CHALMERS
UNIVERSITY OF TECHNOLOGY

Design and evaluation of a software architecture and software deployment strategy

Master's thesis in Automotive Engineering, and Software Engineering and Technology

DAN ANDERSON
ZIWEI HUANG

Department of Mechanics and Maritime Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

MASTER'S THESIS IN AUTOMOTIVE ENGINEERING, AND SOFTWARE ENGINEERING AND TECHNOLOGY

Design and evaluation of a software architecture and software deployment strategy

DAN ANDERSON
ZIWEI HUANG

Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2018

Design and evaluation of a software architecture and software deployment strategy
DAN ANDERSON
ZIWEI HUANG

Master's Thesis 2018:14
Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: + 46 (0)31-772 1000

Cover:
The Chalmers formula student 2017 race car that was used for evaluating the deployment strategy.

Chalmers Reproservice
Göteborg, Sweden 2018

Design and evaluation of a software architecture and software deployment strategy

DAN ANDERSON
ZIWEI HUANG
Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology

Abstract

The thesis is motivated by the *Chalmers formula student driverless 2018* (CFSD'18) project, a pilot project with the aim to deliver a qualified self-driving formula race car and compete in the Formula student Germany 2018 competition. The thesis goal was to design and evaluate the software development and deployment process for a purely microservices-based distributed system on the CFSD'18 self-driving race car. The work coincides closely with the vehicle laboratory Revere, operated by Chalmers University of Technology and the University of Gothenburg. It explores the use of OpenDLV based on libcluon, applying microservice architecture in the CFSD'18 project. A docker-based deployment strategy is investigated and evaluated. A project specific deployment strategy is designed considering hardware related physical constraints and Formula student Germany competition rules. Performance of certain microservices has been measured and evaluated. The experience from this thesis indicates the OpenDLV platform, libcluon, and the docker ecosystem are portable, efficient and adaptive choices for a distributed embedded system, in particular autonomous vehicle projects.

Keywords: Software deployment, microservices, OpenDLV, libcluon, Docker, self-driving, Formula student

Acknowledgment

We would like to express gratitude towards our examiner Ola Benderius and our supervisor Christian Berger for their professional guidance and support during the project. Our grateful thanks also extend to the CFSD'18 team for their cooperation and making this project possible. The assistance given by Björnborg Nguyen was greatly appreciated.

We would also like to thank Revere for sharing their resources and offering valuable help throughout the project.

Göteborg May 2018

DAN ANDERSON
ZIWEI HUANG

Thesis advisor: Christian Berger
Thesis examiner: Ola Benderius

Contents

1 Introduction

Self-driving vehicle technology is a popular and important topic in both automotive industry and academic field. An increased amount of companies and university laboratories are working in this field. Chalmers University of Technology and the University of Gothenburg are operating and maintaining the vehicle laboratory Revere comprising 1/10 scaled cars, a Volvo XC90, and a Volvo FH truck to conduct studies with automated driving [1]. To support continuous development in different vehicle platforms the Revere lab implemented a vehicle independent software framework called OpenDLV. The open software framework handles hardware communication, safety and override functions, sensor fusion, and other base self-driving concept functions.

The motivation of this thesis comes from *Chalmers formula student driverless* (CFSD) project. The main work of the one-year project is to upgrade an electrical powered race car to a driverless vehicle. The goal of the project is to deliver a qualified driverless vehicle to participate and achieve top five in Formula student Germany competition hosted from 6th until 12th of August 2018 in Hockenheim.

The CFSD'18 team consisted of 12 second-year master students from various background and three main supervisors from Chalmers University of Technology and the University of Gothenburg. The vehicle laboratory Revere provided rich resources including both technical and non-technical support for the CFSD'18 team. The project started in August of 2017.

This thesis focuses on software deployment strategy in the distributed heterogeneous computer networks. As the authors were members of the CFSD'18 team, this thesis use practical experience from the CFSD project to explore, design and evaluate software deployment strategy in the distributed heterogeneous computer networks.

1.1 Problem domain & motivation

The CFSD project is a student-driven project with support from supervisors and the vehicle laboratory Revere. The design, manufacturing, implementation, and testing were all done by the CFSD'18 team, which was a big challenge since this was the first year to build a driverless race car at Chalmers.

Fortunately, for the software environment part, the CFSD'18 team does not need to start from scratch. The vehicle laboratory Revere has provided an autonomous driving software framework OpenDLV. Since the whole OpenDLV framework is a layered structure, the work for the team is to develop the vehicle and competition specific software on the top layer and reuse the foundation part for basic communication, sensor fusion, and other functions. The layers are shown in Figure 1. However, this was the initial approach for first few months. With the introduction of libcluon, the stacked layered architecture is gradually transformed to purely microservice based structure.

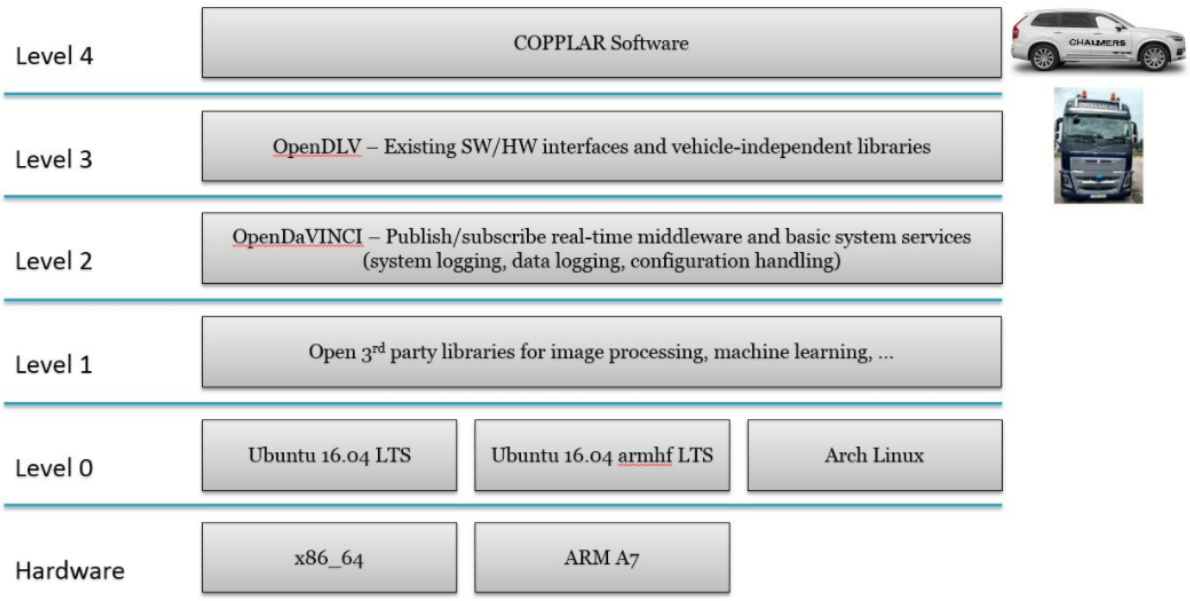

Figure 1, Layered Software Environments at Revere [2].

Software deployment involves installation, configuration, packaging components, uploading, activating and deactivating. The whole software deployment process will become complicated in the distributed heterogeneous computer networks. This thesis is to explore the already adopted software deployment strategy at Revere and use it as the base to adjust and design an efficient and traceable software deployment strategy for CFSD project. Furthermore, evaluate the designed software deployment strategy and contribute to the cross-platform development at the vehicle laboratory Revere.

1.2 Research goal & research questions

The aim of the research was to evaluate development and deployment strategies for a purely microservice-based distributed system on the example of self-driving race car project CFSD'18.

The microservice software structure is gradually adopted in the project. The previous version of OpenDLV ecosystem built on top of OpenDaVINCI was designed as layered structure, while the new version OpenDLV ecosystem is refactored and designed as microservices using libcluon.

Through mining potential Docker-based technique and experimenting on continuous deployment, the following three research questions will be answered.

RQ1 What deployment strategy has the smallest impact on building, releasing, deploying and running software bundles on the target system?

RQ2 How can the software be deployed in the fastest possible way while preserving full traceability: As docker-compose or as docker stack+docker swarm while considering hardware properties (x86_&4 vs armhf)?

RQ3 How can live data be visualized and recorded for further offline analysis?

1.3 Contributions

The thesis explores the use of OpenDLV, based on libcluon, to apply a microservice architecture in CFSD'18 project. Docker-based deployment strategy is investigated and compared. Performance of certain microservices have been measured and evaluated.

All the result and experience concluded in the thesis will contribute to the cross-platform development at the vehicle laboratory Revere as a large practical use case. The thesis will also work as a good pre-study resource of software architecture deployment for CFSD team in the following years. Furthermore, the thesis indicates the OpenDLV platform, libcluon, docker ecosystem are portable, efficient and adaptive choices for distributed embedded system, in particular autonomous vehicle projects. Researchers interested in using OpenDLV, libcluon, and Docker can find some insights from the thesis.

1.4 Limitation

Since the thesis focuses on the CFSD'18 project, the functionality of the software was designed based on the Formula student Germany 2018 competition rules. In contrast to real-work scale for OEMs, the thesis looked more closely into aspects and constraints that matter for competitions (for example, fast but reliable deployment); results, though, are of interest for the automotive domain in general. Also, the software deployment strategy was constrained by the competition rules.

1.5 Structure of the thesis

The rest of the thesis is structured as follows: Section 2 introduces the background of the work, explaining the concepts and techniques related to this thesis. The related work is summarized in Section 3. The research methodology and the experiment settings are specified in Section 4. Next, Section 5 presents the result from the experiment, and the discussion on the result and experience can be found in Section 6. Then, the threads to validity are discussed in Section 7. Finally, Section 8 concludes the thesis and indicates further work.

2 Background

In this section, concepts and techniques related to the thesis are present and explained in general. It covers both technical terms for the thesis and non-technical information about the CFSD'18 project.

2.1 Formula student

Formula student Germany (FSG) is an international design competition for students annually since 2006. The content of the competition is to design and manufacture a single-seat formula race car to compete against teams from all over the world. The challenge of the competition is not to build the fastest race car, instead, is to deliver the best overall package of design, construction, performance, and business planning [3]. Formula Student has traditionally been about building a combustion engine car. In 2010 Formula student Germany started a new class for electrical vehicles and in 2017 they introduced a new class of competition, the Driverless Vehicle class.

The first Chalmers formula student project was initialized in 2001, building a combustion engine car for the Formula Student UK competition in 2002. In 2013 CFS made the first prototype of an electric vehicle by converting an old combustion car. In 2015 the project goal was changed to produce an electric vehicle instead of combustion vehicle. In 2017 the next step was taken, and Chalmers decided to build its first driverless race car by converting the electric car from 2017, see Figure 2. The aim was to compete in the FSG Driverless Vehicle class 2018.

Figure 2, Chalmers formula student 2017 competing in skidpad at Formula student Netherlands.

2.2 Distributed embedded system

One definition of embedded system from Marwedel [4] is “Embedded systems are information processing systems embedded into enclosing products”. Common examples including embedded system are cars, trains, planes, and communication equipment.

The new term “cyber-physical system” appears more when the emphasis is put on the link to physics. The definition according to Lee [5] is “Cyber-Physical Systems (CPS) are integrations of computation and physical processes”. So, the CPS can be understood as a close combination system of embedded system and the target physical environment.

A distributed system is understood as a group of computation nodes working in a dedicated network to solve a problem. A typical distributed system can be a data center with thousands of servers or the control systems on the car.

2.3 Software deployment

Software deployment is generally interpreted as several interrelated activities to make the target software system available to run. In a typical software development lifecycle, a project normally starts from requirement specification and then goes through design, developing, and testing. As shown in Figure 3, software deployment can be considered as the final phase of the cycle. However, considering iteration and agile approach to software development, the deployment phase can continuously provide feedback for requirement and other steps to adjust and improve.

The whole software deployment process will become complicated in the distributed heterogeneous computer networks. This thesis is to explore the already adopted software deployment strategy at Revere and use it as the base to adjust and design an efficient and traceable software deployment strategy for the CFSD’18 project.

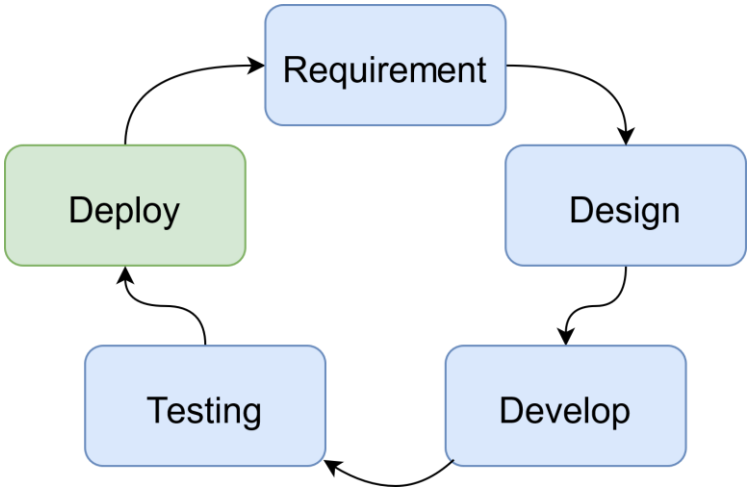

Figure 3, Software development lifecycle.

2.4 Microservices

In a monolithic architecture, an application or system is built as a single unit [6]. Consequently, modules in a monolithic application cannot be executed independently [7]. Monolithic applications have good performance. However, when increasing applications are being deployed to the cloud and distributed system, microservice architecture seems a better choice. Compared with putting all functionality into a single process in monolithic way, microservice make it possible to separate each element of functionality [6].

Microservice pattern enable developers to build solutions with speed and safety in a scaled way [8]. As microservice is a newly appeared software technique terminology, there is no formal and strict definition for it. One proposed definition of microservice involving architecture perspective is:

> "A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices [8, p. 6]."

2.5 Container-based virtualization

In practice, microservices and containers are naturally paired, contributing to greater modularity realization, code reuse and scalability of a distributed system [9]. Docker [10] is a popular platform for container-based virtualization. The core components for docker are docker engine, docker image, docker container, and repository.

A container image is an executable that includes everything needed to run it. It's a lightweight stand-alone solution used to run the same software regardless of the environment. The function of a container is like a virtual machine, but instead of running a separate operating system the container only contains the important software required to run the program and is sharing the kernel with the host, Figure 4. [10].

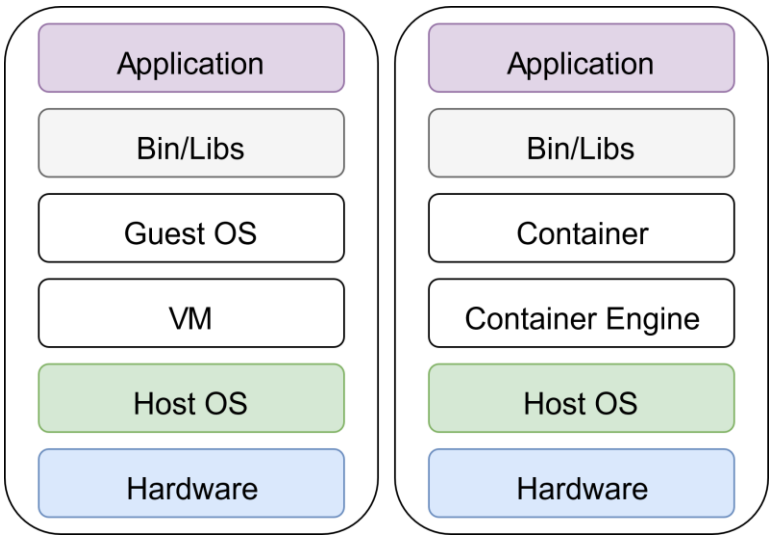

Figure 4, Layered structure of a container based virtualized in comparison with a virtual machine setup.

2.6 OpenDLV

To support continuous development in different vehicle platforms, Revere implemented a vehicle independent software framework called OpenDLV. The open software framework handles hardware communication, safety and override functions, sensor fusion, and other base self-driving concept functions. Revere has successfully used OpenDLV on 1/10 scaled cars, a Volvo XC90, and a Volvo FH truck [1].

2.6.1 OpenDaVINCI

OpenDaVINCI is an open source development architecture for virtual, networked, and cyber-physical system infrastructures. OpenDaVINCI is a compact middleware, written entirely in standard C++. It runs on a variety of POSIX-compatible OS and Windows [11].

OpenDaVINCI powered a variety of different scaled car for international self-driving research. One example is Berkeley´s AGV, the self-driving car from University of California, Berkeley's CHESS group in a joint research project with RWTH Aachen. OpenDaVINCI also powered the ARM-based self-driving miniature car "Meili" that won the 2013 international CaroloCup competition.

As shown in Figure 1, the OpenDLV system is stacked on the top of OpenDaVINCI.

2.6.2 libcluon

libcluon [12] is the world's first header-only middleware for distributed systems for robotic applications. It is a lightweight and efficient library written in modern C++ library to glue distributed software components together. The usage is simply done by including the cluon-complete.hpp into the project. libcluon contains well-designed features, including several native implementations of data serialization and deserialization.

The newest OpenDLV ecosystem by the writing time is realized with libcluon, transformed from previous layered stacked structure [13]. The software structure is entirely based on microservices. Because of strong focus on continuous deployment, Docker ecosystem is explored and widely used for building and storing images for microservices.

3 Related work

Software applications are increasingly integrated to heterogenous collection of software components; thus, deployment becomes an essential step of software life cycle [14]. Carzaniga's [14] study pointed out that the general problems related to software deployment systems involve evolution of both hardware and software system, components dependency and distributed heterogeneous platforms.

Many studies explored the possible approaches to evolve software deployment. New deployment approach should be able to perform on different platforms and networks and decentralize control for both software producers and consumers [15]. Software Dock is proposed as a cooperative framework to support deployment process via release docks and field docks. In robotics research, a model-based approach and a domain specific language are presented to achieve more flexible deployment on a wide range of robot platforms [16]. Moreover, there are already well-developed robot software distributed frameworks like ROS [17] and OpenRTM.

Agile software development methodology aims to improve the efficiency of software development process through early and fast delivery and close collaboration with customers [18]. Involving the feedback from end-users to the development process early help to reduce redundant and unclear features. Continuous deployment is to deploy the newest product to customers as soon as the new code has been generated. Continuous deployment has been successfully performed in organizations such as Facebook [19] and GitHub [20]. The common benefits from continuous deployment are reducing risks for each release, involving feedback from use early, avoiding useless features. However, adopting continuous deployment can introduce some social and technical challenges [21]. One problem regarding infrastructure is that proper software and hardware are required to handle continuous deployment. Testing can also be a challenge because continuous deployment needs continuous quality tests and code review. Some social challenges come from team coordination, team experience, customer adoption and others [21].

Microservices, inspired by service-oriented computing, are small application available to be tested, scared and tested separately [22]. Ebert [23] points out microservices have been increasingly adopted by industries for transformation from function-oriented legacy architectures to modern flexible service-based system. Ebert [23] also states that "The International Data Corporation has forecasted that by 2021, 80 percent of application development on cloud platforms will be with microservices."

Microservice has been increasingly adopted in software system for its attractive benefits. Some of the most important advantages are faster product delivery, increased infrastructure automation and organized scalability [24]. By splitting applications to multiple microservices, each microservice is supposed to run, test and deploy individually. This methodology is also in line with common agile principle of continuous delivery.

Some experience about migration a monolithic architecture towards microservices in real-world cases have been collected and reported. One case study [25] is done in collaboration with Danske Bank's FX Core currency conversion system. According to the banking domain experience report, the migration process was highly business-driven to facilitate proper definition and boundary of function and service. Containerization technology Docker and product in its ecosystem including docker-compose, docker swarm cluster, docker registry have been widely used for automation, orchestration, and integration. Learned from the banking case study, because of the challenging management of many independent microservices, the importance of well-designed software architecture and automation needs to be emphasized.

Microservice is not a perfect software structure for all applications and has drawbacks in real production process. Some interviews [26] have been done with practitioners owning experience in microservice-based systems to analyze existing harmful aspects of the microservice design pattern. Three drawbacks of the microservices are recognized as very harmful: (1) splitting applications based on traditional technical layers instead of business services, (2) hardcoded endpoints in the microservice network, and (3) ambiguous data ownership. The most frequent challenging part is to split and redesign a monolithic application, particularly when the horizontal layered structure has been a habit of development team.

One popular tool to cooperate with microservice architecture is Docker. Docker is an open-source platform to facilitate consistent development and deployment, based on container virtualization technology. Two typical usecases of choosing Docker are projects looking for high capacity and continuous integration [27]. Docker owns a wide range of customers including ADP, AR Group, and Cornell University. Containerization has been reviewed, discussed and supported as a lightweight virtualization solution in edge cloud environment [28]. However, little research of containerization technology can be found in distributed embedded system, in particular self-driving cars area.

The Revere laboratory operated by Chalmers University of Technology and the University of Gothenburg has adopted containerized deployment and microservice in their research on autonomous vehicles [29]. The build, test, and deployment process are encapsulated via VirtualBox, Docker, and Jenkins [1]. A few limitations come from lack of proper overlay network, conflicts of specific features, and pulling regularly large amount of data for end users [29]. The Revere laboratory provided the CFSD'18 team technical support on the already developed self-driving software environment OpenDLV. The main challenge for this work was to explore any possible reusable part, to find the bottleneck of the current deployment strategy, and to extend the functionalities to fulfill the Formula student Germany competition rules.

4 Research methodology

Scientific research is mutually nested by knowledge questions and practical problems [30]. In design science, a practical problem in this mutual problem nesting hierarchy is always on the top-level [31]. Based on this thesis, the clear practical problem is to deliver a qualified Formula self-driving race car and complete the final competition Formula student Germany in August 2018. More precisely, this thesis will focus on the distributed embedded software deployment strategy on the CFSD'18 race car.

Knowledge can be learned from exploring effects produced from the interaction between an artifact and a problem domain [30]. To achieve the research goal, practical questions and knowledge questions should be carefully balanced. This thesis will adopt a top-down analysis way by performing a design in the practical problem domain, then exploring generalizable distributed embedded software deployment strategy. The practical design is to design an efficient, stable and traceable software deployment strategy for the CFSD'18 race car.

To build connection between the practical problem and the bottom generalized knowledge question, evaluation, and continuous experimentation will be performed on the practical design on the CFSD'18 race car. Evaluation will include the reflection on relationship between the software structure and deployment strategy. Also, key indicators of the performance of deployment process will be explored and discussed. Furthermore, continuous experiment will be tried to collect informative feedback and use logged data to improve the design of software iteratively.

4.1 Goals

The aim of the research is to evaluate development and deployment strategies for a highly modular and purely microservices-based distributed system on the example of self-driving race car project the CFSD'18.

4.2 Study design

To answer the research questions, systematically mining into Docker-based orchestra technique and experimenting on continuous deployment have been done. The traceability is treated as a key factor of the continuous deployment process.

In a distributed software system, container orchestration is required to facilitate the process of deploying complex multiple containers on a group of hosts/machines. A benchmarking is performed on the potential choice for container orchestration, Docker Swarm + Docker Stack.

To make most of live data and facilitate offline evaluation, continuous performance measurement strategy is designed to monitor the whole system load. The measurement process itself should be designed as fully automated.

The microservice software structure is gradually adopted in the project. The previous version of OpenDLV ecosystem built on top of OpenDaVINCI is designed as layered structure, while the new version OpenDLV ecosystem is refactored and designed as microservices using libcluon.

The thesis will also explore the impact of transforming the existing OpenDLV multi-layer software stack to fully decoupled, cluon-based microservices. Compare the two architectural differences and reflect on the differences. Furthermore, the software build process will be discussed.

4.3 Experimental material

The software platform is OpenDLV. Two versions of OpenDLV have been investigated. The previously developed OpenDLV is a stacked layered structure with OpenDaVINCI as the base. The newly OpenDLV has transformed to fully microservices structure realized with libcluon.

Docker is heavily used for software deployment. The completed project source code is built as docker image. Docker image provides the possibility to encapsulate everything need to execute.

The key hardware includes two computational nodes and three sensors for perception.

4.3.1 Computer – x86_64

The computer used in the car has an AMD Razer 7 1700 CPU with 8 cores and 3GHz clock frequency [32]. The motherboard has ports for USB 3.0 and Ethernet that will be used to connect the sensors. The x86_64 computer will run Arch Linux and run the proxy microservices for camera, LiDAR, and IMU.

Computer hardware:

- AMD Ryzen 7 1700 3.0 GHz
- GeForce GTX 1060 Mini ITX OC 6GB
- Gigabyte GA-AB350N-Gaming WiFi (ITX)
- Corsair Hydro Series H60 High-Performance Liquid CPU Cooler
- 2x16gb G.Skill Flare X series 2400MHz DDR4 (not ECC)
- Samsung 850 EVO 500GB 2.5" SATA-600
- M4-ATX 6-30V DC/DC (250 Watt) PSU

4.3.2 Computer – Beaglebone Black

The Beaglebone Black is the computer used for low-level interface with the car. It has a 1GHz ARM Cortex-A8 processor. The *general purpose input and outputs* (GPIO) are limited to 3.3V and 4mA [33]. To make the GPIOs usable for this application a custom designed PCB was connected to the Beaglebone, see Figure 5. The board converts the inputs and outputs to the voltage and current levels required. It also includes a CAN transceiver that enables the two CAN bus channels available on the Beaglebone. The CAN bus allowed the autonomous system to communicate with the ECU and motor controllers of the car. The

Beaglebone is connected to the autonomous network through a wired Ethernet connection. The Beaglebone run Arch Linux and the microservices needed to control the input and outputs.

Specifications of Beaglebone I/O-Cape:

- 10 Digital outputs (controls GND, max 500mA)
- 3 Digital input 24V (max 31V)
- 1 Analog out 0-10V (max 20mA)
- 3 PWM (controls GND, max 500mA)
- 2 CAN-bus channels
- 3 Analog input 0-10V
- 2 Analog input 0-5V

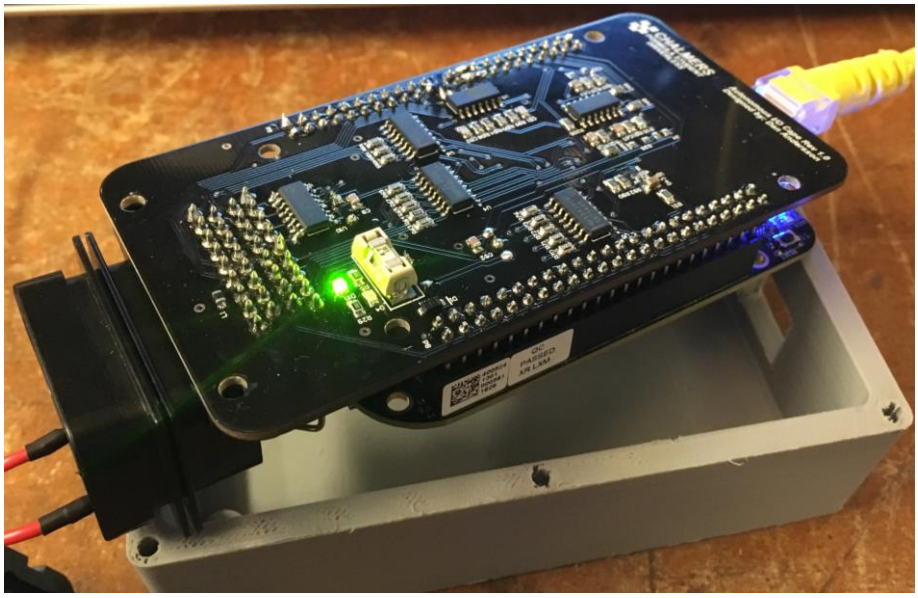

Figure 5, Beaglebone (lower PCB) with a custom designed input and output cape (top PCB).

4.3.3 Sensors for perception

The camera is a ZED stereo camera from Stereo Labs, see Figure 6, and has a USB 3.0 interface [34].

Figure 6, The stereo camera used for detecting the cones.

The lidar is a Velodyne 16 from Velodyne, see Figure 7. It has its own interface box that's communicates through Ethernet [35].

The IMU/GPS system is an Ellipse 2N from SBG systems, see Figure 8, which uses sensor fusion to estimate a more accurate position. The interface can be either CAN-bus or USB, but for this project serial-over- USB is used [36].

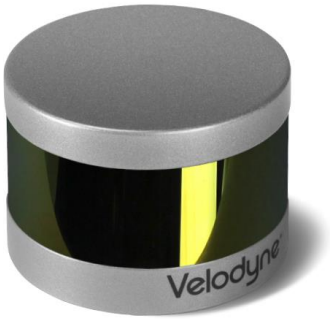

Figure 7, The LiDAR used for detecting cones.

Figure 8, The IMU/GPS system used in the car.

4.4 Deployment method

This section presents the designed deployment method. The Docker-based deployment strategy is investigated and compared to docker-compose and docker swarm + docker stack. A project specific deployment strategy is designed considering hardware related physical constraints and Formula student Germany competition rules.

4.4.1 Docker-compose

Docker is a fast-growing software containerization platform. Docker platform provides a series of tools and technology to facilitate software containerization development. Docker-compose is an efficient tool to define, configure and run a multi-container application in a single file. It makes it possible to run a complex application involving multiple microservices by one docker-compose.yml file. Furthermore, the docker-compose file helps define multi-configuration for running images.

The technical way to run multiple containers is to add all microservices that are supposed to run in the docker-compose.yml file and run the command "docker-compose up".

Docker-compose provides great features to be used in the docker-compose.yml file. All features can be found in docker compose reference documentation online. Version three is the newest version of the compose file. However, the third version removes the function "group_add" [37], which is needed for modules using shared memory. Therefore, it was decided to use the second version in this project.

The snippet in Code 1 is an example of docker-compose.yml file, used for testing the velodyne16.

```
version: '2'

services:
    #proxy-velodyne16
    velodyne16:
        image: chalmersrevere/opendlv-device-lidar-vlp16-
multi:v0.0.2
        network_mode: "host"
        volumes:
            - .:/opt/opendlv.data
        working_dir: "/opt/opendlv.data"
        command: "opendlv-device-lidar-vlp16 --
vlp16_ip=0.0.0.0 --vlp16_port=2368 --cid=111 --verbose"
```

Code 1, Example of a docker-compos. yml file.

4.4.2 Docker swarm & Docker stack

Docker swarm provides the possibility to create a cluster of containers running on multiple machines. Docker version later than version 1.12.0 support swarm mode for natively managing a cluster of Docker Engines [38]. The highlight features include natively cluster management, decentralized design, and load balancing.

Docker stack enables a group of interrelated services to be orchestrated together. In other words, a group of services can be defined to be allocated and scaled on different computing nodes using one single file.

These two docker technique are beneficial to distributed system with multiple machines on fast software deployment. Compared with docker-compose, the docker swarm and docker stack provide the possibility to deploy multiple services to a group of docker engines instead of one docker engine.

Docker stack enables user to define which docker engine to run the specific container on. The example showed in Code 2 is a part of docker-stack.yml for docker stack. In this example, the container will be run on the manager node.

```
services:
    communication-test-send:
        image: communication-test-send:latest
        command: "communication-test-send --cid=112"
        deploy:
          placement:
            constraints: [node.role == manager]
```

Code 2, Example of docker-stack.yml file for running docker stack on the manager node.

More investigation has been done to check whether docker swam and docker stack suits OpenDLV deployment requirements. In the swarm mode, overlay network is used to connect containers. However, the overlay network does not support multicast and it is still an open issue opened from 2015 [39]. Hence, it can't be used in OpenDLV since the communication between microservices rely on multicast.

To solve the problem that overlay network in swarm mode does not support multicast, third party plugins have been tried. One popular third-party plugin, Weave Net [40] Docker plugin, declared that it supports multicast in swarm mode. However, after contacting with Weave Net Docker plugin development team, it was found the plugin currently only works for x86_64. Furthermore, the plugin version for armhf platforms will not be released in the near future. Considering that there is one armhf computer in the CFSD system, it is clear that the Weave Net Docker Plugin cannot solve the problem of multicast in overlay network.

Though docker swarm and docker stack are suitable for software deployment to a group of docker engines, lack of multicasting in the overlay network currently makes them impossible to be integrated with OpenDLV. Therefore, docker-compose was chosen as the deployment tool for this work.

4.4.3 Deployment strategy on the CFSD project

In the target system, there is a x86_64 computer, an armhf computer (Beaglebone black), a Velodyne16 LiDAR, a ZED camera and, an IMU/GPS device, see Figure 9. The arm computer will start up directly when power is supplied to the unit. Once the autonomous system starts it will start the x86_64 computer using wake-on-lan. This is done to reduce power consumption of the system. The arm computer uses 100mA while the x86_64 uses 1A on idle. The x86_64 computer acts as a router since it has the possibility to connect to a WAN using WiFi and bridge it to the Ethernet port. A service computer will be able to connect to the system by connecting to the switch.

The x86_64 and arm computer will need to maintain its own docker-compose.yml file separately. In the docker-compose.yml, specify the exact microservices that will run on the computation node. Furthermore, each computer maintains its own local docker registry storing previous stable images. This make it possible to quick roll back if needed.

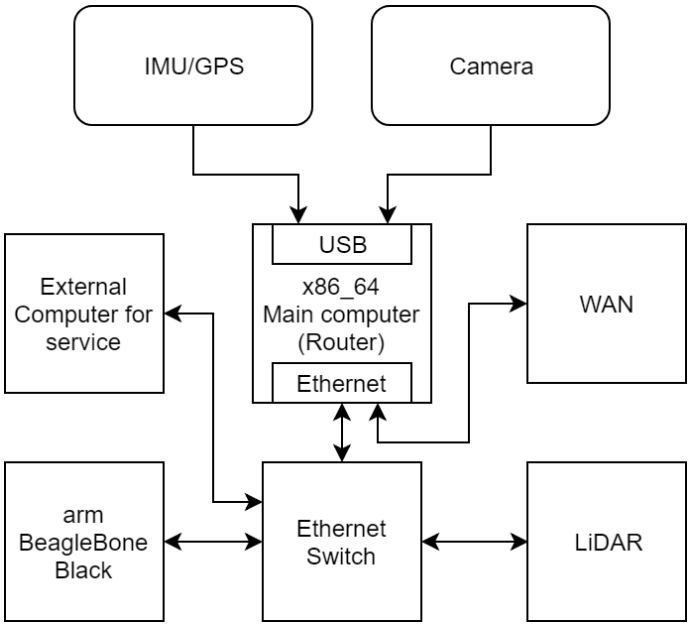

Figure 9, physical layout of the five devices and an Ethernet switch.

On the two computers, the operating system is the pre-installed OpenDLV.OS, which is an Arch Linux based operating system capable of running the containerized OpenDLV framework. The installation process is automated by available scripts in the OpenDLV.OS repository on the GitHub [41] which makes it possible to quickly recover from accident hardware failure and run containerized OpenDLV framework on a fresh computer.

The concrete software deployment flow is shown in Figure 10. The source code of each microservice is pushed to GitHub. In the CFSD'18 project, each microservice is housed in its individual repository. GitHub facilitates the tracebility by git commit hash, which is used to label docker images later. In each repository on GitHub, a travis.yml is maintained to automatically run tests, build and push images to Docker Hub. Although Docker Hub itself provides the basic functionality to hook a GitHub repository and automate the image building, Travic-CI is chosen here for more freedom in the automation process. The ChalmersFSD organization on Docker Hub owns all microservices of the CFSD'18 project. Each image is tagged with its microservice name and git commit hash (seven characters). The next step is to write and configure an appropriate docker-compose.yml file to define microservices that will run on the race car. Lastly, the command "docker-compose up" is used to start up all containers on one node.

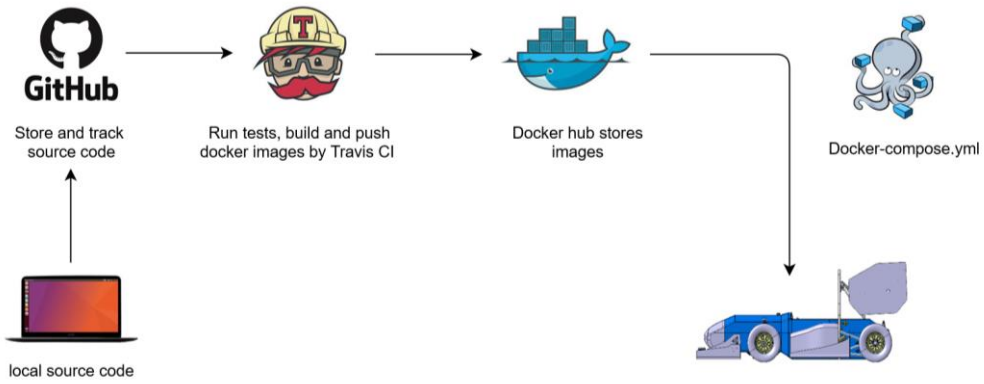

Figure 10 Software deployment process in CFSD'18.

4.4.4 Performance evaluation

OpenDLV [13] is a modern open source software environment to support the development and testing of self-driving vehicles. OpenDLV provides a large number of proxy microservices for commonly used devices on self-driving vehicles. The proxy microservices provide the function for hardware and software interfaces in OpenDLV.

As shown in Figure 9 in Section 4.4.3, the two computational notes in the system are a x86_64 and a Beaglebone Black. The performance of microservices running on the system is logged and evaluated as part of continuous experimentation, where the result is used as feedback to software design and implementation. To enable the possibility of continuous experimentation, highly automated scripts

and process have been designed and implemented. A bash script was design for logging the CPU load and then automatically plot it using a plot script, as listed in Appendix A. The plotting script was implemented in Python 3. When building the docker image the image was tagged with the git commit hash key. The tag was then used as label in the plot to keep track of the performance of a specific update.

5 Results and data analysis

This section presents the data collected from the microservice performance measurements using the CFSD'18 deploymeny strategy and evaluation scripts.

5.1 Performance evaluation of Beaglebone Black

The performance of the Beaglebone proxy modules was measured by logging the CPU load for 5min. To analyze how much the standard messages effect the CPU load three different tests were done. The first test was without any modifications to the proxy modules and the results are presented in Figure 11. The second test was done with preventing the microservices to read messages sent by itself, a feature introduced in libcluon to v0.0.90, shown in Figure 12. The third test was to arrange the microservices on different CIDs to avoid them from processing unnecessary information, as shown in Figure 13. After updating to libcluon v0.0.90 the CPU load was decreased by about 6%. After updating and adjusting the CID usage the CPU load was decreased by additional 28%, giving a total decrease of 34%.

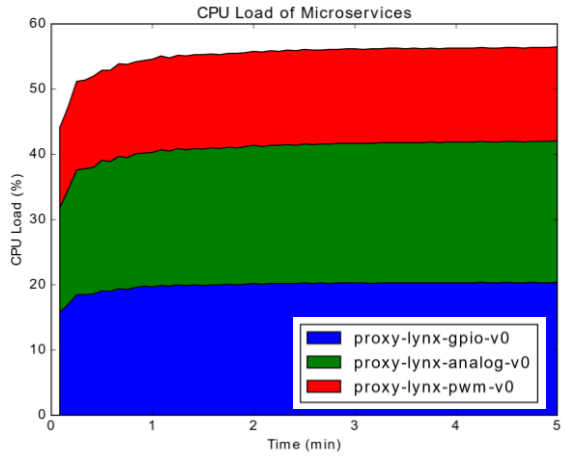

Figure 11, CPU load of the three proxy microservices running on Beaglebone. Before upgrading to libcluon v0.0.90.

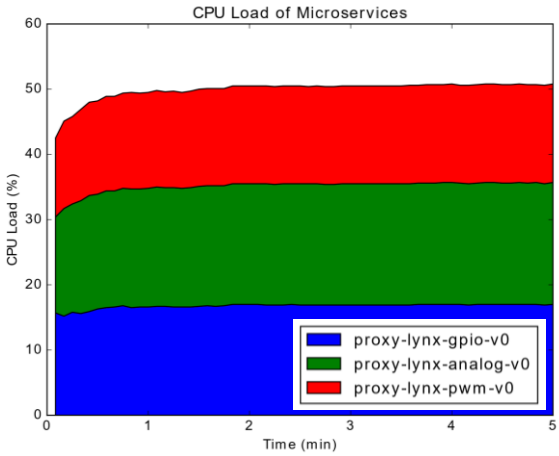

Figure 12, CPU load of the three proxy microservices running on Beaglebone. After upgrading to libcluon v0.0.90.

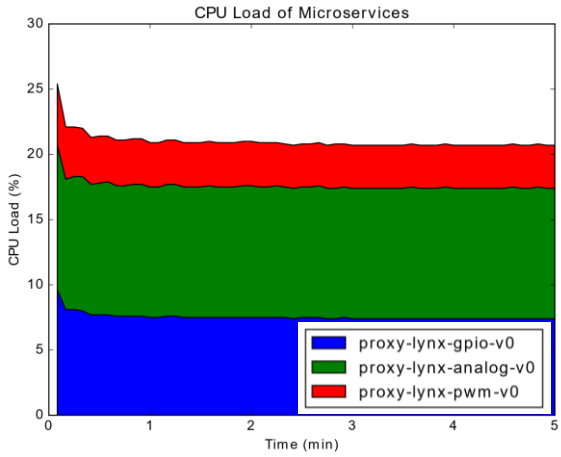

Figure 13, After upgrading to libcluon v0.0.90 and using appropiate conference ID. CPU load of the three proxy microservices running on Beaglebone.

5.2 Preformance evaluation of the x86_64 computer

The performance of proxy microservices on the x86_64 was measured in the same way as for the Beaglebone, by logging the CPU load for 5min. At the time of testing the available microservices was for the ZED camera, the velodyne 16 and the IMU.

In the plots generated the labels on the right are the image name used in the docker-compose.yml for the measurement. To be precise, the green label represents the microservice odsupercomponent. The three proxy microservices using libcluon are not depend on odsupercomponent. The odsupercomponent is part of the previous OpenDaVINCI-based version of OpenDLV, and was still included in the experiment to see how much CPU load it would cost if running OpenDaVINCI rather than libcluon. As displayed in Figure 14, the heaviest microservices in the experiment is proxy-camera, consuming around 26% of CPU load. The odsupercomponent consumed approximate 5% even though it was actually not needed for the system.

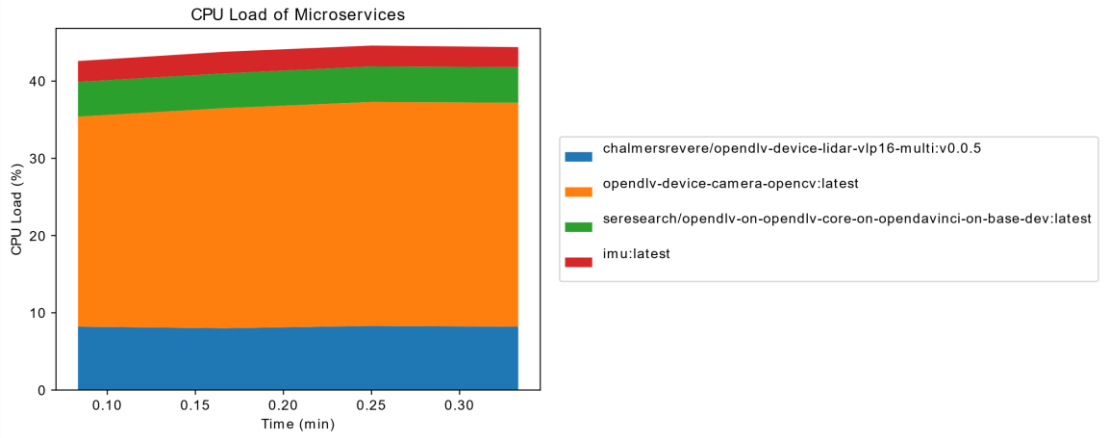

Figure 14, CPU Load bar chart of four microservices on the x86_64.

5.3 Performance evaluation of proxy-camera

Recently, the old version of the proxy microservices using OpenDaVINCI have been gradually refactored to use libcluon. An evaluation experiment was designed to compare the performance of different software structure and implementation. The performance evaluation was conducted for the rather heavy microservice proxy-camera. In the experiment, four versions of proxy-camera have been evaluated. The details of the four proxy-camera versions and the differences of implementation among them can be found in the Table 1. The seven characters (e.g. 6dda418) is the hash key of commitment on GitHub. The version master.0af4299 used in this experiment is using libswscale v3.4.2.

Table 1, Different versions of proxy-camera.

Proxy-Camera Version	**Implementation details**
old proxy	OpenDaVINCI+OpenDLV+SystemV IPC& Semaphores & OpenCV
ubuntu.6dda418	OpenCV 2.x + libcluon (POSIX shared memory)

v4l.2728dde	v4l2 low-level interface + libcluon + libswscale (supplied from Alpine Linux)
master.0af4299 (3.4.2)	v4l2 low-level interface + libcluon + libswscale (self-compiled)

CPU load was observed for 5 minutes. All the data were logged without verbose (i.e. no image display). Figure 15 below shows the performance of four different versions of proxy-camera. From the chart, the v4l.2728dde and master.0af4299 (3.4.2) are more than 10% lower than old proxy and ubuntu.6dda418. The exact procentage value of the CPU load, after 5 minutes, is shown in

Table 2. The implementation approach using "v4l2 low-level interface + libcluon + libswscale" shows promising performance, which decreases the CPU load more than 10% compared with old proxy (using OpenDaVINCI).

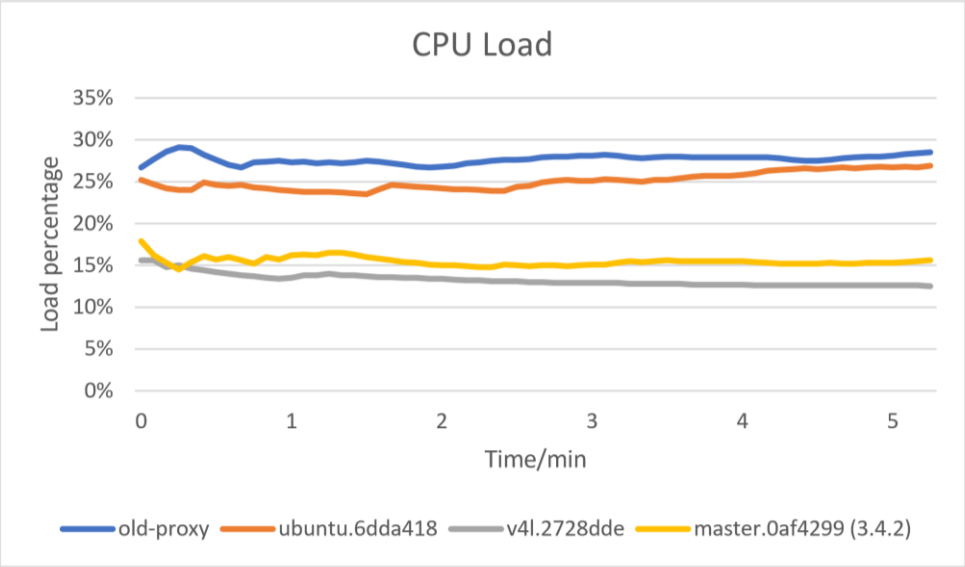

Figure 15, CPU Load line chart of 4 versions of proxy-camera.

Table 2, CPU load percentage of 4 versions of proxy-camera after 5 minutes.

Proxy-Camera Version	Load percentage after 5 mins
old proxy	28.5%
ubuntu.6dda418	26.9%
v4l.2728dde	12.5%
master.0af4299 (3.4.2)	15.6%

Furthermore, deeper investigation was performed on the version master.0af4299 (3.4.2). The memory consumption and CPU load were monitored for 30 minutes. From Figure 16 and Figure 17, it can be found that the memory consumption is stable at 0.7% and the CPU load is stable around 15.9%.

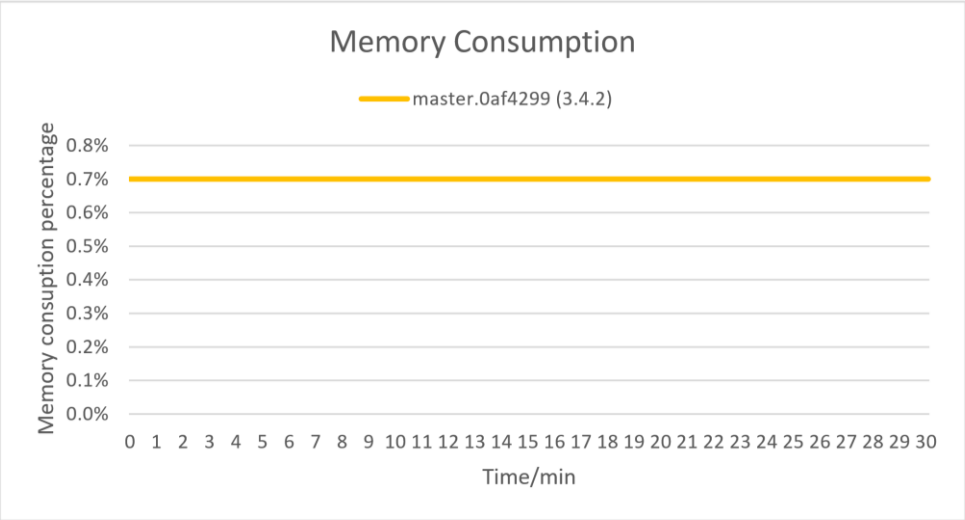

Figure 16, Memory Consumption of master.0af4299 (3.4.2) for 30 minutes.

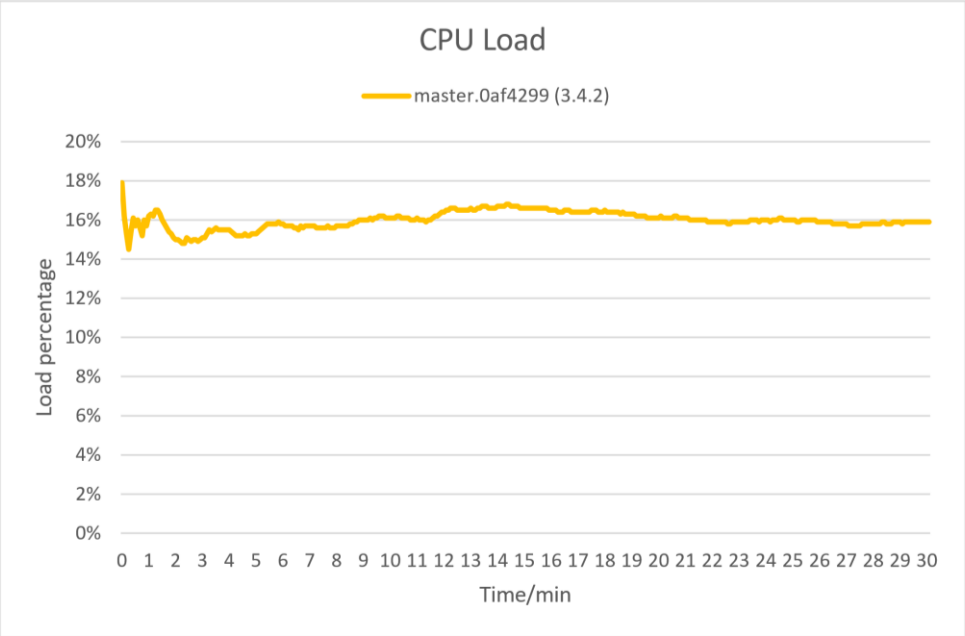

Figure 17, CPU load of master.0af4299 (3.4.2) for 30 minutes.

6 Discussion

In this section, we summarize our reflections about results and answer the three research questions from our work in the CFSD'18 project.

To answer the first research question about the efficient and traceble deployment strategy, the microservice software architecture and deployment workflow is discussed in 6.1 and 6.2. In 6.3, docker related orchestra technology is discussed to answer the second research question regarding fast deployment process. The third research question on live data is answered in 6.4 by explaining the automated log process and its benefits to software development.

6.1 Microservice architecture

libcluon [12], the header-only middleware for distributed systems for robotic applications, greatly contributes to the upgrade of OpenDLV ecosystem to transform from stacked layered architecture to purely microservices-based architecture.

Docker images of microservices using libcluon are much lighter than microservices based on OpenDaVINCI. The size of docker images using libcluon is usually about 10MB, while the images for OpenDaVINCI is in the range of 100MB to 1GB. The reason for the great size difference is that the layered architecture covers everything from the bottom layer to the microservice layer and requires all dependencies for all microservices to be present in the image, while libcluon generates a single executable per image that contains only the minimum set of dependencies

Using libcluon and microservice architecture have brought many benefits to the CFSD'18 project regarding development and deployment process. Firstly, developers have much more freedom to design the microservices they are working on since they are complety independent. For example, the team member of the CFSD'18 working on perception (lidar, camera, and IMU processing) appreciate the appearance of libcluon. When using OpenDLV based on OpenDaVINCI, they have suffered a lot from including new libraries to the OpenDLV layered structure. The layered structure is not flexible for new onboard developers to add new libraries they plan to use. However, libcluon offers the flexibility to newly involved developer to add new libraries and better control of the microservices. Secondly, libcluon improves the portability of created docker images. Using libcluon, each microservices can be built as separate images only containing the target microservices. Before, the layered structure based on OpenDaVINCI created a large image covering all microservices together. Furthermore, the ability to build images independently for each microservice enhances traceability in the development and deployment process. One simple approach to realize traceability is to tag docker images with git commit hash key. The high traceability of code and images will greatly save time and effort spent in testing, deployment, updating and other many related activities.

6.2 Software deployment workflow in the CFSD'18

The software deployment strategy in the CFSD'18 combines several platforms to automate the deployment process, see Figure 10 in Section 4.4.3. GitHub, Travis, Docker Hub and docker related container technology provide the possibility to automate the deployment process with high tracebility and robustness of the system. The process not only helps to reduce a lot of manual work but also reduce risks and uncertainties when deploying to target machines.

The use of OpenDLV.OS increases the robustness of containerized OpenDLV framework. The installation of OpenDLV.OS is automated with available settings to be customized. After installation, the computer is ready to run all containers in the OpenDLV ecosystem. This shows the possibility to quickly turn a fresh computer to an active working embedded system.

One of the most important reasons to use docker in projects in the Revere lab including the CFSD'18 is that one docker image can encapsulate everything needed for the piece of software. This excellent feature make it possible to ensure that the execution environment of related code is exactly the same, reducing potential problems caused by execution environment. As a result, developers can better focus on the code implementation itself and the following steps like testing and deployment can benefit from the encapsulation feature as well.

The containerized technology Docker has been widely used in several projects supported by Revere lab. The good experience of using Docker ecosystem from Revere lab encouraged the CFSD'18 project to persist on using Docker.

6.3 Containerized software deployment process

Docker swarm and docker stack are great tool for running containers on a group of machines with traceability. In the CFSD'18 project, there are a x86_64 and a armhf, which is a rather simple system. However, the truck and dolly combination in Revere lab has up to fifteen computers. Using docker swarm and docker stack will be much more efficient than only using docker-compose for the truck with dolly combination. The key point is that only one docker-compose.yml is needed to deploy multi services to specific nodes if using docker swarm + docker stack. Otherwise, each docker engine needs to maintain one docker-compose.yml file and run separately.

However, further research done to explore docker orchestra approach indicates limitation of docker swarm + docker stack. As explained in Section 4.4.2, docker swarm + docker stack are not suitable for OpenDLV ecosystem by the writing time. The limitation comes from lack of support for multicast in the overlay network created by docker swarm. The possible third-party plug-in WeaveNet to support multicast in the docker swarm only has the release for x86_64 platforms. Instead, docker-compose has been chosen as the tool for deployment even though it is not the best for a system containing multiple target machines. More precisely, each target machine owns its own docker-compose.yml and has to be

triggered separately. This leads to docker-compose introduces limitation to fully automation of the deployment process.

6.4 Automatic test and evaluation

An automated logging script was designed to answer the third research question; How can live data be visualized and recorded for further offline analysis? The script automatically logs the CPU load, plots charts and stores raw data in a folder labeled with date and time. The script was useful when continuously evaluating the live performances microservces running on the two computers.

When evaluating the performance of microservices on the Beaglebone it was found that the number of standard messages sent has high impact on the CPU load even if it's not further processed after the message handler. This is most likely because the module extract all messages to check whether they are relevant before discarding or processing them. This problem can be avoided by selecting different CIDs for different microservices, hence prevent them to process irrelevant messages.

When code is frequently updated, for example, several times a day, traceability of original code and related images becomes an important factor to ensure the correctness and efficiency of testing. Proper approach of improving traceability can also reduce the risks of misleading communication between developers and testers. Furthermore, traceability of evaluation make it possible to clearly show the result to the whole team. Otherwise, it is easy for team members to feel confused on which version of the microservices have been tested and measured.

Git is a good tool to facilitate version control of source code. The git commit hash key (the short version, seven characters) can be directly used as the record of code version. From experience, it shows a good practice to use git commit hash key as part of tag when building docker images, especially for frequent testing and comparison between different versions.

However, there are several constraints to realize fully automated testing for hardware proxy microservices. One concern is that the correctness of proxy-camera needs to be manually tested first, including checking right image, right color, display size and other visible parameters. Another constraint is that the camera and the computer used for testing sometimes are remote from developers. Then at least one tester is needed to be present in the lab where the camera and computer is to manually set up physically connection between hardware. In other words, the testing cannot be directly triggered by developers when code has been updated unless the hardware are always in the active state (power on).

7 Threats to validity

In this section, the four aspect of validity threats concluded by Runeson and Höst [42] are discussed for the experiments conduct in this thesis.

The validity of a study represent the degree of truth that other researchers can trust and rely on. It presents that the study was performed in an unbiased environment under some conditions. Alternatively, the study result should be able to be reproduced if other researchers do the same experiment using the same settings. Potential factors that might affect the experiment results in this study will also be mentioned and explained to better control the experiment.

7.1 Construct validity

Construct validity refers to the modification between how the thesis actually conduct and the approach designed and stated in the report. The hardware and software used in the thesis are exactly the experiment materials specified in Section 4.3. To trace the microservices performance experiment, the git hash of the code generating the image were recorded. In reality, other research groups might use different hardware, for example, different cameras and computers. This can be a thread to conclude the similar performance result.

7.2 Internal validity

Internal validity concerns whether potential relations between different factors exist. In the thesis, when conducting performance test, the unrelated processes running on the x86_64 computer and Beaglebone Black was reduced to a minimum. The purpose is to reduce the risk that other unrelated processes on the computation node affect the observation target microservices. The controlled environment adds internal validity.

However, all the performance tests are done in the indoors laboratory environment. These hardware and software will be running on the CFSD race car later. One threat to internal validity could be that during the tests a 450W power supply was used instead of the 250W power supply. Limiting the power might have an effect on the computational performance. Another related threat could be that the stability of the two power supplies can vary.

7.3 External validity

External validity is concerned with the possibility of generalization of the findings and interest to other people outside the CFSD'18 and Revere lab. The thesis summarizes the experience of using microservice architecture in a self-driving race car project. The features, usability, and performance of some microservices in OpenDLV ecosystem have been extended, tested and evaluated. All the experience in this thesis indicates the OpenDLV platform, libcluon, docker ecosystem are portable, efficient and adaptive choices for distributed embedded system, especially autonomous vehicle projects.

7.4 Reliability

Reliability refers to the dependence of the data collection and result analysis. All experiment hardware are static and software environment is controlled by docker encapsulated image and git commit hash key as reference. These practices improve the reliability of the collected data and analysis. Also, because the data logging and present process has been highly automated, the risk of manually introduced errors in data collection has been greatly lowered.

8 Conclusion & future work

The thesis is motivated by the CFSD'18 project, delivering a qualified self-driving formula race car. The goal was to investigate the software architecture in the system, design and evaluate the software development and deployment process for a highly modular and purely microservices-based distributed system on the example of self-driving race car project CFSD'18.

In order to answer the research questions, the work follows the designed methodology in Section 4.2 and conducts experiments in a traceable way.

Firstly, the deployment strategy of the CFSD'18 project has been designed by considering the hardware physically connection and Formula student Germany competition constraints. The workflow covers several platforms including GitHub, Travis, Docker Hub and docker-related technology. With OpenDLV.OS preinstalled on the computer, the only file that needs maintenance is the docker-compose.yml file. All docker images are labeled with git commit hash key to facilitate traceability, which makes it easy to rollback to previously images.

Secondly, docker-based strategies have been explored, tested and compared. Docker-compose is suitable for deploying multiple microservices at once on one docker machine. This is already chosen as part of deployment strategy in the Revere lab in previous work. While docker swarm + docker stack are designed for simple, scalable deployment on multiple docker machines. This should be a great choice for a complex distributed system. However, the limitation of applying docker swarm in deploying microservices using OpenDLV comes from the lack of support for multicast in the docker overlay network.

Thirdly, the microservices performance on the x86_64 and arm platform Beaglebone have been measured and evaluated. To reduce the risk of operation errors in the testing and make the testing efficient, automation scripts for data logging and analysis has been implemented. The performance evaluation shows the promising result of using OpenDLV and libcluon. All experiments have been performed in a traceable and controlled environment to improve the validity of the thesis.

All the result and experience concluded in the thesis will contribute to the cross-platform development at the vehicle laboratory Revere as a large practical use case. The thesis indicates the OpenDLV platform, libcluon, docker ecosystem are portable, efficient and adaptive choices for distributed embedded system, especially autonomous vehicle projects.

Further work can be explored in using docker swarm + docker stack as a deployment method when the multicast in the swarm network is available. Also, comparing the practical use of OpenDLV and libcluon in several different scaled projects in Revere lab might help to improve the adaptivity and functionality of OpenDLV ecosystem. Another opportunity for further work is to evaluate the messages handler in OpenDLV to reduce the CPU load.

References

[1] C. Berger, "An Open Continuous Deployment Infrastructure for a Self-driving Vehicle Ecosystem," in *IFIP Advances in Information and Communication Technology*, vol. 472, Cham, Springer, 2016, pp. 177-183.

[2] Revere, "OpenDLV," 25 October 2016. [Online]. Available: https://www.chalmers.se/en/researchinfrastructure/revere/Resources/Pages/OpenDLV.aspx.

[3] F. S. Germany, "About: Formula Student Germany," 11 05 2018. [Online]. Available: https://www.formulastudent.de/about/concept/.

[4] P. Marwedel, Embedded System Design, Switzerland: Springer International Publishing, 2018.

[5] E. A. Lee, "Computing Foundations and Practice for Cyber-Physical Systems: A Preliminary Report," University of California at Berkeley, Berkeley, 2007.

[6] M. Fowler, "Microservices," 25 March 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html.

[7] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, Cham, Springer, 2017, pp. 195-216.

[8] I. Nadareishvili, R. Mitra, M. McLarty and M. Amundsen, Microservice Architecture: Aligning Principles, Practices, and Culture, Sebastopol: O'Reilly Media, Inc., 2016, p. 6.

[9] J. Stubbs, W. Moreira and R. Dooley, "Distributed Systems of Microservices Using Docker and Serfnode," in *2015 7th International Workshop on Science Gateways*, Budapest, 2015.

[10] Docker, "What is Docker," Docker, 2018. [Online]. Available: https://www.docker.com/what-container. [Accessed 13 02 2018].

[11] C. Berger, "OpenDaVINCI," CSE Chalmers, [Online]. Available: http://opendavinci.cse.chalmers.se/www/. [Accessed 12 05 2018].

[12] C. Berger, "libcluon:Github.com," Christian Berger, [Online]. Available: https://github.com/chrberger/libcluon. [Accessed 12 05 2018].

[13] C. Berger, "OpenDLV: Github," Chalmers Revere, [Online]. Available: https://github.com/chalmers-revere/opendlv. [Accessed 12 05 2018].

[14] Carzaniga, Antonio; Fuggetta, Alfonso; Hall, Richard S.; Heimbigner, Dennis; van der Hoek, Andre; Wolf, Alexander L.; COLORADO STATE UNIV FORT COLLINS DEPT OF COMPUTER SCIENCE, "A Characterization Framework for Software Deployment Technologies," Defense Technical Information Center, Fort Belvoir, Virginia, United States, April 1998.

[15] R. S. Hall, D. Heimbigner and A. L. Wolf, "A cooperative approach to support software deployment using the Software Dock," in *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, 1999.

[16] N. Hochgeschwender, L. Gherardi and A. Shakhirmardanov, "A model-based approach to software deployment in robotics," in *Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo, 2013.

[17] "ROS," Open Source Robotics Foundation, [Online]. Available: http://www.ros.org/.

[18] K. Beck, J. Grenning, R. C. Martin, M. Beedle, J. Highsmith, S. Mellor, A. v. Bennekum, A. Hunt, K. Schwaber, A. Cockburn, R. Jeffries, J. Sutherland, W. Cunningham, J. Kern, D. Thomas, M. Fowler and B. Marick, "Manifesto for agile software development," 2001.

[19] Feitelson, Dror G.; E. Frachtenberg Facebook; K. L. Beck Facebook, "Development and Deployment at Facebook," *IEEE Internet Computing,* vol. 17, no. 4, pp. 8-17, 2013.

[20] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo, Italy, 2015.

[21] G. G. Claps, R. B. Svensson and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software Technology,* vol. 57, pp. 21-31, January 2015.

[22] J. Thönes, "Microservices," in *IEEE Software (Volume: 32, Issue: 1, Jan.-Feb. 2015)*, IEEE, 2015, pp. 113-116.

[23] X. Larrucea, I. Santamaria, R. Colomo-Palacios and C. Ebert, "Microservices," in *IEEE Software (Volume: 35, Issue: 3, May/June 2018)*, IEEE Software, 2018, pp. 96-100.

[24] M. Fowler, "Microservices," 24 04 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed 12 05 2018].

[25] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen and M. Mazzara, "From Monolithic to Microservices: An Experience Report from the Banking Domain," in *IEEE Software (Volume: 35, Issue: 3, May/June 2018)*, IEEE Software, 2018, pp. 50-55.

[26] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," in *IEEE Software (Volume: 35, Issue: 3, May/June 2018)*, IEEE Software, 2018, pp. 56-62.

[27] C. Andersson, "Docker," in *IEEE Software (Volume: 32, Issue: 3, May-June 2015)*, IEEE Software, 2015, pp. 102-105.

[28] C. Pahl and B. Lee, "Containers and Clusters for Edge Cloud Architectures -- A Technology Review," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*, IEEE, 2015, pp. 379-386.

[29] C. Berger, B. Nguyen and O. Benderius, "Containerized Development and Microservices for Self-Driving Vehicles: Experiences & Best Practices," in *IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017.

[30] R. Wieringa, "Design science methodology: principles and practice," in *Proceeding ICSE '10 Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, Cape Town, 2010.

[31] R. Wieringa, "Design science as nested problem solving," in *Proceeding DESRIST '09 Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, Philadelphia, 2009.

[32] AMD, ”AMD Ryzen 7 1700,” AMD, [Online]. Available: https://www.amd.com/en/products/cpu/amd-ryzen-7-1700. [Använd 12 05 2018].

[33] beaglebone.org, "Beaglebone," beaglebone.org, [Online]. Available: http://beagleboard.org/bone. [Accessed 12 05 2018].

[34] S. Labs, "Meet ZED," Stereo Labs, [Online]. Available: https://www.stereolabs.com/zed/. [Accessed 12 05 2018].

[35] Velodyne, "Velodyne 16," Velodyne, [Online]. Available: http://velodynelidar.com/vlp-16.html. [Accessed 12 05 2018].

[36] S. Systems, "Ellipse2-N: Miniature INS/GPS," SBG Systems, [Online]. Available: https://www.sbg-systems.com/products/ellipse-n-miniature-ins-gps. [Accessed 12 05 2018].

[37] Docker, ”Docker-Compose Versions,” Docker, [Online]. Available: https://docs.docker.com/compose/compose-file/compose-versioning/#version-1-to-2x. [Använd 29 05 2018].

[38] Docker, "Swarm mode overview," Docker, [Online]. Available: https://docs.docker.com/engine/swarm/. [Accessed 14 05 2018].

[39] Weaveworks, "Multicast in Overlay driver," github, [Online]. Available: https://github.com/docker/libnetwork/issues/552. [Accessed 14 05 2018].

[40] Weaveworks, "Integrating Docker via the Network Plugin (V2)," Weaveworks, 2018.

[41] C. Berger, ”OpenDLV.OS,” Github, 14 05 2018. [Online].

[42] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study," in *Empir Software Eng (2009)*, Springer, 2008, pp. 131-164.

Appendix A

CPU logging script

```
#Example of usage:
#sh cpuload.sh gpio
#for logging data for module 'gpip'
cmd=$1
delay=5
loops=$[300/$delay]
folder="$(date "+%Y%m%d-%H%M")"
mkdir "$folder" 2>/dev/null
image="$(docker ps --filter "name=$cmd" --format '{{.Image}}')"
container="$(docker ps --filter "name=$cmd" --format '{{.Names}}')"
pid="$(docker inspect -f '{{.State.Pid}}' $container)"

file="$folder/$cmd.cpu"
file2="$folder/$cmd.mem"

date | tee -a $file $file2;

ps -p $pid -o cmd= | tee -a $file $file2;
echo "image=$image" | tee -a $file $file2
echo "Start of logging" | tee -a $file $file2
for i in `seq 1 $loops`; do
         ps -p $pid -o %cpu= | tee -a $file;
         ps -p $pid -o %mem= | tee -a $file2;
         sleep $delay;
done

echo "End of logging" | tee -a $file $file2

python3 plot.py 1 $folder
python3 plot.py 2 $folder
```