CAN-bus system for vehicle actuation and data logging with Arrowhead Framework

Andreas Månsson

Computer Science and Engineering, bachelor's level
2019

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Supervisors: Ulf Bodin, Niklas Karvonen and Jan van Deventer
CAN-bus system for vehicle actuation and data logging with Arrowhead Framework
Keywords

Arrowhead Framework
Autonomous vehicles
Service prosumer
RAMI 4.0
Abstract

The use of micro controllers in automotive application have exploded during the last half century. What was initially a set of mechanical systems that formed a vehicle have now become a collection of computers on wheels. The reason is quite obvious: micro controllers use several inputs to optimize the performance of systems; for example an engine control or an active safety system.

The different inputs and outputs to these electronic units (electronic control unit, ECU) are of interest to other such units thereby justifying the need of inter-ECU communications. The Controller Area Network (CAN) bus has been developed to facilitate this communication. It is a message based protocol and is very resilient. It is however relatively slow and limited in terms of security. Security is assured only by trying to keep the message identification tags confidential and the bus physically separated to other network.

A couple of decades ago our society embraced the Information Technology (IT) revolution. It allowed people to have extensive access to information. From a technology point of view, IT is based on the use of the Internet, which has been initially designed by the US military for robust applications. It is fast and its security is sufficiently high that we use it to communicate with our banks where we keep all our life savings.

The aim of this thesis has been to combine these technologies such that a vehicle with a CAN bus could offer services (just like a bank does) over the Internet. The goal then is to transform a CAN bus to become a service provider over the Internet. The services are the broadcasted CAN messages made available to authorized interested parties and can post information and actuations to the ECUs connected to the CAN bus. A vehicle in that case becomes a cyber physical system.

To make this transformation possible, we use the open source Arrowhead Framework, which is based on a Service Oriented Architecture (SOA). The available services are made known via a Service Registry and Orchestration service prosumers. Concretely, the work in this thesis project has been to develop (i.e., to design and implement) a CAN service prosumer that is Arrowhead Framework compliant. It has been successfully tested with another service prosumer, which is an Arrowhead Framework compliant data logger.

The driving motivation for the thesis project are construction equipment machines, such as wheel loaders and excavators, which are vehicles with booms or arms. The aspiration is that they not only drive autonomously but also dig autonomously. This ambition shall require large amount of data to be exchanged, something that a CAN bus cannot handle.
CAN-bus system for vehicle actuation and data logging with Arrowhead Framework
## Contents

1 Introduction .......................................................... 1
   1.1 Background ..................................................... 1
   1.2 Motivation ..................................................... 3
   1.3 Problem Definition ........................................... 5
   1.4 Task ............................................................. 5
   1.5 Delimitation .................................................... 6

2 Related work ......................................................... 7

3 Theory ........................................................................... 9
   3.1 System of systems ............................................... 9
   3.2 Service-oriented Architecture ................................ 9
   3.3 Arrowhead ......................................................... 9
      3.3.1 Orchestrator ................................................. 10
      3.3.2 Service registry ........................................... 10
      3.3.3 Authenticator .............................................. 10
      3.3.4 Provider ..................................................... 11
      3.3.5 Consumer .................................................... 11
      3.3.6 Service prosumer .......................................... 11
   3.4 CAN bus ............................................................ 12
      3.4.1 OBD2 ......................................................... 12
   3.5 Software principles ............................................. 13
      3.5.1 Cohesion and coupling ................................... 13

4 Implementation ........................................................ 15
   4.1 Tools ............................................................... 15
   4.2 System architecture .............................................. 15
   4.3 System components .............................................. 16
      4.3.1 CanBusProvider .......................................... 17
      4.3.2 CanBusApp ............................................... 18
      4.3.3 Client-common ........................................... 20

5 Evaluation .............................................................. 23
   5.1 System field-test ............................................... 23
   5.2 Evaluating the system ........................................... 25

6 Discussion ............................................................... 27
7 Conclusions and future work

7.1 Conclusion

7.2 Future work

7.2.1 Improvement on network communication

7.2.2 Protocol for the earth-moving machine

7.2.3 Improvement on reading from the CAN bus

7.2.4 Automated test suite

7.2.5 Security

Bibliography

A Appendixes

A.1 A detailed UML over CanBusApp

A.2 A detailed UML over Can bus provider

A.3 A detailed UML over the total system

A.4 Example code-snippets

A.4.1 An API call
1 Introduction

The vision for future construction equipment is autonomous vehicles (see figure 1.1). They will need much computer power and computer communication. This thesis work is about making such solution possible through service-oriented architecture (SOA). That is Electronic control units (ECU) signals and states offered as services, i.e., information and actuation over the vehicle’s internet. More specifically, we bridge today’s current technology (CAN bus) to the internet with SOA by using a service prosumer. The CAN service prosumer developed here offers the CAN messages as services over the internet.

Figure 1.1: Future vision of a wheel loader, Volvo CE and LEGO.

1.1 Background

EISLAB (Embedded Intelligent Systems lab) at Luleå University of Technology together with Volvo CE (Construction Equipment) is involved in developing a system for remotely controlling wheel loaders (see figure 1.2) with two different solutions for automatic loading.
Siddarth Dadhich wrote his doctoral thesis on "Automation of Wheel-Loaders" [1]. In his thesis he explains the two solutions and the system he implemented for autonomous bucket filling. The two solutions are based on machine learning, where the solutions differs in that one of the solutions uses "imitation learning" and the other "reinforcement learning".

Five steps to consider when automating machines are:

- Manual operation,
- In-sight tele-operation,
- Tele-remote operation,
- Assisted tele-remote operation,
- Fully autonomous operation.

Each of these steps requires a set of features to properly function and with each of the steps the automation system gradually is adapted towards a more mature solution, which enables extensive and more efficient automation.

This type of automation system typically includes an on-board subsystem of:

- Cameras,
- Sensors,
- Actuators,
- Wireless communications,
- A processing platform.

Together with an off-board environment for remote drivers, comprising of:
• Cockpit,
• Video screen,
• Mechanisms for tactile motion feedback.

Additionally an Industrial Control System (ICS) providing integration with other parts in the industrial automation system.

In such a system as described there is a need to extract data from the vehicles sensors to process the next step in the automation process and also to let the automation process actuate on the machine. Because of that there is a need for a communication system for data collection and actuation on the vehicle to enable the automation process.

1.2 Motivation

The system this thesis focuses on implementing is a CAN-bus system. The autonomous system needs to have a CAN-bus application system to read the data coming from sensors on the earth-moving machine as well as actuating the machine. The result of implementing this system would make progress with the autonomous solution for earth-moving machines.

Making the software with regards to adaptation, update and reuse is important when thinking about the man hours that needs to be spent on upgrading, modifying or adapting the software. An assumption can be made that if the problems defined in this thesis is solved, lowering the time that has to be consumed for such tasks would reduce the engineering cost. With the following attributes of adaptation, update and reuse, certain characteristics can be defined as following for these attributes as such:

• Adaptation: To be able to easily adapt to different scenarios and handle updates which changes the scenario for the system. A different scenario could be using a different programming language for the implementation or a different machine to machine solution, meaning instead of CAN bus using Open Platform Communication-Unified Architecture (OPC-UA), which is another machine to machine protocol.

• Update: To be able to easily update, add and remove functionality within the system. An example would be to add a new data-resource in the provider for consumers to consume.

• Reuse: To be able to easily use one or more components from the system in another different system. This could be to reuse the CAN bus solution in another provider as an example.

Making the software with regards to update and adaptation also opens the door for testing the CAN-bus solution on different types of vehicles and not just one. This makes it easier to not only test but to verify before implementing on different, more fragile or time-sensitive projects but also combines the potential development of two different projects in to one single solution, again, saving engineering time and resources.
A thorough documentation would save engineering costs and resources for other users that wishes to modify or extend the software to their own liking. As an example if EISLAB wants to use this software hopefully the documentation that is required for them to modify and extend the software will make the procedure easier.

This project is being implemented using Arrowhead Framework, which is a framework which specifically targets the industry and uses a SOA. Arrowhead is used because of the need to quickly being able to switch the two different automation solutions at run-time and because problems could occur when different collaborators or third-parties continues to build on the solution over time. With a risk to subsequently implement faulty code, which could damage the machines and void guarantees. In the aid to avoid this, the Arrowhead framework and open-source could help to control what needs to be certified by the original equipment manufacturer before being implemented [2].

Using SOA also makes it easier to offer each individual available CAN message as a service, that is available over the internet for service consumers, view figure 1.3. Additionally it will enable an easy control of which consumers may be able to consume what service and an easy way to add new features to the system, i.e, features as in a new CAN message offered as a service.

![Figure 1.3: CAN bus application system with Service-oriented architecture, Where CAN messages are offered as a service over the internet.](image)

The result of implementing the CAN-bus system would be to enable the automation process and thus have the potential to make the environment around the machine safer for workers and could potentially increase productivity. The environment would be safer when fewer humans are involved at the excavation site and since the machines does not need to rest, the productivity would increase.
1.3 Problem Definition

The problem this thesis tries to solve is to partly enable the vision Volvo CE has to automate earth-moving machine. In today’s technology the CAN bus system is inflexible and slow. By integrating the CAN bus into a Service-oriented architecture system and presenting each individual CAN message as a service over the internet, a flexibility that is more up to date by today’s standard can be achieved. By enabling the CAN to communicate over the internet, it introduces more flexibility in distributing the information needed to log data and enable an autonomous solution that is not possible with the CAN bus alone today.

1.4 Task

The task during this thesis involves in implementing a CAN-bus application system to allow for both collecting data and actuate earth-moving machines in the Arrowhead-framework. Figure 1.4 refers to the system architecture and where the CAN-bus application fits in the automation system as a whole. The CAN-bus application needs to produce data for the database system for storage and also send the data back to the control application system for further actuation decisions. Furthermore consuming data from the control application system for actual actuation on the wheel-loader.

The software will be implemented with the purpose of being easily updated and adapted for different implementations but also different kinds of earth-moving vehicles. With such a solution there is also a need for a clear and structured documentation of the software design so that other users may extend on the software after this thesis. This leads us to the task of implementing software for a CAN-bus system application in Arrowhead-framework with regards to:

- adaptation,
- update,
- reuse

and thorough documentation of the software for easier understanding and for people who wishes to extend the software or reuse.
Delimitation

Security will be considered in the development but is not covered in this thesis. Arrowhead does have security built into it and thus does not need any focus in this thesis. Additionally considering the time-frame security will not be used in the implementation, but is a point for future development. Security needs to be addressed in further development due to the nature of information that comes out of a CAN-bus, which could be sensitive data.

Because ease of access and due to time, the solution is only be tested on a Skoda Octavia via its diagnostic port. No access to a Volvo CE wheel-loader and a short time-frame made testing on a wheel-loader impossible.

This thesis is not a finished solution and has to be extended upon after the thesis for a ready solution. As the software is made with the sought after attributes mentioned earlier, together with documentation, it does make it easier for users to extend on the software at a later date.
2 | Related work

Collecting vehicle data within a framework for smart solutions is not a new and unique idea. The agricultural sector has been collecting information from machines among many things with their Farming management information system [3]. In the article *Multi-level automation of farm management information systems* they implement a can bus communication solution within Fispace platform [4], which is based on FIWARE. FIWARE [5] is an open-source framework which compared to Arrowhead tries to solve the same problems of accelerating development of smart solutions. This solution collects vehicle data for further analysis via a CAN bus, similar to this thesis. However they do not actuate on the machine via the CAN bus, only collect information, unlike the solution in this thesis, which does both. The main difference is in scope, FIWARE is more general for all solutions while Arrowhead is more towards industry solutions.

A similar case where SOA-principles was applied to automotive software systems, this time to prove that it is feasible to run a CAN-based communication on a SOA-based system and to implement a Distributed Driver Assistance Systems which could be lane-keeping assistance, electronic stability program for the vehicle as an example [6]. The main differences between the job done in the article *A CAN-based Communication Model for Service-Oriented Driver Assistance Systems* and this thesis is that they created the systems for SOA from scratch with their own communication model. Being more of an analysis of the performance and possibility about using SOA within CAN-based communication. This thesis focuses more on implementing a service prosumer within an existing SOA framework, Arrowhead. With implementation being more of the focus, with less time to analyze because of the short time-frame.

Apart from CAN bus there are other machine to machine protocols, as for example OPC-UA. OPC-UA is an industrial standard for machine to machine protocols and is used widely. An example of SOA with OPC-UA is a project that was done by trying to design and implement SOA for optimizing industrial applications [7]. What the article *OPC-UA and DPWS interoperability for factory floor monitoring using complex event processing* describes is that they create a SOA for industrial purposes, with the help of OPC-UA servers. The resulting implementation could sustain a high number of connections and perform well during a real manufacturing process. The similarity between the OPC-UA solution and this thesis is the use of a System-oriented architecture to achieve flexible and interoperable systems. The key difference however is that the goal of the OPC-UA solution is creating the SOA from scratch and analyzing the performance in a manufacturing process with OPC-UA protocol for communication between machines. This thesis uses Arrowhead Framework for SOA properties, with focus on implementing a solution that uses CAN bus for machine to machine communication.
CAN-bus system for vehicle actuation and data logging with Arrowhead Framework
3 | Theory

3.1 System of systems

System of systems is a collection of sub-systems that together form a more complex system that can handle tasks that none of the systems can accomplish on its own [8]. System of systems fundamentally needs interoperability between the systems and if that is enabled you can pair sub-systems to achieve a complete solution, as illustrated in figure 3.1.

![Figure 3.1: System of systems illustrated as lego blocks.](image)

3.2 Service-oriented Architecture

SOA is the idea that organizations and software should be able to provide the functionality as a service, which then consumers of said service can request and access.

The environment being unpredictable in the sense that consumers can be of any type, interoperability is essential. Because of this, it’s well suited to use a web-service when serving the customer with resources or functionality [9].

3.3 Arrowhead

The Arrowhead framework is a framework which provides local clouds with system of systems to create an Internet of Things(IoT)-based automation. Local cloud is
the collection of systems which works together via the internet, which means the systems does not need to be in the same physical location. In Arrowhead there’s three mandatory systems or prosumers that needs to be in the local cloud [2]. This is:

- Orchestrator.
- Service registry.
- Authenticator.

Each of these systems makes the local cloud possible by each fulfilling certain duties to enable the system of systems. The Arrowhead framework is built with a SOA. Following is about some core concepts in the Arrowhead framework.

### 3.3.1 Orchestrator

The Orchestrator system is the orchestrator of the systems, its the link that keeps the Service registry and the Authenticator system together. It enables remote control of which service instances a consumer should/can consume. This means that the Orchestrator can tell the consumer where to access a service by asking the service registry and tell you if you have permission to access the service or not by asking the Authenticator.

### 3.3.2 Service registry

The Service registry system in Arrowhead is what makes it possible for service providers to publish each of its services. By publishing it in the Service registry a consumer would be able to find the currently available service for consumption. Meaning, if a consumer would like to consume the service \( X \), consumer would first ask the Orchestrator if service \( X \) is available. The Orchestrator would then ask the Service registry if service \( X \) is published, if it is the Service registry will tell the Orchestrator if it exists and where it exists for the consumer to access it and the Orchestrator will forward this information to the consumer.

### 3.3.3 Authenticator

The Authenticator system is the system that helps providers handle which consumer can access what specific service. This means that the provider offering a service which only consumer A can access, the Authenticator will deny all other consumers that is not consumer A. So if consumer B which is trying to consume the service, asks the Orchestrator if he can access the service, the Orchestrator will ask the Authenticator whether consumer B can access or not, Authenticator will reply to the Orchestrator that the consumer can not access the service and the Orchestrator will forward this information back to the consumer.

Figure 3.2 explains a case where the user is not allowed to access a service in the Authenticator, while 3.3 depicts a case where the user is allowed to access the service.
3.3.4 Provider

A service provider in the Arrowhead Framework is an Arrowhead system that produces a service for consumers to consume. It can be as simple as offering a service to tell the time if asked, to more advance services offering actuation on vehicles.

3.3.5 Consumer

A service consumer in the Arrowhead Framework is an Arrowhead system that consumes a service produced by a provider system. It could be a LCD clock asking for the time every 100 ms from a service to display on the LCD as an example.

3.3.6 Service prosumer

An Arrowhead Framework system that produces and consumes services. All Arrowhead Framework systems are prosumers.
3.4 CAN bus

Controller Area Network (CAN) was developed by Bosch Automotive in the 80s to facilitate inter ECU (Electronic control unit) communication [10]. The CAN-bus which are connected to the ECU’s, can broadcast messages to all the nodes that are connected to the CAN-bus for information about the sensors which are connected in the car, as vehicle speed, engine coolant temperature and so on. The major drawback with CAN-buses that exists today is that they send data with a speed limit of 1 Mbit/s. In the future however a new version of CAN-bus called CAN FD will be demanded in new cars as that would increase the transmission rate to 8 Mbit/s, as well as increase the data packet size from regular CAN 8 bytes to 64 bytes. [10]

3.4.1 OBD2

OBD stands for On-board diagnostics. It is an automotive term that refers to the reporting capability and self-diagnoses of vehicles. OBD is a higher-level protocol of how the communication is supposed to be conducted. OBD2 can use many three different kind of buses to communicate over but in this project the focus is on CAN bus. [11]

The message structure of a OBD2 message is shown in Table 3.1.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Bytes</th>
<th>Mode</th>
<th>PID</th>
<th>Ah</th>
<th>Bh</th>
<th>Ch</th>
<th>Dh</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 bit</td>
<td>64 bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Message structure of an OBD2 message.

each of the column means:

- Identifier: 11-bit identifier to distinguish between response or request messages.

- Bytes: The number of bytes of data and information about the packet to follow after this.

- Mode: What mode you want to retrieve the data from. There are 10 different modes. Mode 1 shows current data, mode 2 contains data since last error call on the CAN etc.

- PID: Parameter IDs are codes which are used to request specific data from the vehicle, as for example RPM, speed etc.

- Ah-Dh: These are the data bytes containing the answer in the requests. For every request there are different conversion formulas to get the bytes to a readable form as integers, floats etc.

An example of how a response message might look can be found in table 3.2.
Table 3.2: Example of a response OBD2 message, which contains the speed of the vehicle, which is 50km/h.

| 7DF | 03 | 41 | 0D | 32 | aa | aa | aa |

Observe here that the first field is the identifier, it tells us that it is a response message. After that we get how many bytes of data that follows, which is three bytes of data. Then what follows is the mode, when a response message is sent 0x40 is added to what you asked for, so if you asked for mode 0x01 the response message will show you 0x40+0x01 which will be 0x41. Second to last we see 0x0D which is the PID for vehicle speed. Last we see 0x32 which is the actual value to describe the vehicle speed. After that we reach the limit specified earlier of how much data which would follow the 0x03, which means that the following data is redundant data we don’t acknowledge, i.e, we don’t do anything with the rest of the data which is 0xAA, 0xAA and 0xAA.

### 3.5 Software principles

The following is a short theory behind the principles of cohesion and coupling within software engineering which is used in the implementation to achieve adaptation, update and reuse.

#### 3.5.1 Cohesion and coupling

Cohesion is to which degree the software elements is keeping its focus on performing a single task, high cohesion would be a focused module that tries to complete a single task (focused), while low cohesion would be a module that tries to complete several tasks (unfocused).  

Coupling is to which degree the modules are dependant on each other, low coupling would be that the modules in the system depend very slightly on each other to work, while high coupling would be a system where modules depend on each other to work at all.
CAN-bus system for vehicle actuation and data logging with Arrowhead Framework
4 | Implementation

This section is a technical overview of the implemented system, it includes the tools used to implement the system, the systems architecture and the necessary components in the system.

4.1 Tools

The tools used to implement the system is as follows:

- Programming language: Java 11.0.2.
- Dependency manager: Apache Maven 3.6.0.
- Framework: Arrowhead framework 4.1.2.
- Operating system: Ubuntu 18.04.2 LTS.
- CAN drivers: Kvaser Linux driver and SDK V5.27.776.
- CAN connection device: Kvaser BlackBird SemiPro HS and Kvaser PCIEcan 2xHS v2.

The BlackBird is a USB CAN solution and Kvaser PCIEcan is a PCI-express card that is plugged into the PCI-express port in a stationary computer. The USB solution was used in the field test and the PCI-express card was used first to test the solution in a car, before getting access to the USB CAN solution.

4.2 System architecture

This part focuses on a broad overview of the implemented system and its architecture.

A system was found and built upon from Arrowheads Github [13]. The foundation was for a simple service provider which simply gave a response to a request that was hard-coded. Building upon this made it easier to achieve a desirable solution within the given time-frame.

A big effort was made to keep each part of the system as loosely coupled and with high cohesion as possible with focus on the CAN bus application part of the system. This to make it easier for adaptation, update and reuse. The main idea was that if there was a need for a change, one could specifically target the component that
needed to be modified without having to much problems in the other components of the system.

In this case as seen in the module diagram of the system in figure 4.1 if there was a need to switch from CAN bus to another serial communication method the modification would need to be done in CanBusApp, which holds everything related to the communication with the CAN, while CanBusProvider would have to see to very few changes to continue working as previously. However, because of the inner workings of Arrowhead, Client-common is not available for modifications to the same degree as the CanBusApp. Making changes or replacing features in Client-common could present issues unforeseen. We also notice in the module diagram in figure 4.1 the dependencies on the system. Where CanBusProvider has dependencies on Client-common and CanBusApp for the system to work.

![Module diagram of the system and its components.](image)

The CanBusApp is the essential part which is not connected to Arrowhead in the sense that it doesn’t contain any functionality towards Arrowhead directly, thus this is the part that was focused on most to be loosely coupled and to achieve high cohesion.

### 4.3 System components

This part is a brief overview of the thinking that went into implementing the systems and how it relates to the key problems that was introduced in the introduction. For each system the core classes will be discussed in broader strokes, what it achieves and as previously stated, how it relates to the key problems. After the core classes has been introduced there will be an abstracted class UML of each of the systems. For each of the UML there is a detailed overview in the Appendixes.
4.3.1 CanBusProvider

The Provider component is responsible for everything arrowhead related. This means it has to register the services that is provided to both Service registry and Authenticator. It also manages the web-server that is started to be able to handle requests from consumers. Following will be a description of the core classes in the Provider component.

CanBusProviderMain

As the provider system boots up there is a need for services to be registered. The way this is achieved is by abstracting the registration of these services to its own entity. That way the main class of this component is only responsible for starting the server, web-server, creating the objects that will handle the registration itself and then listen for inputs from the terminal. This leads to a more clear work-flow for the main class which makes the software easier for modifications.

ServiceRegistrator and AuthorisationRegistrator

These classes solely handles the registration of services to the Service registry and the Auth registry in the Arrowhead systems.

Previously before building upon the software this was all instantiated within the main class. To achieve a better experience for someone that wishes to extend on the software it was essential to separate these to its own entity. It handles the registration by reading from a JSON file where each of the services being registered has the information that is needed for each of their registration. This way its easy to add, remove and modify existing services without having to re-compile the code.

CanApiResource

The CanApiResource class is the application programming interface (API) for all of the services. This is where paths to the services are executed and where the requests returns are defined. Having this isolated to its own class is particularly important due to the fact that having an overview of each path and the option to modify what each service offers is important for the system to be modifiable.
4.3.2 CanBusApp

The CanBusApp component is responsible for handling the CAN communication with the vehicle and handling everything CAN related. This component is creating a handle to the device which plugs into the car and makes communication possible. It is able to send and receive specific messages and is holding information regarding what data the application is able to retrieve. It also maintains everything that is needed for the system as reading in PID codes and storing the formulas needed for data conversions. A sequence diagram involving serving a consumer with a request for the revolutions per minute (RPM) from the vehicles engine can be viewed in figure 4.3, where the CanBusApiResource uses the components in the CanBusApp to help serve the consumer with the RPM request.
Main class

To achieve an adaptable and reusable solution in the CanBusApp component, there was a need for low coupling and high cohesion. To achieve this, the main class of the component would be solely responsible for CAN communication by creating a CAN-message handle to manage the communication and by doing this, making it easier to adapt if there would be a change of equipment to handle the communication to the CAN on the vehicle.

In this project a Kvaser BlackBird SemiPro HS was used because that is what Volvo CE is using, but if that would change, the low coupling would make it so that it would not greatly affect other parts of the system in the case of changing equipment, since connecting, sending and receiving is done in one part of the system.

ReadServiceJson

The next step beyond the main class, the software needed a way to read in the PID codes that would be available for requests by the consumer, therefore a class called ReadServiceJson was made which would have a responsibility of reading a JSON file to extract the PID codes for each of the requests and store them in variables in the class for further use.

The result of this is that if RPM was asked for, the PID code for RPM was stored in ReadServiceJson as a variable and could be used as such by the main class when requesting specific vehicle data. Once again further making sure there is low coupling to make it more adaptable and re-usable in case there would need to be a greater or minor change in what PID codes to read. It also goes towards high cohesion since the focus on the class is specific and does not have extra functionality apart from reading in and offering the PID codes.

By making the software work this way there is no need for re-compiling code when making PID code adjustments, however adding a new PID code to read would demand some small adjustments, as a variable is needed to read the value in and to use the variable in a request to retrieve the data.
FormulaCollection

The same principle as with a way to read the PID codes applies to having formulas to calculate the resulting data. However this was implemented in a slightly different manner.

As a received data message from the CAN contains bytes of data where for example \( rpm = (\text{byte}A \times 256 + \text{byte}B)/4 \) to receive the correct output of the RPM from the data, there is a need to have these formulas ready to make the calculation. In the class FormulaCollection there needs to be a function where the formula is applied for each PID request to get the correct output. Therefore FormulaCollection consists of small functions to calculate the data and return the answer.

![Figure 4.4: The CanBusApp abstracted class diagram](image)

4.3.3 Client-common

This module is a collection of code resources that is common between the Provider and the Consumer. It also contains code for the underlying help functionality within
Arrowhead to make the system work, if one would want to use that.

The common code resources has to be in both the Provider and Consumer due to the agreement between the two to use a shared data structure when communicating. What happens is that the message you send to each other in Arrowhead, is a class that is de-serialized from an object of that class to a JSON message and then sent over an HTTP request. So when the message arrives at the consumer its easy for the consumer to easily serialize that back to the original data-structure of that common class.
CAN-bus system for vehicle actuation and data logging with Arrowhead Framework
The resulting system following the implementation discussed in chapter 4, together with thorough documentation should suffice to give the end user confidence enough for adapting, updating and reusing the software.

However considering the time frame, only the CAN bus application component is fully subjected for the software attributes adaptation, update, reuse and not the CanBusProvider or client-common. Worth noting that the CanBusProvider was however modified to be easier to understand and has to some degree high cohesion because of the separated functionality to register the Authenticator Registractor and the Service Registractor. This because the CanBusApp will be subject for more changes than any other component. Considering that a provider in an Arrowhead system will most likely work in the same way over time and client-common having underlying structure and common resources for Arrowhead systems, will also stay similar over time, hence the rate of which these components will need to be updated will probably be less compared to CanBusApp. That said, with thorough documentation and as previously stated that most changes are likely to be in the CAN bus application component, it was most likely the right strategy considering the short time-frame.

5.1 System field-test

In terms of system functionality the system was field-tested in a car, the setup consisted of two laptops, a cellphone with a hotspot running and a car, as can be seen in figure 5.1. The laptops relied on connecting via the cellphone’s hotspot for internet, with one of the laptops connected into the car’s diagnostic port, running the CAN-bus application system and the other laptop running a database system. The connection to the car’s diagnostic port was made with the USB tool called Kvaser BlackBird SemiPro HS.
With the systems up and running, the database system (running either MongoDB or InfluxDB as a database), first asked the orchestration system in the local cloud for the RPM service. This was successfully done and the URL needed for the CAN-bus API was sent to the database system. After that the database system started polling the CAN-bus system for information regarding the vehicle's engine RPM. This was also a success and data was sent accordingly. The database system continued to poll data from the CAN-bus system for an extended period of time, revving the engine and visualizing the data on the screen so anomalies could be seen.

Thanks to the visualization, there was two particular issues that was found during the field-test, one of which was identified. Java does not support unsigned bytes. Since unsigned bytes do not exist in Java there were issues with overflows where suddenly the RPM would be -50 which is obviously wrong. This was solved by not casting to integers in an early stage of the conversion, which made the conversion formulas work correctly and give a proper result.

The other issue that was learned about and sadly never had time to investigate properly was that sometimes the RPM spiked up to a value of 5000 for only one data frame, the frequency of which it happened was somewhere along the lines of once every 1-2 seconds, where data was polled roughly at 15 times a second, where one data frame from the CAN resulted in such a value. This might also be because of some casting of bytes to integers but there’s no way of telling without more experiments.

As just mentioned, the database system was polling the CAN-bus system at 15-20 times per second, this is without limiting the software in terms of polls per second. What that means is that the database continued to poll the CAN-bus system as fast as it could. The speed might seem a bit slow, but considering it was over a hotspot connection and the systems was running on laptops it is decent enough for a field-test. Higher speeds will be needed in the finished solution, considering there is a machine learning algorithm that will need to poll for data. But just by running the systems on a factory-class computer will undoubtedly increase the speed of the polling frequency.

There was no time to actually test actuation on a vehicle due to the lack of vehicles to test on. However in theory it should not differ greatly from retrieving data.
from the CAN-bus since the CAN-bus does not differ who is sending the information and thus should react when trying to actuate on the vehicle. However that is only theoretical and needs to be tested in practicality.

5.2 Evaluating the system

The problem definition for this thesis as stated in chapter 1.3 asks for a system which could serve the vision of Volvo CE towards automating an earth-moving machine. A system that can read and actuate on a CAN-bus through the internet and act as a service provider.

The system that was implemented is a system able to serve data from a CAN-bus in a vehicle to another system via the internet. The flexibility to serve each potential CAN message is implemented so that the software can be extended with a fully-fledged protocol. There has been no tests to actuate on a vehicle, but should theoretically be doable in the same manner as requesting data except from a mode change.

The protocol for retrieving CAN messages is made in JSON, which means that no programming skills is needed to fill in the protocol. Some skill is needed to make each API call however and to implement the conversion formulas as functions, but considering there is already an API call made (engine RPM), it should not be hard following the already existing structure, view Appendix A.4.1 for an example API call.

The system registers each of the services it defines in another JSON, where the user specifies a URL (among other things) for the service API, so that a consumer can request that specific service. Likewise the user can specify what consumers should be able to access the service, this is also done via a JSON. Next to no programming experience is needed to add services as long as they know the information necessary for a service. Previously this was not implemented (register multiple services in one system) or at least not publicly showcased in any of the documents on the Arrowhead Github.

There are issues that needs to be addressed which will be brought up in section 7.2, but the system works as of the thesis date, as a working example or a demo, but for a full working solution the software needs to be extended.
CAN-bus system for vehicle actuation and data logging with Arrowhead Framework
As previously mentioned in Chapter 5, the field test was an important step in confirming that the provider system worked towards a consumer system.

The implication of the evaluation means that we can confirm that it works to have a system implemented to serve towards automation of vehicles within Arrowhead, or at least collecting real time information from vehicles in this sort of system. Due to the lack of vehicles to try to actuate on, actuation is not tested and needs to be tested at a later stage. Additionally it describes that there is still work to be done for a fully functional solution, the field-test was successful in showing that it works and that it could be a good solution, but it also shows the flaws.

Sometimes displaying the wrong value is not an option in an autonomous solution where everything needs to be correct for the machine to take the necessary precautions for safety measures. Furthermore the field-test was done by only requesting one type of data from the vehicle, which was the engines RPM. A fully implemented solution should be capable to have several different requests for the consumer to consume. The software as of right now is capable of requesting different types of data from the vehicle, but due to the time frame of the thesis, it was never tested.

What it does not tell us and should be further investigated is whether or not it is an optimal solution in a larger scale, which would require an extension of this work to investigate at a later point. Questions that does need to be answered are:

- What if there is a need to request several different values very quickly from the CAN bus, will there be errors or will the software work as intended?
- Will the CAN bus component poll the data from the CAN fast enough to serve the machine learning algorithm and the data logger?
- Does the Provider component with the underlying web-server handle the requests fast enough for a functional system? Does "first come, first serve" work on the web-server or does it need to be asynchronous?

Another implication is also that to some degree a confirmation of the software attributes of adaptation and update as asked for in the tasked defined in chapter 1.4. Being able to make big changes in the CAN bus component during the field test without affecting the Provider part at all. Additionally with documentation throughout the thesis of the inner workings of the system should suffice in giving the end-user confidence enough for adaption, update and reuse overall in the system. However that simply does not confirm that the software is capable of adaptation, update and reuse to a full extent. This will be tried and proved when other developers actually try to extend on the software for either preparation of deployment or reusing parts of the system for another application.
7 | Conclusions and future work

7.1 Conclusion

In conclusion, the thesis asked for a software system that is able to serve CAN-bus data via the internet, as provided by a service for consumers to use.

The final outcome of this thesis is a system that is implemented within Arrowhead Framework that is capable of reading data from a vehicle via a CAN bus and offering data for consumers within Arrowhead Framework as a provided service. Actuation was not tested but should theoretically work, needs to be tested and defined in the software.

The system was built to support adaptation, update and reuse, which is attempted by using low-coupling and high cohesion within the classes of the components together with the provided documentation.

The resulting system is not a complete solution and needs more work before being deployed, what needs to be done can be read in chapter 7.2. However it does work as a proof of concept of this sort of system within Arrowhead which has a purpose to actuate and offer data to consumers for storing information.

7.2 Future work

This section will talk about potential future work that needs to be addressed before considering deployment.

7.2.1 Improvement on network communication.

As the system should support automation software with machine-learning having quick and effective communication is essential. Currently the way the systems communicate via Arrowhead, is via web-servers with HTTP requests. HTTP requests have a big overhead.

A possible solution would be to make a UDP socket-based solution for communication, however the implication on that would be that since Arrowhead is built with web-servers as a communication means and the safety is managed as such, the work could become substantial regarding security management.

7.2.2 Protocol for the earth-moving machine.

As of right now the application is only tested with a car and in these tests only the engines RPM was thoroughly field tested. In the future there has to be a fully
fleshed out protocol that can support requests for many different things involving the earth-moving machine.

A way to implement this protocol is present, but there is still a lot of work to implement the protocol since all the PID need to be stored and all the conversion formulas need to be implemented.

### 7.2.3 Improvement on reading from the CAN bus.

A big part of the implemented system was to read from a CAN. This was field-tested in a car to see whether or not it read the data correctly. It worked, however there needs to be more work done to read correctly. As of right now it reads from the CAN when requested to, but a future approach that might prove better is to have a separate thread that keeps reading and updating the values, so that upon request it’s simply a matter of reading the latest fetched value. This might not be a solution if there’s a huge amount of different requests that needs to be done. However introducing such a system also introduces real-time properties that is essential to keep in mind. Race-conditions is one of these problems that needs to be considered if introducing such an implementation. In time-sensitive matters where you have software to automate a vehicle safety is top priority and it needs to be implemented with care. Additionally a fleshed out error handling protocol for the communication with the CAN bus needs to be implemented.

### 7.2.4 Automated test suite

To be able to deploy the system and know it is working as intended correctly, an automated test suite for the system would be essential. When updating the system it is easy to miss something which could result in a malfunction within the system, to avoid this an automated test suite could spot that fault before it got deployed and save countless man-hours of work in the end. As of right now only manual tests has been done, mostly during field-tests.

### 7.2.5 Security

In this thesis, security was not considered. What needs to be addressed in future development of the system is to make the system run in Secure mode in Arrowhead. The only aspect that touched security during the time of the project, was the Authentication registration that was re-arranged. But the Authenticator within Arrowhead is not enough to use in a real deployment. There is a need for a real token system for the services that Arrowhead has built in, but due to complications of getting Secure mode to run, it was ignored.
Bibliography


Appendixes

A.1 A detailed UML over CanBusApp

Figure A.1: CanBusApp detailed class diagram
A.2 A detailed UML over Can bus provider.

Figure A.2: CanBusProvider detailed class diagram
A.3 A detailed UML over the total system.

Figure A.3: Full system detailed class diagram
A.4 Example code-snippets

A.4.1 An API call

```java
@GET
@Path("enginerpm")
@Produces(MediaType.APPLICATION_JSON)
public Response getRPM(@Context SecurityContext context,
                     @QueryParam("token") String token, @QueryParam("signature") String signature) {
    if (context.isSecure()) {
        RequestVerification.verifyRequester(context, token, signature);
    }
    Message msg = null;
    int data = -1;
    try {
        msg = canBus.getFromCan(pids.getPIDCode("enginerpm"));
    } catch (Exception e) {
        System.out.println("Failure");
    }
    if (msg != null) {
        data = formula.getRpm(msg.data[3], msg.data[4]);
        System.out.println("RPM: " + data);
        return Response.status(200).entity(new IOMessage(data, "rpm", 0,
                                                      System.currentTimeMillis())).build();
    } else {
        return Response.status(200).entity(new IOMessage(data, "rpm", -1,
                                                      System.currentTimeMillis())).build();
    }
}
```

This is one of the API calls that are defined, specifically it is the API call for engine rpm.