution as such a feature is vital to the introduction of methods like platooning.

Keywords: VANET, CAN, MIB, SNMP, On-Board-Unit, SNMP Agent, configuration, error management, Platooning, OBD-II, ...

RESUMO

Em média são vendidos mais de 60 milhões de automóveis por ano em todo mundo e, eventualmente, todos eles irão necessitar de manutenção.

Com a complexidade destes veículos sempre a aumentar, qualquer trabalho de manutenção tem aumentado na sua dificuldade como no tempo despendido e. como tal, há uma necessidade de um conjunto de ferramentas de diagnóstico que sejam rápidas e de confiança para acelerar este processo.

Ao mesmo tempo, com o aumento de investimento e desenvolvimento de Vehicular ad hoc networks (*VANET*s), há necessidade de uma aplicação que permita a leitura de sensores em tempo real e mudança de estado de atuadores de um veículo com delay mais baixo possível, permitindo a introdução de métodos de condução como platooning, que obriga a que vários veículos de diferentes marcas e modelos acelerem e travem ao mesmo tempo o que permite a que a distância entre eles baixe visto que o tempo de reação de tal sistema é baseado na latência do método/protocolo de comunicação e não nas capacidades do condutor.

O principal objectivo deste projeto é criar e testar uma especificação Management Information Base (*MIB*) para ser implementada num agente (*SNMP*) Simple Network Management Protocol dentro de uma On-Board-Unit (*OBU*) que permita a uma empresa ou individuo o acesso a toda a informação acumulada através dos sensores do veiculo e, ao mesmo tempo, permitir a configuração do veículo e a sua gestão de erros.

Este sistema vai utilizar a tecnologia Controller Area Network (*CAN*) para aceder e acumular dados dos sensores do veiculo para que estes sejam acedidos em tempo real através de uma aplicação. Esta aplicação irá comunicar com a *OBU* do veiculo através do protocolo *SNMP*. Esta solução deverá ser capaz de gerir mais pedidos de informação que tecnologias ou protocolos standard já existentes, como On Board Diagnostics (*OBD-II*), sendo também mais rápido que estes. Adicionalmente, esta solução deverá também incluir alguma maneira para que utilizadores ou outras entidades numa *VANET* possam activar/desactivar actuadores específicos visto que tal funcionalidade é vital para a introdução de métodos de condução como platooning.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

A

ACC Adaptive Cruise Control.

ACK Acknowledgment.

AGENTX Agent Extensibility.

API Application Programming Interface.

B

BGP Border Gateway Protocol.

C

C-ITS Cooperative Intelligent Transport System.

CACC Cooperative Adaptive Cruise Control.

CAFE Corporate Average Fuel Economy.

CAN Controller Area Network.

CANFD Controller Area Network Flexible Data-Rate.

CRC Cyclic Redundancy Check.

CSM Community-based Security Model.

D

DBC CAN Bus DataBase.

DTLS Datagram Transport Layer Security.

E

ECU Electronic Control Unit.

EOBD European On-Board Diagnostics.

ETSI European Telecommunications Standards Institute.

I

IAB Internet Architecture Board.

INMF Internet Network Management Framework.

IOT Internet of Things.

IP Internet Protocol.

ITS Intelligent Transport System.

ITS-LCI Intelligent Transport System Local Common Interface.

L

LIN Local Interconnect Network.

M

MIB Management Information Base.

MOST Media Oriented System Transport.

N

NM Network Manager.

NMS Network Manager System.

O

OBD On-Board Diagnostics.

OBD-II On-Board Diagnostics-II.

OBU On-Board Unit.

OID Object Identifier.

OSPF Open Shortest Path First.

P

PDU Protocol Data Unit.

PID Parameter ID.

POTS Plain Old Telephone Service.

PSTN Public Switched Telephone Network.

R

RIP Routing Information Protocol.

RSU Road Side Unit.

S

SAE Society of Automotive Engineers.

SMI Structure of Management Information.

SNMP Simple Network Management Protocol.

SSH Secure Shell.

T

TCP Transmission Control Protocol.

TLS Transport Layer Security.

U

UDP User Datagram Protocol.

USM Used-based Security Model.

V

V2I Vehicle to Infrastructure.

V2V Vehicle to Vehicle.

V2X Vehicle to Everything.

VACM View-Based Access Control Model.

VANET Vehicular ad-hoc Network.

1

INTRODUCTION

The automotive industry has for over two decades used *CAN* (Controller Area Network) to manage and configure sensors and actuators in industrial environments as well as to allow universal communication between industrial components developed by different manufacturers. While it's not the only technology to provide these features, it's the most widely used in industrial environments, and more importantly the automotive industry.
More specifically, this technology is used in the internal architecture of vehicles and, as such, it does not allow any third-party applications or hardware to access the *CAN* bus directly. This presents an unique problem to overcome, "How can a third party entity gain access to this network so it can monitor data being transmitted over *CAN* and change the state of any actuator in it?".
The most common method that's currently used to overcome this problem is by accessing the *CAN* bus indirectly through (*OBD*) (On-Board-Diagnostics) technologies and its variants and while this solution allows a third party to read sensor data and send commands to *ECU*s, it comes with its own drawbacks, both in terms of security, performance and versatility which, while minor inconveniences as far as the average user or repair technician is concerned, does restrict its integration in more advanced communication systems, such as *VANET*s (Vehicular ad-hoc Network).

SNMP (Simple Network Management Protocol) [1] is widely used to manage and configure equipment and services, not only in internet networks but also in all other applications domains. Through its long years as a standard, it has shown to be a stable and efficient protocol that can be used by network managers to manage, monitor, and configure equipment while being lightweight enough that even devices with lacking computing power can use it.

Through the use of *SNMP*, and an accompanying custom *MIB* (Management Information Base), another possible solution to the aforementioned problem will be presented that doesn't compromise in security and efficiency while providing the same functionalities in regards to configuration and monitoring of sensors and actuators. This solution will be non proprietary, i.e universal, by being a program that can be integrated and setup on an already existing vehicle *OBU* and as such won't require vehicle manufacturers to redesign the internal architecture of their vehicles. This type of solution has already been proposed [2][3] and, in a way, this project will try to study its effectiveness.

1.1 MOTIVATION

An *ITS* (Intelligence Transportation System) is an advanced application which aims to provide services that enable users to be better informed and make safer, more coordinated, and 'smarter' use of transport networks. This includes, for example, calling for emergency services when an accident occurs or using cameras to enforce traffic laws depending on conditions. Somewhat more relevant for this project is that an *ITS* application will also allow for cooperative systems on the road, that is, communication between car-to-car, car-to-infrastructure and vice versa. For this use-case, data obtained from a vehicle can be used to detect events, such as rain or congestion, and based on these events take preventive actions with the objective of increasing road safety. Such a system needs to overcome a couple of obstacles before it can be introduced as a standard on all vehicles. One such obstacle is how to quickly and reliably collect/transfer data from a vehicle's sensors in a network, and while there are already some solutions in-place that partially overcome this, they are not without drawbacks. These solutions usually come in two "flavors":

- Built-in telematics- Communication with the outside world is done via an proprietary Internet connection and a GSM module.
- Brought-in telematics- Communication with the outside world is done via a device that is plugged in to the OBD port.

As mentioned, each one of these two methods has its own drawbacks, where the former makes use of proprietary internet connection and software which means any customer will only have access to content that the manufacturer allows the customer to have, the latter is limited by low throughput caused by *OBD-II* standard and requires specialized hardware and software to use.

Protocol Nrº	Protocol	Max Throughput
1	SAE J1850 (PWM)	41.6kbps
2	SAE J1850 (VPW)	10.4kbps
3	ISO 9141-2	10.4 kbps
4	ISO 14230-4 (KWP 2000)	10.4kbps
5	ISO 15765-4 (CAN)	500kbps

Table 1: Throughput of main standards supported by *OBD-II*[4]

This low throughput is the major obstacle that prevents *OBD-II* from being used in *VANET* since, when *OBD-II* is being used to collect data from a vehicle it must first send a request, Parameter ID (*PID*), to the *OBU* for the data, wait for the response and only then send a second *PID*. This means that as the number of requests increase, so will the refresh rate of each individual request, more specifically, if there's only one *PID* enabled, the maximum refresh rate is 50ms but if we were to increase the number of *PID*s to the maximum, that is 20 *PID*s per second, we would be looking at a refresh rate of 1000ms per *PID* [5][6][7].

This limitation in number of sensors that can be monitored at the same time, low throughput and low refresh rate when the number of sensors being monitored increases makes the *OBD-II* interface a problematic source of sensor data if we were to use it in *VANET*s. This low refresh rate is especially damning when we consider that even assuming a busload of 70%, which is generally considered the "real-world" maximum, it means that for example, an SAE J1939 data frame may occur every 0.77 ms @ 250 kbps or 0.39 ms @ 500 kbps [8], significantly faster than the 50ms refresh rate on *OBD-II*.

One other obstacle to overcome is how to get *OBU*s made by different manufacturers to communicate with each other in an environment where each manufacturer relies on their own adapted proprietary architecture, which renders any interoperability between applications by different manufacturers difficult to achieve.
And lastly, how can we allow for third-party software makers to develop and implement their own *C-ITS* applications to compete in the automotive market independently of the manufacturer[9] without disclosing any details on a manufacturers proprietary architecture.

A proven technology like *SNMP* working alongside *CAN* can be used to overcome these obstacles while also adding an extra level of security since access to the *MIB* objects of the *SNMP* agent integrated in the *OBU* will only be possible to application modules/*SNMP* managers in the vehicle's local network since it is not wise to allow outside entities direct access to sensors/actuators of a vehicle[3].

1.2 OBJECTIVES

The main objective of this dissertation will be the development of a solution that allows any authorized entity to obtain data from sensors and allow the configuration of actuators connected to it, through an *SNMP* agent integrated in the *OBU* and accompanying *MIB*, in a way that does not compromise in performance and security regarding further development of host-based distributed applications. This solution should bypass any bottlenecks caused by the slow throughput of *OBD-II*, which will allow for real-time diagnostics to be performed on a whole fleet of vehicles while also allowing for better and easier integration of the vehicle's sensors with *VANET*s, it should also give access to any car manufacturer or end-user a solid base with which to build upon and develop their own diagnostics/configuration software.
In essence this solution will be comprised of four different parts:

- MIB.
- SNMP Agent.
- SNMP Manager.
- Host Based Distributed Application.

The *MIB* will define the database that will be used for managing vehicle's sensors and actuators, while both the SNMP Agent and SNMP Manager will communicate with each other using *SNMP* and the *MIB*, thus, allowing for data collected from the vehicle's sensors to be transmitted to whichever device holds the SNMP Manager. Finally a Host Based Distributed Application should provide an easy to use interface that allows a user to read sensor data and configure actuators in real time. The *SNMP* Manager can also be included in the Host Based Distributed Application. Additionally a decoder will also be developed so that we are able decode messages sent in the vehicle's internal network by *ECU*s into human readable data.

Both the *MIB* and *SNMP* agent will be integrated directly in the *OBU* while the *SNMP* manager and host based distributed application should be installed in the vehicle's private network, directly in the *OBU* and/or outside it. Any communication between other distributed *ITS* applications and the system should be done via this host based distributed application.

To accomplish this, an iterative methodology based on document research, solution proposal, implementation and testing will be followed:

- Thoroughly research state-of-the-art access technologies to *CAN* bus, namely *OBD* and its variants.
- Research recent projects that have attempted to overcome the inherent limitations of *OBD* based solutions.
- Research advantages and disadvantages of an *SNMP* based solution.
- Define a *MIB* that can monitor and configure sensors and actuators connected to an *OBU*. Such a *MIB* has to allow access to low level *ITS* functions implemented by the vehicles manufacturer and, at the same time, be transparent in relation to the chosen electronic communication bus, be it *CAN*, Flexray or any other.
- Develop a prototype *SNMP* agent and manager so that it's possible to test the validity of the introduced contexts.

1.3 DOCUMENT LAYOUT

This dissertation is divided into five chapters covering areas related to *SNMP*, *OBD-II* and *CAN* architecture, proposed approach, development decisions, results and finally a conclusion and future work:

- Chapter 1 - Introduction: In this chapter the context of the dissertation is set, as well as the motivation and objectives to be met. The layout of the dissertation is also explained
- Chapter 2 - Related Technologies and R&D Works: In this chapter the already existing technologies and solutions are explained and discussed.
- Chapter 3 - Proposed Approach: In this chapter a proposed approach will be presented based on the results of the discussions in Chapter 2.
- Chapter 4 - Developing a Prototype: This chapter will explain in detail the most critical steps in the development of this solution while also presenting and discussing test results.
- Chapter 5 - Conclusion: This chapter will contain the conclusions of this dissertation and any potential future work that could improve the presented solution.

2

RELATED TECHNOLOGIES

Ever since the first communications networks were introduced, in the shape of telephone networks (*POTS*), there has been a need for Network Managers (*NM*), which at that time were telephone operators, to detect network-affecting problems, like equipment failure or traffic overloads and fix them by rerouting/blocking traffic from entering the congested network or alerting a technician to initiate maintenance activities. Nowadays, telephone networks have been replaced by *PSTN* while the humble telephone operator was replaced by routing protocols like *RIP*, *OSPF* or *BGP*.

Another area in which *NM*s became relevant was in industry, where networks are extremely important as a way to keep production lines running as efficiently as possible. In this area, the role of an *NM* is to ensure that every cog in the machine is running properly and at optimum condition. To fulfill this role a protocol that was simple, centralized and had low consumption of resources on the managed devices was needed. While a variety of protocols were proposed, most were rejected except for *SNMP*.

With the 1970's fuel crisis, the automobile industry realized that fuel efficiency would become a consumer and government requirement. In response to this, the United States Congress established a set of standards named Corporate Average Fuel Economy (*CAFE*) which all companies had to comply. At the same time, computer controls as well as sensor technology were coming of age and by marrying electronics, sensors and software one could create automotive computer controlled systems and with this the fuel efficiency race was on. With the advent of these computer controlled systems a need to diagnose and monitor the performance characteristics of individual components was born [10] and it was due to this need that solutions like *OBD* were developed.

With all these sensors, and their corresponding *ECU*s, being added to a vehicle, its wiring became an issue. To fix this a field bus system based on serial communication was developed and introduced in the 1980's called *CAN* which, as promised, reduced wiring while also increasing reliability and improved service and maintenance features [11].

With the introduction of *VANET*s, a need for an efficient and safe way to access sensor data from a vehicle to use in *ITS* applications has surfaced, however this need can't be easily met with *OBD* based solutions but it can be done if we were to replace *OBD* with *SNMP* as the interface technology [3][2][9]. As per [2], *SNMP* enabled micro controller boards can communicate with multiple sensors while also forwarding data to an on-board *SNMP* manager. The *CAN* interface would also allow for efficient and reliable data transfer between the *SNMP* enabled micro controller boards and *SNMP* manager, additionally, off board communication can be handled by cellular technologies. Although a promising start, [2] does not specify how this solution is to be implemented or the structure of its *MIB* and its proposal of multiple *SNMP* agents in a vehicle, one per type of sensor, goes against normalized *ITS* architectures proposed by institutions like *ETSI* or the one proposed in [9].
In [3], the use of *SNMP* in the context of vehicles with *OBU*s capable of communicating and integrating with *VANET*s is mentioned although, once again, no *MIB* is proposed or any details on how this solution may be integrated.
Out of the three research papers, [9] is the one that provides the most insight into how *SNMP* can be integrated into a modern *ITS* architecture and as such it will form the basis of this solution despite also lacking a *MIB* specification.

2.1 SNMP

Originally developed and introduced by university researchers in the 1988, *SNMP* is a standard protocol for network management. It's used by Network Administrators to monitor and map network availability, performance and error rates [12] and uses *UDP* as the transport protocol, however *TCP* can be used as well.

It was aproved as an internet standard by Internet Architecture Board (*IAB*) in 1990 to address a clear and growing need for network management [1]. In this solution, *SNMP* will be used to allow an Agent integrated in the *OBU* to communicate and transfer data to a manager, installed somewhere in the vehicle.

SNMP comes in 3 main versions:

- SNMPv1-Released in 1990.
- SNMPv2-Released in 1993.
- SNMPv3-Released in 1999.

SNMPv1

SNMPv1, as the name implies, is the first publicly available version of *SNMP* and while it accomplished its goal of being an open, standard protocol, it was lacking in key areas. More specifically, it only supports 32-bit counters, which hampers its ability to manage modern networks, and has poor security features.

SNMPv2

Released in 1993, the main differences between this version and SNMPv1 are improved error handling, the addition of SET operations and support for 64-bit counters. Security features are exactly the same as in SNMPv1, which is in the form of community strings, a simple password that devices need to be able to talk to each other and transfer information.

SNMPv3

SNMPv3 is the newest version of *SNMP*, released in 1999, and its new features primarily revolve around enhanced security. Each *SNMP* entity now has an Engine ID which is used to generate a key for authenticated messages. Besides authentication SNMPv3 now also supports encrypting of *SNMP* messages with USM/VACM.

2.1.1 *USM/VACM*

Introduced as part of *SNMP*v3, *USM/VACM* allows for better message security and access control respectively.
USM, User-Based Security Model, provides message security by:

- Data integrity checking, to ensure that the data was not altered in transit.
- Data origin verification, to ensure that the request or response originates from the source from which it claims to have come from.
- Message timeliness checking and, optionally, data confidentiality, to protect against eavesdropping.

While View-Based Access Control Model (*VACM*) is the ability to control exactly what data an individual user can read or write.
It should also be noted that these security models can be used concurrently with the older community string based security [1].

With these functionalities enabled it's required that users provide the following information when invoking commands.

Parameter	Command-line Option	Description
engineID	-e <EngineID>	Engine ID of the SNMP agent
securityName	-u <Name>	User name
authProtocol	-a <MD5|SHA>	Authentication Type
authKey	-A <PASSPHRASE>	Passphrase
securityLevel	-l <authNoPriv|AuthPriv|noAuthNoPriv>	Security Level
privProtocol	-x <none|des>	Privacy protocol
privPassword	-X <password>	Password

Table 2: SNMPv3 Security Parameters [1]

2.1.2 *How it works*

An *SNMP* network is usually composed by three different parts:

- Manager- Installed in a Network Management System *NMS*:
  - Queries Agents.
  - Sets variables on Agents.
  - Gets responses from Agents.
  - Acknowledges asynchronous events from Agents.
- Agent- Installed in a device that is to be monitored:
  - Collects management information about its local environment.
  - Stores and retrieves management information as defined in the *MIB*.
  - Signals an event to the Manager.
  - Acts as a proxy for some non–SNMP manageable network node.
- MIB:
  - Contains an information database describing the managed device parameters.
  - Both the Agent and Manager share this database and the latter uses it to request specific information from the former.

These requests are done via a set of operations:

- GET- Retrieves the object instance from an Agent.
- GETNEXT- Retrieves the next object variable.
- GETBULK- Retrieves a large amount of objects variables, without needing several GETNEXT operations.
- SET- Tells the *NMS* to modify the value of an object variable.
- TRAPS- Alerts the *SNMP* manager about a condition on the network.
- INFORMS- Traps that include a request for confirmation of receipt from the *SNMP* manager.

In summary, we have a Manager installed in an *NMS* which communicates with Agents installed in one or more devices, where each Agent is keeping track of several parameters of that device via a *MIB* which is in turn shared between both Manager and Agent thus allowing the former to request specific information from the latter.

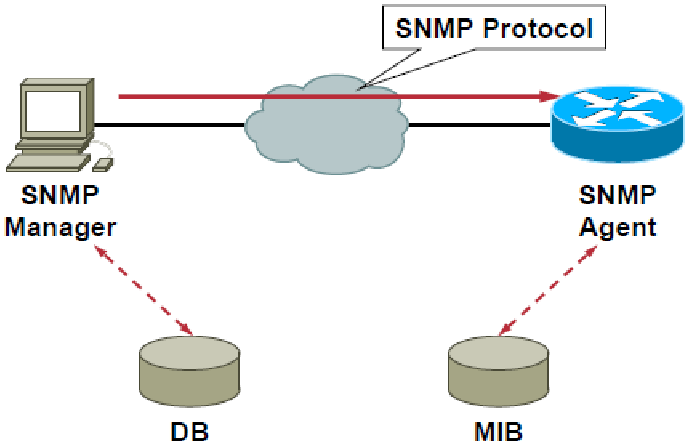

Figure 1: Simplified SNMP architecture. From [13].

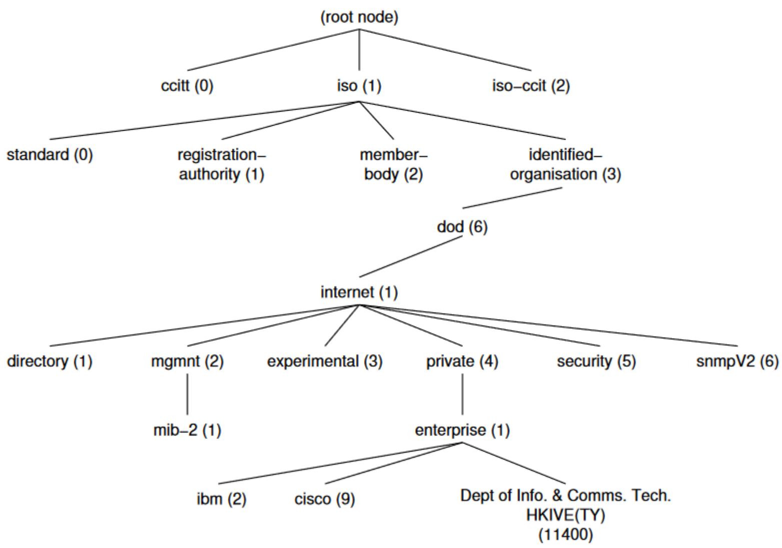

Figure 2: Management Information Base. From [14].

2.2 SNMP MESSAGE FORMAT

The Structure of Management Information (*SMI*) is a standard that defines all types of objects that can be included in *SNMP* messages.

In this project the latest version, *SMI*v2, will be used.

The main propose of the SNMP is to define rules that allow agents and managers to exchange management information. An SNMP message is formed by a single *PDU* sent over *UDP* (port 161 and 162 are standards).

In figure 3 the *SNMP*v3 *PDU* format is presented.

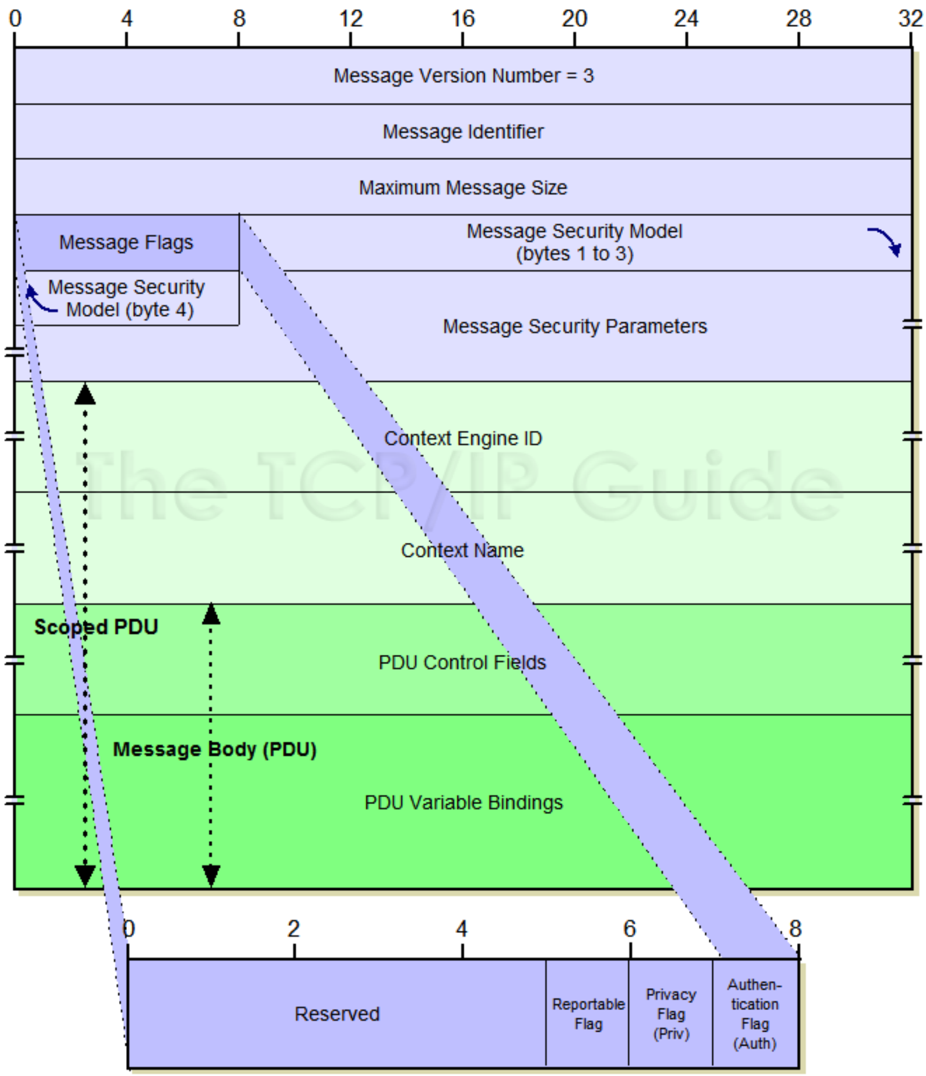

Figure 3: SNMPv3 PDU. From [15]

Field	Syntax	Size (bytes)	Description
Msg Version	Integer	4	Describes the SNMP version number of this message, used for ensuring compatibility between versions. For SNMPv3, this value is 3
Msg ID	Integer	4	A number used to identify an SNMPv3 message and to match response messages to request messages
Msg Max Size	Integer	4	The maximum size of message that the sender of this message can receive. Minimum value of this field is 484.
Msg Flags	Octet String	1	A set of flags that controls processing of the message
Msg Security Model	Integer	4	An integer value indicating which security model was used for this message. For the user-based security model (the default in SNMPv3) this value is 3.
Msg Security Paramenters	-	Variable	A set of fields that contain parameters required to implement the particular security model used for this message. The contents of this field are specified in each document describing an SNMPv3 security model. For example, the parameters for the user-based model are in RFC 3414.
Scoped PDU	-	Variable	Contains the *PDU* to be transmitted, along parameters that identify a *SNMP* context.

Table 3: SNMPv3 PDU Message Fields

2.3 MIB

In an *SNMP* based solution, a *MIB* is a local database of information relevant to the network management that both the manager and agent maintain. This *MIB* will contain the definition and information regarding the proprieties of managed resources/services, called objects or variables, these object can be divided into two types, scalar objects which define a single object instance and tabular objects that define multiple related object instances that are grouped in *MIB* tables. Additionally different *OID*s can be grouped into *MIB* groups if needed. The structure of the *MIB* information and allowable data types is defined by the structure of management information (*SMI*). This *SMI* identifies how resources within the *MIB* are defined and named.
A *MIB* object is defined by the following keywords:

- Syntax- Defines the abstract data structure corresponding to the object type. For example, Unsigned32, Integer, etc.
- Max-Access- Defines whether the object value may only be retrieved but not modified (read-only) or whether it may also be modified (read-write).
- Description- Contains a textual definition of the object type. The definition provides all semantic definitions necessary for interpretation.

Additionally each object in a *MIB* has an object identifier (*OID*)[16], which the management station uses to request the object's value from the agent. An OID is a sequence of integers that uniquely identifies a managed object by defining a path to that object through a tree-like structure called the OID tree or registration tree. When an SNMP agent needs to access a specific managed object, it traverses the OID tree to find the object. In this tree the top-level *OID*s belong to different standard organizations while the lower levels are allocated by associated organizations.

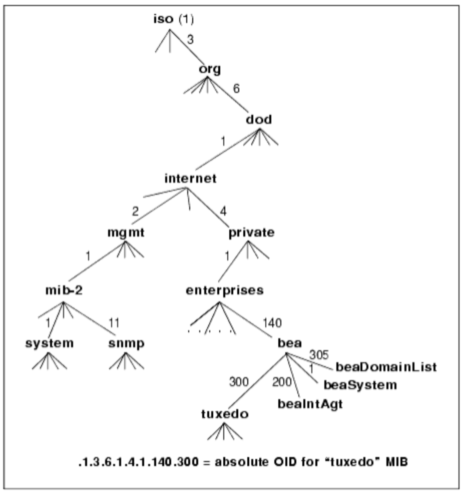

Figure 4: *SNMP MIB* Object Identifier Hierarchy and Format. From [17]

2.4 OBD-II

First introduced in the early 1980s *OBD* is an automotive term referring to a vehicle's self-diagnostic and reporting capability. *OBD* systems give the owner/repair technician access to the status of various vehicle sub-systems thus allowing for easier repairs and even preventive maintenance of failing sub-systems.

Circa 1994, *OBD-II* specification is developed and by 1996 is made mandatory in the US. In Europe, *EOBD* is developed and made mandatory by 2001/2004 for all petrol and diesel vehicles respectively. *EOBD* is essentially a copy of *OBD-II*, using the same connector and signal protocols [18]. *OBD-II* will be the protocol with which the proposed solution will be compared

Figure 5: *OBD-II* connector. From [18]

As mentioned in 1, *OBD-II* supports five different signal protocols:

- SAE J1850 PWM- Pulse Width Modulation 41.6kbps, used by FORD Motor Company and Mazda.
- SAE J1850 VPW- Variable Pulse Modulation 10.4/41.6kbps, used by General Motors.
- ISO 9141-2- 10.4kbps, used by Chrysler and some Asian/European Manufacturers between 2000-2004.
- ISO 14230-4 KWP2000- Keyword Protocol 2000 10.4kbps, commonly used after 2003 by various manufacturers.
- ISO 15765-4 CAN-BUS- From 2008 onward all vehicles sold in the US must implement *CAN* as one of their signalling protocols.

This standard has some disadvantages that prevent it from being used in *VANET*s. First of all it requires specialized hardware to connect with the vehicle's own *OBD-II* connector, then it requires specialized software that is able to read the messages/write to the car's *OBU*, has some serious security flaws [19], it has too much latency to be of use in time sensitive applications and finally it's limited in the number of *ECU*s it can query at the same time and in the *ECU*s it can query.

2.5 CAN

A *CAN* provides a cheap durable network that allows vehicle devices to speak through the Electronic Control Unit (*ECU*) while allowing it to only have one *CAN* interface instead of several analog inputs for all devices in the system.

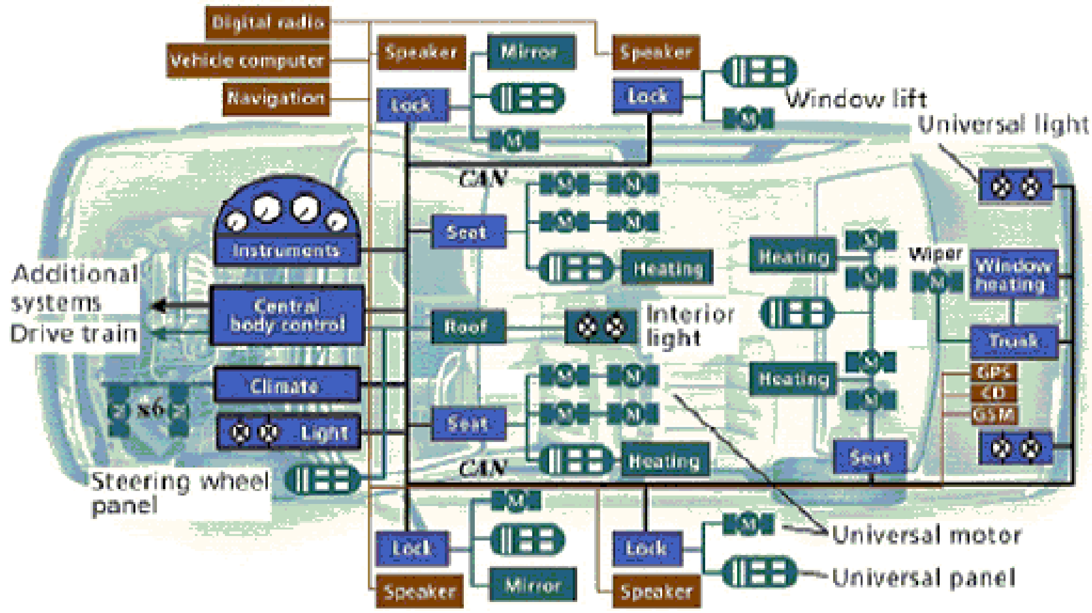

Figure 6: Example of a vehicle *CAN* network. From [20]

Originally developed between 1983 and 1986, it would in 1991 see Bosch publish its latest specification in the form of *CAN* 2.0. This specification has two parts, *CAN* 2.0A and 2.0B, where the former uses 11-bit identifiers and the latter 29-bit identifiers [21]. Being one of the five protocols supported by *OBD-II*, *CAN* would see wide range of adoption in the car manufacturing world, especially since *OBD-II* became mandatory in the US and Europe.

In 2012 Bosch extended the *CAN* standard by releasing Controller Area Network Flexible Data-Rate (*CANFD*). This specification uses a different frame format which allows for different data length and switching to a faster bit rate after arbitration is decided while maintaining backwards compatibility with *CAN* 2.0. *CAN* will be the protocol used when testing the performance of the proposed solution.

Lastly *CAN* provides a fast interface, 1Mbit/s if the bus length is less than 40 meters, which should provide latency's of around 330 μs, which includes, for bus transport, 130μs if it's a full length *CAN* message or 53μs if it's a short *CAN* message, and around 100μs for both the transmitting and receiving node to prepare the transfer/reception of a message [22].

2.5.1 *How it works*

The *CAN* bus is a broadcast type of bus. This means that all nodes can "hear" all transmissions made by other nodes. There is no way to send a message to just a specific node; all nodes will invariably pick up all traffic. The *CAN* hardware, however, provides local filtering so that each node may react only on the messages directed to it.

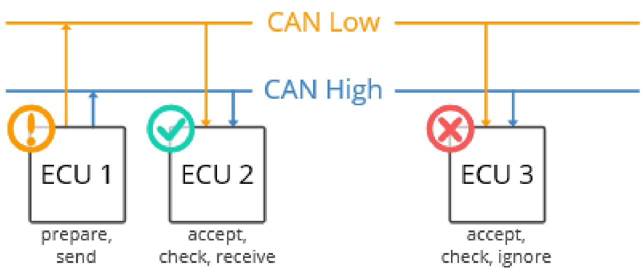

Figure 7: Example of *CAN* communication. From [23]

Since all *CAN* controllers share the same bus there needs to be some process where two or more of them agree on who is allowed to use the bus. Any *CAN* controller can start a transmission if it detects an idle bus. This may result in two or more different nodes starting to send a message at the same time causing a conflict. This conflict is resolved in the following manner:

- The transmitting node monitors the bus while it is sending a message.
- If a node detects a message with a dominant level when it is sending a recessive level itself, it will quit the arbitration process and become receiver.
- This arbitration is performed over the whole arbitration field and when this field is sent only one node will be transmitting.
- Once the message is transmitted the other node can restart the transmission of its own message.

2.5.2 *Message Types*

There are four different types of messages, or frames, that can be transmitted by a node on a *CAN* bus:

- Data Frame- Most common message type, used to send data on the *CAN* bus.
- Remote Frame- Used to solicit the transmission of the corresponding Data Frames.
- Error Frame- Used to request re-transmission of a message.
- Overload Frame- Used to indicate when a node is too busy.

2.5.3 *Message Fields*

A *CAN* frame can have up to 7 fields:

- Start of Frame- Marks the beginning of a frame, consists of one bit only.
- Arbitration Field- Contains an Identifier and a Remote Transmission Request bit. Total size of 11 or 29-bit.
- Control Field- Indicates the total number of bytes on Data Field.
- Data Field- Depending on the type of frame, will either be empty or contain the data that is getting transmitted up to a maximum of 8 bytes.
- CRC Field- Contains *CRC* Sequence, and *CRC* delimiter.
- ACK Field- Contains two bits with an ACK Slot and ACK Delimiter.
- End of Frame- Contains a series of recessive bits.

As previously mentioned there are three main specifications being used in the *CAN* standard.

CAN2.0A

This version is mostly used in light vehicles and features an 11-bit identifier.

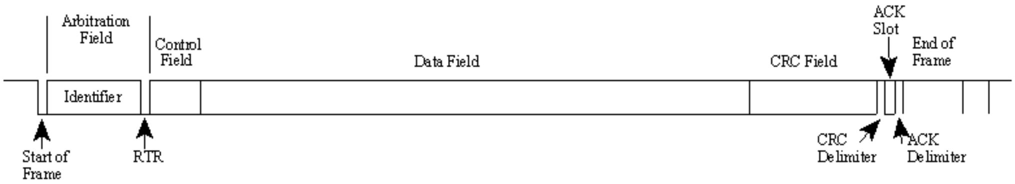

Figure 8: *CAN*2.0A message. From [24]

CAN2.0B

This version is mostly used in heavy vehicles, like trucks and buses, and features an 29-bit identifier.

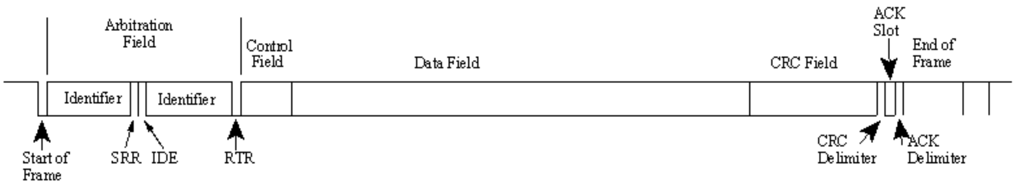

Figure 9: *CAN*2.0B message. From [24]

CAN FD

Newest specification of *CAN* standard, it's expected to provide up to five times the speed of classical *CAN*, at 5Mbit/s, will increase the maximum data payload from 8 bytes to 64 bytes and is expected to appear on all vehicles from 2019 onward.

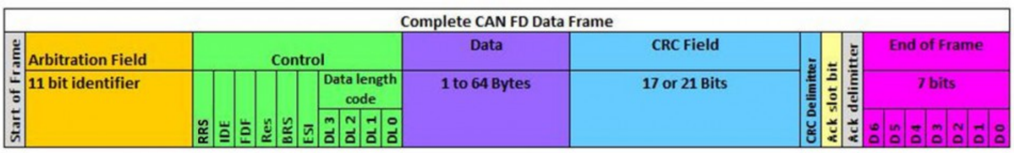

Figure 10: *CANFD* message. From [25]

Other Network Types

Before exploring other communication network protocols it's important to to clearly define *SAE* network classes. These range from A (lowest speed) to C (highest speed). Every protocol will fit into at least one of these classes but some like *CAN*, class B and C, can actually fit into two [26]:

- Class A- low speed (less than 10Kb/s) for convenience features such as body and comfort.
- Class B- medium speed (between 10 and 125Kb/s) for general information transfer, such as emission data, instrumentation.
- Class C- high speed (greater than 125Kb/s) for real-time control such as traction control, brake by wire, etc.

While *CAN* it the most widely used in-vehicle communication network protocol, it doesn't mean it's the only one in existence. Many of these protocols were created by different companies or consortium's a way to meet the different performance requirements throughout a vehicle.

LIN

Local Interconnect Network (*LIN*) is a low-cost serial communication system used as *SAE* class A network, which is a network that is used when the needs, in terms of communication, do not require the implementation of higher-bandwidth multiplexing networks such as *CAN*[27]. The typical applications involving *LIN* include controlling doors (e.g., door locks, opening/closing windows) or controlling seats (e.g., seat position motors, occupancy control). The maximum data rate of *LIN* is 20Kb/s.

MOST

Media Oriented System Transport (*MOST*) is a *SAE* class C network with a data rate of 25Mb/s that provides point-to-point audio and video data transfer with different possible data rates. *MOST* supports end-user applications like radios, GPS navigation, video displays and entertainment systems and has become the de-facto standard for transporting audio and video streams within vehicles[28].

FlexRay

Also a *SAE* class C network, FlexRay is an automotive network communications protocol developed by the FlexRay Consortium, now disbanded, which was a consortium of vehicle manufacturers that included the likes of:

- BMW.
- Daimler.
- Volkswagen.
- General Motors.

The aim of this consortium was to develop a faster and more reliable automotive network communications protocol than *CAN*.

FlexRay specifies three different bit-rates, all of which are faster than classical *CAN* [29]:

- 10Mbit/s.
- 5Mbit/s.
- 2.5Mbit/s.

A FlexRay signal can carry up to 254 bytes, much higher than both classical *CAN* and *CANFD* at 8 and 64 bytes respectively, and has three *CRC*, which allows it to be more reliable and flexible than its main competitor.

However it hasn't seen wide adoption due to higher installation costs and problems with extending network length, being only used by high-end manufacturers like BMW, Audi, Bentley and Mercedes in their latest top of the range models, and as such it's expected to be replaced by Ethernet systems in the near future.

2.6 SNMP AND IOT

Now, more and more objects are becoming embedded with sensors and gaining the ability to communicate. Many *IOT* devices have sensors that can register changes in temperature, light, pressure, sound and motion. To monitor/configure this myriad of sensors, *SNMP* based solutions can prove to be ideal due to the fact that *SNMP* was designed to be used in resource constrained devices [30]. These same requirements also exist when it comes to *OBU*s since they are also resource constrained devices, although not at the same level as *IOT* sensors and as such frameworks have already been proposed that provide remote vehicular sensor monitoring through *SNMP* [2][3]. By nature, this will require *OBU*s with an *SNMP* agent integrated and luckily modern ones do already come with such an agent incorporated in them to configure/monitor some aspects of their every day running however this agent is unable to access sensors/actuators of the vehicle.

2.7 SUMMARY

In this chapter a general overview of the *CAN* protocol and *SNMP* was given, as well as how each of these two protocols work, their architecture, real world uses and how different versions of each protocol, and competitors, compare to each other so that the best options would be chosen for the project. Additionally a short introduction to the *OBD-II* standard was given, since it will be compared throughout this dissertation with the developed solution.

3

SNMP-BASED SOLUTION

In this chapter the requirements, design goals and main issues to overcome will be presented as well as the proposed solution to meet them. This solution will be based on the performance of already existing protocols, *SNMP* versions, *SNMP* commands, security features and more. Lastly the system architecture will also be presented as well as any *API* that aids in the development of this project.

Since the most commonly used protocol within a vehicles' internal network is *CAN*, this solution will be developed with it in mind, in essence, using both *SNMP* and *CAN* to allow outside entities to access the internal network of a vehicle.
This solution needs to be universal, that is, it can be used by any manufacturer independently of its proprietary internal architecture, and it must allow:

- Monitoring sensor data from a vehicle.
- Control over a vehicles actuators in a way that is:
  - As direct as possible.
  - As safe as possible.
  - As reliable as possible.
  - With as little delay as possible.
  - With as high of a data rate as possible.

Due to security reasons, no manufacturer will allow direct access to actuators or sensors of a vehicle to any third party applications, and as such, the only current options are to access these components via the *OBU*, or any other application or mechanism that the manufacturer may or may not provide. Due to this, the only truly universal method to obtain sensor/actuator access is to do it through *OBD-II* port, however this is limited both in terms of performance and in terms of what sensors/actuators we can access.
This project hopes to develop another method to access to these components with the use of *SNMP* and a custom made *MIB*, since through these we can overcome most of the limitations of *OBD-II* and mitigate others, i.e. security requirements can be met through the use of *SNMP*v3, by avoiding constant and active polling we can also increase data rate and decrease delay.

Nevertheless, even with *SNMP*, it is not wise to allow direct access to the agent that is integrated in the *OBU* since it can be dangerous to allow an outside entity direct access to the vehicles actuators and instead only allow applications in the vehicle's local network to directly access the *SNMP* agent in the *OBU*. These applications, each with their own *SNMP* manager, can be roughly split into three types:

- Applications that are only useful to the driver, passenger or other internal vehicle applications/services. These applications can also obtain additional data from external sources, *V2X*, to implement features like:
  - Adaptive Cooperative Cruise Control.
  - Cooperative Cruise Control.
  - Emergency Breaking.
- Applications that contribute to the implementation of another distributed service, or application system, that requires *V2X* communications to trade information between the multiple components in it. This includes features like:
  - Platooning.
  - Cooperative Mapping.
  - Cooperative Traffic Management.
- Applications that serve as a proxy to allow indirect access to external services. This includes:
  - Billing tolls and parking spaces.
  - Information gathering services for brand/dealer clouds.
  - Transport fleet monitoring systems.
  - Vehicle diagnostics and inspection

The overall architecture of this project shall include an *SNMP* Agent instegrated in a vehicle *OBU*, this Agent will manage and store data captured from the vehicles *ECU* in a *MIB*. This *MIB* will be shared between the Agent and Manager, thus allowing for the latter to request data from the former. The same *MIB* can also be used by the Manager to send Set messages that change data in an Agent which will in turn allow for error management and configuration of an vehicle's actuators. This Manager can then be used in both *ITS* applications or a manufacturers' diagnostic software.

In addition to this, since the primary goal of any *C-ITS* application is to both save lives and improve traffic flow[31] any application developed to work in this environment needs its communication with other entities on the road, be them *OBU*s, *RSU*s or even pedestrians, to be both fast and secure and as such this solution should provide an alternative that improves on the performance, when compared to other universal solutions like *OBD-II*.

3.1 SYSTEM ARCHITECTURE

As shown in Figure 11 this solution will require one or more applications to be installed in the car, but outside of the *OBU*, and one *SNMP* Agent that is integrated directly on the vehicles *OBU*.

The *SNMP* Agent main function will be to store data recorded by the vehicles' sensors and, based on requests from the application, transmitting that data to the application, which may then forward it to the relevant entities. If this application is being used to communicate to outside entities its *SNMP* manager should also serve as a gateway application, filtering, pre-processing and even allowing/blocking access to certain functionalities based on the entity as a security feature.

In the architecture represented in Figure 11 we have a car, its local network, *OBU* and *ITS-LCI* (Intelligent Transport System Local Common Interface). As part of the *OBU* we have a *SNMP* agent and three types of services modules[9]:

- Communication Services Module - This interface module will permit sharing of all medium-access technologies supported by the OBU by all application environments in the *ITS* station and deployed on resources outside the *OBU*, or hosts

- Information Services Module- This interface module will permit access and manipulation of data generated by all sensors, actuators and other devices in the vehicle, indirectly or directly connected to the *OBU*

- Function Services Module- This interface module will permit access to lower-level functionality procedures. These are functions that the manufacturers, due to security, safety, performance and liability issues, should have the responsibility and the desire to implement (or closely control its implementation).

The *SNMP* agent is integrated in the *OBU* and is part of the Information Service Module. Certain internal *OBU* services included in the Function Service Module can communicate with the *SNMP* agent directly, through *SNMP*v3. These services can then be accessed with other access technologies that are defined in the *ITS-LCI*.
Otherwise, one can directly access the *SNMP* agent through *SNMP* manager(s) outside the *OBU*. These will be integrated within one of the three types of applications mentioned above and no matter the type of application, it has to be connected to the vehicles internal network while communication with external distributed applications/services is done indirectly though modules or proxies.
Finally, if and when sensor data is being monitored by local network applications where authentication is not essential, which can happen if the development and installation environment of applications in the vehicles local network is more "closed-off" and secure, using *SNMP*v2c instead of *SNMP*v3 to monitor sensor data can be more efficient in terms of delay and throughput.

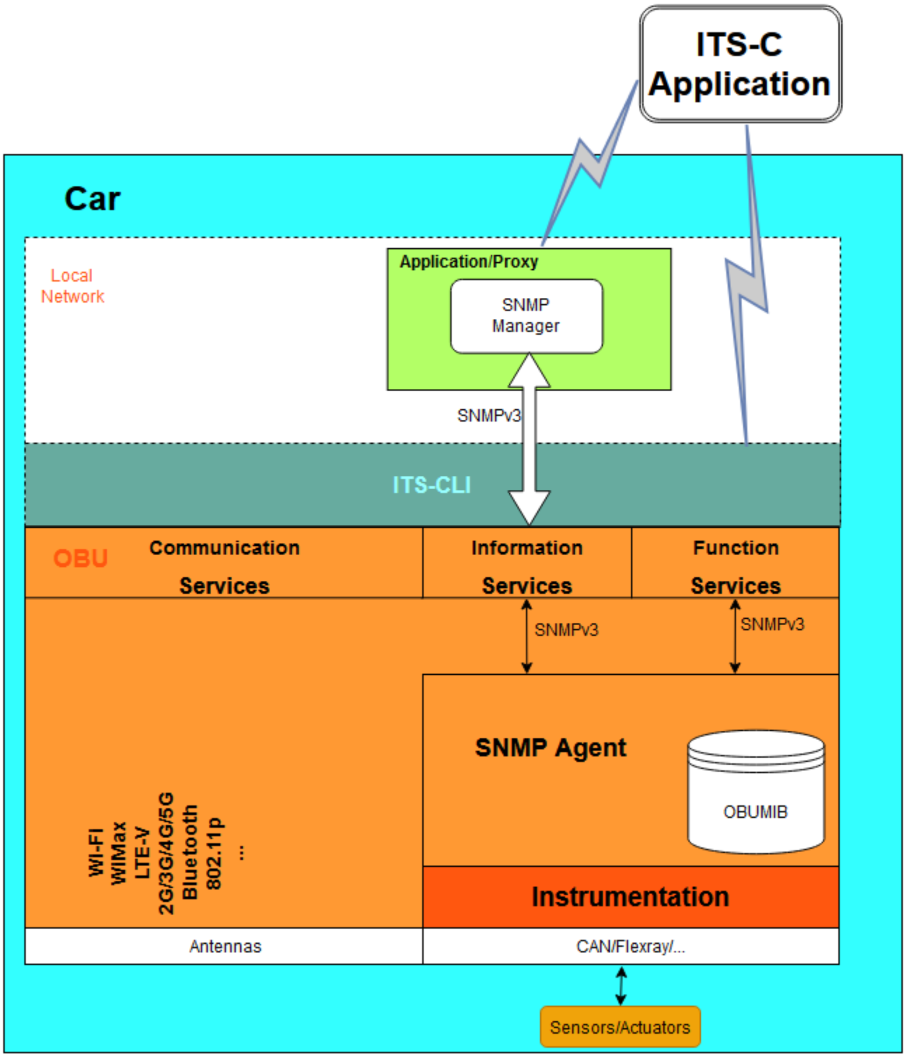

Figure 11: System Architecture

3.2 MANAGEMENT INFORMATION BASE

The *MIB* that was developed for this project will contain tabular objects that were grouped up based on their function:

- System Group- This group will contain all the objects/tables related to vehicle/OBU metadata.
- Sensor Group- This group will contain all objects/tables related to sensor readings.
- Error Group- This small group will contain all objects/tables related to error management.
- Actuator Group- This small group will contain all objects/tables related to actuators.

Note: For simplicity sake certain nodes were not implemented in the prototype and as such, the *OID*s defined in this dissertation will differ from the *OID*s used in the prototype. This does not invalidate the results obtained with this prototype.

3.2.1 *System OBU Group*

This group will contain all information regarding the installed *OBU*, for example: Date of installation, Version Number, Runtime, alongside capabilitiesTable and connectedVehiclesTable, these objects will serve mostly as a way to store the vehicles' "metadata", this includes capabilities, vehicle ID, mileage, age, country of origin, etc. These tables will be populated based on information provided by the manufacturer on a per vehicle basis and as such it's the least developed part of this project.

systemOBUGroup		
Object	SMI type	OID
numberOfCapabilties	INTEGER	1.3.6.1.3.8888.1.1
capabilitiesTable	Table	1.3.6.1.3.8888.1.2
numberOfConnectedVehicles	INTEGER	1.3.6.1.3.8888.1.3
connectedVehiclesTable	Table	1.3.6.1.3.8888.1.4
sysOBUDateandTime	OBUDateandTime	1.3.6.1.3.8888.1.5
sysOBUNMonRequest	Unsigned32	1.3.6.1.3.8888.1.6
sysOBUNEventRequest	Unsigned32	1.3.6.1.3.8888.1.7
sysOBUNConfRequest	Unsigned32	1.3.6.1.3.8888.1.8
sysOBUNErrors	Unsigned32	1.3.6.1.3.8888.1.9
sysOBUVehicleID	String[...]	1.3.6.1.3.8888.1.10
sysOBUDistanceType	Integer/Enum	1.3.6.1.3.8888.1.11
sysOBUTotalDistance	Unsigned32	1.3.6.1.3.8888.1.12
sysOBUCountry	Integer/Enum	1.3.6.1.3.8888.1.13

Table 4: systemOBUGroup

Capabilities Table

This table will list the vehicle/*OBU* capabilities, including all available services.

capabilitiesTable		
Object	SMI type	OID
CapabilitiesID	Unsigned32	1.3.6.1.3.8888.1.2.1..1
SetOfCapabilitiesID	Unsigned32	1.3.6.1.3.8888.1.2.1..2
SpecificCapabilitiesID	Unsigned32	1.3.6.1.3.8888.1.2.1..3
CapabilityValue	String[...]	1.3.6.1.3.8888.1.2.1.4

Table 5: capabilitiesTable

Connected Vehicles Table

This table will be store all information regarding the vehicle itself and the Id of the entities it's connected to.

connectedVehiclesTable		
Object	SMI type	OID
VehicleID	Unsigned32	1.3.6.1.3.8888.1.4.1.1
LocalID	String[...]	1.3.6.1.3.8888.1.4.1.2
GlobalID	String[...]	1.3.6.1.3.8888.1.4.1.3
AssociatedOBUorRSU	String[...]	1.3.6.1.3.8888.1.4.1.4
LocalOrRemote	Integer/Enum	1.3.6.1.3.8888.1.4.1.5
Capabilities	Unsigned32	1.3.6.1.3.8888.1.4.1.6

Table 6: vehiclesTable

3.2.2 *Sensor Group*

This group will consist of all tables that are in any way related to reading and storing data obtained from sensors. It will include the both the tables containing the actual readings as well as any auxiliary table that aid in the reading/understanding the recorded value and, lastly, any table related to the request itself.

Map Type Table

The first part of this group are the tables that identify the sensors/actuators in the vehicle. Each one of these will be assigned an unique entry, identified by an ID, where the name, interface, description, precision, maximum delay, maximum sampling frequency, unit and proprietary ID are stored. As a way to optimize memory usage, both the description and unit will be stored in auxiliary tables. This is done since it's possible for sensors/actuators to share the same description and/or unit, in which case we only need to indicate the index of the table where the description/unit is stored instead of repeating the same data thus saving on memory usage.

This table will map proprietary manufacturers *ECU*s into generic types defined on genericTypesTable and sampleUnitsTable. In essence it will be used to identify which sensor/actuator of which *ECU* is being read as well as the unit in which the values are being stored.

mapTypeTable		
Object	SMI type	OID
MapTypeID	Unsigned32	1.3.6.1.3.8888.2.10.1
ProprietaryTypeID	Unsigned32	1.3.6.1.3.8888.2.10.2
GenericMapTypeID	Unsigned32	1.3.6.1.3.8888.2.10.3
SampleUnitMapID	Unsigned32	1.3.6.1.3.8888.2.10.4
Precision	Integer	1.3.6.1.3.8888.2.10.5
MaxSamplingFrequency	Unsigned32	1.3.6.1.3.8888.2.10.6
MaxMapDelay	OBUDateandTime	1.3.6.1.3.8888.2.10.7
DataSource	Integer/Enum	1.3.6.1.3.8888.2.10.8
InterfaceSource	Integer/Enum	1.3.6.1.3.8888.2.10.9

Table 7: mapTypeTable

Sample Units Table

This table will be used to identify which unit a stored value was recorded in, i.e "Km/h". This will define the coding algorithm for the Precision object on the mapTypeTable. This table is used so that a given unit isn't getting repeated in memory for every sensor that uses it, instead only the ID of the entry that stores that unit will be repeated.

sampleUnitsTable		
Object	SMI type	OID
SampleUnitID	Unsigned32	1.3.6.1.3.8888.2.14.1.1
UnitDescription	String[...]	1.3.6.1.3.8888.2.14.1.2

Table 8: sampleUnitsTable

Generic Types Table

This virtual/enumeration table will contain a generic description of the type of data a certain sensor is generating, for example: "Vehicle velocity". This table exists so that an user can know what is the function of every sensor/actuator.
Much like sampleUnitsTable, this table is used to optimize memory usage since there can be more than one sensor/actuator sharing the same description.

genericTypesTable		
Object	SMI type	OID
GenericTypeID	Unsigned32	1.3.6.1.3.8888.2.11.1..1
TypeDescription	String[...]	1.3.6.1.3.8888.2.11.1.2

Table 9: genericTypesTable

These first three tables, mapTypeTable, genericTypesTable and sampleUnitsTable, will be populated on start-up based on the contents of an CAN Bus DataBase (*DBC*) file and are used to provide information regarding the sensor whose data is being stored.

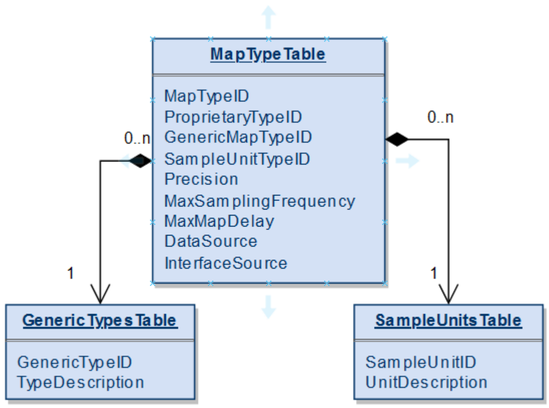

Figure 12: mapTypeTable and its auxiliary tables

This next set of tables will be the ones that will store all information regarding a specific request. Due to this factor, they are the most important tables in the whole *MIB* since the whole functionality of monitoring sensors is dependant on them. When developing this solution there were two lines of thought that were considered.

- Sensor reading driven approach
- Request driven approach

The first one is probably the simplest since the idea would be that all data read from *CAN* interface would be stored in the *MIB* so that outside entities could access it at will, however it didn't take long to realize how inefficient and resource consuming this approach would be so it was scraped.

The second approach was much more sensible since the *MIB* would only store data that other entities have requested. While this approach had a slight downside, since it would be more complex to implement, its advantages far outweighed the disadvantages.
In this approach, the first step lays in outside entities creating entries in a table so that whenever a new message arrives to the *OBU* the entries of that table would be checked against the source sensors of the message, if there's no request on that specific sensor the message would be ignored, otherwise the relevant data would be added to the *MIB*. The entity that made the original request would then only need to access the data related to its request.

Request Monitoring Data Table

The following table was the one that was created to fullfill the role of storing all requests made to the system. This table will store a variety of usefull information relating to requests, from the last sample that was recorded for it, LastSampleID, to how many samples are associated to it, NOfSamples. These requests can be limited either by a timestamp or by a maximum number of samples, EndTime and MaxNOfSamples respectively. Additionally, the current status of the request is also stored, which will aid in identifying which requests are active or not, it will also store the *SNMP* username of the user that made the request in RequestUser. Lastly, it will store the IDs of any related tables in the *MIB*.

requestMonitoringDataTable		
Object	SMI type	OID
RequestID	Unsigned32	1.3.6.1.3.8888.2.2.1.1
RequestControlID	Unsigned32	1.3.6.1.3.8888.2.2.1.2
RequestMapID	Unsigned32	1.3.6.1.3.8888.2.2.1.3
RequestStatisticsID	Unsigned32	1.3.6.1.3.8888.2.2.1.4
SavingMode	Integer/Enum	1.3.6.1.3.8888.2.2.1.5
SamplingFrequency	Unsigned32	1.3.6.1.3.8888.2.2.1.6
MaxDelay	Unsigned32	1.3.6.1.3.8888.2.2.1.7
StartTime	OBUDateandTime	1.3.6.1.3.8888.2.2.1.8
EndTime	OBUDateandTime	1.3.6.1.3.8888.2.2.1.9
WaitTime	OBUDateandTime	1.3.6.1.3.8888.2.2.1.10
DurationTime	OBUDateandTime	1.3.6.1.3.8888.2.2.1.11
ExpireTime	OBUDateandTime	1.3.6.1.3.8888.2.2.1.12
LastSampleID	Unsigned32	1.3.6.1.3.8888.2.2.1.13
NOfSamples	Counter32	1.3.6.1.3.8888.2.2.1.14
MaxNOfSamples	Unsigned32	1.3.6.1.3.8888.2.2.1.15
LoopMode	Integer/Enum	1.3.6.1.3.8888.2.2.1.16
Status	Integer/Enum	1.3.6.1.3.8888.2.2.1.17
RequestUser	String[...]	1.3.6.1.3.8888.2.2.1.18

Table 10: requestMonitoringDataTable

RequestStatisticsDataTable

This table was created since there may be a need to provide ”on the fly” statistical information regarding a specific request. Not all requests will need this feature and as such it is entirely optional, its existence being decided when a request is being created. These statistics include minimum recorded value, maximum recorded value, average recorded value, how many samples were recorded and for how long has the request been active.

requestStatisticsDataTable		
Object	SMI type	OID
StatisticsID	Unsigned32	1.3.6.1.3.8888.2.6.1.1
DurationTimeStatistics	OBUDateandTime	1.3.6.1.3.8888.2.6.1.2
NOfSamplesStatistics	Counter32	1.3.6.1.3.8888.2.6.1.3
MinValue	Unsigned32	1.3.6.1.3.8888.2.6.1.4
MaxValue	Unsigned32	1.3.6.1.3.8888.2.6.1.5
AvgValue	Unsigned32	1.3.6.1.3.8888.2.6.1.6

Table 11: requestStatisticsDataTable

Request Control Data Table

The decision of using a request driven approach did bring with it an issue that needed to be resolved where the same entity could create duplicate requests on the same sensor.

To fix this issue, among others, the following table was created. The idea is simple, every request in requestMonitoringDataTable will have a related entry in this new table, this entry can be shared among multiple requests only if those requests are made on the same sensor. In essence, the entries in this table serve as a sort of summary to the various requests made to the system and serves as a quick and easy way to know which sensors are currently being monitored and the status of that monitoring. Both this table and requestMonitoringDataTable will be used to prevent duplicate requests on the same sensor by the same entity.

requestControlDataTable		
Object	SMI type	OID
RequestControlID	Unsigned32	1.3.6.1.3.8888.2.4.1.1
RequestControlMapID	Unsigned32	1.3.6.1.3.8888.2.4.1.2
SettingMode	Integer/Enum	1.3.6.1.3.8888.2.4.1.3
CommitTime	OBUDateandTime	1.3.6.1.3.8888.2.4.1.4
EndControlTime	OBUDateandTime	1.3.6.1.3.8888.2.4.1.5
DurationControlTime	OBUDateandTime	1.3.6.1.3.8888.2.4.1.6
ExpireControlTime	OBUDateandTime	1.3.6.1.3.8888.2.4.1.7
ValuesTableID	Unsigned32	1.3.6.1.3.8888.2.4.1.8
StatusControl	Integer/Enum	1.3.6.1.3.8888.2.4.1.9

Table 12: requestControlDataTable

Samples Table

The next issue that needed solving was how sensor data was going be stored in the *MIB*, how would the entries in requestMonitoringDataTable point to it and how to prevent duplicate sensor readings from being added to the *MIB* when there are multiple requests on the same sensor. While several ideas were considered the one that was most promising was a solution similar to linked lists where every time a new reading from a sensor was added to the *MIB*, all active requests on that sensor would be changed so that LastSampleID points to this new reading. Then, to "link" all those sensor readings into a linked list, this new entry only needs to store the index of the entry it just replaced in LastSampleID.

To prevent duplicate sensor readings from being added to the *MIB* when there are multiple requests on the same sensor a simple checksum is calculated. This checksum, alongside MapTypeSamplesID, is used to check if a particular sample was already added to the *MIB*. If it isn't in the system it can be added, otherwise the only thing that needs changing is LastSampleID on all requests made for to this sensor. This way duplicate entries can be prevented, thus reducing memory usage.

In essence, this table will be used to store all sensor data relating to active requests in the *MIB*. Among the information it stores is an index that points to an entry requestControlDataTable identified by RequestSampleID, as mentioned before it also stores the ID of the last sample recorded from the same sensor, which can then be used to group all recorded samples relating to a specific sensor, additionally, it stores the ID that points to an entry in mapTypeTable which will help understand the data being stored and, finally, it also stores a simple checksum which is created based on a timestamp and the name of the *ECU* that sent the data which will allow samples from obtained from different sensors but from the same *ECU* in the same message, to be identified as such.

samplesTable		
Object	SMI type	OID
SampleID	Unsigned32	1.3.6.1.3.8888.2.8.1.1
RequestSampleID	Unsigned32	1.3.6.1.3.8888.2.8.1.2
TimeStamp	OBUDateandTime	1.3.6.1.3.8888.2.8.1.3
SampleFrequency	Unsigned32	1.3.6.1.3.8888.2.8.1.4
PreviousSampleID	Unsigned32	1.3.6.1.3.8888.2.8.1.5
SampleType	Integer	1.3.6.1.3.8888.2.8.1.6
SampleRecordedValue	Integer	1.3.6.1.3.8888.2.8.1.7
MapTypeSamplesID	Unsigned32	1.3.6.1.3.8888.2.8.1.8
SampleCheckSum	String[...]	1.3.6.1.3.8888.2.8.1.9

Table 13: samplesTable

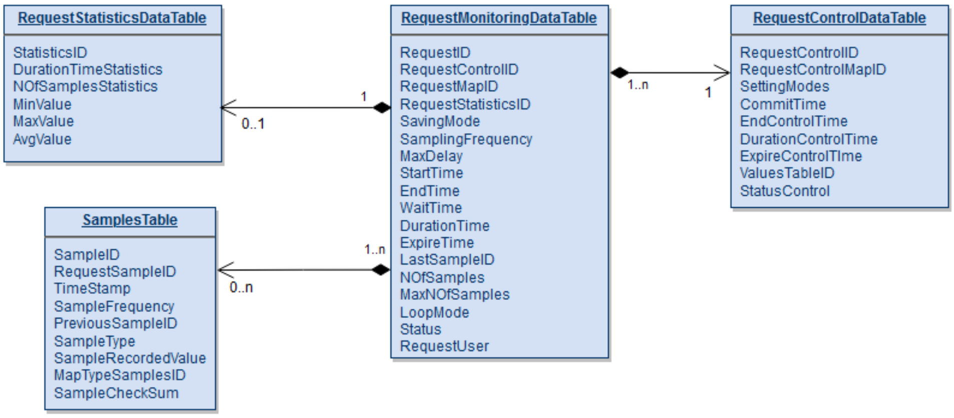

Figure 13: requestMonitoringDataTable and its relationships

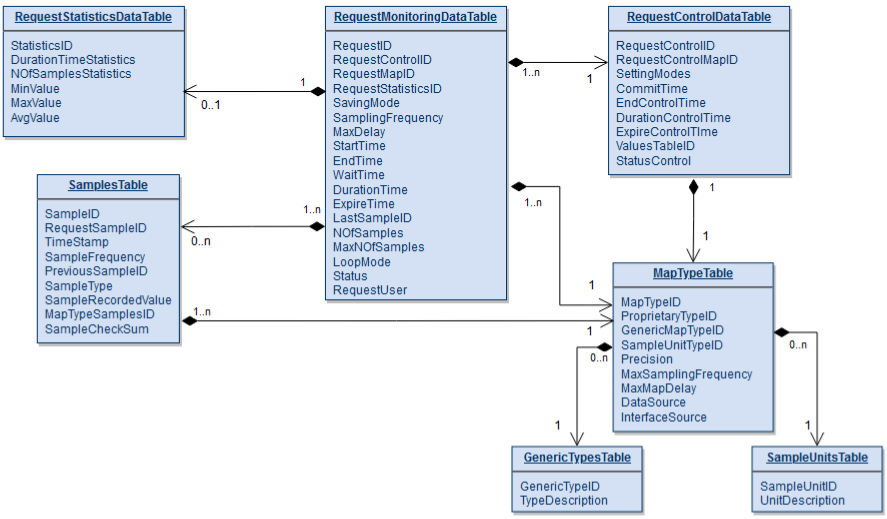

Figure 14: Sensor Group

3.2.3 *Error Group*

This small group will be used to help diagnose the reasons why certain requests were not set. These reasons can range from invalid mapTypeTable ID to duplicate requests on the same object by an user. It will be similar to mapTypeTable in the sense that there's a main table where the errors are added and and auxiliary table where the descriptions of the errors are stored. This way if there several instances of the same error the same description won't be repeated over and over again, only the index that points to that description will be repeated.

Error Table

This first table is where the errors that are currently active will be stored so as to allow the user to know why their request was not set. It will store a timestamp of the error as well as an expire time to delete this error entry. It will also store the username of the user whose request triggered the error as well as the errorDescriptionTable ID.

errorTable		
Object	SMI type	OID
errorID	Unsigned32	1.3.6.1.3.8888.3.2.1.1
errorTimeStamp	OBUDateandTime	1.3.6.1.3.8888.3.2.1.2
errorDescriptionID	Unsigned32	1.3.6.1.3.8888.3.2.1.3
errorUser	String[...]	1.3.6.1.3.8888.3.2.1.4
errorExpireTime	String[...]	1.3.6.1.3.8888.3.2.1.5

Table 14: errorTable

Error Description Table

This auxiliary table will store a simple description of the error as well as an error code. It is populated on start-up based on the contents of a text file.

errorDescriptionTable		
Object	SMI type	OID
errorDescrID	Unsigned32	1.3.6.1.3.8888.3.4.1.1
errorDescr	String[...]	1.3.6.1.3.8888.3.4.1.2
errorCode	Unsigned32	1.3.6.1.3.8888.3.4.1.3

Table 15: errorDescriptionTable

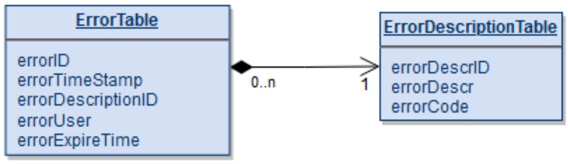

Figure 15: Error Group

3.2.4 *Actuator Group*

This final group is intended to allow actuators to be activated/deactivated with as quickly as possible, for example if the lead vehicle in a platoon were to break, all other vehicles in that same platoon need to break with minimal delay, so as to avoid a crash. This could be done by sending a message to the *OBU* of every vehicle in the platoon, which in turn would send *CAN* messages to the relevant nodes of the vehicle's *CAN* network to activate the brake actuators.

Sadly, due to proprietary reasons, there's not much insight into how these *CAN* messages would look like, as such, this portion of the *MIB* will probably require some changing before it's implemented in a real vehicle.

Command Template Table

This table is populated on start-up based on the contents of a text file. It includes a short description of what the command will do, the target node, in hexadecimal, and a template of the command, also in hexadecimal.
e.g. AA BB CC DD EE ** ** H1, where '*' indicate where the user input will be added.

commandTemplateTable		
Object	SMI type	OID
commandTemplateID	Unsigned32	1.3.6.1.3.8888.4.2.1.1
commandDescription	String[...]	1.3.6.1.3.8888.4.2.1.2
targetNode	String[...]	1.3.6.1.3.8888.4.2.1.3
commandTemplate	String[...]	1.3.6.1.3.8888.4.2.1.4

Table 16: commandTemplateTable

Command Table

This final table will store all commands that were not yet sent to the *CAN* network. It contains the user input, an ID to commandTemplateTable and the *SNMP* username of the user that sent the command.

The command will first be validated, then the *CAN* message will be created with the user input and the template, following that, the message will be sent over the *CAN* network to the target node and finally the entry will be deleted from commandTable.

If validation or the transmission of the *CAN* message fails, a new entry in errorTable will be created and the entry in commandTable will be deleted.

commandTable		
Object	SMI type	OID
commandID	Unsigned32	1.3.6.1.3.8888.4.4.1.1
templateID	Unsigned32	1.3.6.1.3.8888.4.4.1.2
commandInput	INTEGER	1.3.6.1.3.8888.4.4.1.3
commandUser	String[...]	1.3.6.1.3.8888.4.4.1.4

Table 17: commandTable

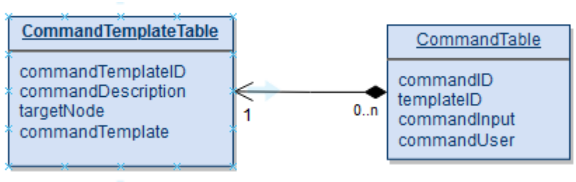

Figure 16: Actuator Group

These three groups can now be merged together into a full *MIB* providing us with a full view of the whole tree.

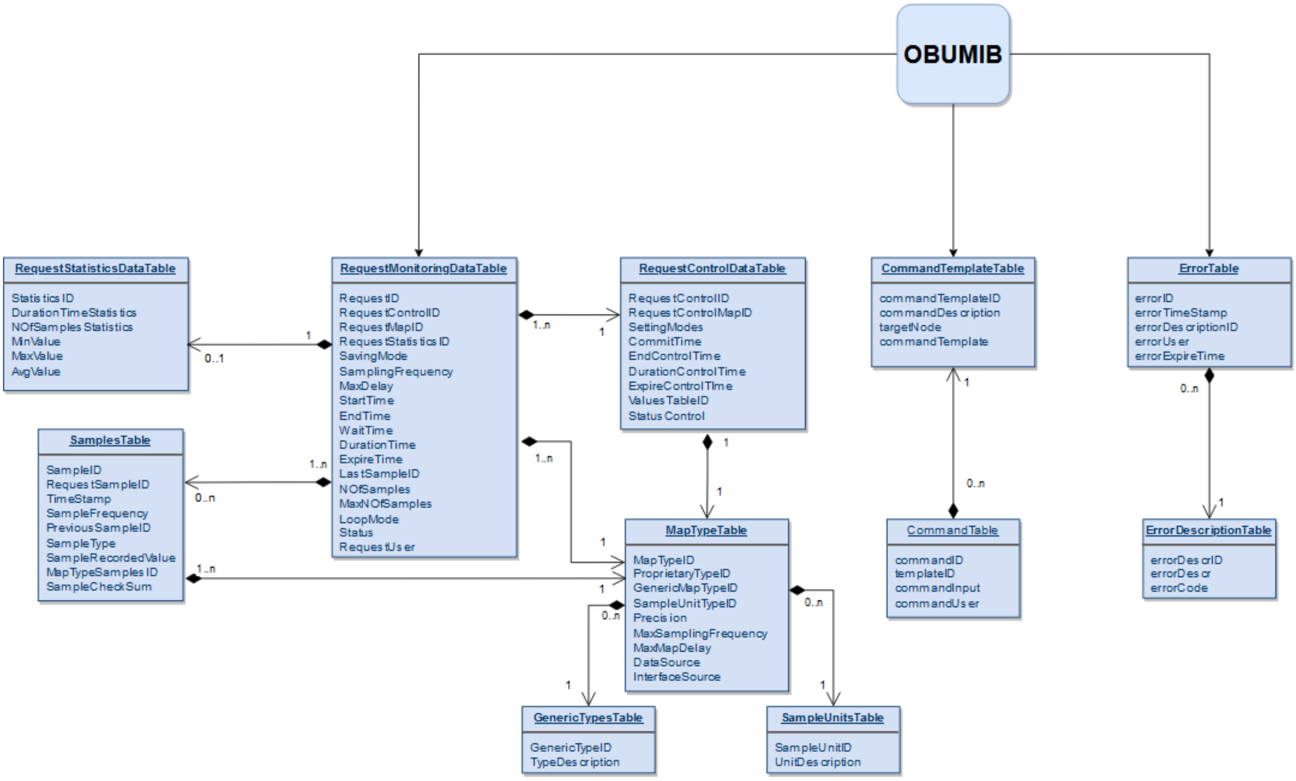

Figure 17: Full *MIB*

The following tables are presented as an example, in them we have three requests and some of the samples recorded for them.

requestMonitoringDataTable							
ReqID	ReqControlID	ReqMapID	ReqStatID	...	LastSampleID	NOfSamples	...
0	0	199	2	...	18	10	...
1	1	50	3	...	8	8	...
2	0	199	0	...	18	4	...

Table 18: Example requestMonitoringTable

- Request 0 - Request made for sensor whose ID is 199, it has statistics enabled, index nº2, and 10 recorded samples. Its latest sample is sample nº 18.
- Request 1 - Request made for sensor whose ID is 50, it has statistics enabled, index nº3, and 8 recorded samples. Its latest sample is sample nº 8.
- Request 2 - Request made on the same object as Request 0, confirmed by the fact that "ReqMapID" and "ReqControlID" are equal, it does not have statistics enabled and has 4 recorded samples. Much like request 0, its latest sample is nº 18.

samplesTable					
sampleID	...	PreviousSampleID	...	SampleRecordedValue	...
...	...	...	...	...	...
15	...	14	...	2000	...
16	...	15	...	2500	...
17	...	16	...	2450	...
18	...	17	...	2300	...

Table 19: Example samplesTable

If, for example, we wanted to get all the samples related to request 2 all we have to do is go to sample nº18 and then, based on its ”PreviousSampleID”, go to the sample indicated by it. By repeating this process 4 times, the number of samples indicated by ”NOfSamples”, we can obtain all samples related to this request. In this case the samples related to request 2 are:

- Sample 18 - 2300
- Sample 17 - 2450
- Sample 16 - 2500
- Sample 15 - 2000

To aid in the understanding of this data we can then use ”ReqMapID” to check ”genericTypeID” and ”sampleUnitID” related to this request. These two indexes will point to the unit description and type description of the request which may, for example, be ”rpm” and ”Actual engine speed which is calculated over a minimum crankshaft angle of 720 degrees divided by the number of cylinders.” respectively. This indicates that both Request 0 and Request 2 are being used to collect/monitor rpm data.

3.2.5 *Structure of Management Information*

With all tables, and their contents, defined all that is needed is to do is convert that information into an *SMI* specification. As an example, a portion of the full *MIB* will be presented, where the column ”SavingMode”, of the table requestMonitoringDataTable is defined.

In this example, we can verify that ”SavingMode” is an object that can be read, written or created and that this object is of the type ”Integer” with two valid options, 0 and 1, meaning ”permanent” and ”volatile” respectively. Additionally we can also verify that ”SavingMode” is the 5th column of the table requestMonitoringDataTable.

Max-Access Value	Description
read-create	Object can be read, written or created
read-write	Object can be read or written
read-only	Object can only be read
accessible-for-notify	Object can be used only using SNMP notification (SNMP traps)
not-accessible	Used for special purposes

Table 20: Max-Access values

```
 ...
savingMode OBJECT-TYPE
  SYNTAX  INTEGER {
       permanent(0),
       volatile(1) }
  MAX-ACCESS read-create
  STATUS  current
  DESCRIPTION
    "This object will identify the mode in which a specific request will be saved
    "
  -- 1.3.6.1.3.8888.1.1.5
::= { requestMonitoringDataEntry 5 }
...
```

Listing 3.1: Partial MIB Specification

The full *MIB* definition developed for this project can be found in Full MIB Specification.

3.3 SUMMARY

In this chapter, an overview of the potential use cases for this solution as well as its objectives was given followed by an overview of the the system architecture. Additionally, the most important tables of the *MIB* were presented, including an introduction on how these tables are defined in the *SMI* specification and the reasoning behind the specification of those tables. Finally a brief example of how these tables can be used to monitor sensor data was given.

4

PROTOTYPE DEVELOPMENT & TESTING

In this chapter a summary of the development steps of the prototype created for this project will be presented starting with the choice of language and API, followed by a short explanation into how a *CAN* message are decoded. Following this a more in-depth dive into how the sub-agent processes both *CAN* messages and manager requests will be given.
Additionally, regarding the manager, its functionalities will be presented alongside the reasoning behind the choice of protocol that will provide authentication and privacy to the data being transmitted, followed by a short explanation into how the *SNMP* manager communicates with the sub-agent.
Finally, the results of this solution will be presented alongside a comparison with *OBD-II*.

The first step in this type of development is deciding on the programming language that is to be used. In this case the chosen programming language for this phase of the project is C/C++, since its response times were roughly $\frac{3}{4}$ those in Java while also being less computationally intensive. Additionally C/C++ is the preferred language to develop software modules within *OBU*s which makes it the best choice for this phase of the project.

Response Time (ms)				
	1attr/method	Nattrs/method	NMOs/method	1MO/method
C/C++	0.8	1	6	37
Java	1	2	8	45

Table 21: Comparison between C++ and Java response times [32]

- 1attr/method - Response times for retrieving a single attribute from an object.
- Nattrs/method - Response times for retrieving all eight object attributes from an object.
- NMOs/method - Response times for retrieving all objects from all *TCP* connections (40 in total) using GETBulk.
- 1MOs/method - Response times for retrieving all objects from all *TCP* connections (40 in total) using GETNext.

The next step in the development was creating the *MIB*, which was already presented in the previous chapter. This *MIB* should allow for sensor data to be read, actuators to be activated/deactivated, any requested data to be stored and, finally, all information regarding the vehicle or its *OBU* to be stored.

4.1 NET-SNMP

The *API* that was chosen for the development of this phase of the project is included in the Net-SNMP application suite. This suite includes:

- Command-line applications to retrieve and manipulate information from *SNMP* capable devices.
- Graphical *MIB* browser (tkmib).
- *SNMP* Agent (snmpd) that supports Agent Extensibility (*AgentX*) protocols.
- Library for developing new *SNMP* applications with C and Perl *API*s.

Lastly this suite also includes a tool (mib2c) that is designed to take a portion of the *MIB* tree (as defined by a *MIB* file) and generate the template C code necessary to implement the relevant management objects within it.

Through this tool every table defined in 3.2 will be converted to C code, so that our custom *AgentX* SubAgent can handle communication with the manager and manage all of our tables.

4.2 GENERATING VIRTUAL CAN MESSAGES

Before creating the *SNMP* sub-agent, a simple program that can write *CAN* messages into a virtual *CAN* interface was created. Since the *SNMP* sub-agent will also be connected to this interface it will be able to receive *CAN* messages in real time much like it would happen in real life. This simple program will read *CAN* messages from a .trc file containing raw *CAN* bus logs and write the contents of that file into the previously created *CAN* interface. To do this two simple structures were created.

```c
typedef struct can
{
  double timestamp;
  unsigned char *id;
  int dlc;
  unsigned char data[MAXDATALENGTH];
} can;
typedef struct canlist
{
  int capacity;
  int current;
  can *list;
} canlist;
```

These structures are populated based on the contents of the log file and then iterated through to write those messages into the *CAN* interface. Naturally, these *CAN* messages are encoded by the *ECU*s prior to being sent, as such, the agent will need to decode them so they can be stored in the *MIB*. Additionally the timestamp of the original *CAN* messages can be used to replicate the time interval between the arrival of two consecutive messages.

4.3 SNMP AGENT

One feature within *SNMP* that will be used in this phase of the project is that of sub-agents. These are independent *SNMP* daemons, that are transparent to the network management station[33], that register to the master agent the *MIB* modules they want to take care of[34], additionally, since the *OBU* may need to support other *MIB*s besides the one developed above, including the MIB-II Standard, it is recommended to manage custom *MIB*s, through sub-agents especially since that is the only way to make use of the Net-SNMP API.

In the Net-SNMP API this feature is handled by an *SNMP* agent (snmpd) that supports *AgentX* protocols. From this point on, the terms "agent" and "sub-agent" may be used interchangeably but they both indicate the same software module that was developed and implements the custom "OBUMIB".

When it comes to actually building the *SNMP* sub-agent in C code, the Net-SNMP website does come with some very useful tutorials and example code snippets that were used in the development of the sub-agent [35], which can, at its most basic level, be divided into 3 parts:

- Registering Sub-Agent with the master Agent.
- Registering and initializing the MIB tables the Sub-Agent will handle.
- A main loop which will handle the requests by the manager.

```
int agentx_subagent=1; /* change this if you want to be a SNMP master agent */
...
if (agentx_subagent) {
  /* make us a agentx client. */
  netsnmp_ds_set_boolean(NETSNMP_DS_APPLICATION_ID, NETSNMP_DS_AGENT_ROLE, 1);
}
...
/* initialize tcpip, if necessary */
SOCK_STARTUP;
init_agent("example-demon");
/* initialize mib code here */
...
init_snmp("example-demon");
...
keep_running = 1;
...
/* your main loop here... */
while(keep_running) {
  agent_check_and_process(0); /* 0 == don't block */
}
snmp_shutdown("example-demon");
SOCK_CLEANUP;
return 0
```

Listing 4.1: Sample sub-agent code

The next step was to convert the *MIB* tables into C code, this was done by another tool included in the the Net-SNMP API called mib2c. To use it we first needed to add the *MIB* file to the *SNMP* agent[36] and then simply use the command below to create a skeleton .c code and accompanying header file.

```
mib2c -c mib2c.array-user.conf <TargetTable>
```

It should be noted that since both vehiclesTable and systemOBUGroup fall outside the purview of this project they won't be converted into C code. However, if and when this solution is deployed in the real world, these two tables should be reintegrated as they will contain information that is valuable in the real world.

Included in these files is the function needed to initialize a table, usually named init_<targetTable>() which will be run by the sub-agent to initialize and register the table in the *SNMP* agent.

In the main loop, the sub-agent will need to manage the tables and user requests. Since handling user requests is done by the API with the function *agent_check_and_process*(), we could focus on managing the tables themselves. This included:

- Decoding *CAN* Messages.
- Managing Requests.
- Storing Sensor Readings.

4.3.1 *Decoding* CAN *Messages*

Much like in real life, a raw *CAN* message is not human readable and will first need to be decoded before it can be stored on the *MIB*. The *CAN* messages we focused on are called Data Frames, since these are the ones that include sensor readings.

These *CAN* messages usually contain the following information:

- Timestamp- Timestamp of when message was transmitted/received.
- ID- ID of component/part that transmitted the message.
- DLC- Size of data being transmitted, in bytes.
- Data- Data that is being transmitted.

```
40,425 18FFB5F2 8 3A 82 FF 5C C6 80 11 05
40,431 18F005F6 8 FF FF FF FB FF FF 20 50
40,431 14FFB4F6 8 00 FF 16 F0 FF FF FF FF
40,433 18FFB6F2 8 00 00 00 00 F1 12 FF FF
```

Figure 18: Example CAN messages

As it stands these messages didn't provide a lot of information since both the ID and Data fields are in hexadecimal and as such they needed to be decoded to be usable.

Sadly, due to proprietary reasons, decoding these fields is not a matter of converting the Hex data to Strings or Integer, instead requiring a file that indicates the rules on how to get human readable information from these messages [37].

Such a file usually comes in a *DBC* format and manufacturers do not provide the end user with this information, requiring projects like opendbc to have a glimpse on how data is decoded. Unfortunately even projects like opendbc have limited coverage over *DBC* files with sometimes faulty or even incorrect information being included in them. Nevertheless a complete CAN 2.0B file DBC was found, in this case for the J1939 standard which uses 29-bit identifiers and is used on heavy-duty vehicles [38].

Inside such a file, a series of lines similar to figure 19 can be found. Lines starting with BO_ denote a message and contain the following:

- *DBC* ID: ID in decimal.
- Name: Name of *ECU*.
- Length: Length of data in bytes.
- Sender: Name of the transmitting node, Vector__XXX if no name is available.

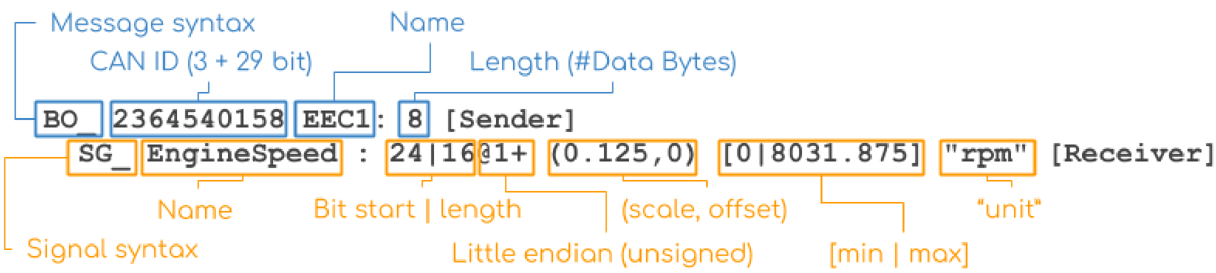

Figure 19: Example information contained in DBC files. From [37]

Below these Messages, there will be one or more signals and the rules to decode them. These signals are denoted by SG_ . These lines will contain the following fields:

- Name: Name of value that is being recorded. For example "Engine Temperature".
- Bit Start: The bit start counts from 0 and marks the start of the signal in the data payload.
- Length: The bit length is the signal length.
- Endian: The @1 denotes little-endian/Intel, @0 denotes big-endian/Motorola.
- Signed: The value type is denoted as unsigned by an '+', '-' denotes signed.
- Scale,Offset: The (scale,offset) values are used in the physical value linear equation.
- Min,Max,Unit: The [min|max] and unit are optional meta information.
- Receiver: The receiver is the name of the receiving node (again, Vector__XXX is used as default).

Starting for example with the message "0CF00400 29 7D 87 68 13 00 F4 87", we would first need to match the ID "0CF00400" to an DBC ID. To do this the mask "0x1FFFFFFF" needs to be applied to the 32-bit DBC ID to get the 29-bit CAN ID, which can then be mapped against the message. This is done by going through all messages in the *DBC* file and applying the mask to the *DBC* ID.
$2364540158 \,\&\, 0x1FFFFFFF = 0CF00400$

Note: For 11-bit IDs a simple conversion from Hex to Decimal is needed to map the DBC ID to CAN ID.

With the DBC ID matched to the CAN ID, we could now identify the *ECU* that sent the message, in this case it was EEC1, along with all signals that can be used to decode the message. In this example the signal "EngineSpeed" in figure 19 will be used.

According to the decoding rules set for EngineSpeed, the relevant data starts on bit 24 and has a length of 16 bits, it is a signed value in little-endian with a scale of 0.125 and offset of 0. The minimum value is set to 0 and maximum to 8031.875. Finally the unit is "rpm".

We start then by extracting the relevant data from "29 7D 87 68 13 00 F4 87", which means it starts on byte 3 (when counting from 0) and has a length of 2 bytes. This equals to "68 13" however since the signal is in little-endian we needed to reorder the byte sequence to "13 68".

To obtain the physical value we had to first convert this Hexadecimal value to decimal, which is 4968, and then apply a linear conversion with the scale and offset.

$physical\_value = Offset + Scale * RawValue$

$621 = 0 + 0.125 * 4968$

Thus, we could now conclude that one of the signals included in the *CAN* message "0CF00400 29 7D 87 68 13 00 F4 87" can be decoded to "EEC1 is reporting that engine speed is at 621 rpm". This process can then be repeated for all signals in that message, thus allowing those results to be stored in the *MIB* at will.

Decoding CAN *Messages in C*

To properly decode *CAN* messages we first needed to load all necessary information from the *DBC* file into memory, this was done with the use of structs. More specifically, these 4 structs:

- BO_List -This struct will contain all messages contained in the *DBC* file.
- BO -This struct will contain all information regarding a particular message.
- SG_List -This struct will contain all signals of a particular message.
- SG -This struct will contain all information, e.g. decoding rules, of a particular signal.

These structs are created and populated on start-up and are later used to both decode *CAN* messages and populate mapTypeTable, genericTypesTable and sampleUnitsTable.

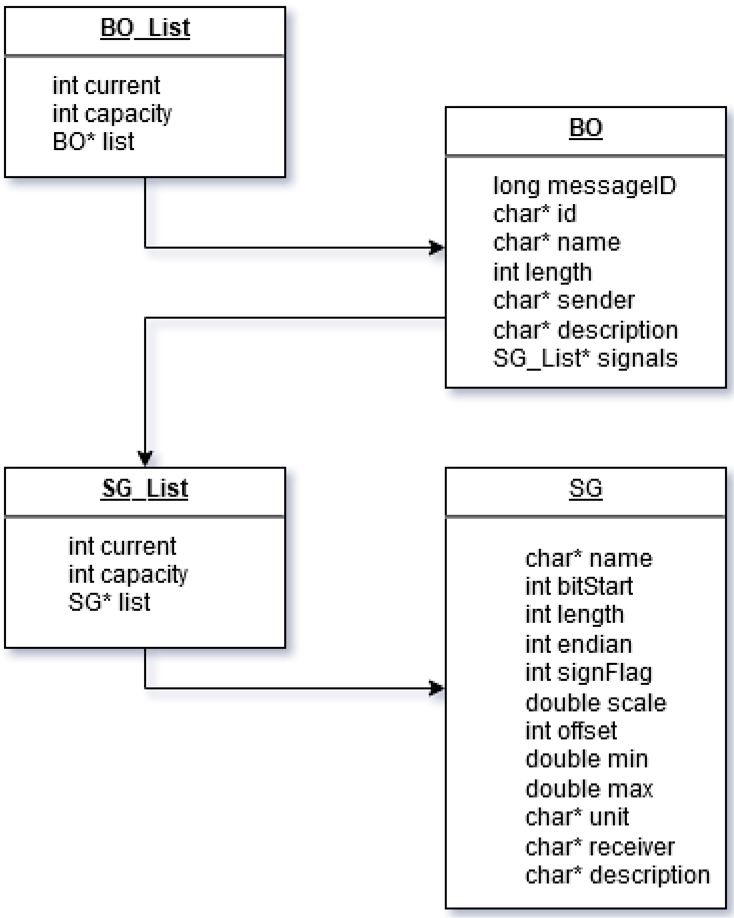

Figure 20: DBC Structs

With these structs loaded we could now start decoding *CAN* messages. This is handled by a child process created before the main loop in 4.1.

This child process is continuously reading data from the *CAN* network and whenever a new message is received, it will first decode it into another struct which is then sent to the parent process so that it can be processed. This struct is far simpler than the ones in figure 20, containing the name of the message, number of signals in message and several lists which will contain the decoded data.

```
char* name;          /*message name*/
int signals;         /*number of signals in message*/
char** signalname;   /*name of all signals in message*/
double* value;       /*decoded values of all signals in message*/
char** unit;         /*units used by all signals in message*/
```

Listing 4.2: Decoded *CAN* struct

When the parent process receives this struct, it will first create a timestamp and checksum and then, for every signal, check if there's any active request for it. If there's a request for a particular signal, and this reading hasn't yet been added into the *MIB*, then a new entry in samplesTable will be created while at the same time requestControlDataTable, requestMonitoringDataTable and requestStatisticsDataTable will be updated.

4.3.2 *Managing Requests*

To accomplish the main objectives of this phase of the project, that is, allowing users to read sensor data from the *CAN* network and allow user to change the state of actuators in real-time, we needed to allow the outside world to interact with the *MIB*.

This can be done by allowing users to, through a manager, use *SNMP* set commands on both commandTable and requestMonitoringDataTable. Since human error is always a possibility, some validation of user inputs must be included when processing entries from commandTable and requestMonitoringDataTable.

To achieve this two functions were created whose main objective is managing all entries, both in terms of validating user inputs and deleting said entries whenever their role is fulfilled.

The simpler of the two is checkActuators(), which is included in the files pertaining to commandTable. This function, besides sending out *CAN* message to target *ECU*s, will also be used to validate manager requests and ensure that the command was received by the target *ECU*.

To do this, it will first traverse commandTable in search of entries, when a entry is found it will check if the indicated templateID is in the *MIB*, as an entry of commandTemplateTable, if not, an entry in errorTable will be created followed by the deletion of this entry in commandTable.

Then it will check whether or not the command input is valid, at least as far as the target node is concerned, if it's invalid, an entry in errorTable will once again be created followed by the deletion of this entry in commandTable. If it's a valid input a *CAN* message will be created by adding the command input, in hexadecimal, to the command template and finally the message will be sent to the *CAN* network.

To ensure that the message was received successfully by the receiver the *CAN* standard does include a feature similar to *TCP* where the receiver will send an acknowledgement to the transmitter to confirm the receipt of the message, in *CAN* this is done by sending a data frame with a dominant bit during the *ACK* slot [39]. If such a data frame is not received, a new entry in errorTable will be created followed by the deletion of the entry in commandTable.

If the message transmission is successful, the entry in commandTable will now be deleted since it's no longer needed.

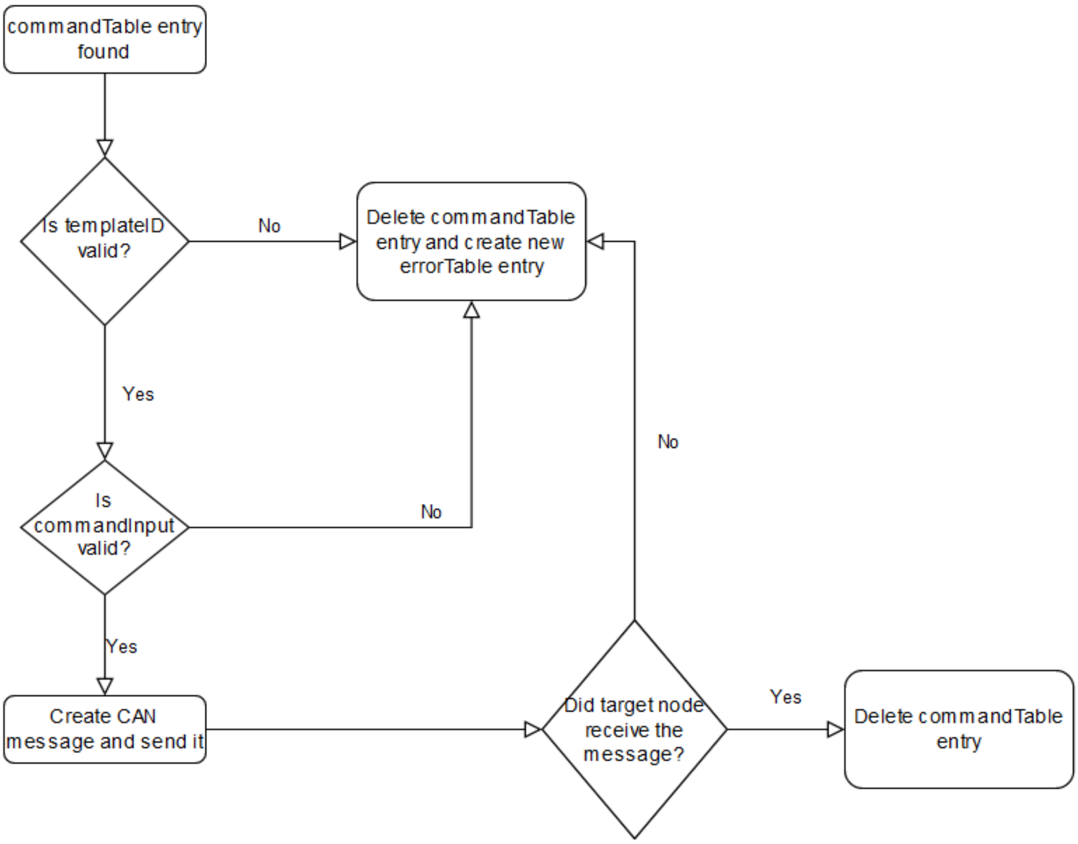

Figure 21: How an entry in commandTable is managed

The second function, checkTables(), is included in the files pertaining to requestMonitoringDataTable and besides ensuring that the manager requests are valid, this function will also change the status of an entry and create/delete any auxiliary entries to a specific request. These status are the following:

- 0 - Off.
- 1 - On.
- 2 - Set.
- 3 - Delete.
- 4 - Ready.

To further understand how an entry in requestMonitoringDataTable is validated we first needed to know what columns in the table does the user have access to. These were:

- requestMapID - Points to an entry in mapTypeTable which identifies the sensor whose data is being recorded.
- requestStatisticsID - Points to an entry in requestStatisticsDataTable, 0 indicates no entry should be created while 1 indicates the opposite.
- savingMode - Indicates whether this entry is volatile or permanent.
- startTime - Indicates when the monitoring should start.
- waitTime - Indicates an (optional) time to prepare the system to handle the request.
- durationTime - Indicates how long should the monitoring go on for.
- expireTime - Indicates how long after the request ended should data be kept in the *MIB*.
- maxNofSamples - Maximum number of samples to be recorded.
- loopMode - Whether or not the request should be restarted after deletion.

When a new request is set by a manager, its entry in requestMonitoringDataTable will have a status of 2 (SET). While in this state, the request will first be validated and, if successful, its status will be changed to 4 (READY).

If it fails validation an entry will be created in errorTable and the entry in requestMonitoringDataTable deleted.

Validating Manager Requests

To validate manager requests we needed to go to every single column set by the request and check whether or not it's valid. This is done via the following steps:

- 1-Check if startTime was set by the manager, if not use current system time as startTime.
- 2-Compare current system time with startTime and check if startTime is between 00:00:00 and 23:59:59.
- 3-Check if waitTime is between 00:00:00 and 23:59:59.
- 4-Check if durationTime is between 00:00:00 and 23:59:59.
- 5-Check if expireTime is between 00:00:00 and 23:59:59.
- 6-Check if maxNofSamples is greater than 0.
- 7-Check if savingMode is either 0 or 1.
- 8-Check if the manager who created this entry has already created another entry for the same object.
- 9-Check if requestStatisticsID is either 0 or 1.
- 10-Check if requestMapID points to a valid entry in mapTypeTable.
- 11-Check if loopMode is either 1 or 2.

If it fails at any of these points, the status of the request will be changed to 3 (DELETE) with an appropriate entry in errorTable being created explaining where the request failed.

Once validated, the status of the entry is changed to 4 (READY) and the next step is creating an entry in requestControlDataTable. Since there can be multiple requests on the same object we had to first check if there's an entry in requestControlDataTable with the same requestControlMapID as the requestMapID in the request. If an entry is found then there's no need to create a new entry in requestControlDataTable, requiring only updating the request to take into account the ID of that entry. If no entry is found, then a new entry in requestControlDataTable will be created, returning it's ID so it can be updated in the request.

The next step involves adding endTime to the entry, by adding startTime+waitTime+durationTime. This endTime indicates at which time should the entry be set to 0 (OFF).

Following that, startTime+waitTime is compared to current system time to know if the entry can be set to 1 (ON). If so, prior to changing status to 1, we had to first check whether or not the request will need an entry in requestStatisticsDataTable, creating one if it's indeed necessary. Finally the status will change to 1 (On).

In addition to these previously mentioned features, checkTables() also fullfill any roles regarding managing a request. These include:

- Updating in requestControlDataTable based on the status, savingModes and times of entries in requestMonitoringDataTable related to it.
- Changing status from 1 (ON) to 0 (OFF) when current system time is after endTime.
- Changing status from 0 (OFF) to 3 (DELETE) when current system time is after expireTime.
- Deleting an entry when it's status is 3 (DELETE).

This last feature will include multiple steps. First and foremost, the startTime of the entry that is to be deleted will be compared to the commitTime of its related entry in requestControlDataTable. Since the times in requestControlDataTable will always be equal to the oldest request on this object still in the *MIB*, we could find out if this entry is the oldest for this object or not.

If there's indeed an older request for the same object in the system then we don't need to delete any sample, and only need to delete the entry in requestMonitoringDataTable and requestStatisticsDataTable, if this last one exists.

If both startTimes are equal, it can mean one of three options:

- It's the oldest request among several others on the same object.
- It's the joint oldest request among several others on the same object.
- It's the only request on this object.

For the first of these situations, we must first find the second oldest request on this object and once we find it we must update the entry in requestControlDataTable to take into account that request. Following that we will delete all samples that predate this second oldest request and finally delete the entry in requestMonitoringDataTable and all those uniquely related to it.

For the second situation we only need to delete the entry in requestMonitoringDataTable and requestStatisticsDataTable, if it exists.

For the last one, all entries in samplesTable related to this request will be deleted alongside the entries in requestControlDataTable, requestMonitoringDataTable and, if it exists, requestStatisticsDataTable.

Giving us the following flowchart:

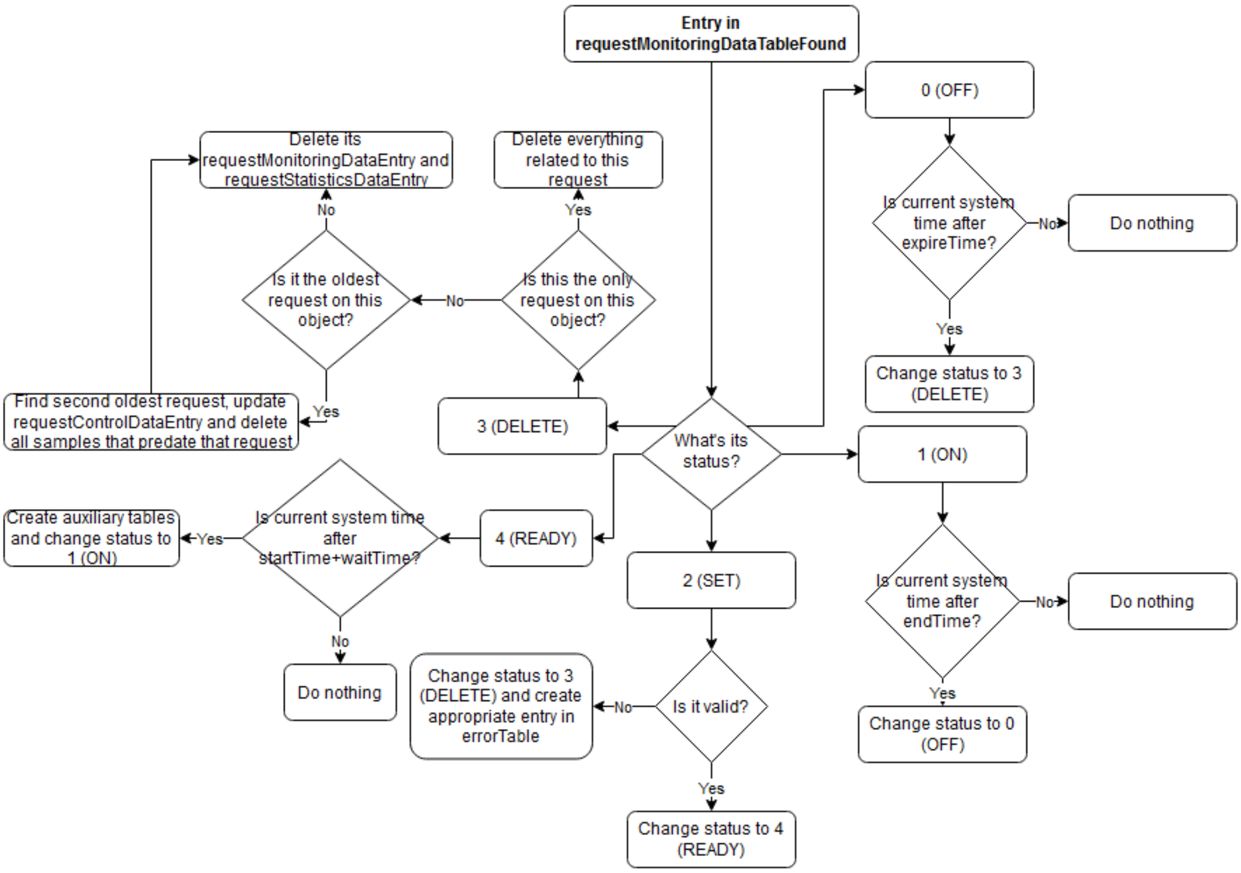

Figure 22: How is an entry in requestMonitoringDataTable managed

It should be noted that after an entry with loopMode set to 1 (YES) is deleted, a new copy of that request will be added to the system.

4.3.3 *Storing Sensor Readings*

In this section we will dive into how data from the *CAN* network is stored but, before going into details we must first give a short introduction to containers, at least as far as the NET-SNMP *API* is concerned.

Containers

In essence, containers are a generic data interface similar to a database, and just like one, you use an index or key to access and sort data. These containers will keep our rows in memory while also sorting them, when rows are added or removed, providing a specific row for GET/SET requests without requiring *OID*s and, as such, significantly simplifying the process of obtaining data from tables. This simplification allows developers to concentrate on operation the data rather than having to deal with *SNMP* GET/SET details[40].

These containers can be used to traverse tables and also include some easy to use operations that allowed us to add, remove or find an entry in a table based on its index or even iterate through all entries on a table:

- CONTAINER_INSERT - Given an container and an object, it will add the object to the container.
- CONTAINER_REMOVE - Given an index and a container, it will remove the object matching that index.
- CONTAINER_FIND - Given an index and a container, it will return the object matching that index.
- CONTAINER_ITERATOR - Allows iterating through the contents of a container.

Since requestMonitoringDataTable is one of the two tables that the user can directly change, the files pertaining to this table will be the ones to, for the most part, handle how *CAN* messages will be stored and how auxiliary entries in other tables are created.

This was already touched upon in 4.3.2, more specifically the function checkTables(), however, in this section the focus will be on the function checkSamples() which should provide a slightly deeper dive into the internal logic of the program.

```c
 ...
/* initialize mib code here */
BO_List *boList = readDBC(FILE_LOCATION);
...
int fd[2];
if (pipe(fd) < 0)
    exit(1);
canDecoder = fork();
/* you're main loop here... */
if (canDecoder == 0)
    parseCAN(boList, fd);
else
    while (keep_running)
    {
        checkTables();
        if (agent_check_and_process(0) > 0)
            checkActuators();
        decodedCAN dc;
        int retval = fcntl(fd[0], F_SETFL, fcntl(fd[0], F_GETFL) | O_NONBLOCK);
        r = read(fd[0], &dc, sizeof(decodedCAN));
        if(r>0 && dc.signals>=0){
            /*Create checksum->check and timestamp->s*/
            for (int i = 0; i < dc.signals; i++)
                checkSamples(signalname, dc.value[i], dc.signals, s, check);
        }
    }
```

Listing 4.3: Final Sub-Agent code

Unlike checkTables(), that is executed once per loop, checkActuators() is only executed whenever a *SNMP* message is received, while checkSamples() is only executed when a *CAN* message is successfully decoded. Once the parent process receives the decoded message it will, for every signal, traverse requestMonitoringDataTable looking for entries whose status is 1 (ON) that are recording this signal.

This was done by first obtaining the entry in mapTypeTable that requestMapID points to and then comparing its data source with the signal name. If they match the next step is to traverse samplesTable and check if this particular sample has already been added by comparing the requestMapID and checksum with the ones included in samplesTable, if an entry is found with both items matching then this sample has already been added to the system.

If the sample has been found, the only thing we needed to do is update lastSampleID within requestMonitoringDataTable to the ID of this new sample and it's entry in requestStatistics-DataTable, if it exists. Otherwise the sample will be added to samplesTable and then both requestControlDataTable, requestMonitoringDataTable and requestStatisticsDataTable, if it exists, will be updated. This is done to prevent repeat samples of being added to the system.

For example this is the function that is used to check if a sample is already added to the system.

```
/*This function will check if a sample was already added to the table by
   comparing the checksum, if it exists its index will be returned*/
int checkSampleChecksum(char *checksum, unsigned long id)
{
    int res = 0;
    void *data;
    netsnmp_iterator *it;
    it = CONTAINER_ITERATOR(cb.container);
    if (NULL == it)
        exit;
    for (data = ITERATOR_FIRST(it); data; data = ITERATOR_NEXT(it))
    {
        samplesTable_context *samples = data;
        if (strcmp(checksum, samples->sampleChecksum) == 0 && id == samples->
    mapTypeSamplesID)
        {
            res = samples->sampleID;
            break;
        }
    }
    ITERATOR_RELEASE(it);
    return res;
}
```

Listing 4.4: Ensuring repeat samples are not added

On the other hand the function that adds a new entry to samplesTable looks like this.

```
/*samplesStruct is a struct similar to samplesTables_context that is used to aid
    in the creation of new entries*/
void insertSamplesRow(samplesStruct *req)
{
    samplesTable_context *ctx;
    netsnmp_index index;
    oid index_oid[2];
    index_oid[0] = req->sampleID;
    index.oids = (oid *)&index_oid;
    index.len = 1;
    ctx = NULL;
    /* Search for it first. */
    ctx = CONTAINER_FIND(cb.container, &index);
    if (!ctx)
    {
        // No dice. We add the new row
        ctx = samplesTable_create_row(&index, req);
        CONTAINER_INSERT(cb.container, ctx);
    }
}
```

Listing 4.5: Adding entries to a table

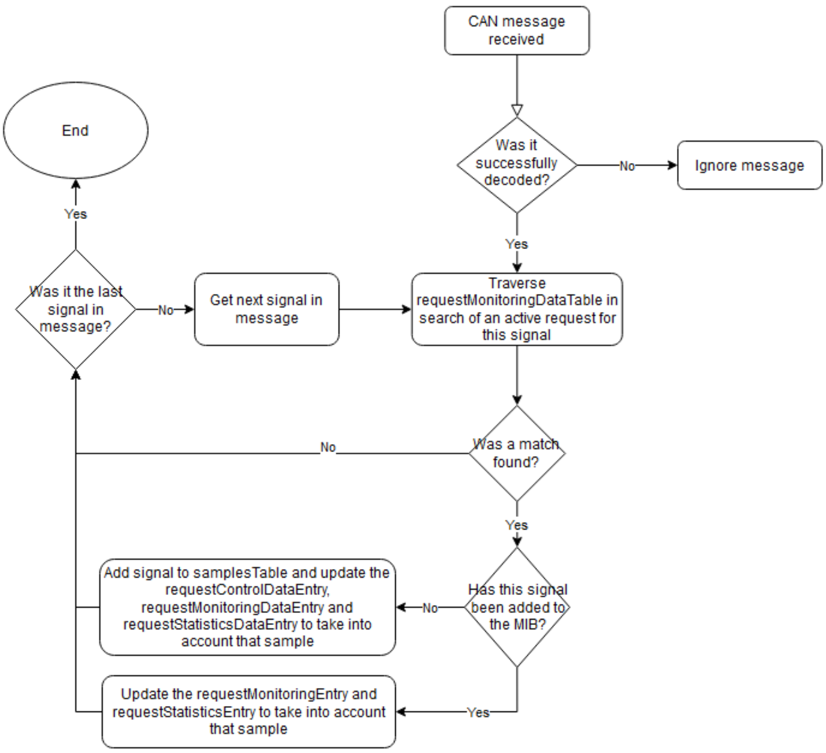

Figure 23: How a signal is stored in the database

4.3.4 *Saving Modes*

Another functionality that was added was that of saving modes. These are part of requestMonitoringDataTable and requestControlDataTable, as setting mode and commit mode respectively, and are used to determine which requests present in the system are to be saved in a cache file when the system is turned off.

Once the shutdown procedure has begun, requestMonitoringDataTable will be traversed so as to delete all entries whose setting mode is set to "Volatile". Once such a request is found, it's status will be changed to "Delete" so that all remaining entries in the system are those with the setting mode as "Permanent".

The remaining entries in requestMonitoringDataTable will then be stored in a cache file as well as the entries in requestControlDataTable, requestStatisticsDataTable and samplesTable that are in the system. This is done by first adding all those entries to the following struct which is then written to a file.

```
/*statisticsCache, controlCache and samplesCache are similar to monitoringCache
   */
   typedef struct monitoringCache
   {
       requestMonitoringDataTable_context **items;
       int current;
       int capacity;
   } monitoringCache;
   typedef struct systemCache
   {
       monitoringCache mc;
       statisticsCache sc;
       controlCache cc;
       samplesCache rc;
   } systemCache;
```

Listing 4.6: Cache struct

On start-up, if a cache file is present, the contents of the file will be read and added back to the system.

4.4 MANAGER

To test the *MIB* and agent presented above, a *SNMP* manager was needed and, as referenced in chapter 4.1, while the Net-SNMP suite includes a generic manager, it is not adequate for this particular role and as such a custom Manager had to be made. This manager was then used to communicate with the agent through *SNMP* commands to:

- Create new requests.
- View an existing request.
- Edit an existing request.
- View any table in the system.
- View active errors in the system.
- Send commands to specific actuators.

To achieve this we will only needed to use two *SNMP* commands, "snmpset" and "snmpbulkget", which means we needed to write our own functions that would create the corresponding *PDU*, send it and handle the response from the agent, additionally a simple terminal based interface was also created to allow users to more easily use the manager.

4.4.1 *Authentication and Privacy*

Security is one of the main requirements set on chapter 4, as such, it must be included on this solution. There are multiple protocols that can be used for this purpose like *SSH*, *TLS*, *DTLS*. Each one of these protocols come with their own downsides as far as raw performance is concerned however they should still be considered [41].
Additionally, as mentioned in table 2, *USM* provides a solid security suite for *SNMP*v3 allowing for three levels of security [42]:

- noAuthNoPriv mode (nn), *USM* provides no authentication and no encryption services and is from a security perspective comparable to the *CSM*, Community-based Security Model.
- authNoPriv mode (an), *USM* provides message authentication, message integrity, and timeliness checking services but no encryption.
- authPriv mode (ap), *USM* provides message authentication, message integrity, and timeliness checking services plus encryption of the payload of *SNMP* messages.

As such, when it comes to security and privacy, it was decided that SNMPv3 with Auth and Priv should be used since it allowed adequate levels of security with better performance compared with other methods.

By enabling Privacy we can ensure that, for example, any communication between two vehicles is encrypted, thus preventing any third party from reading any data between the 2 entities and potentially causing harm to any of those vehicles occupants by changing the contents of the data being transmitted. Likewise, with Auth, we can ensure that only authorized entities can communicate with our *CAN* agent, thus preventing unauthorized access to our vehicles' sensor and actuator data.

As per the Net-SNMP *API*, for the manager to be able to communicate with the agent, it must first establish a session with it, this is done with the *API* function "snmp_open" which takes a "session" struct as input. It's in this session struct that we define which protocol to use, and which functionalities we want enabled. In our case we used:

- SNMPv3.
- SHA-1 Authentication.
- AES Encryption.

```
...
/* set the SNMP version number */
session.version = SNMP_VERSION_3;
/* set the SNMPv3 user name */
session.securityName = strdup(snmpusername);
session.securityNameLen = strlen(session.securityName);
/* set the security level to authenticated and encrypted */
session.securityLevel = SNMP_SEC_LEVEL_AUTHPRIV;
/* set the authentication method to SHA */
session.securityAuthProto = usmHMACSHA1AuthProtocol;
session.securityAuthProtoLen = USM_AUTH_PROTO_SHA_LEN;
session.securityAuthKeyLen = USM_AUTH_KU_LEN;
...
/* set the encryption method to AES */
session.securityPrivProto = snmp_duplicate_objid(usmAESPrivProtocol,
USM_PRIV_PROTO_AES_LEN);
session.securityPrivProtoLen = USM_PRIV_PROTO_AES_LEN;
session.securityPrivKeyLen = USM_PRIV_KU_LEN;
...
ss = snmp_open(&session); /* establish the session */
```

Listing 4.7: Creating SNMPv3 session

Naturally, if we prefer SNMPv2 all that is neded to do is change the contents of that "session" struct.

4.4.2 *GetBulk and Set*

SNMP comes by default with commands that allow for a manager to get or set data from an agent and in the case of the former there are multiple commands to choose from, Get, GetNext and GetBulk. Out of this three, GetBulk is the most suitable this particular use case, where multiple *MIB* object need to be obtained in quick succession, as by using it we can limit network congestion since, otherwise, we would require several Get and GetNext messages to achieve the same results as GetBulk [41]. To further lower network congestion *TCP* can also be considered since there would be less re-transmissions thus increasing reliability and efficiency while also lowering congestion[43], albeit at a cost to performance.

The version of "snmpbulkget" that was created for this phase of the project will simply create a *PDU* with the target table OID, send it to the agent and then, assuming everything went according to plan, add the contents of the response to the following struct.

```
/*example table OID*/
static oid commandTableOid[] = {1, 3, 6, 1, 3, 8888, 12};
typedef struct table_contents
{
  struct table_contents *next;
  netsnmp_variable_list *data; //netsnmp_variable_list is part of netsnmp API
} table_contents;
```

Listing 4.8: Table Contents struct

This linked list can then be traversed to obtain all the information from table, which allowed us to, for example, list all active errors in the system in a human friendly manner.

The "Set" command can be used to change data in an Agent's *MIB*, thus allowing for actuators to be activated/deactivated at will and new requests for data to be made or old ones edited. "snmpset" will follow a similar logic presented for "snmpbulkget", where a "snmpset" *PDU* is created, with a list of OIDs for the target columns, their values and the data type. This *PDU* is then sent to the agent and its response handled accordingly.

With the *SNMP* session created and both "snmpset" and "snmpbulkget" available, all that is left to develop is the terminal based interface that will allow the user to create, view and otherwise manage requests and commands in the system.

4.4.3 *View any table in the system*

This functionality was mostly included for debugging and, as such, is the simplest one of them all since it will just send a "bulkget" message to the agent and print the results. These printed results will be similar to the "snmpbulkget" command included in the NET-SNMP daemon.

```
CommandTemplateTable
OBU-MIB::commandTemplateID.0 = Gauge32: 0
OBU-MIB::commandTemplateID.1 = Gauge32: 1
OBU-MIB::commandTemplateID.2 = Gauge32: 2
OBU-MIB::commandDescription.0 = STRING: "This command will change status of brake actuators"
OBU-MIB::commandDescription.1 = STRING: "This command will change status lock actuators"
OBU-MIB::commandDescription.2 = STRING: "This command will turn AC on at a set temperature"
OBU-MIB::targetNode.0 = STRING: "AAAAAB"
OBU-MIB::targetNode.1 = STRING: "123456"
OBU-MIB::targetNode.2 = STRING: "AC1111"
OBU-MIB::commandTemplate.0 = STRING: "FF FF FF ** ** FF FF FF"
OBU-MIB::commandTemplate.1 = STRING: "A2 C5 88 ** ** 92 F0 EA"
OBU-MIB::commandTemplate.2 = STRING: "AA BB CC DD EE ** ** H1"
```

Figure 24: Viewing the contents of commandTemplateTable with the manager

4.4.4 *Create new requests*

Before explaining the process of how a new request is created we must first explain why the user can only change to contents of some columns within requestMonitoringDataTable.

This is done mainly to prevent entries from being left ”hanging” by edits to a request, and as such, certain columns can only be changed by the user in the beginning, while others can only be changed at a later date, some can be changed anytime while others can’t be changed at all.

For example, ”requestControlID” can’t be set or edited by the user since it will point to an entry in requestControlDataTable and as such is solely managed by the agent while at the same time ”status” is originally handled by the agent but the user can manually edit it later, with some constraints, likewise some columns are originally set by the user but the user won’t be allowed to change them at a later date like ”requestMapID”.

As previously mentioned, the columns that the user can set when creating a new request are the following:

- requestMapID - Points to the signal whose samples the user wants recorded.
- requestStatisticsID - Points to an entry in requestStatisticsDataTable.
- savingMode - Is it volatile or permanent.
- maxNOfSamples - Maximum number of samples to be recorded.
- loopMode - Should it restart once it’s deleted or not.
- startTime - When should request start.
- waitTime - How long after startTime should the request wait to start.
- durationTime - How long should the request run.
- expiretime - How long after the request ended should it stay in the system.

As such, to create a new request, the user inputs for all of these columns has to be obtained, first of which being ”requestMapID”.

Since a vehicle will contain a considerable number of sensors within it, we had to first send a snmpbulkget message for both mapTypeTable and genericTypesTable, and print the results in a concise and user friendly manner, which will allow the user to know what is the ID of every sensor, alongside its description.

```
196 [EEC1:EngSpeed]
        Actual engine speed which is calculated over a minimum crankshaft angle of 720 degrees divided by the number of cylinders.
```

Figure 25: Sensor Description

The user will then only need to provide the inputs for every single one of those columns and send the snmpset command to the agent.

```
Choose sensor:196
Do you want statistics?(0=No,1=Yes): 1
Choose Saving Mode(0=Permanent,1=Volatile): 0
Please indicate start time in the following format 12:00:00 (empty for current system time):
Please indicate wait time in the following format 12:00:00 (empty for no waitTime):
Please indicate duration time in the following format 12:00:00 (empty for 10 minutes duration):
Please indicate expire time in the following format 12:00:00 (empty for 10 minutes expiration):
Indicate maximum number of samples to be recorded (default is 50):
Should this request restart once it's over?(1=Yes,2=No): 2
Indicate username(Default is manager username):
```

Figure 26: Creating a new request

```
RequestMonitoringDataTable
OBU-MIB::requestID.0 = Gauge32: 0
OBU-MIB::requestMonControlID.0 = Gauge32: 0
OBU-MIB::requestMapID.0 = Gauge32: 196
OBU-MIB::requestStatisticsID.0 = Gauge32: 2
OBU-MIB::savingMode.0 = INTEGER: permanent(0)
OBU-MIB::samplingFrequency.0 = Gauge32: 0
OBU-MIB::maxDelay.0 = INTEGER: 0
OBU-MIB::startTime.0 = STRING: "25/09/2021 16:00:18"
OBU-MIB::endTime.0 = STRING: "25/09/2021 16:10:18"
OBU-MIB::waitTime.0 = STRING: "00:00:00"
OBU-MIB::durationTime.0 = STRING: "00:10:00"
OBU-MIB::expireTime.0 = STRING: "00:10:00"
OBU-MIB::lastSampleID.0 = Gauge32: 0
OBU-MIB::nOfSamples.0 = Counter32: 0
OBU-MIB::maxNOfSamples.0 = Gauge32: 50
OBU-MIB::loopMode.0 = INTEGER: no(2)
OBU-MIB::status.0 = INTEGER: on(1)
OBU-MIB::requestUser.0 = STRING: "snmpadmin"
```

Figure 27: Created request in requestMonitoringDataTable

Note: In the current prototype stage, the column ”requestUser” is also set by the user and not by the agent but in its deployed version that column should be solely handled by the agent.

4.4.5 *View a request*

This functionality is centered around viewing the results of any request present in the system, that means providing the user with all samples relating to a request alongside their respective timestamps, checksums, unit, signal name and, if relevant, statistics.

To do this we first needed to send bulkget messages to the agent for sampleUnitsTable, mapTypeTable and requestMonitoringDataTable which allowed us to show all requests in the system alongside their IDs so that the user can choose which request to inspect.

```
ID->SignalName [Number of Samples] Username
        0->EEC1:EngSpeed [10] snmpadmin
        1->EEC1:ActlEngPrcntTrqueHighResolution [10] snmpadmin
        2->EEC1:EngTorqueMode [21] Utilizador Teste
```

Figure 28: List of requests in the system

After the request is chosen the manager will send "bulkget" messages to obtain the contents of samplesTable and, if the request included statistics, requestStatisticsDataTable and print all samples related to the chosen request in a concise manner.

```
Choose request:0
Request 0 made by user "snmpadmin" on EEC1:EngSpeed
Sample 10: 7775 rpm [25/09/2021 16:03:07] {88A13412}
Sample 9: 7771 rpm [25/09/2021 16:03:06] {36A13412}
Sample 8: 7769 rpm [25/09/2021 16:03:05] {F9A03412}
Sample 7: 7807 rpm [25/09/2021 16:03:05] {CFA03412}
Sample 6: 5749 rpm [25/09/2021 16:03:04] {26A13412}
Sample 5: 7838 rpm [25/09/2021 16:03:03] {AAA03412}
Sample 4: 7837 rpm [25/09/2021 16:03:02] {E69B3412}
Sample 3: 5782 rpm [25/09/2021 16:03:02] {BE9B3412}
Sample 2: 7838 rpm [25/09/2021 16:03:01] {839B3412}
Sample 1: 5811 rpm [25/09/2021 16:03:01] {339B3412}
Statistics- MIN(5749) MAX(7838) AVERAGE(7195)
```

Figure 29: Samples related to a request

These results contain the following information:

- Sample Number.
- Sample Value.
- Sample Unit.
- Sample Timestamp.
- Sample Checksum.

4.4.6 *Edit a request*

Much like ”View Request”, this functionality will also require the user to choose between already existing request on the system, which means it starts by sending a ”bulkget” message for the contents of requestMonitoringDataTable to the manager.

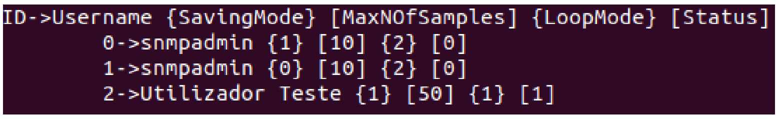

Figure 30: Requests and the current contents of their editable columns

After choosing what request to edit, the user will choose the column to be edited and input the new value. Finally, the manager will send the corresponding ”snmpset” message which if successfully validated will change the corresponding entry.

As mentioned in 4.4.4, some columns can only be changed by the user on creation, some can only be edited after creation, while others can’t be changed by the user at all. When it comes to editing columns in a request, it was decided that the user should only be allowed to change 4 of them:

- Saving Mode (0 or 1).
- Loop Mode (1 or 2).
- Max Number of Samples (>0).
- Status (0 to 4).

While for most of this columns validation is rather straightforward, meaning they only need to meet a simple condition, when it comes to the ”status” some other restrictions must be taken into account based on the current status of the request. For example, if the request status is ”ready” or ”set” it can be changed to ”on”, ”off” or ”delete”, while at the same time, if a request is ”off” it can only be be changed ”delete”. This is done due to the linked list nature of how samples are stored in the *MIB* and will prevent a requests’ status from going backwards in the sense that the proper path of all requests is set->ready->on->off->delete.

If the input given by the user is found to be invalid, a new entry in errorTable will be created while the edit itself will be canceled.

4.4.7 *View Active errors in the system*

This functionality will consist of sending bulkget commands to both errorDescriptionTable and errorTable since the former contains the descriptions of the error while the latter contains the error itself as well as a timestamp and the user who made the error in the first place.

With the contents of these two tables we could now present the errors in a user friendly manner.

```
Error 0=[25/09/2021 16:11:10] EC10[Invalid command template ID] User[snmpadmin]
Error 1=[25/09/2021 16:11:19] EC8[Invalid sensor, check mapTypeTable for valid sensor id's] User[snmpadmin]
Error 2=[25/09/2021 16:11:34] EC6[User has already set a request for samples recorded from this sensor] User[snmpadmin]
Error 3=[25/09/2021 16:11:46] EC15[Current status is On, it can only be manually changed to Delete or Off] User[snmpadmin]
```

Figure 31: Active Errors in the system

4.4.8 *Send Command*

This final functionality will, quite simply, send a "bulkget" message for the contents of commandTemplateTable and print all existing commands in an easy to understand manner. The user will then choose the command it wants to send and input the new value.

```
Command 0 -[Target=AAAAAB] This command will change status of brake actuators
Command 1 -[Target=123456] This command will change status lock actuators
Command 2 -[Target=AC1111] This command will turn AC on at a set temperature
Choose Template: 0
Insert Input:12
0
12
"snmpadmin"
```

Figure 32: Sending a command to the agent

The manager will then create a new entry in commandTable by sending a "snmpset" message containing the input given by the user and the template that was chosen so that the agent can validate, build and send the correct *CAN* message to the network.

```
Command sent: AAAAAB FFFFFF000CFFFFFF
```

Figure 33: Confirmation that the agent sent the *CAN* message

4.5 TESTS AND RESULTS

With development complete the next step is to run a few baseline performance tests and compare them to already existing solutions, more specifically *OBD-II*. In this section, besides the aforementioned tests, the testing environment will also be presented and finally a discussion of the results will be made.

4.5.1 *Testing Environment*

Since this is still just a prototype there's no vehicle or *OBU* where this can solution can be tested in and as such the tests will be performed in an instance of Ubuntu 20.04.2 on VMWare Workstation 16 Pro, version 16.0.0 build-16894299, on Windows 10 Pro.

System Specification (Available)
Intel Core i5-4670 @3.4Ghz 4(2) Cores
16(8)GB DDR3 1600Mhz

Any results obtained in this system won't be totally conclusive of the performance in any *OBU* however it can prove that this solution has promise and is worth further development.

When testing the simulator will write *CAN* messages from raw *CAN* log to a virtual *CAN* interface, which the sub agent will listening into. The manager will be used to create some requests and then view the results of those requests.

When it comes to choosing what signal the system is going to record, a few statistics of the *CAN* logs were created so as to identify which *ECU* was transmitting more messages.

ECU	Number of Messages	Percentage
EEC1	19535	45.6%
EEC2	3907	9.12%
EEC3	3907	9.12%
LFE	1954	4.56%
PTO	1954	4.56%
...	...	...

Table 22: Number of messages per *ECU*

On all four raw *CAN* logs that were obtained to test this solution, the most active *ECU* was EEC1 and as such all requests will be on signals coming from that *ECU*.

4.5.2 *Testing Results*

There are four important metrics that are relevant when it comes to this solution:

- How long does it take to execute a command.
- How long does it take to decode a signal.
- How long does it take to insert a sample in the system.
- How long does it take a manager to get the results of a request.

To measure time taken by any process, we can use clock() function which is available time.h. We can call the clock function at the beginning and end of the code for which we are measuring time, subtract the values, and then divide by CLOCKS_PER_SEC (the number of clock ticks per second) to get processor time, like following.

```
clock_t start = clock();
... /* Do the work. */
clock_t end = clock();
printf("%f\n",(double) (end - start) / CLOCKS_PER_SEC);
```

Listing 4.9: Measuring Time

Executing a command and decoding a message

These two metrics were the simplest ones to measure since, to execute a command we only needed to measure how long it takes to run the function checkActuators(), while to decode we only needed to measure the time it takes to run the function decode().
After inserting the code above in the relevant parts of the program, some commands were added by the manager to the system and the time taken to run those commands were measured. On average, each command took around 80µs to execute. This result only takes into account how long the system takes to read the entry from the table, create a *CAN* message, transmit it and delete the entry from the system. While this functionality is still incomplete, and as such the result is fairly inconsequential, it can still give an idea of the possible performance of this solution.

When it comes to decoding messages, the test consisted of running the simulator and subagent at the same time, measuring how long it took to decode each *CAN* message. which on average took around 30µs. Once again, while it's not representative of real world performance these can be used to measure the validity of this solution, additionally one can expect that in-house decoders that are already in use by vehicle manufacturers are more efficient than the one than the one that was created for this phase of the project.

Inserting a sample in the system

To measure how long it takes to insert samples in the system two types of tests were run, one where there's only one request in the system and another where there were several requests on the system. Since EEC1 is the most active *ECU* in the *CAN* logs that were used in these tests, the requests will all be made for signals of this particular *ECU*.

For the test with a single request, the manager was used to make a request on the signal EngSpeed, which is part of EEC1. The time it took the system to add new entries to the *MIB* was then measured and printed out, giving the following results

```
Time to insert sample 0.000177 s-- EEC1:EngSpeed
Time to insert sample 0.000224 s-- EEC1:EngSpeed
Time to insert sample 0.000062 s-- EEC1:EngSpeed
Time to insert sample 0.000124 s-- EEC1:EngSpeed
Time to insert sample 0.000061 s-- EEC1:EngSpeed
Time to insert sample 0.000105 s-- EEC1:EngSpeed
Time to insert sample 0.000071 s-- EEC1:EngSpeed
Time to insert sample 0.000122 s-- EEC1:EngSpeed
Time to insert sample 0.000057 s-- EEC1:EngSpeed
Time to insert sample 0.000136 s-- EEC1:EngSpeed
```

Figure 34: Inserting samples on a single request

As can be seen above, outside of the first two entries, the amount of time it took to add entries to the *MIB* was between 50µs and 130µs.

Next several requests were created on signals of EEC1, more specifically 10 requests were created by 2 different users on the same set of sensors (EngSpeed, ActualEngPercentTorque, EngTorqueMode, EngStarterMode, DriversDemandEngPercentTorque).

```
Time to insert sample 0.000046 s-- EEC1:ActualEngPercentTorque
Time to insert sample 0.000093 s-- EEC1:DriversDemandEngPercentTorque
Time to insert sample 0.000086 s-- EEC1:EngTorqueMode
Time to insert sample 0.000103 s-- EEC1:EngStarterMode
Time to insert sample 0.000161 s-- EEC1:EngSpeed
Time to insert sample 0.000061 s-- EEC1:ActualEngPercentTorque
Time to insert sample 0.000104 s-- EEC1:DriversDemandEngPercentTorque
Time to insert sample 0.000066 s-- EEC1:EngTorqueMode
Time to insert sample 0.000127 s-- EEC1:EngStarterMode
Time to insert sample 0.000063 s-- EEC1:EngSpeed
Time to insert sample 0.000100 s-- EEC1:ActualEngPercentTorque
Time to insert sample 0.000053 s-- EEC1:DriversDemandEngPercentTorque
Time to insert sample 0.000095 s-- EEC1:EngTorqueMode
Time to insert sample 0.000072 s-- EEC1:EngStarterMode
Time to insert sample 0.000121 s-- EEC1:EngSpeed
Time to insert sample 0.000060 s-- EEC1:ActualEngPercentTorque
```

Figure 35: Inserting samples on multiple requests

Once again out of the 100+ samples that were added into the system, the longest it took was around 200µs while on average the results were similar to those obtained in the first, 90µs.

Obtaining samples from a table

For these measurements the same tests as the section above were used, but this time the measurements were taken by measuring how long it took the manager to run the function bulkget() when attempting to view the results of a request.

For a single request, containing 10 samples, it took around 20ms to obtain the results, while with 10 requests in the system, it took around 40 ms. It should be noted that with bulkget function that was developed for this phase of the project, it will return all entries of a table.

Figure 36: Single request

Figure 37: Multiple requests

Comparing with OBD-II

As it stands the results are rather one sided, while it must be mentioned that *OBD-II* was, as the name implies, originally intended to be used in diagnosing possible issues with a vehicle, the 20 query per second limit and accompanying slow refresh rate will impede its use with any sort of *VANET* application while the prototype presented above will only be limited by the computing power of the *OBU* and available bandwidth, since it's wholly dependent on the ability of the agent to respond immediately to the requests of the manager, which the Standard does not ensure. Nevertheless, as per [44] and [45], the response times presented above are within margin for use in *CACC* (Cooperative Adaptive Cruise Control), *ACC*(Adaptive Cruise Control) and platooning applications.

4.6 SUMMARY

In this chapter the various development steps of the prototype were presented, in addition to this the development of the tools that allow a *CAN* message to be sent, decoded and handled by the prototype was also presented alongside some code snippets and flowcharts for individual functions within the agent.

The development of the manager was also presented, where each functionality, including the choice of security protocol, of this manager was explained and demonstrated as well.

Finally, some metrics were measured so that this solution can be compared with the tools that are currently in use.

5

CONCLUSION

With the ever closing introduction of *VANET*s, the realization that the protocols that are currently in use to obtain data from within a vehicle are wholly unsuitable for this new paradigm as come to the surface, since while those protocols and solutions are quite capable when used for their original purpose, they lack the performance required for future applications.

This means that new unprecedented solutions need to be developed so that those new requirements are met and with this work a solution that does just that has been presented. That is an agnostic and modular architecture that allows the development of cooperative *ITS* applications.

This work proves that *SNMP* can indeed be used to monitor vehicular sensors while also allowing some degree of control over a vehicle via its actuators. This solution provides the basis through which any application that may require access to real-time sensor data or direct access to actuators, so as to change their states in real time, can be developed around, while also being an agnostic and modular architecture that allows the development of cooperative *ITS* applications. Additionally, it can also provide the same functionalities whose requirements are already met by standards like *OBD-II* without requiring the customer or manufacturer to use specialized hardware.

Since this solution is based on *SNMP*, a manufacturer would only need to integrate an *SNMP* sub-agent like the one developed for this project in the *OBU* as well as installing applications that integrate an *SNMP* manager, for example in the local vehicle system (outside the OBU). After setting up the agent and manager, the only requirement left would be an *ITS* application, developed by the manufacturer or a third party, to handle the communication between the different entities within a *VANET*.

From the results discussed in chapter 4 we can prove that, despite being an early prototype, the performance of this solution is a marked improvement over already existing solutions both in latency, number of sensors that can be queried at the same time and refresh rate of any of those queries, while using a proven and reliable protocol in the form of *SNMP*v3. Additionally, for certain use cases where authentication and privacy is not required, *SNMP*v2c can be used instead of *SNMP*v3 which will further improve the performance.

With this, we can safely say that the objectives listed in chapter 1.2 and chapter 3 were met, since a *MIB* that can allow low level *ITS* functions implemented by the vehicles manufacturer and is transparent to the chosen electronic communication bus was successfully created. Additionally, a prototype *SNMP* agent and manager, and accompanying decoder/ generator, that allow for the creation of monitoring requests and commands to the network were also developed to test the validity of this project.

FUTURE WORK

This being said, the present work it still just a prototype and as such it can still be improved on multiple fronts before being deployed. These range from overall stability and performance improvements to refinement of the presented solution, for example when it comes to how this solution changes the state of actuators.

One such improvement is that of the decoder, since the current decoder being used lacks the ability to decode messages from protocols other than *CAN*2.0B and more specifically data frames. As such, the decoder should be refined so that it supports all kinds of *CAN* frames from all *CAN* protocols, or its competitors. The main focus should be to make it compatible with *CAN*FD. Alternatively in-house decoders made by the manufacturer should be used instead of a custom decoder since those will already be capable of achieving the same results with optimum performance.

For security reasons it might be wise to limit the access of certain sensors/actuators based on who the user is since currently there's no such method in place, additionally the current method of obtaining the username of the manager that made a request relies too much on that manager inserting the right name, ideally the username should be obtained by the agent when the packet arrives, this username can be the *IP* address or *SNMP*v2 Community string/*SNMP*v3 User Name. This will most likely require converting the current sub agent based solution to an custom *SNMP* agent.

Some new *SNMP* primitives should be created similar to *SNMP* traps so that when a trap is triggered, it would transmit it in broadcast mode to all relevant entities. Additionally if two entities made a request on the same sensor, instead of the source vehicle transmitting the stored data individually to each of those entities it should transmit it in broadcast mode similar to what *CAN* protocol already uses since it would lower the bandwidth being used. A new primitive similar to "bulkget" should also be created so that it only returns samples related to a specific sensor or request, since currently, it will transmit all data in a table which decreases performance.

Finally, some real world tests should also be performed to validate this solution since the tests performed and presented in this document were done in a computer with performance that does not represent real world capabilities of *OBU*.

REFERENCES

[1] O'Reilly Douglas Mauro, Kevin Schmidt. *Essential SNMP*. O'Reilly, 2 edition, 2003.

[2] G. Mohinisudhan, Sahana K. Bhosale, and Bharat S. Chaudhari. Reliable On-board and Remote Vehicular Network Management for Hybrid Automobiles. In *2006 IEEE Conference on Electric and Hybrid Vehicles*, pages 1–4, 2006. doi: 10.1109/ICEHV.2006.352284.

[3] Joao AFF Dias, Joel JPC Rodrigues, Vasco NGJ Soares, João MLP Caldeira, Valery Korotaev, and Mario L Proença. Network management and monitoring solutions for vehicular networks: a survey. *Electronics*, 9(5):853, 2020.

[4] Ishak Aris, Mohamad Fauzi Zakaria, S. Bashi, and R Sidek. Development of OBD-II Driver Information System. In *International Engineering Convention, Jeddah, Saudi Arabia*, March 2007.

[5] ISO 15031-5:2015. Road vehicles — Communication between vehicle and external equipment for emissions-related diagnostics — Part 5: Emissions-related diagnostic services. Standard, International Organization for Standardization, Geneva, CH, 2015.

[6] How fast can OBD-II data update. `https://customer.cradlepoint.com/s/article/What-Is-the-Maximum-Refresh-Interval-for-OBD2`, . Accessed:17/10/2021.

[7] OBD Knowledge Sampling Rate. `http://www.mx.innova.com/en-US/TechnicalInfo/OBDKnowledge`, . Accessed:30/10/2021.

[8] Wilfried Voss. *A comprehensible guide to J1939*. Copperhill Technologies Corporation, 2008.

[9] Bruno Dias, Alexandre Santos, António Costa, Bruno Ribeiro, Fabio Goncalves, Joaquim Macedo, Maria Nicolau, Oscar Gama, and Susana Sousa. Agnostic and Modular Architecture for the Development of Cooperative ITS Applications. *Journal of Communications Software and Systems*, 14:218–227, 09 2018. doi: 10.24138/jcomss.v14i3.550.

[10] Keith McCord. *Automotive Diagnostic Systems: Understanding OBD I and OBD II*. CarTech Inc, 2011.

[11] Wilfried Voss. *A comprehensible guide to controller area network*. Copperhill Media, 2008.

[12] SolarWinds. A guide to understanding SNMP. In *A guide to understanding SNMP*, 2013.

[13] Cisco Systems. Introduction to SNMP and MIB. 2004.

[14] Nick Urbanik. The Structure of Management Information (SMI).

[15] SNMP Version 3 (SNMPv3) Message Format. http://www.tcpipguide.com/free/t_SNMPVersion3SNMPv3MessageFormat.htm. Accessed:24-12-2020.

[16] Dr. Marshall T. Rose and Keith McCloghrie. Structure and identification of management information for TCP/IP-based internets. RFC 1155, May 1990.

[17] SNMP MIB. https://docs.oracle.com/cd/E13203_01/tuxedo/tux90/snmpmref/1tmib.htm, . Accessed:24/12/2021.

[18] On-board diagnostics. https://en.wikipedia.org/wiki/On-board_diagnostics, . Accessed:12-12-2020.

[19] Aastha Yadav, Gaurav Bose, Radhika Bhange, Karan Kapoor, N Ch Sriman Narayana Iyenger, and Ronnie Caytiles. Security, Vulnerability and Protection of Vehicular On-board Diagnostics. *International Journal of Security and Its Applications*, 10:405–422, 04 2016. doi: 10.14257/ijsia.2016.10.4.36.

[20] Md Arafatur Rahman. *Design of Wireless Sensor Network for Intra-vehicular Communications*. 01 2014.

[21] Steve Corrigan HPL. Introduction to the controller area network (can). *Application Report SLOA101*, pages 1–17, 2002.

[22] Controller area networks and the protocol can for machine control systems. *Mechatronics*, 4 (2):159 – 172, 1994. ISSN 0957-4158. doi: https://doi.org/10.1016/0957-4158(94)90041-8. Special Issue Mechatronics in Sweden.

[23] Simple intro to CAN bus. https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en. Accessed:24-11-2020.

[24] CAN bus the central networking system of vehicles. https://premioinc.com/blogs/blog/can-bus-the-central-networking-system-of-vehicles. Accessed:12-12-2020.

[25] CAN bus serial protocol decoding. https://www.picoauto.com/library/picoscope/can-bus-serial-protocol-decoding. Accessed:26-11-2020.

[26] A survey of automotive networking applications and protocols. 2015.

[27] Françoise Simonot-Lion Nicolas Navet. In-vehicle communication networks - a historical perspective and review. *International Journal of Security and Its Applications*, 09 2013.

[28] Jun Huang, Mingli Zhao, Yide Zhou, and Cong-Cong Xing. In-vehicle networking: Protocols, challenges, and solutions. *IEEE Network*, 33(1):92–98, 2019. doi: 10.1109/MNET.2018.1700448.

[29] FlexRay Consortium. FlexRay Communications System Protocol Specification Version 3.0.1. In *FlexRay Communications System Protocol Specification*, 10 2020.

[30] Huang Hui-Ping, Xiao Shi-De, and Meng Xiang-Yin. Applying snmp technology to manage the sensors in internet of things. *The Open Cybernetics & Systemics Journal*, 9(1), 2015.

[31] D. Jiang and L. Delgrossi. IEEE 802.11p: Towards an International Standard for Wireless Access in Vehicular Environments. In *VTC Spring 2008 - IEEE Vehicular Technology Conference*, pages 2036–2040, 2008. doi: 10.1109/VETECS.2008.458.

[32] G. Pavlou, P. Flegkas, S. Gouveris, and A. Liotta. On management technologies and the potential of Web services. *IEEE Communications Magazine*, 42(7):58–66, 2004. doi: 10.1109/MCOM.2004.1316533.

[33] SNMP agents and subagents. `https://www.ibm.com/docs/en/zos/2.3.0?topic=20-snmp-agents-subagents`, . Accessed:15-07-2021.

[34] Extending Net-SNMP. `https://vincent.bernat.ch/en/blog/2012-extending-netsnmp#extending-net-snmp`. Accessed:15-07-2021.

[35] Writing a Sub Agent. `https://net-snmp.sourceforge.io/wiki/index.php/TUT:Writing_a_Subagent`. Accessed:15-07-2021.

[36] Using and loading MIBS. `https://net-snmp.sourceforge.io/wiki/index.php/TUT:Using_and_loading_MIBS`. Accessed:15-07-2021.

[37] CAN DBC File Database Intro. `https://www.csselectronics.com/screen/page/can-dbc-file-database-intro/language/en`. Accessed:01-05-2021.

[38] J1939 standard. `https://hackage.haskell.org/package/ecu-0.0.8/src/src/j1939_utf8.dbc`. Accessed:14-07-2021.

[39] Robert Bosch. *CAN Specification version 2.0.* 9 1991.

[40] NetSNMP subagent development manual. `https://www.cnblogs.com/shipfi/articles/1033365.html`. Accessed:16-10-2021.

[41] J. Schönwälder and V. Marinov. On the Impact of Security Protocols on the Performance of SNMP. *IEEE Transactions on Network and Service Management*, 8(1):52–64, 2011. doi: 10.1109/TNSM.2011.012111.00011.

[42] Uri Blumenthal and Bert Wijnen. User-based security model (USM) for version 3 of the simple network management protocol (SNMPv3). Technical report, RFC 2574, April, 1999.

[43] X. Du, M. Shayman, and M. Rozenblit. Implementation and performance analysis of SNMP on a TLS/TCP base. In *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No.01EX470)*, pages 453–466, 2001. doi: 10.1109/INM.2001.918059.

[44] Rajesh Rajamani and Steven E Shladover. An experimental comparative study of autonomous and co-operative vehicle-follower control systems. *Transportation Research Part C: Emerging Technologies*, 9(1):15–31, 2001.

[45] Shahriar Hasan, Ali Balador, Svetlana Girs, and Elisabeth Uhlemann. Towards emergency braking as a fail-safe state in platooning: A simulative approach. In *2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall)*, pages 1–5. IEEE, 2019.

ANNEX A: OBU MIB

```
OBU–MIB DEFINITIONS ::= BEGIN

IMPORTS
  experimental,
  MODULE–IDENTITY,
  OBJECT–TYPE,
  Counter32,
  Unsigned32
    FROM SNMPv2–SMI
  TEXTUAL–CONVENTION
    FROM SNMPv2–TC
  OBJECT–GROUP
    FROM SNMPv2–CONF;

obuMIB MODULE–IDENTITY
  LAST–UPDATED "202103121429Z"  –– Mar 12, 2021, 2:29:00 PM
  ORGANIZATION "Universidade do Minho"
  CONTACT–INFO
    ""
  DESCRIPTION
    "SMIv2 MIB module to be used in vehicular OBU"
  REVISION "202103121429Z"  –– Mar 12, 2021, 2:29:00 PM
  DESCRIPTION
    "Initial version."
  –– 1.3.6.1.3.8888 ––
::= { experimental 8888 }

systemOBUGroup OBJECT–GROUP
  OBJECTS {
    numberOfCapabilities,
    capabilitiesID,
    setOfCapabilitiesID,
    specificCapabilitiesID,
    capabilityValue,
    numberOfConnectedVehicles,
    vehicleID,
    localID,
    globalID,
    associatedOBUorRSU,
    localOrRemote,
    capabilities,
    sysOBUDateandTime,
    sysOBUNMonRequest,
```

```
      sysOBUNEventRequest,
      sysOBUNConfRequest,
      sysOBUNErrors,
      sysOBUVehicleID,
      sysOBUDistanceType,
      sysOBUTotalDistance,
      sysOBUCountry
        }
  STATUS   current
  DESCRIPTION
     "This group includes all objects related to sensor OBU system"
  -- 1.3.6.1.3.8888.1 --
   ::= { obuMIB 1 }

numberOfCapabilities OBJECT-TYPE
  SYNTAX   INTEGER
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
     "This object will count the number of capabilities"
  -- 1.3.6.1.3.8888.1.1
::= { systemOBUGroup 1 }

capabilitiesTable OBJECT-TYPE
  SYNTAX   SEQUENCE OF CapabilitiesEntry
  MAX-ACCESS not-accessible
  STATUS   current
  DESCRIPTION
     "This table will list the vehicle/OBU capabilities, including all available
     services."
  -- 1.3.6.1.3.8888.1.2 --
::= { systemOBUGroup 2 }

capabilitiesEntry OBJECT-TYPE
  SYNTAX   CapabilitiesEntry
  MAX-ACCESS not-accessible
  STATUS   current
  DESCRIPTION ""
  INDEX {
     capabilitiesID }
  -- 1.3.6.1.3.8888.1.2.1
::= { capabilitiesTable 1 }

CapabilitiesEntry ::= SEQUENCE {

   capabilitiesID           Unsigned32,
   setOfCapabilitiesID      Unsigned32,
   specificCapabilitiesID   Unsigned32,
   capabilityValue          OCTET STRING }

```

```
capabilitiesID OBJECT–TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX–ACCESS read–only
  STATUS  current
  DESCRIPTION
    "This object will identify a capabilities table row"
  –– 1.3.6.1.3.8888.1.2.1.1
::= { capabilitiesEntry 1 }

setOfCapabilitiesID OBJECT–TYPE
  SYNTAX  Unsigned32
  MAX–ACCESS read–only
  STATUS  current
  DESCRIPTION "This column will identify capabilities relevant to a specific
    subsystem, e.g: Front Sensors"
  –– 1.3.6.1.3.8888.1.2.1.2
::= { capabilitiesEntry 2 }

specificCapabilitiesID OBJECT–TYPE
  SYNTAX  Unsigned32
  MAX–ACCESS read–only
  STATUS  current
  DESCRIPTION "This column will be used to indentify a specific capability"
  –– 1.3.6.1.3.8888.1.2.1.3
::= { capabilitiesEntry 3 }

capabilityValue OBJECT–TYPE
  SYNTAX  OCTET STRING
  MAX–ACCESS read–only
  STATUS  current
  DESCRIPTION ""
  –– 1.3.6.1.3.8888.1.2.1.4
::= { capabilitiesEntry 4 }

numberOfConnectedVehicles OBJECT–TYPE
  SYNTAX  INTEGER
  MAX–ACCESS read–only
  STATUS  current
  DESCRIPTION
    "This object will count the number of connected vehicles"
  –– 1.3.6.1.3.8888.1.3
::= { systemOBUGroup 3 }

connectedVehiclesTable OBJECT–TYPE
  SYNTAX  SEQUENCE OF ConnectedVehiclesEntry
  MAX–ACCESS not–accessible
  STATUS  current
  DESCRIPTION
    "This table will list the connected to this vehicle"
  –– 1.3.6.1.3.8888.1.4 – –
```

```
::= { systemOBUGroup 4 }

connectedVehiclesEntry OBJECT-TYPE
  SYNTAX  ConnectedVehiclesEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION ""
  INDEX {
     vehicleID }
  -- 1.3.6.1.3.8888.1.4.1
::= { connectedVehiclesTable 1 }

ConnectedVehiclesEntry ::= SEQUENCE {
   vehicleID              Unsigned32,
   localID            OCTET STRING,
   globalID               OCTET STRING,
   associatedOBUorRSU     OCTET STRING,
   localOrRemote          INTEGER,
   capabilities           Unsigned32}

vehicleID OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
     "This object will identify a connectedVehicles table row"
  -- 1.3.6.1.3.8888.1.4.1.1
::= { connectedVehiclesEntry 1 }

localID OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
     ""
  -- 1.3.6.1.3.8888.1.4.1.2
::= { connectedVehiclesEntry 2 }

globalID OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
     ""
  -- 1.3.6.1.3.8888.1.4.1.3
::= { connectedVehiclesEntry 3 }

associatedOBUorRSU OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
```

```
  STATUS   current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.4.1.4
::= { connectedVehiclesEntry 4 }

localOrRemote OBJECT-TYPE
  SYNTAX   INTEGER
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.4.1.5
::= { connectedVehiclesEntry 5 }

capabilities OBJECT-TYPE
  SYNTAX   Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.4.1.6
::= { connectedVehiclesEntry 6 }

sysOBUDateandTime OBJECT-TYPE
  SYNTAX   OBUDateandTime
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.5
::= { systemOBUGroup 5 }

sysOBUNMonRequest OBJECT-TYPE
  SYNTAX   Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.6
::= { systemOBUGroup 6 }

sysOBUNEventRequest OBJECT-TYPE
  SYNTAX   Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.7
::= { systemOBUGroup 7 }
```

```
sysOBUNConfRequest OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.8
::= { systemOBUGroup 8 }

sysOBUNErrors OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.9
::= { systemOBUGroup 9 }

sysOBUVehicleID OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.10
::= { systemOBUGroup 10 }

sysOBUDistanceType OBJECT-TYPE
  SYNTAX  INTEGER
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.11
::= { systemOBUGroup 11 }

sysOBUTotalDistance OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.12
::= { systemOBUGroup 12 }

sysOBUCountry OBJECT-TYPE
  SYNTAX  INTEGER
  MAX-ACCESS read-only
  STATUS  current
```

```
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.1.13
::= { systemOBUGroup 13 }

sensorGroup OBJECT-GROUP
  OBJECTS {
    numberOfRequests,
    requestID,
    requestMapID,
    requestMonControlID,
    savingMode,
    samplingFrequency,
    maxDelay,
    startTime,
    endTime,
    waitTime,
    durationTime,
    expireTime,
    lastSampleID,
    loopMode,
    nOfSamples,
    status,
    requestUser,
    numberOfRequestsControl,
    requestControlID,
    requestControlMapID,
    settingMode,
    commitTime,
    endControlTime,
    durationControlTime,
    expireControlTime,
    valuesTableID,
    statusControl,
    numberOfRequestsStatistics,
    statisticsID,
    durationTimeStatistics,
    nOfSamplesStatistics,
    minValue,
    maxValue,
    avgValue,
    numberOfSamples,
    sampleID,
    requestSampleID,
    timeStamp,
    sampleFrequency,
    previousSampleID,
    numberOfMapTypes,
    mapTypeID,
    proprietaryTypeID,
```

```
    genericMapTypeID,
    sampleUnitMapID,
    precision,
    maxMapDelay,
    maxSamplingFrequency,
    interfaceSource,
    dataSource,
    numberOfGenericTypes,
    genericTypeID,
    typeDescription,
    numberOfSampleUnits,
    sampleUnitID,
    unitDescription,
    sampleRecordedValue,
    sampleType,
    mapTypeSamplesID,
    maxNOfSamples,
    requestStatisticsID,
    sampleCheckSum }
  STATUS  current
  DESCRIPTION
    "This group includes all objects related to sensor data retrieval"
  -- 1.3.6.1.3.8888.2 --
  ::= { obuMIB 2 }

numberOfRequests OBJECT-TYPE
  SYNTAX  INTEGER
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.2.1
::= { sensorGroup 1 }

requestMonitoringDataTable OBJECT-TYPE
  SYNTAX  SEQUENCE OF RequestMonitoringDataEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION
    "This table will list all information regarding a requests on a specific
    object."
  -- 1.3.6.1.3.8888.2.2
::= { sensorGroup 2 }

requestMonitoringDataEntry OBJECT-TYPE
  SYNTAX  RequestMonitoringDataEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION ""
  INDEX {
```

```
    requestID }
  -- 1.3.6.1.3.8888.2.2.1
::= { requestMonitoringDataTable 1 }

RequestMonitoringDataEntry ::= SEQUENCE {
  requestID             Unsigned32,
  requestMonControlID   Unsigned32,
  requestMapID          Unsigned32,
  requestStatisticsID   Unsigned32,
  savingMode            INTEGER,
  samplingFrequency     Unsigned32,
  maxDelay              INTEGER,
  startTime             OBUDateandTime,
  endTime               OBUDateandTime,
  waitTime              OBUDateandTime,
  durationTime          OBUDateandTime,
  expireTime            OBUDateandTime,
  lastSampleID          Unsigned32,
  nOfSamples            Counter32,
  maxNOfSamples         Unsigned32,
  loopMode              INTEGER,
  status                INTEGER,
  requestUser           OCTET STRING}

requestID OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-create
  STATUS  current
  DESCRIPTION
    "This object will identify an individual request"
  -- 1.3.6.1.3.8888.2.2.1.1
::= { requestMonitoringDataEntry 1 }

requestMonControlID OBJECT-TYPE
  SYNTAX  Unsigned32
  MAX-ACCESS read-create
  STATUS  current
  DESCRIPTION
    "This object will identify the requestControlDataEntry related to a request"
  -- 1.3.6.1.3.8888.2.2.1.2
::= { requestMonitoringDataEntry 2 }

requestMapID OBJECT-TYPE
  SYNTAX  Unsigned32
  MAX-ACCESS read-create
  STATUS  current
  DESCRIPTION
    "This object will identify the mapTypeTable related to a requests"
  -- 1.3.6.1.3.8888.2.2.1.3
::= { requestMonitoringDataEntry 3 }
```

```
requestStatisticsID OBJECT–TYPE
  SYNTAX   Unsigned32
  MAX–ACCESS read–create
  STATUS   current
  DESCRIPTION
    "This object will identify the requestStatisticsDataEntry related to a
    request"
  –– 1.3.6.1.3.8888.2.2.1.4
::= { requestMonitoringDataEntry 4 }

savingMode OBJECT–TYPE
  SYNTAX   INTEGER {
       permanent(0),
       volatile(1) }
  MAX–ACCESS read–create
  STATUS   current
  DESCRIPTION
    "This object will identify the mode in which a specific request will be saved
    "
  –– 1.3.6.1.3.8888.2.2.1.5
::= { requestMonitoringDataEntry 5 }

samplingFrequency OBJECT–TYPE
  SYNTAX   Unsigned32
  MAX–ACCESS read–create
  STATUS   current
  DESCRIPTION
    "This object will store the sampling frequency"
  –– 1.3.6.1.3.8888.2.2.1.6
::= { requestMonitoringDataEntry 6 }

maxDelay OBJECT–TYPE
  SYNTAX   INTEGER
  MAX–ACCESS read–create
  STATUS   current
  DESCRIPTION
    "This object will store the maximum delay allowed"
  –– 1.3.6.1.3.8888.2.2.1.7
::= { requestMonitoringDataEntry 7 }

startTime OBJECT–TYPE
  SYNTAX   OBUDateandTime
  MAX–ACCESS read–create
  STATUS   current
  DESCRIPTION
    "This object will store the start time of a certain request"
  –– 1.3.6.1.3.8888.2.2.1.8
::= { requestMonitoringDataEntry 8}
```

```
endTime OBJECT–TYPE
  SYNTAX   OBUDateandTime
  MAX–ACCESS read−create
  STATUS   current
  DESCRIPTION
    "This object will store the end time of a certain request"
  –– 1.3.6.1.3.8888.2.2.1.9
::= { requestMonitoringDataEntry 9 }

waitTime OBJECT–TYPE
  SYNTAX   OBUDateandTime
  MAX–ACCESS read−create
  STATUS   current
  DESCRIPTION
    "This object will store the wait time of a certain request"
  –– 1.3.6.1.3.8888.2.2.1.10 ––
  ::= { requestMonitoringDataEntry 10 }

durationTime OBJECT–TYPE
  SYNTAX   OBUDateandTime
  MAX–ACCESS read−create
  STATUS   current
  DESCRIPTION
    "This object will store the duration time of a certain request"
  –– 1.3.6.1.3.8888.2.2.1.11
::= { requestMonitoringDataEntry 11}

expireTime OBJECT–TYPE
  SYNTAX   OBUDateandTime
  MAX–ACCESS read−create
  STATUS   current
  DESCRIPTION
    "This object will store the expire time of a certain request"
  –– 1.3.6.1.3.8888.2.2.1.12
::= { requestMonitoringDataEntry 12}

lastSampleID OBJECT–TYPE
  SYNTAX   Unsigned32
  MAX–ACCESS read−create
  STATUS   current
  DESCRIPTION
    "This object will store the ID of the last sample to be recorded"
  –– 1.3.6.1.3.8888.2.2.1.13
::= { requestMonitoringDataEntry 13 }

nOfSamples OBJECT–TYPE
  SYNTAX   Counter32
  MAX–ACCESS read−only
  STATUS   current
  DESCRIPTION
```

```
    "This object will store the total number of samples recorded"
  -- 1.3.6.1.3.8888.2.2.1.14
::= { requestMonitoringDataEntry 14 }

maxNOfSamples OBJECT-TYPE
  SYNTAX   Unsigned32
  MAX-ACCESS read-create
  STATUS   current
  DESCRIPTION
    "This object will store the max number of samples to be recorded for a
    request"
  -- 1.3.6.1.3.8888.2.2.1.15
::= { requestMonitoringDataEntry 15 }

loopMode OBJECT-TYPE
  SYNTAX   INTEGER {
       yes(1),
       no(2)}
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will identify whether the request will loop or not"
  -- 1.3.6.1.3.8888.2.2.1.16
::= { requestMonitoringDataEntry 16 }

status OBJECT-TYPE
  SYNTAX   INTEGER {
       off(0),
       on(1),
       set(2),
       delete(3),
       ready(4) }
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will identify the current status of a request"
  -- 1.3.6.1.3.8888.2.2.1.17
::= { requestMonitoringDataEntry 17 }

requestUser OBJECT-TYPE
  SYNTAX   OCTET STRING
  MAX-ACCESS read-create
  STATUS   current
  DESCRIPTION
    "This object will store the expire time of a certain request"
  -- 1.3.6.1.3.8888.2.2.1.18
::= { requestMonitoringDataEntry 18}

numberOfRequestsControl OBJECT-TYPE
  SYNTAX   INTEGER
```

```
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.2.3
::= { sensorGroup 3 }

requestControlDataTable OBJECT-TYPE
  SYNTAX  SEQUENCE OF RequestControlDataEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION
    "This table is used to identify and store information regarding all requests
    on an object."
  -- 1.3.6.1.3.8888.2.4
::= { sensorGroup 4 }

requestControlDataEntry OBJECT-TYPE
  SYNTAX  RequestControlDataEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION
    ""
  INDEX {
    requestControlID }
  -- 1.3.6.1.3.8888.2.4.1
  ::= { requestControlDataTable 1 }

RequestControlDataEntry ::= SEQUENCE {

  requestControlID     Unsigned32,
  requestControlMapID  Unsigned32,
  settingMode          INTEGER,
  commitTime           OBUDateandTime,
  endControlTime       OBUDateandTime,
  durationControlTime  OBUDateandTime,
  expireControlTime    OBUDateandTime,
  valuesTableID        Unsigned32,
  statusControl        INTEGER }

requestControlID OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "ID of a certain request"
  -- 1.3.6.1.3.8888.2.4.1.1
::= { requestControlDataEntry 1 }

requestControlMapID OBJECT-TYPE
```

```
  SYNTAX   Unsigned32
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will identify the requestControlMapID related to a certain
    request"
  -- 1.3.6.1.3.8888.2.4.1.2 --
  ::= { requestControlDataEntry 2 }

settingMode OBJECT-TYPE
  SYNTAX   INTEGER {
       permanent(0),
       volatile(1) }
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will identify the mode in which a specific request will be set"
  -- 1.3.6.1.3.8888.2.4.1.3 --
  ::= { requestControlDataEntry 3 }

commitTime OBJECT-TYPE
  SYNTAX   OBUDateandTime
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will store the commit time of a certain request"
  -- 1.3.6.1.3.8888.2.4.1.4 - -
  ::= { requestControlDataEntry 4 }

endControlTime OBJECT-TYPE
  SYNTAX   OBUDateandTime
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will store the end time of a certain request"
  -- 1.3.6.1.3.8888.2.4.1.5 --
  ::= { requestControlDataEntry 5 }

durationControlTime OBJECT-TYPE
  SYNTAX   OBUDateandTime
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will store the duration time of a certain request"
  -- 1.3.6.1.3.8888.2.4.1.6 --
  ::= { requestControlDataEntry 6 }

expireControlTime OBJECT-TYPE
  SYNTAX   OBUDateandTime
  MAX-ACCESS read-only
```

```
  STATUS  current
  DESCRIPTION
    "This object will store the expire time of a certain request"
  -- 1.3.6.1.3.8888.2.4.1.7 --
  ::= { requestControlDataEntry 7 }

valuesTableID OBJECT-TYPE
  SYNTAX  Unsigned32
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will identify the lastSampleID of the respective value related
    to a specific request"
  -- 1.3.6.1.3.8888.2.4.1.8 --
  ::= { requestControlDataEntry 8 }

statusControl OBJECT-TYPE
  SYNTAX  INTEGER {
      inactive(0),
      active(1) }
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will be used to check if there's any request on this object
    still active"
  -- 1.3.6.1.3.8888.2.4.1.9 --
  ::= { requestControlDataEntry 9 }

numberOfRequestsStatistics OBJECT-TYPE
  SYNTAX  INTEGER
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.2.5
::= { sensorGroup 5 }

requestStatisticsDataTable OBJECT-TYPE
  SYNTAX  SEQUENCE OF RequestStatisticsDataEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION
    "This table will be used to store relevant statistics regarding a certain
    request"
  -- 1.3.6.1.3.8888.2.6
::= { sensorGroup 6 }

requestStatisticsDataEntry OBJECT-TYPE
  SYNTAX  RequestStatisticsDataEntry
  MAX-ACCESS not-accessible
```

```
  STATUS   current
  DESCRIPTION ""
  INDEX {
    statisticsID }
  -- 1.3.6.1.3.8888.2.6.1
::= { requestStatisticsDataTable 1 }

RequestStatisticsDataEntry ::= SEQUENCE {

   statisticsID              Unsigned32,
   durationTimeStatistics    OBUDateandTime,
   nOfSamplesStatistics      Counter32,
   minValue                  INTEGER,
   maxValue                  INTEGER,
   avgValue                  INTEGER }

statisticsID OBJECT-TYPE
  SYNTAX   Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "Statistics ID of a certain request"
  -- 1.3.6.1.3.8888.2.6.1.1
::= { requestStatisticsDataEntry 1 }

durationTimeStatistics OBJECT-TYPE
  SYNTAX   OBUDateandTime
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will store the duration time of a certain request"
  -- 1.3.6.1.3.8888.2.6.1.2
::= { requestStatisticsDataEntry 2 }

nOfSamplesStatistics OBJECT-TYPE
  SYNTAX   Counter32
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will store the number of samples recorded"
  -- 1.3.6.1.3.8888.2.6.1.3
::= { requestStatisticsDataEntry 3 }

minValue OBJECT-TYPE
  SYNTAX   INTEGER
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "Minimum value recorded by a certain request"
  -- 1.3.6.1.3.8888.2.6.1.4
```

```
::= { requestStatisticsDataEntry 4 }

maxValue OBJECT-TYPE
  SYNTAX INTEGER
  MAX-ACCESS read-only
  STATUS current
  DESCRIPTION
    "Maximum value recorded by a certain request"
  -- 1.3.6.1.3.8888.2.6.1.5
::= { requestStatisticsDataEntry 5 }

avgValue OBJECT-TYPE
  SYNTAX INTEGER
  MAX-ACCESS read-only
  STATUS current
  DESCRIPTION
    "Average value recorded by a certain request"
  -- 1.3.6.1.3.8888.2.6.1.6
::= { requestStatisticsDataEntry 6 }

numberOfSamples OBJECT-TYPE
  SYNTAX INTEGER
  MAX-ACCESS read-only
  STATUS current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.2.7
::= { sensorGroup 7 }

samplesTable OBJECT-TYPE
  SYNTAX SEQUENCE OF SamplesEntry
  MAX-ACCESS not-accessible
  STATUS current
  DESCRIPTION
    "This table will store all values requested by a certain RequestSampleID
    which identifies the respective requestMonitoringDataTable."
  -- 1.3.6.1.3.8888.2.8
::= { sensorGroup 8 }

samplesEntry OBJECT-TYPE
  SYNTAX SamplesEntry
  MAX-ACCESS not-accessible
  STATUS current
  DESCRIPTION
    ""
  INDEX {
    sampleID }
  -- 1.3.6.1.3.8888.2.8.1
::= { samplesTable 1 }
```

```
SamplesEntry ::= SEQUENCE {
  sampleID              Unsigned32,
  requestSampleID       Unsigned32,
  timeStamp             OBUDateandTime,
  sampleFrequency       Unsigned32,
  previousSampleID      Unsigned32,
  sampleType            INTEGER,
  sampleRecordedValue   INTEGER,
  mapTypeSamplesID      Unsigned32,
  sampleCheckSum        OCTET STRING }

sampleID OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will identify a specific recorded value"
  -- 1.3.6.1.3.8888.2.8.1.1
::= { samplesEntry 1 }

requestSampleID OBJECT-TYPE
  SYNTAX  Unsigned32
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will be used to identify the request on requestControlDataTable"
  -- 1.3.6.1.3.8888.2.8.1.2
::= { samplesEntry 2 }

timeStamp OBJECT-TYPE
  SYNTAX  OBUDateandTime
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will identify the time at which a value was recorded"
  -- 1.3.6.1.3.8888.2.8.1.3
::= { samplesEntry 3 }

sampleFrequency OBJECT-TYPE
  SYNTAX  Unsigned32
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will store the sample frequency"
  -- 1.3.6.1.3.8888.2.8.1.4
::= { samplesEntry 4 }

previousSampleID OBJECT-TYPE
  SYNTAX  Unsigned32
  MAX-ACCESS read-only
```

```
  STATUS   current
  DESCRIPTION
    "This object will store the ID of the previously recorded sample from the
    same request"
  -- 1.3.6.1.3.8888.2.8.1.5
::= { samplesEntry 5 }

sampleType OBJECT-TYPE
  SYNTAX   INTEGER {
        short(0),
        medium(1),
        long(2) }
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will store the type of data being recorded.
     short=16bit
    medium=32bit
     long=64bit"
  -- 1.3.6.1.3.8888.2.8.1.6 --
  ::= { samplesEntry 6 }

sampleRecordedValue OBJECT-TYPE
  SYNTAX   INTEGER
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will store sensor readings"
  -- 1.3.6.1.3.8888.2.8.1.7 --
  ::= { samplesEntry 7 }

mapTypeSamplesID OBJECT-TYPE
  SYNTAX   Unsigned32
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will point to the description of a signal."
  -- 1.3.6.1.3.8888.2.8.1.8 --
  ::= { samplesEntry 8 }

sampleCheckSum OBJECT-TYPE
  SYNTAX   OCTET STRING
  MAX-ACCESS read-only
  STATUS   current
  DESCRIPTION
    "This object will be used to store a checksum of an recorded value, this
    checksum will be created based on timestamp and the name of the CAN node, and
    will so as to identify multiple readings from the same CAN message"
  -- 1.3.6.1.3.8888.2.8.1.9 --
  ::= { samplesEntry 9 }
```

```
numberOfMapTypes OBJECT-TYPE
  SYNTAX  INTEGER
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.2.9
::= { sensorGroup 9 }

mapTypeTable OBJECT-TYPE
  SYNTAX  SEQUENCE OF MapTypeEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION
    "This table will map proprietary manufacturers ECUs into generic types
    defined on genericTypesTable."
  -- 1.3.6.1.3.8888.2.10
::= { sensorGroup 10 }

mapTypeEntry OBJECT-TYPE
  SYNTAX  MapTypeEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION ""
  INDEX {
    mapTypeID }
  -- 1.3.6.1.3.8888.2.10.1
::= { mapTypeTable 1 }

MapTypeEntry ::= SEQUENCE {
  mapTypeID               Unsigned32,
  proprietaryTypeID       Unsigned32,
  genericMapTypeID        Unsigned32,
  sampleUnitMapID         Unsigned32,
  precision               INTEGER,
  maxSamplingFrequency    Unsigned32,
  maxMapDelay             INTEGER,
  dataSource              OCTET STRING,
  interfaceSource         OCTET STRING }

mapTypeID OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will identify a certain Map Type"
  -- 1.3.6.1.3.8888.2.10.1.1
::= { mapTypeEntry 1 }
```

```
proprietaryTypeID OBJECT–TYPE
  SYNTAX   Unsigned32
  MAX–ACCESS read–only
  STATUS   current
  DESCRIPTION ""
  –– 1.3.6.1.3.8888.2.10.1.2
::= { mapTypeEntry 2 }

genericMapTypeID OBJECT–TYPE
  SYNTAX   Unsigned32
  MAX–ACCESS read–only
  STATUS   current
  DESCRIPTION
    "This object will contain the generic type of data recorded, this generic
    type of data is stored on the genericTypesTable"
  –– 1.3.6.1.3.8888.2.10.1.3
::= { mapTypeEntry 3 }

sampleUnitMapID OBJECT–TYPE
  SYNTAX   Unsigned32
  MAX–ACCESS read–only
  STATUS   current
  DESCRIPTION
    "This object will identify the unit in which samples are taken, this unit is
    stored on the sampleUnitsTable"
  –– 1.3.6.1.3.8888.2.10.1.4
::= { mapTypeEntry 4 }

precision OBJECT–TYPE
  SYNTAX   INTEGER (0| 1..9999999)
  MAX–ACCESS read–only
  STATUS   current
  DESCRIPTION
    "This object will identify the precision of a particular sensor"
  –– 1.3.6.1.3.8888.2.10.1.5
::= { mapTypeEntry 5 }

maxSamplingFrequency OBJECT–TYPE
  SYNTAX   Unsigned32
  MAX–ACCESS read–only
  STATUS   current
  DESCRIPTION
    "This object will identify the maximum sampling frequency of a particular
    sensor"
  –– 1.3.6.1.3.8888.2.10.1.6
::= { mapTypeEntry 6 }

maxMapDelay OBJECT–TYPE
  SYNTAX   INTEGER
  MAX–ACCESS read–only
```

```
  STATUS  current
  DESCRIPTION
    "This object will identify the maximum delay of a particular sensor"
  -- 1.3.6.1.3.8888.2.10.1.7
::= { mapTypeEntry 7 }

dataSource OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will identify the sensor, for example 'FMCW Sensor'"
  -- 1.3.6.1.3.8888.2.10.1.8
::= { mapTypeEntry 8 }

interfaceSource OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will identify the interface from which data is being read, for
    example 'CAN 2.0'"
  -- 1.3.6.1.3.8888.2.10.1.9
::= { mapTypeEntry 9 }

numberOfGenericTypes OBJECT-TYPE
  SYNTAX  INTEGER
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.2.11
::= { sensorGroup 11 }

genericTypesTable OBJECT-TYPE
  SYNTAX  SEQUENCE OF GenericTypesEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION
    "This table will contain a generic description of the type of data a certain
    sensor is generating, for example: 'Vehicle velocity'."
  -- 1.3.6.1.3.8888.2.12
::= { sensorGroup 12 }

genericTypesEntry OBJECT-TYPE
  SYNTAX  GenericTypesEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION ""
  INDEX {
```

```
    genericTypeID }
  -- 1.3.6.1.3.8888.2.11.1
::= { genericTypesTable 1 }

GenericTypesEntry ::= SEQUENCE {
  genericTypeID    Unsigned32,
  typeDescription OCTET STRING }

genericTypeID OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will identify a certain type description"
  -- 1.3.6.1.3.8888.2.11.1.1
::= { genericTypesEntry 1 }

typeDescription OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will contain generic information regarding the types of data
    that can be recorded"
  -- 1.3.6.1.3.8888.2.11.1.2
::= { genericTypesEntry 2 }

numberOfSampleUnits OBJECT-TYPE
  SYNTAX  INTEGER
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.2.13
::= { sensorGroup 13 }

sampleUnitsTable OBJECT-TYPE
  SYNTAX  SEQUENCE OF SampleUnitsEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION
    "This table will contain the unit with which a sensor is recording data, for
    example: 'Km/h'.
    This will define the coding algorithm for the Precision object on the
    mapTypeTable."
  -- 1.3.6.1.3.8888.2.14
::= { sensorGroup 14 }

sampleUnitsEntry OBJECT-TYPE
  SYNTAX  SampleUnitsEntry
```

```
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION ""
  INDEX {
    sampleUnitID }
  -- 1.3.6.1.3.8888.2.14.1
::= { sampleUnitsTable 1 }

SampleUnitsEntry ::= SEQUENCE {

  sampleUnitID     Unsigned32,
  unitDescription OCTET STRING }

sampleUnitID OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will identify a certain unit description"
  -- 1.3.6.1.3.8888.2.14.1.1
::= { sampleUnitsEntry 1 }

unitDescription OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will contain the units in which sensors record their data"
  -- 1.3.6.1.3.8888.2.14.1.2
::= { sampleUnitsEntry 2 }

errorGroup OBJECT-GROUP
  OBJECTS {
    numberOfErrorDescriptions,
    numberOfErrors,
    errorID,
    errorTimeStamp,
    errorDescriptionID,
    errorDescrID,
    errorDescr,
    errorUser,
    errorExpireTime,
    errorCode }
  STATUS  current
  DESCRIPTION
    "This group includes all objects related to errors"
  -- 1.3.6.1.3.8888.3 --
  ::= { obuMIB 3 }

numberOfErrors OBJECT-TYPE
```

```
  SYNTAX  INTEGER
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.3.1
::= { errorGroup 1 }

errorTable OBJECT-TYPE
  SYNTAX  SEQUENCE OF ErrorEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION
    "This table will contain information regarding active error codes."
  -- 1.3.6.1.3.8888.3.2
::= { errorGroup 2 }

errorEntry OBJECT-TYPE
  SYNTAX  ErrorEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION
    ""
  INDEX {
    errorID }
  -- 1.3.6.1.3.8888.3.2.1
::= { errorTable 1 }

ErrorEntry ::= SEQUENCE {

   errorID            Unsigned32,
   errorTimeStamp     OBUDateandTime,
   errorDescriptionID Unsigned32,
   errorUser          OCTET STRING,
   errorExpireTime    OCTET STRING }

errorID OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will identify all currently active reported errors"
  -- 1.3.6.1.3.8888.3.2.1.1
::= { errorEntry 1 }

errorTimeStamp OBJECT-TYPE
  SYNTAX  OBUDateandTime
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
```

```
    "This object will store the time in which an error was first reported"
  -- 1.3.6.1.3.8888.3.2.1.2
::= { errorEntry 2 }

errorDescriptionID OBJECT-TYPE
  SYNTAX  Unsigned32
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will contain the description of a certain error, this
    description is stored on the errorDescriptionTable"
  -- 1.3.6.1.3.8888.3.2.1.3
::= { errorEntry 3 }

errorUser OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will be used to store the user whose actions triggered an error"
  -- 1.3.6.1.3.8888.3.2.1.4
::= { errorEntry 4 }

errorExpireTime OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will be used with errorTimeStamp to delete an error entry after
    the expire time has been passed"
  -- 1.3.6.1.3.8888.3.2.1.5 --
  ::= { errorEntry 5 }

numberOfErrorDescriptions OBJECT-TYPE
  SYNTAX  INTEGER
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.3.3
::= { errorGroup 3 }

errorDescriptionTable OBJECT-TYPE
  SYNTAX  SEQUENCE OF ErrorDescriptionEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION
    "This table will be used to store all possible errors, both active or
    otherwise, and their descriptions."
  -- 1.3.6.1.3.8888.3.4
```

```
::= { errorGroup 4 }

errorDescriptionEntry OBJECT-TYPE
  SYNTAX  ErrorDescriptionEntry
  MAX-ACCESS not-accessible
  STATUS  current
  DESCRIPTION
    ""
  INDEX {
    errorDescrID }
  -- 1.3.6.1.3.8888.3.4.1
::= { errorDescriptionTable 1 }

ErrorDescriptionEntry ::= SEQUENCE {

   errorDescrID  Unsigned32,
   errorDescr    OCTET STRING,
   errorCode     Unsigned32 }

errorDescrID OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will identify a certain error description"
  -- 1.3.6.1.3.8888.3.4.1.1
::= { errorDescriptionEntry 1 }

errorDescr OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will provide a generic description to the error being reported"
  -- 1.3.6.1.3.8888.3.4.1.2
::= { errorDescriptionEntry 2 }

errorCode OBJECT-TYPE
  SYNTAX  Unsigned32
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will contain the current error code that was triggered by user
    action"
  -- 1.3.6.1.3.8888.3.4.1.3
   ::= { errorDescriptionEntry 3 }

actuatorGroup OBJECT-GROUP
  OBJECTS {
    numberOfCommandTemplates,
```

```
    numberOfCommands,
    commandID,
    templateID ,
    commandInput ,
    commandUser ,
    commandTemplateID ,
    commandDescription ,
    targetNode ,
    commandTemplate }
  STATUS current
  DESCRIPTION
    "This group includes all objects related to actuators"
  -- 1.3.6.1.3.8888.4 --
  ::= { obuMIB 4 }

numberOfCommandTemplates OBJECT-TYPE
  SYNTAX INTEGER
  MAX-ACCESS read-only
  STATUS current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.4.1
::= { actuatorGroup 1 }

commandTemplateTable OBJECT-TYPE
  SYNTAX SEQUENCE OF CommandTemplateEntry
  MAX-ACCESS not-accessible
  STATUS current
  DESCRIPTION
    "This table will contain CAN command templates to be used when activating/
    deactivating actuators"
  -- 1.3.6.1.3.8888.4.2
::= { actuatorGroup 2 }

commandTemplateEntry OBJECT-TYPE
  SYNTAX CommandTemplateEntry
  MAX-ACCESS not-accessible
  STATUS current
  DESCRIPTION ""
  INDEX {
    commandTemplateID }
  -- 1.3.6.1.3.8888.4.2.1
  ::= { commandTemplateTable 1 }

CommandTemplateEntry ::= SEQUENCE {
  commandTemplateID     Unsigned32 ,
  commandDescription    OCTET STRING,
  targetNode            OCTET STRING,
  commandTemplate       OCTET STRING }
```

```
commandTemplateID OBJECT-TYPE
  SYNTAX  Unsigned32 (1..99999999)
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will identify a specific command template"
  -- 1.3.6.1.3.8888.4.2.1.1 --
  ::= { commandTemplateEntry 1 }

commandDescription OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will store a short description of what a command does"
  -- 1.3.6.1.3.8888.4.2.1.2 --
  ::= { commandTemplateEntry 2 }

targetNode OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will store the Node ID to which a command will be sent"
  -- 1.3.6.1.3.8888.4.2.1.3 --
  ::= { commandTemplateEntry 3 }

commandTemplate OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    "This object will store a template of a command in hex. eg: FF FF FF ** ** FF
     FF FF, where * indicate where user input will be placed"
  -- 1.3.6.1.3.8888.4.2.1.4 --
  ::= { commandTemplateEntry 4 }

numberOfCommands OBJECT-TYPE
  SYNTAX  INTEGER
  MAX-ACCESS read-only
  STATUS  current
  DESCRIPTION
    ""
  -- 1.3.6.1.3.8888.4.3
::= { actuatorGroup 3 }

commandTable OBJECT-TYPE
  SYNTAX  SEQUENCE OF CommandEntry
  MAX-ACCESS not-accessible
  STATUS  current
```

```
  DESCRIPTION
    "This table will contain all commands that are to be sent into the CAN network"
  -- 1.3.6.1.3.8888.4.4
::= { actuatorGroup 4 }

commandEntry OBJECT-TYPE
  SYNTAX   CommandEntry
  MAX-ACCESS not-accessible
  STATUS   current
  DESCRIPTION ""
  INDEX {
    commandID }
  -- 1.3.6.1.3.8888.4.4.1
::= { commandTable 1 }

CommandEntry ::= SEQUENCE {

  commandID       Unsigned32,
  templateID      Unsigned32,
  commandInput    INTEGER,
  commandUser     OCTET STRING}

commandID OBJECT-TYPE
  SYNTAX   Unsigned32 (1..99999999)
  MAX-ACCESS read-create
  STATUS   current
  DESCRIPTION
    "This object will identify a certain command"
  -- 1.3.6.1.3.8888.4.4.1.1 --
  ::= { commandEntry 1 }

templateID OBJECT-TYPE
  SYNTAX   Unsigned32
  MAX-ACCESS read-create
  STATUS   current
  DESCRIPTION
    "This object will be used to store the ID of the commandTemplate to be used in this command"
  -- 1.3.6.1.3.8888.4.4.1.2 --
  ::= { commandEntry 2 }

commandInput OBJECT-TYPE
  SYNTAX   INTEGER
  MAX-ACCESS read-create
  STATUS   current
  DESCRIPTION
    "This object will be used to store the user inputs that are to be used in the command"
  -- 1.3.6.1.3.8888.4.4.1.3 --
```

```
  ::= { commandEntry 3 }

commandUser OBJECT-TYPE
  SYNTAX  OCTET STRING
  MAX-ACCESS read-create
  STATUS  current
  DESCRIPTION
    "This object will store the username of the user that set this command"
  -- 1.3.6.1.3.8888.4.4.1.4 --
  ::= { commandEntry 4 }

OBUDateandTime ::= OCTET STRING (SIZE (11 | 13))

END
```

Listing 1: Full MIB Specification