Bandwidth reductions gains through Edge Computing in connected cars

Mikael Jarfors | Axel Rosén
mikael.jarfors.590@student.lu.se | axel.rosen.895@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Maria Kihl, Lars Larsson

Examiner: Christian Nyberg

November 9, 2018

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Gathering and analyzing data that is generated in IoT and mobile devices is increasing due to the huge potential value it brings to consumers and car manufacturers. The increase in production of data introduces new problems in terms of bandwidth requirements. Performing computations, filtering, and analyzing data could be introduced to devices to reduce the bandwidth usage, this concept is called Edge computing.

This Masters thesis has tackled the problem of bringing sensor based edge computing to vehicles. The problem motivation raised two questions; How arrival intensity of sensor data affects the system, and what bandwidth reductions gains can be made. Firstly a pipeline was defined, and technologies and frameworks were evaluated to be used in said pipeline in the method chapter.

The PoC was then tested in a real car, in order to prove that it works. It was also used as a baseline for testing the two research questions posed. The PoC was then tested in two rounds, in order to evaluate the different research questions. The First research question was evaluated through having a static system and set of data and varying the sample rate of data to simulate the arrival intensity. The results show that the system had a linear relationship in terms of memory, cpu usage to the arrival rate. For the specific hardware that the system was tested showed that the system was stable up to a sample rate of 10000 Hz.

The main research question was tested with the results from the secondary research question in mind. The sample rate was set to 100Hz and instead, the agent scenarios were varied in order to evaluate what bandwidth reductions gains can be made with three different edge computing levels. The results showed that the bandwidth can be reduced to 0.01% of the original amount when sampling data over 2 hours at 100 Hz. The scenarios had similar CPU usage despite increasing the amount of edge computing done in the agents, which further showed that edge computing is feasible in that car. However, it was also shown that the use case which the agent is based upon dictates what bandwidth reductions gains can be made.

Acknowledgements

First we would like to extend our gratitude to Volvo for enabling us to do our Master thesis at a company that is in the forefront in the field. Further we would like to thank our supervisors Davide Grohmann and Henrik Sölver at Volvo for continuous support. We appreciate the time they took to help us and the guidance they offered whenever we needed it. We are especially grateful for all of the clear and precise design advice.

We would also like to thank all the staff at Lunds University Faculty of Engineering that have been involved in our Master Thesis, especially our supervisors Lars Larsson and Maria Kihl for their endless support, precise feedback and academic knowledge.

Popular Science Summary

The car industry is rapidly evolving and including more technology in its cars each generation. The technology in the cars is also becoming smarter through the use of more sensors such as parking sensors and cameras to aid in driving. In order to be competitive in the market of tomorrow, one of the aspects that can set a car apart from the competition, is a car that becomes smarter with age.

In order to realize this, sensor data in the car needs to be interpreted and dealt with accordingly. With an increasing number of sensors in cars, sending data to the cloud in order to interpret and calculate results from the values is not feasible. There are two main reasons why sending the data is not feasible. The first one being that bandwidth and mobile data from internet providers is expensive, when considering the scale of thousands of cars sending data to the cloud. The second is the nature of network coverage, as cars are normally driven on roads and often between cities. This means that there are locations where network signals are poor or non-existent. The common result of these two issues is that the need for a way to reduce bandwidth usage is high.

Edge computing is the concept of introducing intelligence into the end points of a network instead of having the intelligence in servers. An example of this is in a surveillance camera scenario, where you may have a certain total bandwidth of, for example 10 units. If each connected camera streams costs 1 unit, then you can have 10 cameras in total. However, it may be unnecessary for the cameras to stream constantly. Therefore, if a simple calculation or interpretation is done inside the camera hardware, such as checking if there is movement in the frame, and only then allow the camera to stream.

Applying the concept of edge computing to a car is similar, however there are some fundamental issues that need to be addressed. A car is a unique piece of technology because of its complexity and that it's also responsible for the safety of its passengers. Therefore, cars today have requirements on the response times of certain aspects of a car, such as the time it takes for an airbag to activate, or newer technology such as intelligent braking when the car sensing an obstacle ahead. Both of these scenarios are directly related to the safety of the car's passengers and its environment.

The edge computing system needs

to be robust and isolated in order to be allowed to run in a car, and therefore compromises need to be made. This thesis serves to identify some of the trade-offs and construct an edge computing pipeline with these in mind, and evaluate it.

The evaluation showed that the system can handle realistic sensor collection rates that are viable today, which paved way to evaluate a real scenario. The scenario showed that without edge computing a large amount of packages needed to be sent, and this needed a lot of CPU-power. However without much change to the overall CPU-power need, with edge computing, only 0.01% of the bandwidth was needed to send the processed results instead of the collected data.

Table of Contents

List of Figures

List of Tables

List of acronyms

API - Application Programming Interface
CPU - Central Processing Unit
ECU - Electrical Control Unit
GQM - Goal Question Metric
HTTP - Hypertext Transfer Protocol
IoT - Internet of Things
IPC - Inter-Process Communication
MEC - Mobile Edge Computing
PID - Process Identifier
PoC - Proof of Concept
QoS - Quality of Service
REST - Representational State Transfer
RSS - Resident Set Size
TCP - Transmission Control Protocol
VEC - Vehicular Edge Computing

Chapter 1
Introduction

Gathering, analyzing and keeping track of data has never been more important than it is today. The value that analyzing data brings, has a broad spectrum of interested stakeholders, everybody from sport teams [1] to corporations are avidly looking for new ways to appropriate their data.

Data can be analyzed with different goals in mind. Retail corporations could use it to learn customer behaviors and product corporations could use it for diagnostic purposes. Each goal has its' own challenges and focuses e.g. one goal might benefit from getting the data fast despite losing some data, other goals might benefit more from getting as much data as possible needless of the time aspect.

Cars are getting progressively more intelligent due to an increase in the amount of information a car receives from its environment. Customers in today's car-filled world have increased their expectations on what cars can and should do. Therefore it is important for cars to meet this new elevation of customer standards through technology.

In the automotive industry, data generated from different sensors in the car has increased drastically in recent years. However, there are still challenges in this area that have not been met. These challenges need to be addressed before full utilization of data from sensors in the car can be achieved.

This master thesis will focus on the use of data in a diagnostic process for rapid prototyping similar to what J. Lou et al. [2] investigated. In contrast however, this will be done by utilizing concepts of Edge Computing. The concept of Edge computing relieves the pressure on the network load by e.g. processing, compressing or aggregating data before sending the data to the cloud. When utilizing concepts of edge computing on devices with limited resources, trade-offs such as CPU usage, memory usage and data availability need to be taken into account.

1.1 Problem motivation

Due to the increasing amount of sensors in cars, the amount of data generated by the car is quickly outpacing the bandwidth at which the data can be sent. Therefore the need for a way to handle data is becoming more important. Car manufacturers have traditionally extracted information from cars when the car is at service, through its service providers. However, this leads to two main issues. Firstly a car is serviced only a few times a year which means that the interval at

which the automotive company receives updates are rare and uneven. Secondly the car manufacturers will only receive the data if the service provider has a contract with the manufacturer, which, third-party service providers generally do not. Thus, with the large amounts of benefits of being able to procure the data in real-time and more reliably, through sending the data over the Internet, the need for data collection during services is greatly reduced.

However, this causes a new range of issues, such as the amount of data that is being sent now needs to be sent over cellular connections, which are often, if not always, rate and bandwidth limited. Therefore, intelligent ways to reduce the amount of data sent without removing the essential information that the data would provide are needed. This, for instance can be done through calculations on the data in the car, where the calculated value can be sent instead of the entire data stream or via compression.

Furthermore, the amount of processing power in a car is limited, as the power requirements for the internal components are similar to that of an IoT (Internet of Things) device. The leads to the need for understanding what trade-offs are needed in order to make the necessary bandwidth reductions.

A fair amount of research have been conducted withing this area[3, 4, 5] but few documented implementations on actual devices and discussions regarding trade-offs have been made. It is of interest from the industry that such ideas and concepts can be implemented. This master thesis will explore some areas of Edge Computing and apply it to specific scenarios.

1.2 Aim

The aim of this master thesis was to investigate how, and what, edge computing concepts could be applied to a scenario involving a connected car. Thus reducing the network load from the car to the cloud while still providing the benefits of high data availability. Focus was especially put on developing edge computing concepts that work with a limited amount of resources. A proof of concept (PoC) was designed and developed to prove that the edge computing concepts work in a real scenario. The thesis aims to help developers in the future make decisions about the different techniques discussed in the thesis as well as give an example of how edge computing can be used on architectures similar to the PoC.

1.2.1 Research Questions

The questions that this problem motivation implies and the master thesis researched are:

1. What bandwidth reductions gains can be made by increasing the amount of data processing within the car itself, and what trade-offs in terms of e.g. data availability, processing power requirements, and failure handling does such edge computing processing introduce?

   The data will arrive to the system with a certain intensity, and this will dictate the amount of calculations that is needed, and thus the following research question is implied:

2. How does the arrival intensity of sensor data in a motor vehicle dictate the processing requirements of agents in edge processing/computing?

1.3 Related Works

There has been a substantial amount of research into mobile edge computing, where the focus has generally been on how to move calculations to the edge of networks in cases such as IoT devices or mobile edge computing (MEC) in mobile devices.

One prominent example has been on the increase in the amount of data created at the edges of networks [4]. The problem that the amount of data produced will soon exceed the capacity that is available through mobile networks, and therefore, how does one minimize the power consumption of edge devices and bandwidth usage without compromising quality? Determining how to measuring the quality of the information received was done by S. Wang et al through the loss function in their learning algorithm. Their quality aspect could thus, be measured quantitatively as the loss function in the machine learning algorithm that was being deployed in their edge computing solution. Therefore in order to achieve the desired trade-off between local calculations and global aggregation, the amount of calculations done at the edge was changed in order to achieved the desired loss level.

Concerning vehicular edge computing (VEC), Sun et al. implement learning algorithms that use V2V (Vehicle to Vehicle) and V2I (Vehicle to Infrastructure) and the vehicle as an edge [5]. The discussion is based around task replication and how the communication and calculations can be shared between vehicles. This result concluded that cars are able to reduce the offloading delays using VEC.

1.4 Project Disposition

The master thesis report consists of *Introduction* which introduces the subject the thesis concerns and research questions the thesis is based upon. The next chapter is *Background*, which gives the reader an understanding about the theoretical parts, concepts, models and techniques that the thesis concerns. The report continues with *Method*, a chapter which explains and discusses implementations of the theoretical concepts and models. The next chapter is *Arrival intensity of sensor data*, which concerns the sub research question. In the next chapter *Bandwidth reductions* the main research question is presented with results and a discussion. The report continues with *Discussion*, which is a general discussion regarding all of the results and both research questions. The following chapter is *Conclusions* which is a conclusion of the results and discussion. The last chapter is *Future Work*, which concludes what future works that can be made to improve on the topics that the thesis concerns.

Chapter 2
Background

We have studied the concept of edge computing as it relates to processing data from sensors in motorized vehicles. Since the number of sensors is potentially large, and projected to grow as more intelligent features are demanded by customers, one of the problems that arises concerns the data distribution and handling within the car itself. Processes in the vehicle must be robust, as the computational components may not interfere with vital car components. Furthermore, the hardware in modern cars may have fairly limited performance, and thus, any reasonable solution needs to take energy efficiency and execution speed into account. Finally, for security reasons, programs conducting data processing must be executed in isolation to avoid interfering with each other.

Therefore, the following areas are briefly introduced, Core Concepts, Technologies, Architecture and Profiling methodologies.

2.1 Core Concepts

2.1.1 Robustness

According to J. Armstrong [6], robustness can be identified as six key areas in terms of computer science, isolation, concurrency, fault identification, fault handling, dynamic code upgrade and stable storage.

Isolation is the separation of processes from one another in terms of various resources such as memory, CPU usage, filesystem and other namespaces. In terms of software running in a vehicle, isolation allows for processes to crash without jeopardizing system functionality and uptime. Isolation can be created through different mediums that are available in operating systems such as Control groups (Cgroup) or containers such as Docker.

Concurrency is running several processes simultaneously, such as running processes in different threads. Concurrency allows for threads to crash without compromising on uptime. For example, if a process is run concurrently in two instances, if one of the instances crashes, given that the programming language supports it, the other two will still be able to run. Concurrency is supported natively in many programming languages, and also through separate tools which e.g., runs the entire system in several instances.

Fault detection, is knowing that a fault has occurred, as a system may or may

not be under constant supervision. Fault detection can be done through local logs, or error reporting to a remote location.

Fault identification is knowing where and why an error occurred. This can be done in a number of ways, such as printing stack traces or exit codes as well as the last state of the process.

Dynamic code upgrade, is support for changing the running code without stopping the system. This can be done through starting and stopping programs from a central process manager developed in the system.

Stable Storage, storage that is non-volatile, meaning it survives a crash. This can be done through transactional storage, meaning that the data can not be written as part of a whole to the disk, and instead needs to be written as a logical change or addition.

These six parts are what make a robust system.

2.1.2 Edge computing

With an increasing number of devices gathering data and an increasing need for cloud storage the need for network bandwidth increases. Edge Computing is a paradigm for connected devices, to reduce bandwidth usage on the network, where processing is done in the nodes themselves, instead of in a central server unit. An example of this is in a surveillance camera scenario, where you may have a certain total bandwidth of, for example 10 units. If each connected camera streams costs 1 unit, then you can have 10 cameras in total. However, it may be unnecessary for the cameras to stream constantly. Therefore, if a simple calculation or interpretation is done inside the camera hardware, such as checking if there is movement in the frame, and only then allow the camera to stream. This methodology would allow for more cameras than 10 to be able to use the network, since each camera would not need its full bandwidth in each case, this is called Edge Computing.

Traditionally stream processing systems have been designed for environments supporting clusters of computers. To support edge computing as well as distributed environments, suitable architectural models have emerged. A lot of focus is currently on moving certain stream processing elements closer to where the data is generated, at the edges. Transferring the processed data can be done in different ways, e.g. by batch jobs or streaming. Much of this work is still on a conceptual or architectural level without concrete software solutions. Possible solutions have proposed the use of a simple computer (e.g. Raspberry Pi or similar) where certain data compression and data processing can be handled [7, 8].

2.1.3 Stream Processing

Stream processing is the handling of data when it arrives through callback functionality [9]. Event-based systems is a large use case for stream processing, as the events would trigger callbacks in the stream processor which would in turn handle the event. Sensor data readings can be seen as events and thus stream processing that is useful in a vehicle scenario where data collection is occurring.

Stein et al. introduce how stream processing is used in UI applications where there is a single thread that handles the user input as well as how that thread is

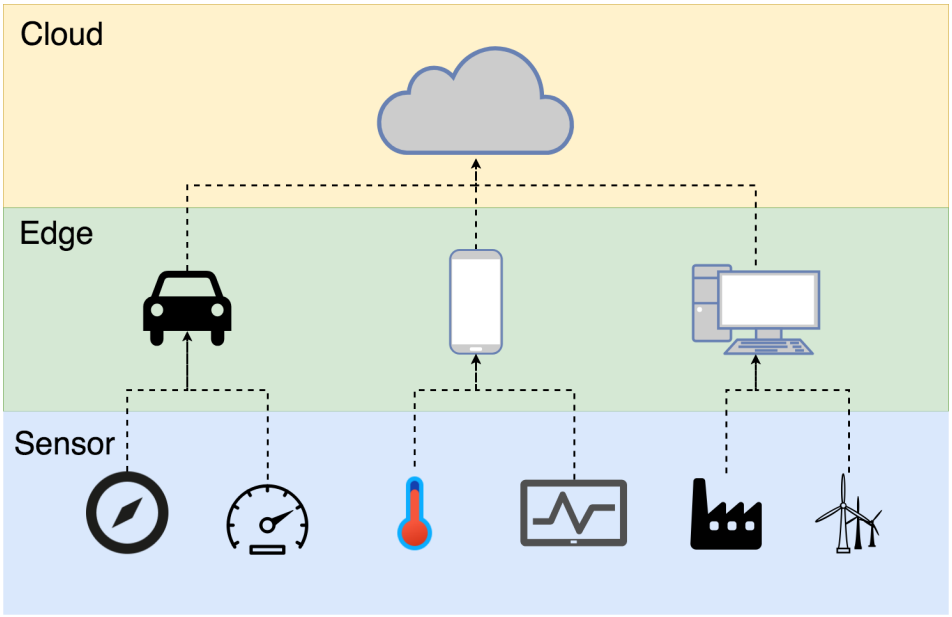


Figure 2.1: Concept of Edge Computing with one cloud, three edges and six sensors

not supposed to interfere with the functionality of the program [9]. This is similar to the commonly used Python compiler CPython which has a *Global Interpreter Lock*, which in practice results in only one thread executing at once since the memory management in CPython is not thread safe [10].

Furthermore, the introduction of event-based programming frameworks such as ReactiveX allows for stream processing in a complete package along with Map Reduce functionality such as filter, map, reduce [11].

2.2 Architecture

Edge computing requires a defined direction of data flow, from the source to the stream processor to the cloud. The publisher-subscriber topology is a quintessential feature of edge computing pipeline design.

2.2.1 Master-Worker Architecture

In computer science the master-worker architecture is a model that can be used for managing tasks and subtasks, where there is one main task, called a master, and several subtasks, called workers. The manager will distribute work to the workers through giving them tasks [9].

Subtasks can be different from each other and work together or totally independently from each other. A scenario could be a worker having a predefined task

that is activated when the worker receives data. Another scenario could be where several workers need access to the same data and return values based on the data.

There are many software architectures that are based on the master-worker model such as the publisher-subscriber. In the publisher-subscriber topology, the publisher is the master, which delegates data to the subscribers. Each of the subscribers, is a worker in the architecture. In edge computing, having a master is important. The master is responsible for being the dealer between the incoming sensor data and the workers.

2.2.2 Agent Model

A common approach to designing an edge computing system is an agent approach, similar to a master-worker architecture. The approach consists of several workers that are referred to as agents, whom receive data from sensors in the manager node. However, in contrast to the master-worker architecture, the agents themselves know their own task and the manager is only responsible for supplying information and not delegation.

An agent is the worker that carries out aggregations, calculations and filtering of the data and chooses what to upload to the server. Agents will receive a stream of data at variable rates, meaning that there will be waiting for data to arrive. Therefore, a solution would be to provide the agents with a event based approach. Therefore, in order to aid the workers' stream of data, an implementation of the MapReduce pattern is implemented for simple servicing of the data.

2.2.3 The MapReduce Pattern

The MapReduce pattern is a combination of two functions, the map and the reduce. According to J. Dean et al. [11] the map, is a user defined function. It takes an input value and produces an intermediate value, which is associated with the same key. The reduce operation is a user defined function, which receives a list of values for a key, normally from a map function, and produces a possibly smaller set of values for said key. In collaboration, these two functions allows the system to reduce the amount of memory needed to handle large amounts of data. Furthermore, the pattern can be used in a stream processing based system, which handles a constantly increasing amount of data. The MapReduce pattern is implemented through the Reactive Extensions (ReactiveX) framework [12].

2.2.4 Publisher - Subscriber Architecture

A publisher-subscriber architecture is a network topology used in a distributed network setting [13]. It works in the manner that there is a central publisher that has direct or indirect local access to the data. Direct access to the data means that the data is being read by the publisher. In contrast, indirect local access is when the publisher is communicated the data, often from several sources. One topic can have several subscribers, such that when a publisher publishes data from a specific topic, all the subscribers for that topic receive updates from the publisher. The topology is based around the concept of multicasting, which takes the assumption that a broadcast message is as expensive as a normal point to point message.

There are different ways to filter the information sent from a publisher. The filtration can be done in the publisher, meaning that the publisher needs to keep track of the subscribers' topics. On the other hand, the publisher can send the data to all subscribers and let each subscriber filter on the desired topic. In this scenario, the publisher does not need to keep track of the subscribers' topics, but each subscriber will need to filter away all the data.

A brief overview of the design is seen in 2.2, where the flow of information is shown from left to right. The publisher-subscriber topology becomes important, when more than one agent is added to the pipeline. This allows for a modular design where a number of agents can be scaled up and down easily. This will allow for the publisher to scale, since adding a subscriber only adds the cost of the subscriber itself.

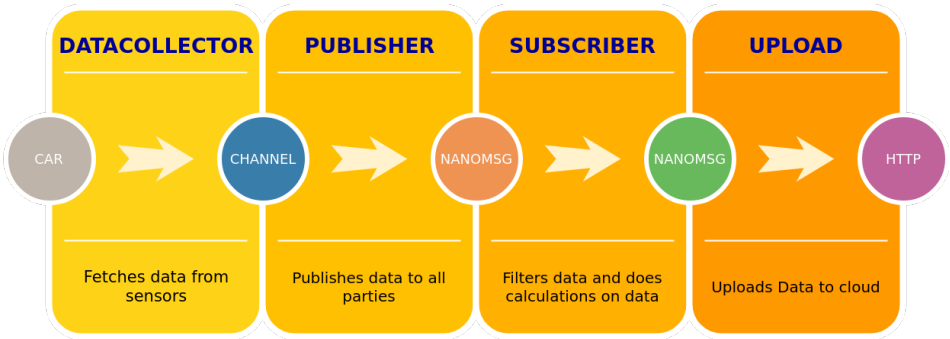


Figure 2.2: Preliminary Design of Agent Pipeline

2.3 Technologies

The defined architecture allows for distributing data through useful and smart topologies, however, the question remains of how to implement the different topologies. Message queues is one answer, as many of the message queue systems have support for a publisher subscriber topology as well as communication between programming languages.

Furthermore, in order to isolate the agents from each other and the rest of the system, agents must be confined to a set amount of resources in terms of the file system, CPU usage, memory usage and network usage. Virtualization and containers are tools that allow for this kind of isolation and will be introduced below.

2.3.1 Frameworks and languages

Golang

Golang is an open source programming language created by Google in 2009. It is a concurrent, garbage collected and fast-compiled language. The idea behind Golang partly originates from the problems that were introduced by multicore processors, networked systems and massive computation clusters, where the problems were

worked around instead of addressed [14]. Similarities between Golang and C can be found, such as that both languages are statically typed, compiled, however, Golang has memory safety, garbage collection and use CSP-styled concurrency. Golang is, in its design, suitable for system software on multicore machines [15].

Reactive Extensions

ReactiveX is a framework that allows for asynchronous event based programming through observables and observers. ReactiveX has bindings for many of the major languages such as Python, Java, and Golang. ReactiveX allows for event based programming through callbacks and lambda functions.

Natively ReactiveX allows for chaining of several functions such as the functions included in the MapReduce pattern such as map which allows for fluent data type conversions and filter which only passes data that satisfies a certain predicate [9]. Reduction mechanics are also available, which allows for merging of data streams. Subsequently, these functions can then create observables which other stream processors can subscribe to, allowing for multiple interpretations of the same data without multiple copies of the data within a single process.

2.3.2 Message Queues

The concept of message queues is central to modern computer science, which facilitates inter-process or inter-thread communication. Whether it is between two processes on the same machine, or two threads in the same process or processes on different machines. Distributed computing is another central concept in computer science, where different nodes or components are distributed within a system, and networked to one another. The concept of message queues is a key element in enabling distributed computing.

Message queues can have several different network topologies such as a pipeline, communication between pairs, and distributing data through publisher subscriber. There are several other topologies, as well as combinations of the previously mentioned topologies. Pipeline means that there is a number of receivers polling for new pushes from several nodes, in a pipeline manner. This topology is an enabler for funneling data through a single point, such as through a network interface to the internet. The Pair topology allows for bi-directional communication between two nodes, which is a commonly useful situation where a pair of nodes are dependent on each other. The publisher-subscriber topology is important when there is a single source of data that needs to be distributed to several interested parties.

This section serves as an introduction into some of the message queues that are readily available. Message queues come in two different architectures, those being with and without brokers. A broker is a middle man in the communication between the server and the client, this introduces additional latency to the communication as a trade-off to introducing intelligence to the network. A broker-less message queue is based around an intelligent endpoint system instead of an intelligent broker.

ZeroMQ

ZeroMQ is a socket based communication technology, implemented in C++ that can be used by most of the modern programming languages through wrappers. It allows for fast and brokerless communication between servers and clients through asynchronous message passing in multiple different network topologies such as in pairs, and in a publisher subscriber scenario [16].

Nanomsg

Nanomsg is also a socket based communication technology that is based upon ZeroMQ whilst being written in C. However Nanomsg has native implementations in Golang, allowing for native communication between C and Golang, through a similar brokerless implementation like ZeroMQ [17].

RabbitMQ

RabbitMQ is a message queue system based on an intelligent broker system where the broker acts as a middle man for all communication between servers and clients in the network. RabbitMQ is developed and written in the Erlang language, and has support for many of the popular programming languages today [18].

Message Queues in a Publisher - Subscriber Architecture

In order to implement these communication channels, a message queue implementation is normally used. Message queues have many implementations that differ in the way that communication is distributed and which distribution patterns are supported. Traditionally, message queues are implemented with a central message broker, and such is the case in RabbitMQ. However, ZeroMQ and Nanomsg are implemented through sockets and thus there is no central broker. The central broker that is available in RabbitMQ allows for complex filtration and intelligence, however it also works as a bottleneck for all communication.

In contrast, the socket based communication of ZeroMQ and Nanomsg allow for inexpensive implementations that work without a dealer process. This allows for smaller implementations and higher throughput at the expense of any intelligence in the broker itself. Thus the intelligence is shifted to the endpoints, the idea of intelligent endpoints is essential for edge computing.

2.3.3 IPC, TCP and INPROC

Communication in the technologies such as ZeroMQ and Nanomsg, can have different transport protocols, such as through the transmission control protocol TCP, intra-process communication (INPROC) and inter-process communication (IPC). TCP is a protocol that allows for communication between the application layer and the internet protocol. TCP is implemented through the use of sockets in both ZeroMQ and Nanomsg. IPC uses named pipes, which are logical files that two or more processes have access to, often with different write and read permissions,

depend on the role in the network topology. INPROC is similar to IPC, however only threads from the same process can have access to the pipe.

2.3.4 Serialization formats

In publisher-subscriber topologies, the information is filtered either on the subscriber side in brokerless message queues, or in the broker. The information is thus bundled as a topic and data tuple, such that an application can filter on a specific topic. This structure is often referred to metaphorically as an envelope. Technologies such as JSON and Protobuf allow for easy envelope support through fields in respective protocol.

JSON

Javascript object notation (JSON) a medium of encoding and serialization of data, in order to structure data in object like fashion. JSON is a string based format, that is readable in its base form [19].

Protobuf

Protocol Buffers (Protobuf) is a also a medium for encoding and serialization of data, which uses interface description to serialize data through a user created schema. Protobuf is not human readable in its base form but can be decoded to become human readable [20].

2.3.5 Virtualization and Containers

Virtualization in the context of computer science is the idea of being able to create virtual instances of a physical computer/machine in software. The technique enables the ability to install different OS on different Virtual Machines(VMs) on the same physical machine. Classic VMs exist as an abstraction layer between the application in the virtual environment and the hardware on the machine that is running the VM [21]. Virtualization allows for isolating a process and its dependencies from the host operating system, which is wanted behavior in a system where processes should not interfere with each other.

Containers, also called Linux Containers, are used for execution of applications in a software component. Through isolation of system resources every container obtains its own process ids, file system namespace and identifiers. Containers provide performance that is close to the performance of the native environment despite being isolated from the rest of it [22].

Docker

Docker is a container management technology that was released 2013. It is a software suite that enables management of light weight containers, through the Docker service. Docker enables the possibility of many different docker containers to be included inside one Docker instance through technologies such as swarming. A Docker container can be tailored to individual needs through specifying operating

systems and dependencies. The dependencies needed for a particular program can be bundled within the docker container. This allows a separate user to simply download the system and run it without having to install any dependencies or have the same operating system. Therefore, it is suitable for application where different parts of the application is dependent on different types of OS or versions of programming languages [23]. Docker enables isolation of the containers from the rest of the system by restrictions on namespaces, CPU usage, network namespaces and memory usage. In addition it also has restrictions for the number of PIDs (Process ID) and IO usage.

Control groups

Control groups, also called Cgroups, is a Linux kernel feature that enables allocation of resources such as CPU, system memory and network bandwidth. Cgroups are managed inside a logical filesystem, and maintained through folder structures. Cgroups can be used by management tools to configure, monitor, and prioritize system resources by dividing the hardware resources appropriately between tasks and programs [24]. Each folder inside the Cgroup hierarchy represents one group, that can have a set of restrictions placed upon that group. Subfolders, are given the same restrictions as the parent folder, and are referred to as subsystems. When running a docker container, one has the option to include flags which correspond to limiting the container in terms of cpu shares and memory allocations. These options are the values that docker writes to the different Cgroup parameters. Therefore, solely using a Cgroup, one can achieve some of the isolation levels that the docker container system uses. However, a docker container is also run as a separate user, which isolates the filesystem to some extent, which is not possible using only Cgroups.

2.4 Profiling methodologies

According to Patel et al. profiling is a technique or methodology used to acquire performance data from the program, whether it be memory, cpu usage or any other useful statistic [25]. They also describe the various methods of performing such measurements, such as inserting code which allows for measuring specific parts of the code. However this method introduces errors in the measurements, because the inserted code introduces an unwanted overhead.

Similarly, one could sample certain aspects of the code at set intervals through either an interrupt, or through a separate information source such as the proc file system in Linux. This method potentially avoids the unwanted overhead, but may introduce unwanted behavior due to the interrupt [25].

Pprof

Pprof is a method of profiling code in Golang [26], using insertion of a Pprof server which can sample the amount of memory in use, and record a call trace in order to evaluate CPU usage. Pprof allows for easy profiling and measurement of the code without having to alter the code that is being profiled. Therefore,

the overhead created by the Pprof server should impact the system minimally, as it is a combination of insertion of code and sampling of code. However, Pprof is a server running as a separate process, and may still have indirect impact on different parts of the system.

Pprof has several measuring capabilities such as measuring the amount of memory that is in use at any given time. This is important due to the fact that the Golang memory allocation model works with a garbage collector. The garbage collector will collect unused memory at regular intervals, and keep it as a form of cache for the next allocation. Therefore, from a sampling source such as htop, the amount of memory that a Golang program uses, will differ significantly from what is currently allocated.

Moreover, the Pprof tool has the ability to see and visualize the call trace of function calls as seen in Figure 2.4 and 2.3, through either the trace or the goroutine analyzer. This can accurately measure how much cpu usage each function has during the last 30 seconds.

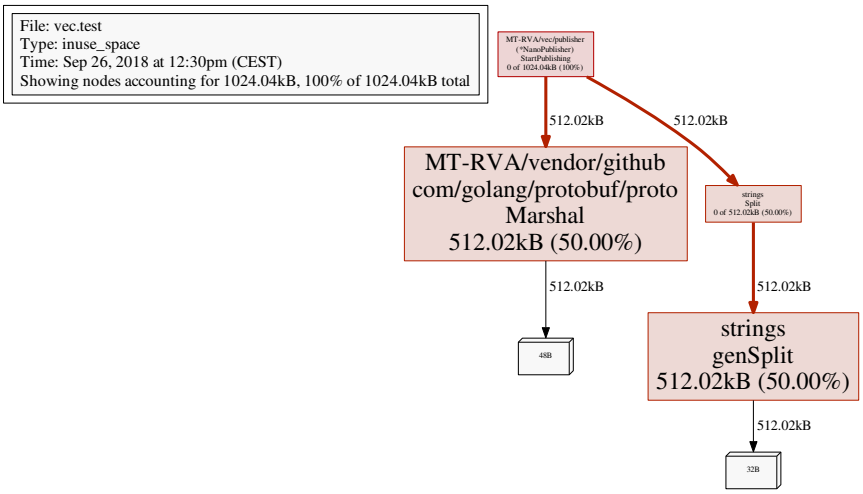


Figure 2.3: Visualizing the memory in use with Pprof. Showing the functions using the most memory.

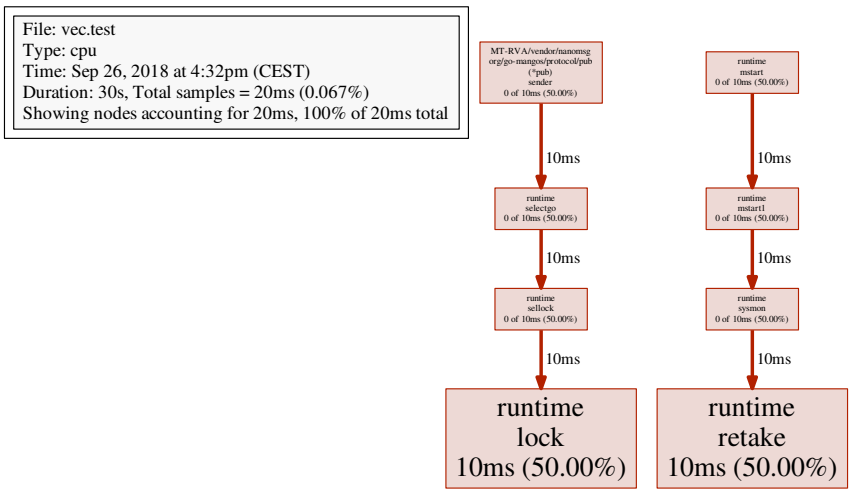


Figure 2.4: Visualizing the CPU usage from call trace with Pprof. Showing the functions with most CPU usage.

Chapter 3
Method

The section will introduce a design, and how to evaluate it from a practical standpoint, based upon the original research question, *What bandwidth reductions gains can be made by increasing the amount of data processing within the car itself, and what trade-offs in terms of e.g. data availability, processing power requirements, and failure handling does such edge computing processing introduce?*.

In order to evaluate the benefits of edge computing in the vehicles, one must account for the unique requirements that are placed on a vehicular system, which create a set of restriction and limitations. The use-case for edge computing in a car is incredibly varied, and thus a system was modelled with this approach. Therefore, the need for a system that supported a single distributor of data with a variable amount of subscribers was necessary. Thus the first part was to identify which technologies support this use case and evaluate which was most suitable.

In addition to the modular nature of the system, some essential functionality was needed in order to dictate how the agents execute. Agent scripts are of low priority in comparison with other systems of a car, such as the systems that are responsible for core functions of the car. Therefore, the agents need to be isolated from the rest of the processes, as well as being robust.

Isolation was the main aim of executional environment, as the agents need to kept from affecting the rest of the system.

These system requirements need to be combined into a proof of concept where benchmarks can be run on different scenarios to establish what trade-offs could be made in order to reduce the amount of bandwidth needed to purvey the same information.

3.1 Assumptions and restrictions

1. The agents need to be modular and support many different use cases.
2. The system needs to be able to handle adding edge computing components dynamically.
3. The agents are restricted to Python development and the rest of the system is restricted to Golang
4. The system is restricted to be able to be compiled to binaries to run on x86 and ARM. This limits the use of technologies in the sense that they need to

be compatible with ARM and x86.

5. The system needs to be scalable in the sense that it needs to scale well with an increasing number of agents.

6. Package sizes are assumed to be be smaller than 1MB, in the range of around a couple of tens of bytes.

7. The system will run in an isolated module, where there is no chance to manually change or restart the system after it is deployed.

8. The system needs to be robust according to [6].

9. The system should be able to handle 10 agents with frequencies at 100 Hz, which is roughly equal to one agent with an arrival intensity of 1000 Hz.

10. The communication with the cloud needs to be compatible with REST API, thus HTTP will be used.

3.2 Choice of design

The high level design proposed in figure 2.2 was used in developing a design that could be used as a template for the implementation. The design 3.1 is a pipeline for information flow, with an intermediary step being the agents. The agents will act as the stream processor in the pipeline, and will only react to relevant data. Moreover the agents are Python scripts that process the data in some way in order to either draw a conclusion from the data or to reduce the amount of data uploaded. The agents need to support the MapReduce pattern in order to reduce the amount of data being uploaded in several ways. The data reduction can either be done through a filtration on values above a certain threshold, a reduce operation such as averaging value, or noticing a pattern of subsequent value. All of these three examples are ways that can significantly reduce the amount of data uploaded if compared to streaming the raw data to the cloud and letting a server process the data.

The designed pipeline is comprised of three main parts: the collection of data, the processing of data and the sending of data to the cloud. The data collector reads data from sensors in the car or other interfaces such as simulators and passes this data to the publisher. In the next step the publisher sends all the data to all subscribers (agents) which then in turn filters out data depending on the topic subscription. The agent will then send the result of its computations to the uploader which will then post the result to the cloud.

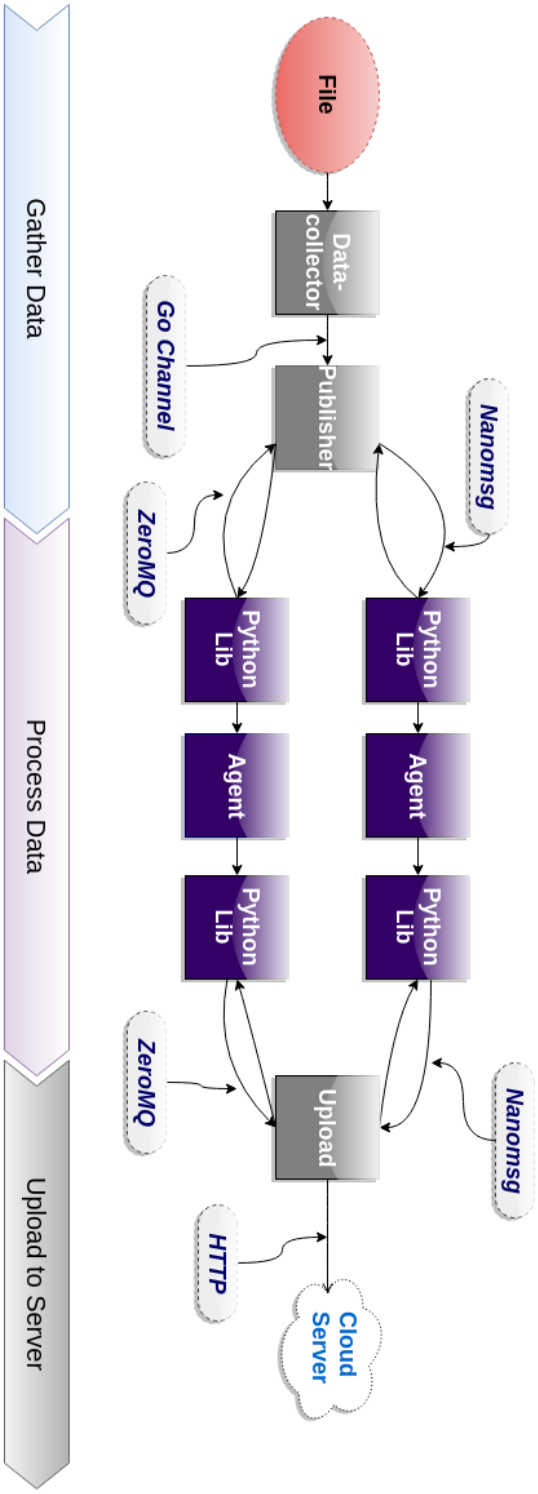


Figure 3.1: Pipeline used for evaluating message queues

3.2.1 Choice of network distribution

Different network distribution techniques can be utilized to achieve similar results but all introduce different trade-offs. The choice of the publisher-subscriber pattern was made because of multiple different reasons. A main reason that publisher-subscriber is a natural fit for this problem is the concept of loose coupling. Loose coupling separates the dependency of components, in this case the publisher is isolated from the subscribers. Isolating components contributes to the overall goal of high robustness in the system. Furthermore, the ease of scalability in publisher-subscriber is needed [13]. The number of subscribers is directly dependent on what data that agents are subscribing to, this makes scalability a key factor.

The decision was made to not have a central broker between the publisher and subscribers. The main reason for this is to reduce the time of the sent data between the publisher and subscribers. This is a trade-off between having an "intelligent" component that could e.g. enable filtering of messages or having a faster subscription service. In this case it brings more value to have a faster service.

3.2.2 Choice of technologies for baseline

The baseline is described as a system without any Edge computing with the purpose of gathering data and sending it to a server, without any package loss and in reasonable time.

The baseline following the design was implemented in Golang. The choice was made to let the datacollector send the data through a Go channel. Go channels were chosen because it is the standard way to send data in Golang, it is essentially a pipe that connects concurrent Go routines. The datacollector and the publisher has each one Go routine.

The research regarding performance of the message queues did not differ greatly [27], which led to the decision to implement two different message queues, ZeroMQ and Nanomsg. These two message queues were implemented between the publisher and subscribers as well as between the agents and the uploader. To handle porting between Golang used in the publisher and Python used in the agents a Python library was implemented as receiver to the publisher. The main part that is taken care of by this component is the conversion of the ZeroMQ and Nanomsg messages from Golang to Python, which is needed because the byte encoding in Golang and Python differs.

The agents were implemented as a pass through medium, meaning that they did nothing other than pass on the messages received one by one. The Python libraries located after the agent sends the messages with ZeroMQ and Nanomsg to the uploader. With the use of HTTP the uploader transfers the messages to a server.

The baseline design can be seen in figure 3.1.

3.2.3 Initial benchmarks

The initial implementation was used to perform benchmarks regarding the message queues, ZeroMQ and Nanomsg, and the communication mediums, Protobuf and

JSON. The goal with these benchmarks was to make a decision of which of these technologies to use.

The results from the benchmark shows a slight advantage for Nanomsg, but not enough to base the decision solely on that. Other factors have to be taken into account, such as implementation difficulty and scalability. Both ZeroMQ and Nanomsg scale well and only differs when the number of requests are large. Since the scale of the numbers of requests in this thesis will be well below that, the scalability can be seen as equally good. Furthermore, the fact that ZeroMQ is written in C++ and Nanomsg is written in Golang. To use ZeroMQ a wrapper had to be used, which poses some potential problems such as a dependency to C++, versioning and updating. The fact that Nanomsg is written in Golang gives it a natural upper hand over ZeroMQ if the only language used is Golang. In this thesis Golang and Python is used and therefore the choice was made to use Nanomsg.

When benchmarking the communication mediums the focus was speed, nanoseconds per operation. The result in Table 3.1 was obtained by running tests with two different operations, parsing and creation, on different number of cores. The results are summarized as a graph in Figure 3.2

Table 3.1: Protobuf and JSON benchmarks with varying number of cores

Test	JSON	Protobuf	Percent Difference	Percent Faster
Parse 1 core	2841 ns/op	899 ns/op	316	Proto - 216
Parse 2 cores	2807 ns/op	891 ns/op	315	Proto - 215
Parse 3 cores	2935 ns/op	930 ns/op	315	Proto - 215
Parse 4 cores	2855 ns/op	924 ns/op	308	Proto - 208
Create 1 core	8047 ns/op	532 ns/op	1512	Proto - 1412
Create 2 cores	8312 ns/op	537 ns/op	1547	Proto - 1447
Create 3 cores	8194 ns/op	560 ns/op	1463	Proto - 1363
Create 4 cores	8235 ns/op	552 ns/op	1491	Proto - 1391

3.2.4 Choice of message queue

To better answer the question of which message queue was most suitable, the method goal question metrics (GQM) were used. Explanations of the scores can be found in the Table A.1 and A.2 in the appendix. Only Nanomsg supports the architectures natively, both ZeroMQ and Rabbitmq need to be cross-compiled from C++ respectively Erlang.

The package size that can be sent differs between ZeroMQ and Nanomsg. ZeroMQ does not have a limit that lies within our scope, package sizes of 100 MB works. Nanomsg has a tipping point of around 1 MB, packages bigger than that might not be delivered properly [27]. Based on the assumption in Section 3.1 item number 6, this is not an issue.

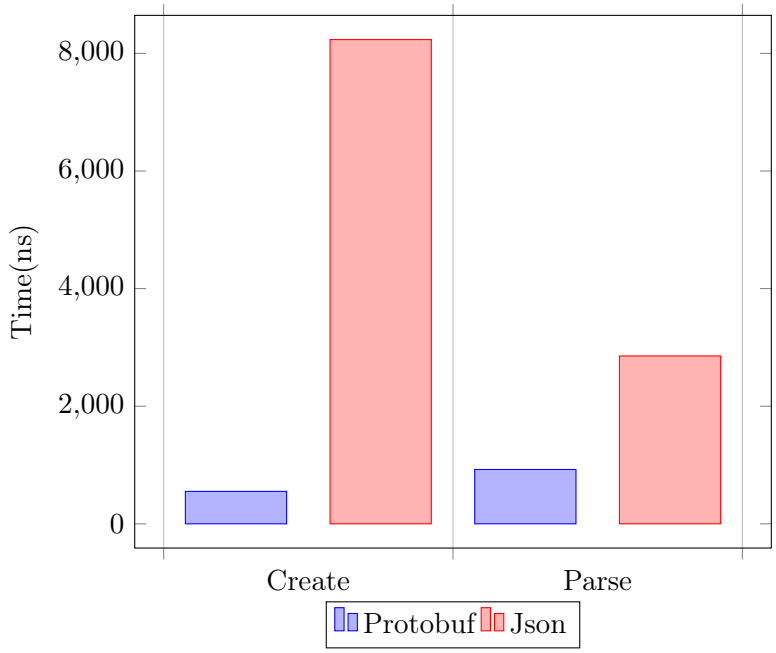


Figure 3.2: Comparison of Create and Parse operations between Protobuf and JSON

Rabbitmq were disregarded due to the fact that it does not compile to ARM7 which is a future requirement.

The results of the GQM is shown in Table 3.2. The combined scores from the GQM show that Nanomsg is the best choice with a score of 8.

3.2.5 Choice of serialization format

The data that is collected and sent, from the datacollector to the cloud, can have multiple use cases. These use cases may require different data which the subscribers has to keep track of in an easy way. Because of this reason the data needs to have a defined structure. A definition for this structure was developed. To send this data over socket based communication the data needs to be serialized. Serialization of data can be done to many different formats which have different trade-offs.

A first selection of different serialization formats was made to narrow down the options. The choice was narrowed down by the language support for Golang and Python. The result of this was two formats, Json and Protobuf. Mainly two factors were taken into account when choosing format, namely memory consumption and processing speed per message. Memory consumption is an important metric in this case when working on devices with limited resources. The memory consumption is lower in Protobuf than in Json, by a factor of up to 4 [28]. Secondly, the speed of the two serialization formats differ. Protobuf could be as much as 5 times

Table 3.2: Goal Question metric evaluation of message queues

Questions	Metrics	ZeroMQ Score	Nanomsg Score	Rabbitmq Score
Does the message queue satisfy the requirements?	Supported by the architectures (x86 and ARM7)?	2	3	1
	Supported by Golang	2	3	2
	Supported by Python	2	2	2

faster than Json at processing time per message for serialization, deserialization, compression and decompression[28]. The performance of serialization formats can differ depending on the specific implementation of that serialization, therefore a benchmark in our specific environment was performed as explained above under "Initial benchmarks". The benchmark 3.2 further supports the benchmarks done by Petersen et al. [28], claiming that Protobuf is faster than Json. Based on these two factors the choice was made to select Protobuf as the serialization format.

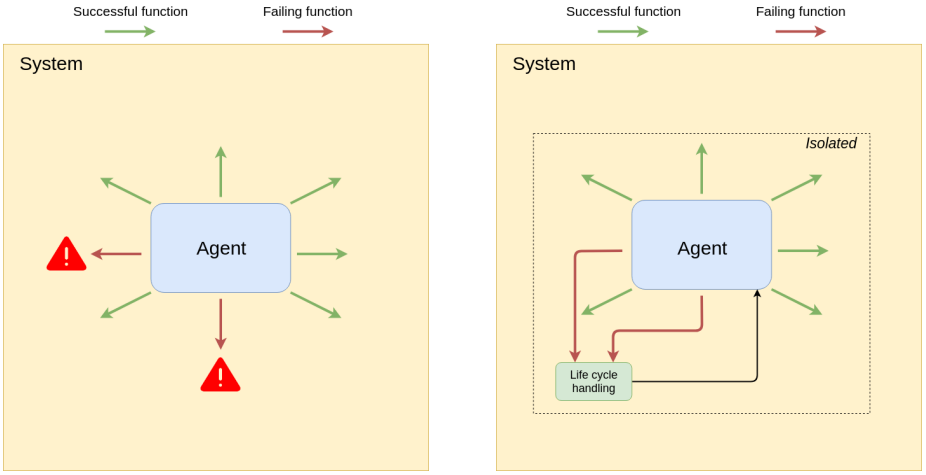
3.2.6 Choice of isolation method

A robust system in the sense that malfunctioning or failing modules should not affect, stop or slow down the rest of the system is essential for this system to work. A main reason for this is because of assumption 3.1 number 8. Furthermore, modules need to be isolated in order to keep them from affecting the rest of the car. Malfunctioning modules must be dealt with individually, therefore a signal handler is introduced in order for a safe shutdown procedure. The life cycles of modules should not affect each other in the meaning that errors of one module should not create errors in another module. To achieve this goal, isolation of specific modules of the system should be used. Isolation could be achieved through different techniques of either virtualization and/or containers.

To perform this isolation, Docker or control groups should be used. Control groups are a versatile isolation method and is compatible with Linux kernels above Linux 3.14. The Docker runtime is required when running a Docker container, and the Docker runtime is not supported by all linux distributions. In order to use Docker as the virtualization method, one would need to be able to spawn Docker containers dynamically during runtime. This may need to be done through a Docker container, which requires the privileged flag to be set. This may be a security concern as the idea is to limit not elevate the permission level of the system.

Control groups are found as an alternative to Docker, as Cgroups provide the isolation in terms of CPU, memory, and network limiting that is needed. Therefore,

as a result, the choice was made to perform the isolation with Control groups.



(a) Unisolated agent **(b)** Isolated agent with life cycle handling

Figure 3.3: Isolation of agent. (a) An unisolated agent with failing functions. (b) An isolated agent with failing functions and life cycle handling.

3.2.7 Trade-offs between requirements and system load

Different requirements that improve the systems' ability to handle more complex edge computing scenarios all come with trade-offs in regard to power consumption and processing power. Based on assumption number 8 in 3.1, the system needs to be robust, meaning that it needed to run concurrently, in isolation as previously mentioned. In addition, the system needs to identify faults and be fault tolerant. The last robustness concept is stable storage, which in the case of agents can be interpreted as the Quality of Service (QoS). The QoS in this case is how well the system performs in regard to how many packages that are dropped over some time period. The QoS can be implemented on the upload side.

Gathered data could have different constraints on time sensitivity and importance relating to the use cases. Some data could be wanted fast and as soon at it is generated, other data could be needed continuously but with less importance and some data could be wanted, but is of little importance when and if it is received. There also needs to be support of how the upload works when there are problems with the network. To fulfil these requirements, three scenarios were identified:

1. Real-time streaming: Stream the data in a way that it reaches the cloud within a given relatively small time frame, without losing any packages.
2. Best effort streaming: Stream the data as soon as it is possible.
3. Batch post: Collect the data during a time frame and send it to the cloud at a specified interval.

These scenarios could be further explained as that data have different priorities and should be uploaded accordingly, and thus a quality of service have been introduced to the system.

A high robustness level is a requirement in edge computing in a connected car. Scenarios could arise where internal systems in a vehicle are run without supervision for months at a time.

Concurrently running parts of the system is a key part of increasing the robustness in the system which easily solvable in Golang through goroutines. Further trade-offs was made to increase the robustness by increasing the number of handlers that deliver data, so that if one would misbehave there is always a handler that could perform the work. This is implemented through the recovery mechanic in Golang, which functions similarly to catching exceptions. However, instead of exceptions, the recover method can catch segmentation faults and restart the worker thread. By having three such worker threads, the system can have three erroneous messages send to the upload module simultaneously before a temporary outage occurs. Important processes in the system that could misbehave, having life cycle handling to enable these processes to start, stop or restart according to the wanted behavior.

To further increase the robustness fault detection and fault identification are needed. Through a supervisor routine which detects agents crashes and sends the exit code, stack trace to the server, one can both detect faults and identify the problem through the stack trace.

3.2.8 Proof of concept

The choices described above boiled down to an implementation of a proof of concept. The proof of concept is comprised of three main parts: the collection of data, the processing of data and the sending of data to the cloud. The datacollector collects data from one of its interfaces that are sensors connected to Electrical Control Units (ECUs), that pass this data to a publishing socket based on the Nanomsg standard. The publisher then sends this data to the interested parties through the socket, which the Python libraries are listening to. The filtration is done and the agents receives the data that they are interested in, and can begin processing the data. After the processing is done the agent can upload the data through a separate socket to the upload service which then sends the data to the cloud. The design of the proof of concept can be seen in figure 3.4

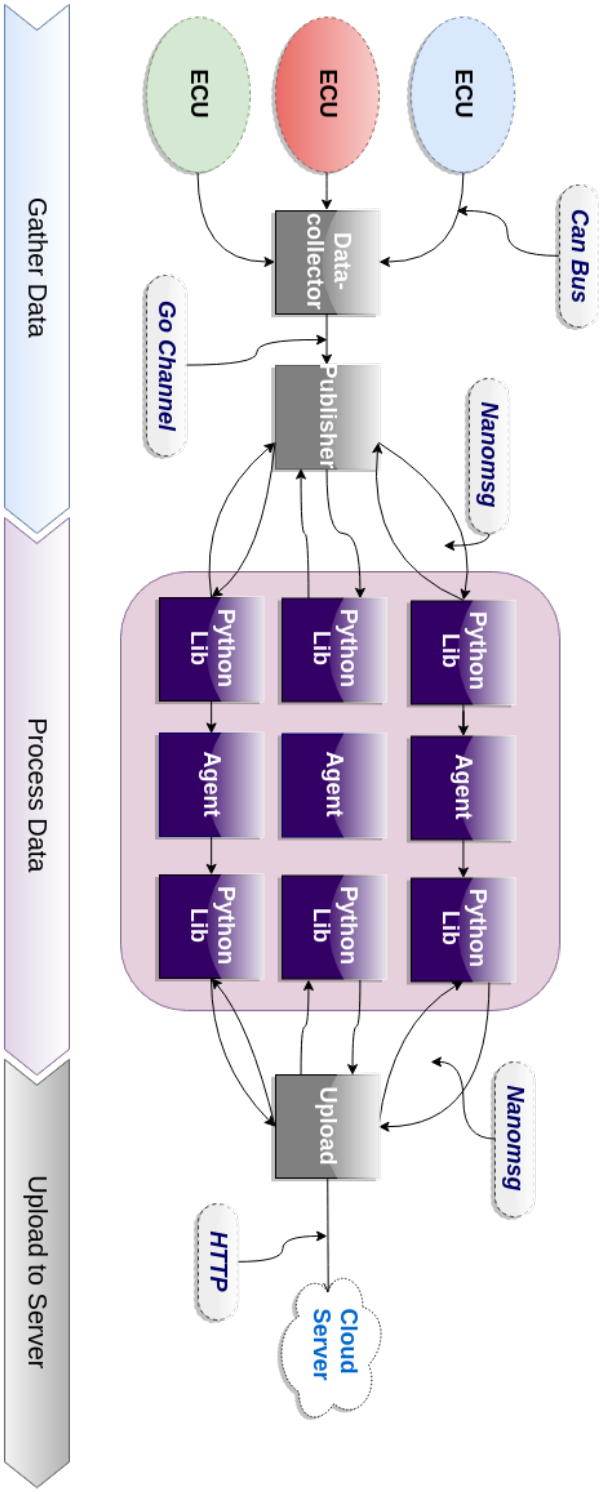


Figure 3.4: Pipeline describing the final design of the system with Nanomsg, publisher-subscriber and isolation of agents.

Chapter 4

Arrival intensity of sensor data

This section outlines the testing methodology for how the arrival intensity affects the system. The tests were based on the proof of concept, and highlights the first research question. The tests were performed to identify how the arrival intensity stresses the system in terms of memory usage, cpu usage as well as throughput. The tests were done through the use of the proof of concept outlined in the method chapter.

4.1 Test methodology

The goal with the test was to identify limits of the system in terms of arrival intensity. Therefore, the test was based on varying the period between data arrival, in a static system.

The system in question ran with the same agent, which sent every message that it received to the cloud server. The message sent will be represented by the topic it represents and the value converted to a float. The pipeline in which the system ran on will be kept the same with one publisher socket running over TCP and one IPC socket responsible for uploading the data.

The cloud server which the data was uploaded to ran on a separate computer. The motivation behind this was to both simulate an accurate representation of how the system will function in the real world and to ensure isolation between the server and the system. The server ran on an isolated network in order to minimize packet loss due to saturation of the network.

In order to simulate a larger data set the data being published was represented as a static CSV file. The static CSV-file was used instead of sampling data from the car to provide a static input data set to minimize its effect. The data-set is previously sampled data from a car. This will thus represent data sizes which fair as an accurate real world representation of the car.

The sample arrival periods was between 1 microsecond and 1 second, and increased with a magnitude of 10 for each test. This gave us a large range data of which to evaluate the system upon, and represent the upper scale of what arrival intensities can be expected in the real world.

In order to get accurate measurements on the memory and cpu usage of the agent control, sampling was done through Pprof. The system was ran with the Pprof tool in server mode, which allowed it to be queried in order to obtain accu-

rate representations of memory usage per process and cpu usage in the currently executing stack. The Pprof values read was the total memory in use and the buffer Goroutine memory usage at the 5 minute mark, as well as the cpu usage the last 30 seconds of the test. This was accompanied by readings from the *htop* program which is a graphical user interface for interpreting the Linux proc file system. The resident set size was read from the *htop* program.

The test itself ran over 5 minutes to represent load across the system, and these individual tests ran 10 times each. The test measured the values of Resident size, Total memory usage, Buffer size usage, CPU usage and Throughput of messages. The tests ran on a system with a Intel Core i5-4250U running at 1.3GHz and 8GB of RAM.

Resident size - is the size of the actual physical memory, i.e. how much memory that is held in RAM.

Total memory - usage is the total memory allocated, measured by the profiling tool Pprof, as seen in Figure 2.3.

Buffer usage - is the memory usage of the publisher routine which is responsible for handling the messages that are are queued for the agents.

CPU usage - is the average CPU usage of the last 30 seconds during the test. The CPU usage is measured by the profiling tool Pprof, as seen in Figure 2.4.

Throughput - is the percentage of messages that are received by the server compared with the theoretical maximum value of received messages.

The choice was made to include these measurements because they all contribute to better understand what and how trade-offs could be made. Resident size, total memory usage, buffer size usage and CPU usage were chosen as they are key factors when it comes to minimizing the impact of edge computing in a connected car scenario. Throughput was chosen as a measurement as it could help draw conclusions of what trade-offs that could be and as it gives an indication of how the system behaves at different arrival times.

The steps for the test were the following:

1. Set up the static parts of the test with the proof of concept
2. Vary the arrival intensity between 10^6, 10^5, 10^4, 10^3, 10^2, 10^1, 10^0 samples per second.
    (a) Perform a benchmark measuring Resident Set Size
    (b) Perform a benchmark measuring total memory usage
    (c) Perform a benchmark measuring buffer memory usage
    (d) Perform a benchmark measuring CPU usage
    (e) Perform a benchmark measuring throughput

4.2 Results

The result Table shows a collection of the measurements over the intensity 1 Hz to 1.000.000 Hz. The graphs below will be visualized on graphs which are semi-log. Semi-log representation was chosen in order to show the exponential relationship at high frequencies.

Table 4.1: Benchmark results for arrival intensities, measuring Resident set size, Total memory in use, Buffer memory usage, CPU usage and Throughput as a percentage of the theoretical maximum

Frequency[Hz]	**RSS[MB]**	**Total[MB]**	**Buffer[MB]**	**CPU[%]**	**Throughput[%]**
1	10.18	0.9832	0.5933	0.055 50	95.00
10	16.07	1.302	0.5120	0.8383	94.50
100	55.30	17.63	4.250	7.530	95.75
1000	389.5	161.1	36.83	43.93	85.25
10 000	1273	515.0	113.6	97.57	17.50
100 000	1610	715.2	168	109.9	1.700
1 000 000	1625	744.7	166.3	114.3	0.1600

The Resident set size can be seen increasing at a linear rate from 1 Hz to 10000 Hz. In this range the Resident set size is 10.185, 16.073, 55.297, 389.50 and 1273.2 MB. The gradient decreases exponentially between 10000 Hz and 1000000 Hz with the Resident set sizes of 1273.2, 1609.8 and 1625.1 MB as seen in the graph in Figure 4.1.

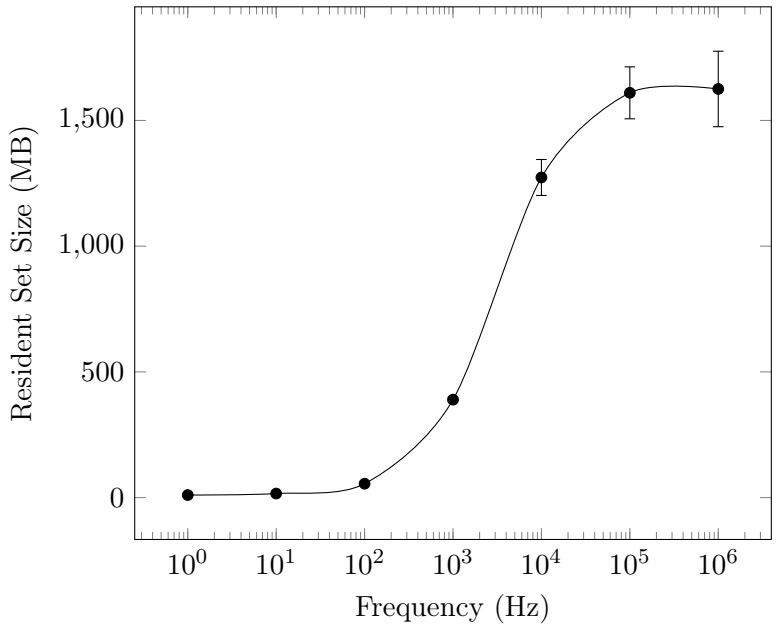


Figure 4.1: The resident set size over different arrival intensities

The Total memory usage can be seen increasing at a linear rate from 1 Hz to 10000 Hz. In this range the Total memory is 0.98317, 1.3020, 17.633, 161.05 and 515.03 MB. The gradient decreases exponentially between 10000 Hz and 1000000 Hz with the Total sizes of 515.03, 715.22 and 744.69 MB as seen in the graph in Figure 4.2.

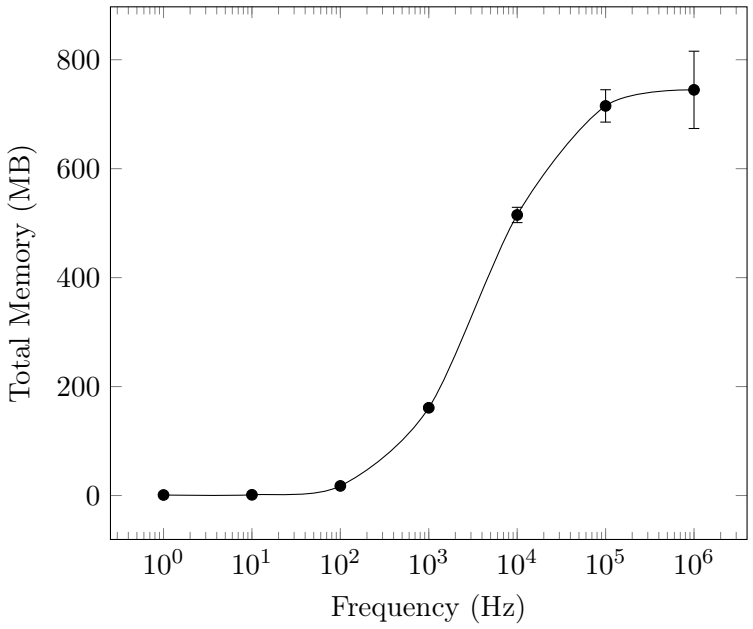


Figure 4.2: The total memory usage over different arrival intensities

The Buffer size can be seen increasing at a linear rate from 1 Hz to 10000 Hz. In this range the Total size is 0.59333, 0.51200, 4.2500, 36.833 and 113.58 MB. The gradient decreases exponentially between 10000 Hz and 1000000 Hz with the buffer sizes of 113.58, 168.00 and 166.29 MB as seen in the graph in Figure 4.3.

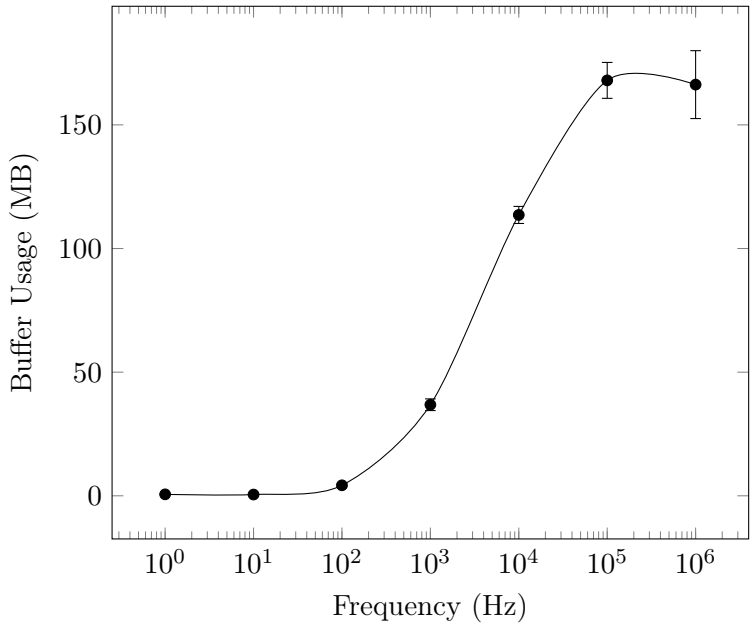


Figure 4.3: The total buffer usage over different arrival intensities

The CPU usage can be seen increasing at a linear rate from 1 Hz to 10000 Hz. In this range the CPU usage is 0.055500, 0.83833, 7.5300, 43.932 and 97.572 %. The gradient decreases exponentially between 10000 Hz and 1000000 Hz with the CPU usages of 97.572, 109.91 and 114.34 % as seen in the graph in Figure 4.4.

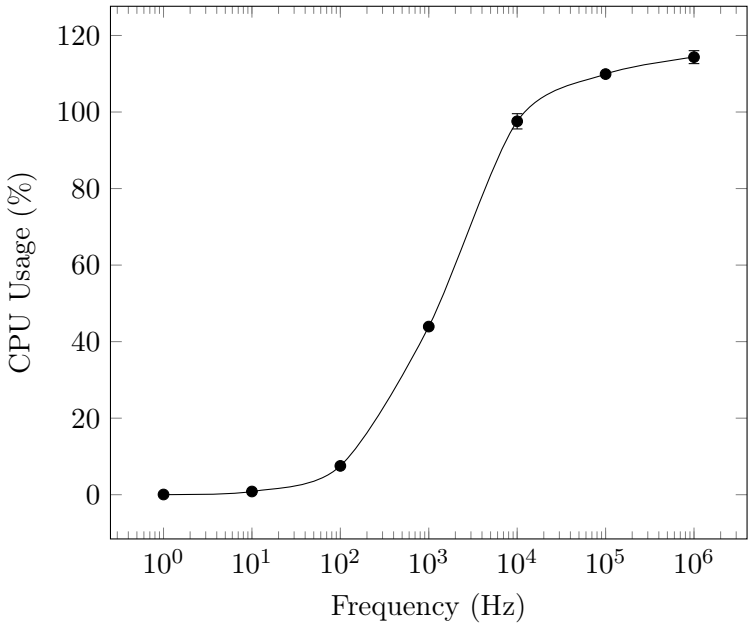


Figure 4.4: The CPU usage over different arrival intensities

The Throughput is within 6% from being at optimal throughput from 1 Hz to 100 Hz. The throughput is lowered down to 85.25 % on 1000 Hz. There is a drop between 1000 Hz and 10000 Hz from 85.25 to 17.50 %. The throughput decreases exponentially towards 0 between 10000 Hz and 1000000 Hz with the throughput of 17.50, 1.700 and 0.1600 % as seen in the graph in Figure 4.5.

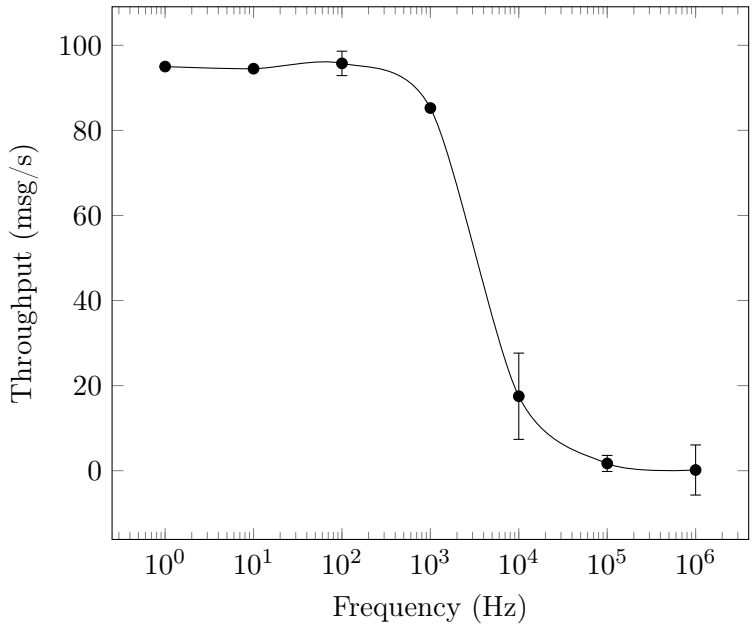


Figure 4.5: The throughput compared to the theoretical maximum of throughput over different arrival intensities

4.3 Discussion

From the results a strong correlation can be found between how the different metrics change depending on the frequency. The changes in the graphs can be seen to appear around 10000 Hz, where the graph tends to change from a linear curve to an exponential. The reason for this is likely due to the arrival intensity of the data exceeding the amount of data that can be handled in the system. When the buffer is full it will block any new data, and at that point the system begins to lose messages. At that point the system could be regarded as an unstable queue where the arrival intensity is greater than the time in the system. A recommendation is therefore to keep the arrival intensity at below 10000 Hz.

The Resident set size is how much memory a process uses as seen from the operating system. Therefore, as seen in the Golang specification, Golang uses a garbage collecting system with a cache. Therefore the allocated memory can be significantly different from the memory observed in htop. The Resident set size increases linearly up 10000 Hz which is expected since the hardware is able to

provide the system with the amount of Random Access Memory (RAM) needed.

The biggest contributor to the Total memory size is the buffer size, which stands for between 60% and 22% in the frequency range of 1 and 10000 Hz. This conclusion is made by looking at the visual representation of Pprof as seen in Figure 4.6. The total memory in use is a more accurate representation of how much memory is currently allocated in functions and goroutines than htop. However, the unused memory, that is stored as a cache will not be returned to the system. Thus this measurement in conjunction with the RSS will give an accurate representation of the effect on other processes in the system.

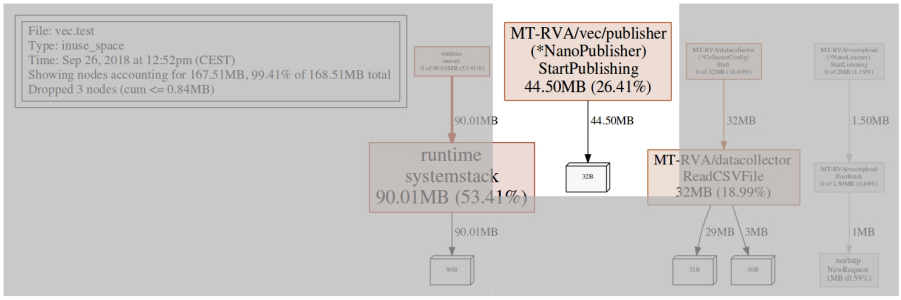


Figure 4.6: Pprof tool highlighting the buffer size

An arrival intensity of 1 Hz showed larger buffer usage than the buffer size for 10 Hz. This can be credited to when the buffer is read, as the buffer difference of 600 bytes can be a couple of messages. When the buffer is empty the size is 0.512 MB. If the buffer is read when it has one message, the buffer size increases by a large percentage in comparison with an empty buffer. In most of the tests, the buffer was empty at 1 Hz, however not on every test. The buffer must have been read between the time the message was added to the buffer and the serving time of a message.

The CPU usage shows a linear climb until the threshold 10000 Hz where it changes to an exponential decline. The system that the tests are run on, is a dual core system and the execution time is shared between the agent control and the agents. Therefore, the amount of CPU usage that the system can obtain at most is around 120% depending on the balance between the agent and the agent control. Therefore, the CPU usage is the primary bottleneck, as the RAM usage as seen from *htop* is far from the maximum of 8GB.

The throughput can be seen as a measure of how well the system performs at different frequencies. A delivery of 100% of the messages is not achieved at any level. This is due to the fact that when the throughput was read, the number of messages read was the number of delivered messages from the datacollector to the server, hence not the messages that were at that time in the buffer. Furthermore, the throughput was calculated as the number of messages delivered over the total test time. The test time was the whole time the test ran, including the start up time for the system. Therefore, the start up time may be responsible for some of the discrepancy from 100%. The throughput can be seen to decrease significantly

in comparison with the theoretical maximum. One of the main reasons for the difference is that the system is not able to manage the high frequency of arriving messages which leads to a lot of messages being thrown away due to a full buffer.

As the arrival intensity increases the amount of stress on the system increases, by a linear amount until there is instability in the system. Therefore, the system in its current state and hardware can be run on a 10000 Hz sampling system. In context with the real world, this is a very high sample rate. For example, you could not expect to be shifting gears every 100 microseconds, thus most of the samples would be redundant. However, it is relevant when running multiple agents. The system can be proven to be stable up to 10 agents with frequencies at 100 Hz.

Chapter 5
Bandwidth reductions

This section outlines the testing methodology for what bandwidth reductions that can be made. The tests were based on the proof of concept, and highlights the primary research question. In order to evaluate the proof on concept in terms of the research question, a number of a appropriate test cases were set up.

From the limitations found in the results and discussion of the arrival intensity of sensor data, four scenarios can be created to represent four use cases with different processing power and edge computer power. With edge computing power means the amount of edge computing that could be performed within the given limitations the scenario introduce.

There are a number of ways to receive data from the car as previously discussed, such as manually downloading data through a cable interface, such as during a service. Secondly, one can send all the relevant data to the cloud. Thirdly, one can do simple calculation such as a filtering of the data in order to receive the wanted set of data. And lastly, the entirety of calculations are done in the car and results are sent to the cloud.

The first scenario will be disregarded in terms of the tests here because the availability of the data is essential to the project. Thus, receiving the data on a yearly basis, when the car is serviced, is not practical.

Thus the following scenarios are created:
No Agent - Sends all the sensor data to the cloud as soon as it is available
No Edge Computing - Sends all the sensor data through an agent to the cloud as soon as it is available
Low Edge Computing - Filters the values such that only a certain interval of values are given
High Edge Computing - Filters the values and does the entirety of the calculations in the car

In order to evaluate these scenarios, a number of metrics needed to be established. In terms of the research question, the memory, CPU usage as well as evaluating the availability of the data.

The memory was measured through three instances. The memory usage of the agent control environment, through Pprof. The memory usage of the agent itself through htop. The CPU usage was also measured through Pprof in terms of the agent control environment, and only through htop in terms of the agent.

The goal of the tests was to identify which parameters that are important and

affect the system the most, and how trade-offs between processing powers could be made while still being able to perform edge computing use cases.

5.1 Test methodology

The pipeline in which the system was run was kept the same, with one publisher socket running over TCP and one IPC socket responsible for uploading the data. The data set was kept the same across the tests, in order to simulate the same scenario. The data was sampled at a rate of 100 Hz which was shown to be stable as seen in the results from the arrival intensity tests in table 4.1.

The system in question will run with four different agent set-ups, the first being without the any edge computing, passing the data from the datacollector straight to the uploader. The second with a simple agent, uploading all of the requested sensor data with the help of the agent to the cloud. The third agent will filter the sensor data on one parameter, and send the data for those specific intervals. The fourth agent will filter and do some calculations on the parameters at the specific intervals and upload the results.

In order for a fair comparison, the scenario with No Edge Computing, will exclude the agent, and pass the data straight to the upload listener. This way the pipeline is kept the same, in all four scenarios.

The steps for the test was the following:

1. Set up the static parts of the test for the proof of concept
2. For each scenario from the four test cases
    (a) Set up hardware constraints according to the scenario
    (b) Set up edge computing complexity by selecting appropriate agent
    (c) Perform a benchmark measuring bandwidth load
    (d) Perform a benchmark measuring CPU usage
    (e) Perform a benchmark measuring memory consumption

5.2 Results

The results are presented in four tables and graphs. The first section shows the CPU usage, the second shows Memory usage, the third shows Bandwidth usage and the forth shows the three named areas summarized in one graph.

Table 5.1: CPU usage in percentage for agent and pipeline, where 100% is maximum CPU usage of one core

Scenario	Agent [%]	Pipeline [%]
No Agent	0	34.77
No Edge Computing	15.33	41.97
Low Edge Computing	28.62	6.662
High Edge Computing	25.2	6.907

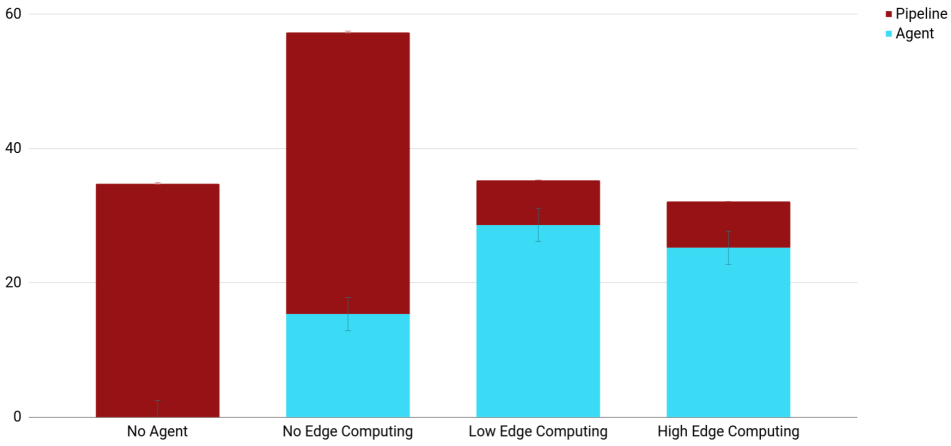


Figure 5.1: CPU usage over different edge computing scenarios

Table 5.1 and the Graph 5.1 shows CPU usage for the different scenarios and how the CPU usage is divided between the agent and the pipeline. The No Agent scenario shows that all CPU usage is in the pipeline. The scenario of No Edge Computing, shows that a substantial amount of CPU is used by the pipeline. The CPU in the other two scenarios with edge computing is clearly used most by the agents and not the pipeline.

Table 5.2: Bandwidth reduction results for memory usage

Scenario	Agent [kB]	Pipeline [kB]
No Agent	0	0.08533
No Edge Computing	41.79	0.256
Low Edge Computing	57.48	8.117
High Edge Computing	57.24	8.532

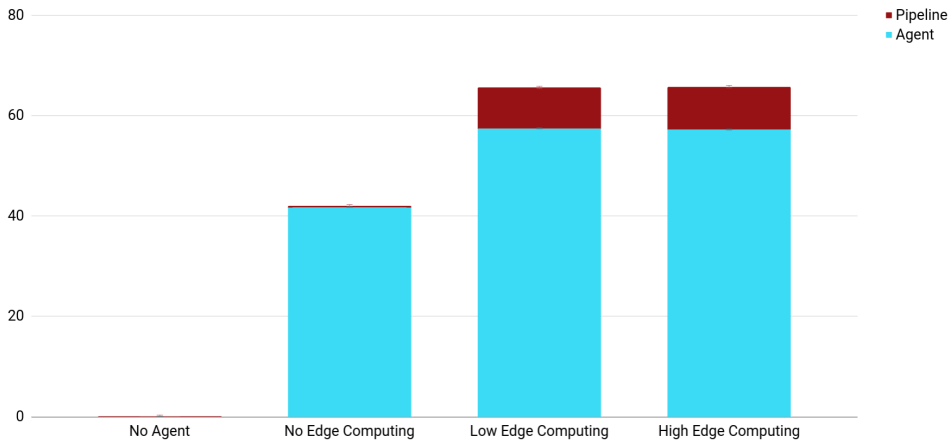


Figure 5.2: Memory usage over different edge computing scenarios

Table 5.2 and the Graph 5.2 shows the memory allocations to the heap for the different scenarios and how the memory usage is divided between the agent and the agent control. The No Agent scenario shows that almost no memory is used and since no agent is used, the pipeline stands for 100%. In the scenario of No Edge Computing, the Agent stands for almost all memory usage, 41.79 kB, and the pipeline is barely visible with its 0.256 kB memory usage. The two scenarios with low respectively high edge computing shows that it is the agent that stands for a considerable amount of the memory usage.

Table 5.3: Results for bandwidth usage over different edge computing scenarios

Scenario	Total usage [kB]
No Agent	10223.1
No Edge Computing	10223.1
Low Edge Computing	22.152
High Edge Computing	1.032

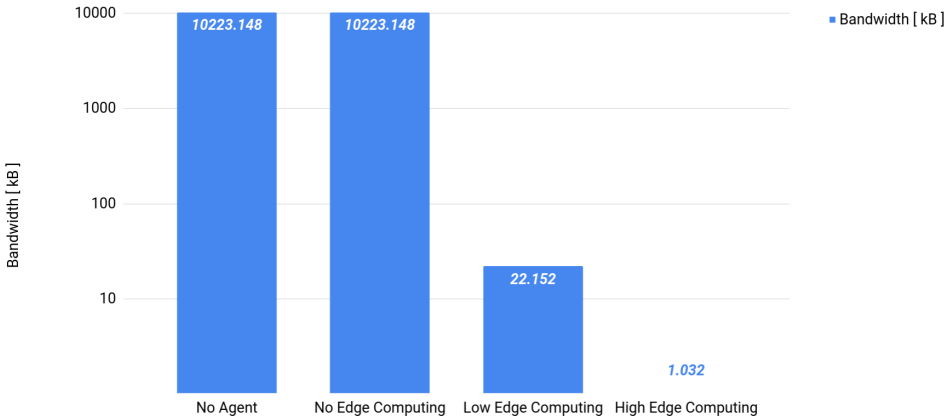


Figure 5.3: Bandwidth usage over different edge computing scenarios

Table 5.3 and the Graph 5.3 shows bandwidth usage for the different scenarios. The No Agent and the No Edge computing scenarios shows that the total bandwidth is the same, 10223.1 kB. The two other scenarios shows that the bandwidth is considerably lowered by the introduced edge computing.

A summary of the metrics CPU, memory and bandwidth are shown in 5.4. The image shows that the low and high edge computing scenarios are similar in x and y positions, i.e. CPU and memory usage, and fairly similar in size, i.e. bandwidth usage. While the No Agent and No Edge computing scenarios are the same in size, they differ in x and y positions.

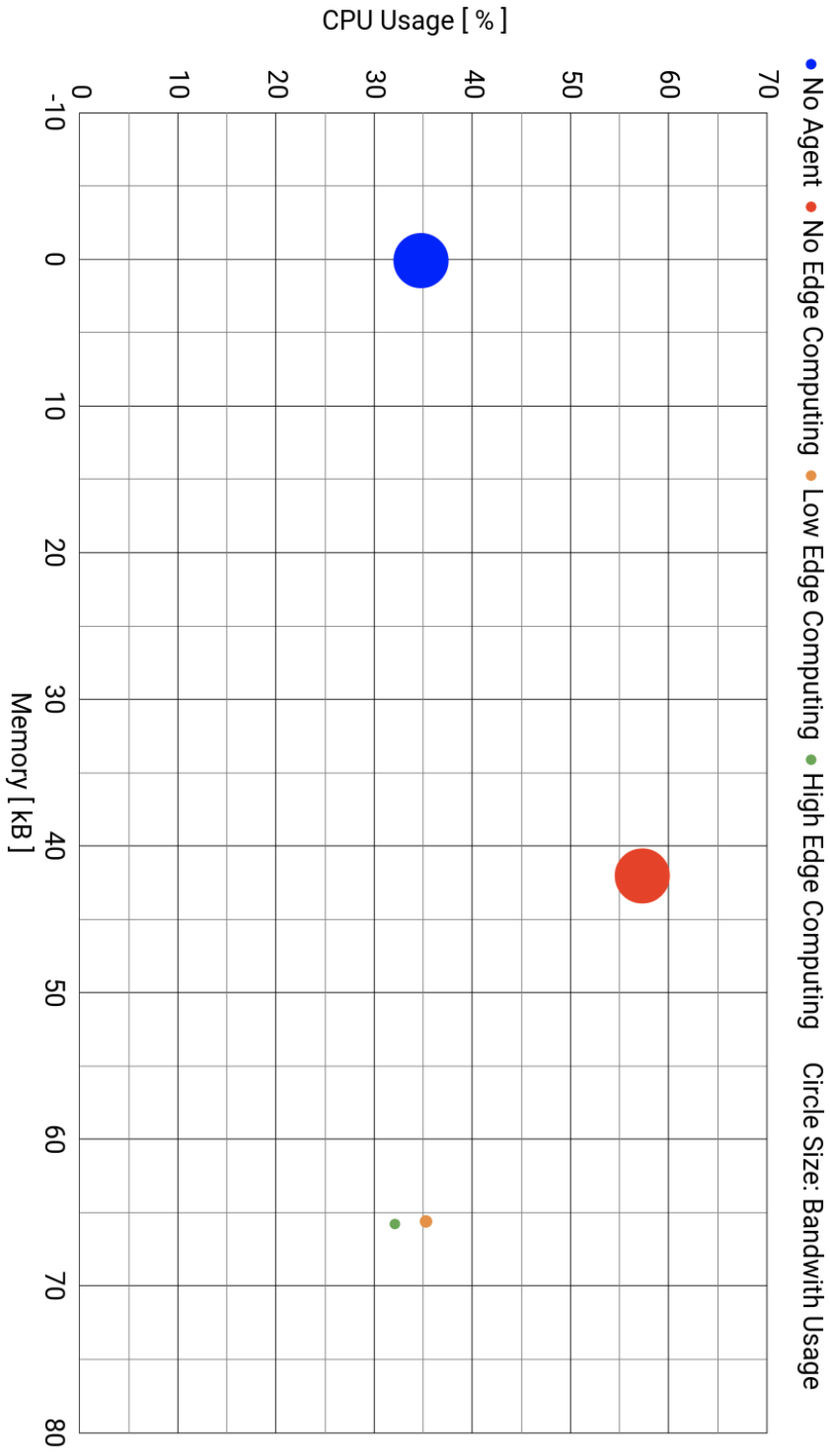


Figure 5.4: CPU, memory and bandwidth over different edge computing scenarios

5.3 Discussion

The results show the agent and the agent control environment as a single measurement. However, they were measured and presented as separate to be able to visualize how the agent behaves by different levels of edge computing. It is important to include the entirety of the system together to indicate how the whole system behaves at different levels of edge computing.

There is a throughput of 100% of the messages in all scenarios, which proves a stable buffer and system. The sizes of the delivered packages are not affected by the different scenarios. The size of the Protobuf message are from 20.3 to 21.5 b/message which is a difference of less than 6%. The consistency in package sizes proves that the data that has been processed does not change in size when the data is from the same data set. The bandwidth was also further reduced by the choice of serialization format which reduced packet size from about 100 bytes in json to 20 bytes in Protobuf. The design choice of using Protobuf instead of Json clearly shows as the better choice.

The CPU usage is almost the same for three of the scenarios, however, for the No Agent scenario all of the CPU usage is in the pipeline since there is no agent. The scenario with no edge computing stands out because it uses more CPU than the rest. This shows that simply having an agent which uploads all the data is not an effective approach. The CPU usage for the No Agent scenario is almost the same as the Edge Computing scenarios due to the fact that it sends a lot more packages through HTTP-requests. This can be seen by looking at the Pprof tool and isolating the areas that are responsible for HTTP-requests. An example from a Pprof benchmark from one of the tests shows that around 56% of the CPU usage is related to HTTP-requests, this is calculated by adding all parts that are connected to HTTP requests, which in one example are the parts A, B, C, D, E in Figure 5.5. The parts were extracted from the Pprof tool and are part of a larger call trace. It also worth noting that the HTTP requests are generally one of the most CPU intensive tasks in the overall pipeline. The No Edge Computing scenario has a higher CPU usage than the other three scenarios. In that scenario the agent is used to receive every message and then pass it through to the uploader. Since the pipeline must send and receive messages to and from the agent the CPU in the pipeline will be higher than in the No Agent scenario. The agent in the No Edge Computing scenario does not do any computations, as a result, the CPU usage in the agent will be lower than in the agents in the two scenarios with Low and High Edge Computing.

The results show that the CPU usage is higher for the scenario Low Edge Computing than High Edge Computing. The reason for the CPU being higher correlates to the number of HTTP-requests being sent, which is greater in the Low scenario compared to the High scenario. This again relates back to indicating that HTTP requests are taxing to the on the system.

From the results, the memory can be seen to be close to 0 for the scenario with no agent. The memory is 8.5 bytes because, at that scenario, the pipeline sends all the messages directly to the server, meaning no buffer is used. The memory increases substantially in the No Edge Computing scenario when keeping in mind that it does the same as the No Agent Scenario. When comparing the memory

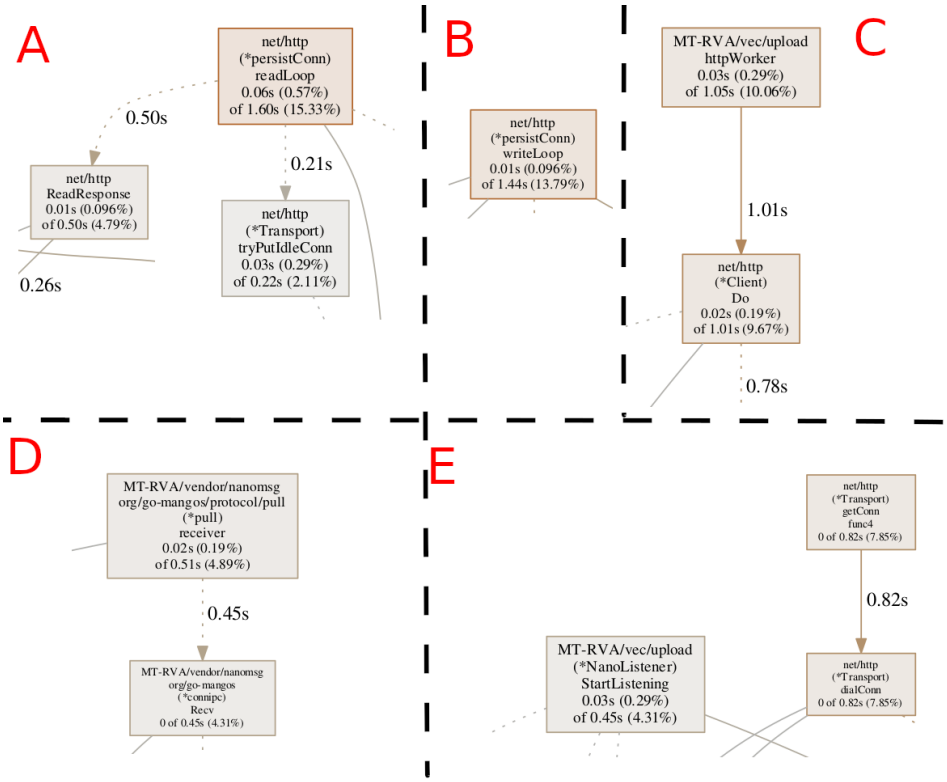


Figure 5.5: HTTP requests isolated from the Pprof tool. (A) Call trace of HTTP response reader. (B) Call trace of HTTP write request. (C) Call trace of a HTTP thread. (D) Call trace of Nanomsg upload socket. (E) Call trace of HTTP connection and listener

usage between the scenarios with edge computing it can be concluded that the baseline is relatively high since the memory usage in the No Edge Computing scenario is more than half of the memory usage in the Low and High Edge Computing scenarios.

The memory shown in the Pprof benchmark is the allocated memory to the heap, which is currently in use. Therefore, one can conclude that most of the memory usage in the No Agent scenario is on the stack. In the scenario with a No Edge Computing, one can observe the minimum amount of memory usage needed to start an agent which is approximately 40 kB. The memory usage for the two edge computing scenarios are within 1 % of each other. In both of the edge computing scenarios, the agent needs to receive all of the data that the pipeline handles, and thus they need to be buffered in the publisher. The buffer is the main contributor to the memory usage, this is the reason that the memory usage for the edge computing scenarios are similar. The memory usage for the simplest of Python agents is also quite high due to the nature of the interpreter.

The bandwidth usage is 102223.1 kB for the No Agent and the No Edge Computing scenarios. This result is expected as both of the scenarios sends all data that is read and proves that the system is stable, i.e. drops no packages. The bandwidth usage is 22.2 kB for the Low Edge Computing scenario and 1.0 kB for the High Edge Computing scenario. This is a reduction from the No Agent and No Edge Computing scenario to 0.2% for the Low Edge Computing scenario and a reduction to 0.01% for the High Edge Computing scenario. This result is achieved by sampling data at a rate of 100Hz over 2 hours. The reduction is thus to 0.2% and 0.01% over that period of time. A longer sampling period would produce a greater reduction. Another representation of this is that it can be shown that the bandwidth used is reduced by 137.5 % every second.

The bandwidth usage is directly dependent on the number of packages sent. The number of packages that are sent are decided by the filtering, aggregations and calculations done in the agent.

If edge computing is present, trade-offs regarding memory are present and will demand more resources from the system. However, it does not make any noticeable difference if the agents perform little or a lot of edge computing since the major component affecting the memory is the buffer. Trade-offs regarding CPU does not appear due to the fact that a majority of the CPU usage is moved from the pipeline to the agent if edge computing is present.

Chapter 6
Discussion

The system has proven that edge computing will reduce the amount bandwidth needed, at very little cost in terms of CPU and memory in comparison with uploading all the sensor data. In comparison with today's solution of retrieving the data at yearly intervals or whenever the car is serviced, the increase of CPU and memory usage is significant. However, the data shows that the amount of CPU usage and memory usage needed is directly proportional to the sample rate (arrival intensity) of the sensor data. Therefore, dynamically adjusting the sample rate to suit the CPU and memory allowance, will allow the system to perform according to the requirements put upon on it. Decreasing the CPU usage and memory usage comes with the trade-off that the data availability will decrease, as well as the risk of errors may increase.

The risk of measurement errors may increase at lower sample rates due to missing essential data. For example in Figure 6.1, if the sample is done at sample 1 and sample 2, one could lose the information that the car has traversed through gears 2-6, and only observe a steady gear position. The possible error varies for each measurement value, since the interval at which the measurement index updates varies. For example, the gear position is only expected to switch as fast as the car can switch gears. Therefore, with varied sample rates according to the individual metric, most of the errors can be kept to a minimum.

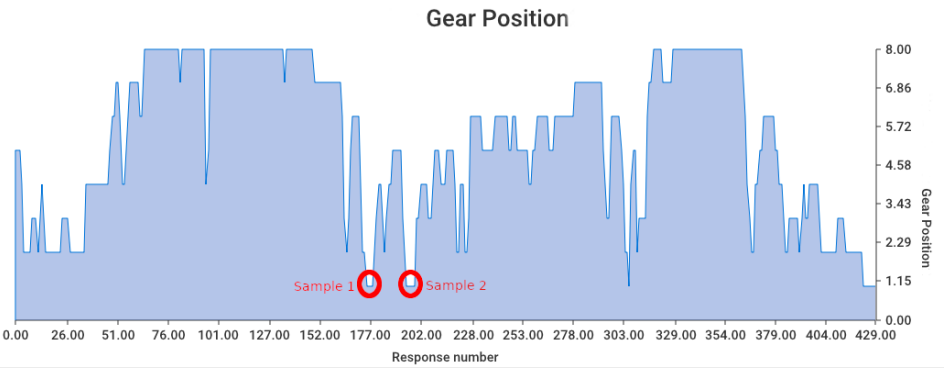


Figure 6.1: Sampling data with potential error

It is clear that the agent's purpose will dictate the potential bandwidth reductions gains. A simple example being that, if instead of uploading a value, the value is buffered and the average is uploaded for every 10th value, the bandwidth needed would be reduced by 10. This indicates, that the more intelligent scenarios found for the agent scripts, the more the bandwidth can be reduced. On the other hand, a poorly designed agent, that does heavy calculations and doesn't reduce the amount of data needed to purvey the information. For example an agent that converts a value from one form to another would result in very poor performance.

The proof of concept has also proven that it is feasible to run edge computing in a car with quite minimal impact to the CPU and system as a whole. The PoC was also proven to scale well as it was tested with twelve agents, each with a different measurement assignment, without any issues. In the future, with hardware more powerful than the current car, more complex agents with heavier calculation loads and higher sample rates should not be an issue.

The pipeline as it stands is filtered on the subscriber side, which means that the publisher is in some way reliant on the speed of the subscriber. The performance of the publisher could be increased if the filtration was done in the publisher.

The system was introduced with robustness requirements, and these requirements have been met to a certain degree. The pipeline functions concurrently, and the entire pipeline can be isolated by being run in a Docker container. This can be done in conjunction with using Cgroups for the agents, whilst in a Docker container. However, there is no life cycle management for the pipeline itself, only for the agent scripts that it executes. In comparison, the agent scripts have complex life cycle management that is made incredible cheap in terms of memory and CPU usage due to the nature of goroutines. Allowing each agent script to have its own supervisor, solves a lot of the robustness criteria without much CPU and memory usage.

Chapter 7
Conclusions

This masters thesis has tackled the problem of bringing sensor based edge computing to vehicles. The problem motivation raised two questions; How arrival intensity of sensor data affects the system, and what bandwidth reductions gains can be made. Firstly a pipeline was defined, and technologies and frameworks were evaluated and compared to be used in said pipeline in the method chapter. Secondly, the pipeline was successfully implemented, and tested in a real car. Therefore, the conclusion can be drawn that there is a possibility for edge computing in vehicles today.

From the results in Chapter 4, which addressed the research question, of what arrival intensity the pipeline can handle, one can draw the conclusion that the system can handle intensities of up to 10,000 Hz. This means that the system exceeds the requirements of being able to run 10 agents with a sample rate of 100 Hz each. However, this is highly dependent on the hardware that is available in the car. Therefore, no concrete conclusion can be made for a general case. However, for the specific hardware used, the requirements could be met. Furthermore, the system showed exponential decline in throughput after the 10,000 Hz intensity, which means that one can draw the conclusion that it is not viable to run higher intensities, because the loss of data is high.

Consequently, from the results in chapter 5, which addressed the research question of what bandwidth reductions gains can be made at what cost. One can first draw the conclusion that using a real world use case, the amount of bandwidth reduction gains that can be made are significant. A corollary is that the overall CPU usage is unaffected in cases that were tested.

However, the use case for the agent is the determining factor in what bandwidth reductions gains that can be made in these scenarios. Since the use case is the determining factor, no conclusion can be made on whether using edge computing is always a bandwidth reducing activity.

There is however, a clear trend in that exchanging a HTTP-request for a calculation can work to reduce the amount of bandwidth needed as well as the amount of CPU usage needed as long as the calculation is reducing or aggregating the data.

Data availability is difficult to measure in terms of the tests run, despite the QoS features that were implemented. The system is designed to always give the agents the latest data, and if the data arrives faster than the agent can handle the oldest data is discarded. Therefore, the conclusion can be drawn that the data

availability will suffer if the agents can not keep up with the arrival intensities.

Failure handling was added to the system through the robustness criterions outlined in Chapter 2, which means the system has some overhead in terms of fault detection, fault identification, and isolation. However, the conclusion can be drawn that the overhead of the robustness aspects was minimal as in the real world use case the pipeline only used approximately 6% of the CPU which was to a great extent the publisher and upload.

In summary, the masters thesis aimed to investigate what bandwidth reductions gains can be made in cars today. Using the results, the bandwidth reductions gains are significant, and the potential sample rate that the system supports are indicators towards the usability of edge computing in a vehicle. In conjunction with the stability and robustness that the car industry requires, this can serve as proof for further use cases for edge computing in cars.

Chapter 8
Future Work

The Master thesis has focused on the research question: *What bandwidth reductions gains can be made by increasing the amount of data processing within the car itself, and what trade-offs in terms of e.g. data availability, processing power requirements, and failure handling does such edge computing processing introduce?* through the perspective of software as the hardware was static. In future iterations, one could look at how different hardware platforms are more suitable towards evaluating edge computing in the car. Research on how different hardware platforms could increase the frequency, i.e. lowering the arrival intensity, and the amount of processing that could be done the car could be made.

Further research regarding filtration of messages could be done. More specifically, if the filtration would be done on the publisher side, the overall performance of the publisher could be improved as well as the traffic on the internal sockets reduced.

Security is an important aspect of edge computing as well as in every part of a car, including gathering, handling and analyzing sensor data. A general approach to security have been followed in this Master thesis, but since the scope of the Master thesis did not cover security aspects, possible problems related to security have not been resolved. In the POC there does not exist a way to dynamically add new agents to the system. This choice was made early on in the Master thesis due to the security constraints it impose and fact that addressing it would not fit in the time schedule of the Master thesis. However in the future, dynamically uploading agents might be a wanted function to have, for instance when the car is serviced. If such functionality should be implemented, relevant security measures must be taken into account, such as preventing upload of malicious agents and further limiting the parts that an agent could access through e.g. stricter control groups.

The trade-offs relating to the research questions that have been tested and discussed are all dependent on the hardware that it was run on. The hardware the POC will run on in the future will change and therefore the trade-offs in this Master thesis might not be directly applicable to future solutions. What trade-offs should be made in future solutions must therefore be reevaluated.

Streaming data in real-time (below a certain delay threshold) has its obvious use cases and advantages over streaming in best effort. One of the main advantages enabled by real-time streaming is the use cases that are made possible by being able to receive data or instructions in real-time. The choice was made to not

include the real-time streaming in the scope due to lack of time.

Storage is an important part in edge computing scenarios where downtime of connection to the cloud could appear. Wireless connection problems over the 3G- and 4G-net are common and thus storage aspects are an important part of edge computing in a connected car scenario. Trade-offs relating to storage have not been researched due to lack of time. Future work needs to address issues such as: How should the system handle data when the wireless connection is inaccessible for a longer period of time? What data should be kept and what data should be thrown away when the local storage in the car is full?

References

[1] László Gyarmati and Mohamed Hefeeda. Analyzing in-game movements of soccer players at scale. *MIT Sloan Sports Analytics Conference 2016*, 2016.

[2] J. Luo, K.R. Pattipati, and L. Qiao andalice S. Chigusa. Towards an integrated diagnostic development process for automotive systems. *2005 IEEE International Conference on Systems, Man and Cybernetics*, 2005.

[3] Raffaele Bolla, Alessandro Carrega, Matteo Repetto, and Giorgio Robino. Improving efficiency of edge computing infrastructures through orchestration models. *Computers 2018, 7(2)*, 2018.

[4] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K Leung, Christian Makaya, Ting He, and Kevin Chan. Adaptive federated learning in resource constrained edge computing systems. Technical report, Tech. Rep., Jul. 2018.[Online]. Available: https://ibm. co/2zNZLmG.

[5] Yuxuan Sun, Jinhui Song, Sheng Zhou, Xueying Guo, and Zhisheng Niu. Task replication for vehicular edge computing: A combinatorial multi-armed bandit based approach. *arXiv preprint arXiv:1807.05718*, 2018.

[6] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology - Mikroelektronik och informationsteknik, 2003.

[7] William Tärneberg, Amardeep Mehta, Eddie Wadbro, Johan Tordsson, Johan Eker, Maria Kihl, and Erik Elmroth. Dynamic placement in the mobile cloude network. *Further Generation Computer Systems*, 70:163–177, 2017.

[8] Roberto Morabito and Nicklas Beijar. Enabling data processing at the network edge through lightweight virtualization technologies. In *Sensing, Communication and Networking (SECON Workshops), 2016 IEEE International Conference on*, pages 1–6. IEEE, 2016.

[9] Benno Stein, Manu Sridharan, Lazaro Clapp, and Bor-Yuh Evan Chang. Safe stream-based programming with refinement types. *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, September 2018.

[10] Global interpreter lock. `https://wiki.python.org/moin/GlobalInterpreterLock`. Accessed: 2018-10-24.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[12] Reactive extensions framework. `http://reactivex.io/`. Accessed: 2018-10-24.

[13] Qin Li, Huibiao Zhu, Jing Li, and Jifeng He. Scalable formalization of publish/subscribe messaging scheme based on message brokers. *Lecture Notes in Computer Science, vol 4937*, 2008.

[14] Rob Pike. Go at google: Language design in the service of software engineering. *SPLASH*, 2012.

[15] Golang. Golang FAQ. `https://golang.org/doc/faq`. Accessed: 2018-10-24.

[16] iMatix. Zeromq. `http://zeromq.org/`. Accessed: 2018-10-24.

[17] Garrett D'Amore. Nanomsg. `http://zeromq.org/`. Accessed: 2018-10-24.

[18] Inc Pivotal Software. Rabbitmq. `https://www.rabbitmq.com/`. Accessed: 2018-10-24.

[19] Introducing json. `https://json.org/`. Accessed: 2018-10-24.

[20] Google. Protocol buffers. `https://developers.google.com/protocol-buffers/`. Accessed: 2018-10-24.

[21] Massimo Cafaro and Giovanni Aloisio. Grids, clouds, and virtualization. In *Grids, Clouds and Virtualization*, pages 1–21. Springer, 2011.

[22] Carlos Arango, Rémy Dernat, and John Sanabria. Performance evaluation of container-based virtualization for high performance computing environments. *arXiv preprint arXiv:1709.10140*, 2017.

[23] Docker Inc. Build, manage and secure your apps anywhere. your way. `https://www.docker.com/`. Accessed: 2018-10-24.

[24] Red Hat Inc. Introduction to control groups (cgroups). `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01`. Accessed: 2018-10-24.

[25] Rajendra Patel and Arvind Rajawat. A survey of embedded software profiling methodologies. *International Journal of Embedded Systems and Applications (IJESA)*, December 2011.

[26] Golang. Package pprof. `https://golang.org/pkg/net/http/pprof/`. Accessed: 2018-10-24.

[27] A. Wegrzynek V. Chibante Barroso, U. Fuchs. Benchmarking message queue libraries and network technologies to transport large data volume in the alice o system. *2016 IEEE-NPSS Real Time Conference (RT)*, 2016.

[28] Bo Petersen, Henrik Bindner, Shi You, and Bjarne Poulsen. Smart grid serialization comparison. In *SAI Computing Conference, London*, 2017.

Appendix A

Goal question metric

Table A.1: GQM criterion - Supports architectures(x86 and ARM7)

Score	Criteria
1	Does not compile
2	Cross-compiles
3	Compiles

Table A.2: GQM criterion - Supports Golang/Python

Score	Criteria
1	No support
2	Support through wrapper
3	Native support