University of Verona

DEPARTMENT OF COMPUTER SCIENCE
Master Degree in Computer Science and Engineering

Model-based design of at-the-edge heterogeneous Robotics Applications based on ROS

Candidate:
Simone Girardi

Thesis advisor:
Prof. Nicola Bombieri

Research supervisor:
Dott. Stefano Aldegheri

Thesis submitted in 2020

Abstract

Developing distributed robotics applications on embedded devices, we have to deal with the heterogeneity of both the software applications and the hardware platforms where these applications run. At the state of the art there are some solutions that help us to develop robotics applications and deploy them on embedded boards. The problem is that none of these solutions allows us to be sufficiently accurate to guarantee the functionality by satisfying non-functional constraints (i.e. performance, accuracy, energy efficiency reliability, data privacy and security, etc.) of the entire system, especially if we want to increase its complexity. To solve this problem, we must to take account of the necessary resources to run the applications and the constraints imposed by the limits of the devices.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

While robots originated in large-scale mass manufacturing working very efficiently behind fences (i.e., not interacting with human operators), they are now spreading to more and more application areas. As these robots operate at human scale (e.g., physical Human-Robot-Interaction in factories, warehouses, and at homes) and perform safety-critical tasks (like driving or surgery), the correctness requirements are stringent and include, beside functional accuracy, also non-functional constraints such as real-time, safety, data privacy, scalability, and energy efficiency [1].

To address all these requirements, robotic SW applications have to be configured to be run across heterogeneous Cloud-Server-Edge computing platforms [2]. Such a design and verification task is very challenging, as it requires high-expertise and time-consuming (re)configuration steps to compile and test the application code for different HW/SW configurations.

In this work, we present a toolchain that relies on a container-based packaging mechanism whereby SW components are abstracted from the environment in which they actually run and orchestrated across heterogeneous HW architectures. We extended the Docker and Kubernetes (KubeEdge) environments, which are well-established in the context of Cloud-native SW applications, to be used in the context of ROS-based robotic applications on Cloud-Edge platforms.

Recently, some open source and a number of proprietary model-based platforms have been proposed to ease the design of robotic systems (e.g., the EU funded Robmosys project, NVIDIA Isaac, Amazon AWS Robomaker). What is missing in these solutions is a toolchain able to address the complexity, the scalability, and the heterogeneity of the HW/SW domains by

considering non-functional constraints. Taking into account in seamless way functional and non-functional constraints is a key feature to design reliable, and safe robotic applications from the specifications to the system deployment. This thesis give two main contributions:

1. Customization, debugging and verification of the ORB-SLAM2 algorithm for efficient execution at the edge.

2. A design flow based on containers and KubeEdge, that integrates the ORB-SLAM2 application into a more complex application for mobile robots.

1.1 Thesis outline

This thesis is organised in two main workflows: *Verification* and *Methodology*. Chapter 2 is an overview over the research contribution in the field of deep learning, edge computing and the other tools discussed in the next chapter of this thesis. The first workflow is described in chapter 3, where a complete inspection and verification was made on the ORB SLAM algorithm to guarantee a deterministic behaviour in a sequential context. The second workflow described in chapter 4 proposes a solution to the problem of integrating heterogeneous robotic applications. With the support of edge computing platforms like KubeEdge, some experiments was made to deploy automatically our integrated system from the host (cloud) to the edge. Finally in chapter 5 the experimental results are showed and discussed.

Chapter 2

Background

2.1 Deep Learning

2.1.1 Motivations

Deep learning has recently been highly successful in machine learning across a variety of application domains, including computer vision, natural language processing, and big data analysis, among others. For example, deep learning methods have consistently outperformed traditional methods for object recognition and detection in the IMAGENET Large Scale Visual Recognition Challenge (ISLVRC) since 2012 [3]. However, deep learning's high accuracy comes at the expense of high computational and memory requirements for both the training and inference phases of deep learning. Training a deep learning model is space and computationally expensive due to millions of parameters that need to be iteratively refined over multiple time epochs. Inference is computationally expensive due to the potentially high dimensionality of the input data (e.g., a high-resolution image) and millions of computations that need to be performed on the input data.

2.1.2 Definitions

As described in [4], the modern term "deep learning" goes beyond the neuroscientific perspective engineering applications on the current breed of machine learning models. It appeals to a more general principle of learning *multiple levels of composition*, which can be applied in machine learning frameworks that are not necessarily neurally inspired. Deep learning is a subset of AI and machine learning and differs in that they can automatically

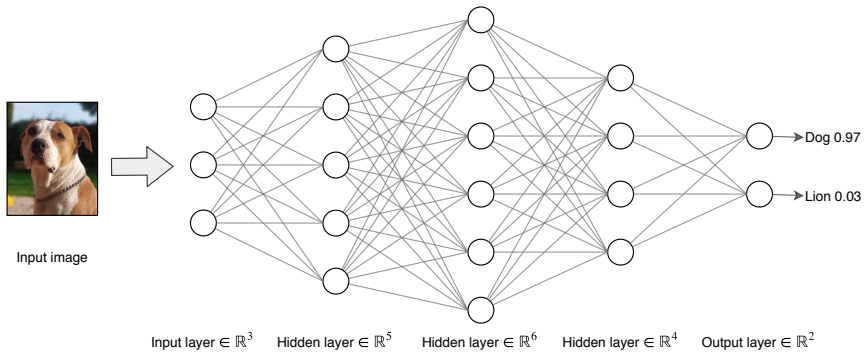


Figure 2.1

DNN example with image classification

learn representations from data such as images, video or text, to be used for classification without introducing hand-coded rules or human domain knowledge. Their highly flexible architectures can learn directly from raw data and can increase their predictive accuracy when provided with more data. A deep learning prediction algorithm, consists of a number of layers, as shown in Fig. 2.1.

In deep learning *inference*, the input data pass through the node's layers in sequence, and each layer performs matrix multiplications on the data. The output of a layer is usually the input to the subsequent layer. After data are processed by the final (fully connected) layer, the output is either a feature or a classification value. When the model contains many layers in sequence, the neural network is known as a deep neural network (DNN). When the matrix multiplications include convolutional filter operations, the model is named convolutional neural networks (CNNs), which is common for image and video processing contexts. There are also DNNs designed especially for time series prediction; these are called recurrent neural networks (RNNs), which have loops in their layer connections to keep state and enable predictions on sequential inputs.

In deep learning *training*, the computation proceeds in reverse order. Given the ground-truth training labels, multiple passes are made over the layers to optimize the parameters of each layer of matrix multiplications, starting from the final layer and ending with the first layer. The algorithm used is typically stochastic gradient descent (SGD). In each pass, a randomly small subset of N input data ("mini-batch") from the training data set, is selected and used to update the gradients in the direction that minimizes

the training loss (where the training loss is defined as the difference between the predictions and the ground truth). One pass through the entire training data set is called a training epoch [5].

There are some considerations to take into account: the first is that there are a large number of parameters in the matrix multiplications, resulting in many computations being performed and thus the latency issues that we see on end devices. The second is that there are many choices (hyper-parameters) on how to design the DNN models (e.g., the number of parameters per layer, and the number of layers), which makes the model design more of an art than a science. Different DNN design decisions result in tradeoffs between system metrics; for example, a DNN with higher accuracy likely requires more memory to store all the model parameters and will have higher latency because of all the matrix multiplications being performed. On the other hand, a DNN model with fewer parameters will likely execute more quickly and use less computational resources and energy, but it may not have sufficient accuracy to meet the application's requirements.

2.1.3 Performance Measurement

How can we evaluate the performance of a neural network? Deep learning can be used to perform both supervised learning and unsupervised learning. The metrics of success depend on the particular application domain where deep learning is being applied. For example, in object detection, the accuracy may be measured by the mean average precision (mAP) [3], which measures how well the predicted object location overlaps with the ground-truth location, averaged across multiple categories of objects. In machine translation, the accuracy can be measured by the bilingual evaluation understudy score metric [6], which compares a candidate translation with several ground-truth reference translations. Other general system performance metrics not specific to the application include throughput, latency, and energy. These metrics are summarized in Table 2.1. Designing a good DNN model or selecting the right DNN model for a given application is challenging due to the large number of hyper-parameter decisions.

Machine learning research typically focuses on accuracy metrics, and their system performance results are often reported from powerful server testbeds equipped with GPUs. For example, Huang et al. [7] compared the speed and accuracy tradeoffs when running on a high-end gaming GPU

Metric	Unit
Latency	s
Energy	mW, J
Concurrent Requests Served	#
Network Bandwidth	Mbps
Accuracy	Application Specific

Table 2.1
Neural Network Performance Metrics

(NVIDIA Titan X). The YOLO DNN model [8], which is designed for real-time performance, provides timing measurements on the same server GPU. Specifically targeting mobile devices, Lu et al. [9] provided the measurements for a number of popular DNN models on mobile CPUs and GPUs (Nvidia TK1 and TX1). Ran et al.[10] further explored the accuracy-latency tradeoffs on mobile devices by measuring how reducing the dimensionality of the input size reduces the overall accuracy and latency. DNN models designed specifically for mobile devices, such as MobileNets [11], report system performance in terms of a number of multiply–add operations, which could be used to estimate latency characteristics and other metrics on different mobile hardware, based on the processing capabilities of the hardware. Once the system performance is understood, the application developer can choose the right model.

2.1.4 Frameworks

Several open-source software libraries are publicly available for deep learning inference and training on end devices and edge servers. Google's TensorFlow [12], released in 2015, is an interface for expressing machine learning algorithms and an implementation for executing such algorithms on heterogeneous distributed systems. Tensorflow's computation workflow is designed as a directed graph and utilizes a placement algorithm to distribute computation tasks based on the estimated or measured execution time and communication time [13]. The placement algorithm uses a greedy approach that places a computation task on the node that is expected to complete the computation the soonest. Tensorflow can run on edge devices, such as Raspberry Pi and smartphones. TensorFlow Lite was proposed in

the late 2017 [14], which is an optimized version of Tensorflow for mobile and embedded devices, with mobile GPU support added in early 2019. Tensorflow Lite only provides on-device inference abilities, not training, and achieves low latency by compressing a pre-trained DNN model. Caffe [15] is another deep learning framework, originally developed by Jia, with the current version, Caffe2, maintained by Facebook. It seeks to provide an easy and straightforward way for deep learning with a focus on mobile devices, including smartphones and Raspberry Pis. PyTorch [16] is another deep learning platform developed by Facebook, with its main goal differing from Caffe2 in which it focuses on the integration of research prototypes to production development. Actually Facebook is working on the merge of Caffe2 and PyTorch frameworks. GPUs are an important key element in efficient DNN inference and training. NVIDIA provides GPU software libraries to make use of NVIDIA GPUs, such as CUDA [17] for general GPU processing and cuDNN [18] which is targeted toward deep learning. While such libraries are useful for training DNN models on a desktop server, cuDNN and CUDA are not widely available on current mobile devices such as smartphones. To utilize smartphone GPUs, Android developers can currently make use of Tensorflow Lite, which provides experimental GPU capabilities. To experiment with edge devices other than smartphones, researchers can turn to edge-specific development kits, such as the NVIDIA Jetson TX2 development kit for experimenting with edge computing, with NVIDIA-provided SDKs used to program the devices.

2.1.5 Challenges

Because of the required competences and effort to choose the best neural network and the best parameters that better fit the applications of our interest, there has also been much recent studies in automated machine learning, which uses artificial intelligence to choose which DNN model to run and tune the hyper-parameters. For example, Tan et al. [19] and Taylor et al. [20] proposed using reinforcement learning and traditional machine learning, respectively, to choose the right hyper-parameters for mobile devices, which is useful in edge scenarios. As described in [2] many challenges remain in deploying deep learning on the edge, not only on end devices but also on the edge servers and on a combination of end devices, edge servers, and the cloud. For example parameters like latency, energy consumption and mi-

gration still the main challenges in the field of deep learning applied to the edge computing.

2.2 Edge Computing

Today, an IoT solution has to cover a much broader scope of requirements. We see that in most cases, organizations opt for a combination of cloud and edge computing for complex IoT solutions. Cloud computing typically comes into play when organizations require storage and computing power to execute certain applications and processes, and to visualize telemetry data from anywhere. Edge computing, on the other hand, is the right choice in cases with low latency, local autonomous actions, reduced back-end traffic, and when confidential data is involved.

2.2.1 Motivations

As written in [21] there are at least three reasons to consider the adoption of edge computing. The first is that putting all the computing tasks on the cloud has been proved to be an efficient way for data processing since the computing power on the cloud outclasses the capability of the things at the edge. However, compared to the fast developing data processing speed, the bandwidth of the network has come to a standstill. With the growing quantity of data generated at the edge, speed of data transportation is becoming the bottleneck for the cloud-based computing paradigm. The second reason is that almost all kinds of electrical devices will become part of IoT, and they will play the role of data producers as well as consumers, such as air quality sensors, LED bars, street lights and even an Internet-connected microwave oven. It is safe to infer that the number of things at the edge of the network will develop to more than billions in a few years. Thus, raw data produced by them will be enormous, making conventional cloud computing not efficient enough to handle all these data. This means most of the data produced by IoT will never be transmitted to the cloud, instead it will be consumed at the edge of the network. Fig. 2.2 shows the conventional cloud computing structure.

However, this structure is not sufficient for IoT. First, data quantity at the edge is too large, which will lead to huge unnecessary bandwidth and computing resource usage. Second, the privacy protection requirement will

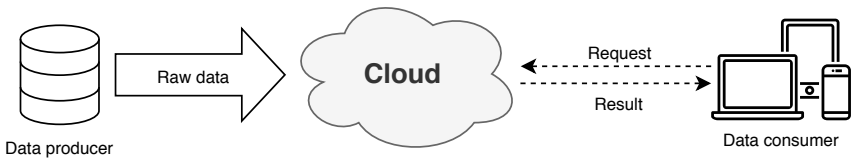


Figure 2.2
Cloud computing paradigm

pose an obstacle for cloud computing in IoT. Lastly, most of the end nodes in IoT are energy constrained things, and the wireless communication module is usually very energy hungry, so offloading some computing tasks to the edge could be more energy efficient. Another valid consideration to take into account is that in the cloud computing paradigm, the end devices at the edge usually play as data consumer, for example, watching a YouTube video on your smart phone. However, people are also producing data nowadays from their mobile devices. The change from data consumer to data producer/consumer requires more function placement at the edge.

2.2.2 Definitions

Edge computing refers to the enabling technologies allowing computation to be performed at the edge of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoT services. The rationale of edge computing is that computing should happen at the proximity of data sources. From a certain point of view, edge computing could interchangeable with fog computing [22], but edge computing focus more toward the things side, while the first focus more on the infrastructure side. Fig. 2.3 illustrates the two-way computing streams in edge computing. In the edge computing paradigm, the things not only are data consumers, but also play as data producers. At the edge, the things can not only request service and content from the cloud but also perform the computing tasks from the cloud. Edge can perform computing offloading, data storage, caching and processing, as well as distribute request and delivery service from cloud to user. With those jobs in the network, the edge itself needs to be well designed to meet the requirement efficiently in service such as reliability, security, and privacy protection.

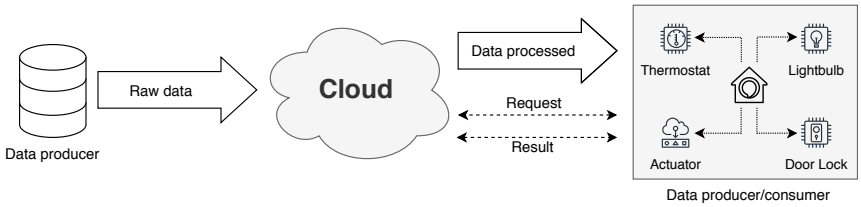


Figure 2.3
Edge computing paradigm

2.2.3 Performance Metrics

In edge computing, we have multiple layers with different computation capability. Workload allocation becomes a big issue. We need to decide which layer to handle the workload or how many tasks to assign at each part. There are multiple allocation strategies to complete a workload, for instances, evenly distribute the workload on each layer or complete as much as possible on each layer To choose an optimal allocation strategy, there several optimization parameter to consider. These metrics are very similar to those described in 2.1, including latency, bandwidth and energy. How to measure these performance is strongly dependent by the environment, resources and application case.

2.2.4 Challenges

In cloud computing, users program their code and deploy them on the cloud. Usually, the program is written in one programming language and compiled for a certain target platform, since the program only runs in the cloud. However, in the edge computing, computation is offloaded from the cloud, and the edge nodes are most likely heterogeneous platforms. In this case, the runtime of these nodes differ from each other, and the programmer faces huge difficulties to write an application that may be deployed in the edge computing paradigm. Another important challenge is related to the *naming*. In edge computing, one important assumption is that the number of things is tremendously large. At the top of the edge nodes, there are a lot of applications running, and each application has its own structure about how the service is provided. Similar to all computer systems, the naming scheme in edge computing is very important for development, addressing, things identification, and data communication. However, an ef-

ficient naming mechanism for the edge computing paradigm has not been built and standardized yet. Edge practitioners usually needs to learn various communication and network protocols in order to communicate with the heterogeneous things in their system. In terms of *service management* at the edge of the network, there are four fundamental features that should be supported to guarantee a reliable system, including differentiation, extensibility, isolation, and reliability. To protect the data *security* and usage *privacy* at the edge of the network, several challenges remain open. First is the awareness of privacy and security to the community: ip camera, health monitor, or even some Wi-Fi enabled toys could easily be connected by others if not protected properly. Second is the ownership of the data collected from things at edge. Just as what happened with mobile applications, the data of end user collected by things will be stored and analyzed at the service provider side. Third is the missing of efficient tools to protect data privacy and security at the edge of the network. The highly dynamic environment at the edge of the network also makes the network become vulnerable or unprotected and more tools are still missing to handle diverse data attributes for edge computing.

2.3 Robotics Tools and Platforms

This section present the main robotics platforms analysed and explored during the development of this thesis.

2.3.1 ROS

The Robot Operating System (ROS) [23], is a open-source framework based on the component-based software engineering paradigm that provides the middleware for inter-process communication. Initially developed by the Stanford Artificial Intelligence Laboratory its development continued at Willow Garage, a robotics research institute, and now it's maintained and improved under the action of the ORSF foundation [23]. As a meta-operating system, ROS offers features such as hardware abstraction, low-level device control, implementation of commonly used functionalities, message communication between processes and package management. It uses an asynchronous publish/subscribing mechanism made possible by message standardisation and encapsulation that make the external interface of every

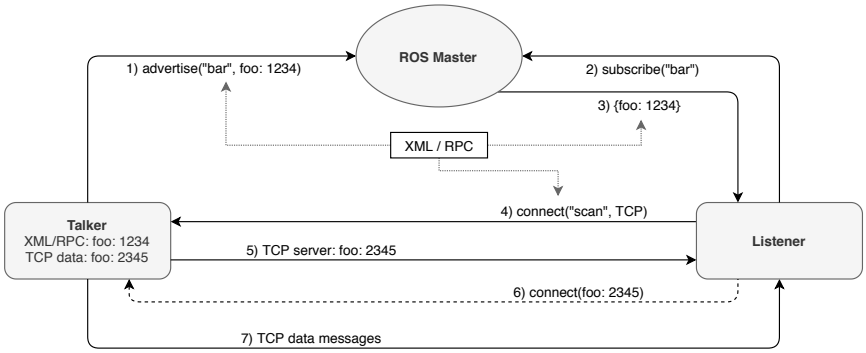


Figure 2.4

Example of nodes and topic communication

node as general as possible, allowing quick nodes exchange and, thus, great architectural flexibility. Each independent block, called *node*, executes a particular task of a process and can communicates with other nodes through *topics*. This allows to create complex architectures by aggregating many simpler entities and simplifies the use of different tasks or different methods for the same task. Additionally to the message-passing system, the core ROS component, called *roscore*, maintains a global execution time for the nodes to achieve synchronisation. Each node executes separately with its own internal clock driven by the set execution rate. At every message sent/received, containing the internal time information, the core component updates the global time by following the execution status provided by these messages. For more clarity about how ROS communication works an example is provided in figure 2.4. When a publishing node will announce to the master that it is publishing over a topic and the subscribing node will say to the master that it want to listen to a topic. Then if the publisher and the subscriber are on the same topic, the master will transfer data in order for the two other node to communicate directly via TCP/IP or UDP.

The reason to choose ROS for the development of robotics applications resides mainly in its high modularity. Furthermore it is a well known standard in the scientific community of the robotics researches and developers.

2.3.2 NVIDIA Isaac SDK

The Isaac SDK is the main software toolkit provided by NVIDIA and allows developers to brought new opportunities for developing robotics so-

lutions as well as researching topics in this field. As described in [24] it's comprised of the following:

- Isaac Robot Engine: A framework which allows you to easily write modular applications and deploy them on your robots.
- Isaac GEMs: A collection of robotics algorithms from planning to perception, most of them GPU-accelerated.
- Applications: Various example applications from basic samples which show specific features to applications that facilitate complicated robotics use cases.
- Isaac Sim for Navigation: A powerful virtual robotics laboratory and a high-fidelity 3D world simulator that accelerates research, design, and development by reducing cost and risk.

The main reasons we would consider to use this platform reside in its high modularity and high performance solutions that it is able to offer on NVIDIA Jetson platforms. Others important reasons to adopt this solution are related the **C API** that allows the communication with Isaac apps from languages other than C++, and the **ROS bridge** that offers the capability to interact with ROS nodes applications.

2.3.3 CoppeliaSim

From the creators of V-Rep [25], CoppeliaSim [26] seems to be the definitive solution for development of robotics applications. The robot simulator CoppeliaSim, with integrated development environment, is based on a distributed control architecture: each object/model can be individually controlled via an embedded script, a plugin, ROS nodes, BlueZero nodes, remote API clients, or a custom solution. This makes CoppeliaSim very versatile and ideal for multi-robot applications. Controllers can be written in C/C++, Python, Java, Lua, Matlab, Octave or Urbi. CoppeliaSim can be used as a stand-alone application or can easily be embedded into a main client application: its small footprint and elaborate API makes CoppeliaSim an ideal candidate to embed into higher-level applications. An integrated Lua script interpreter makes CoppeliaSim an extremely versatile application, leaving the freedom to the user to combine the low/high-level functionalities to obtain new high-level functionalities.

2.4 Docker

While container technologies have been around for a long time, they have become more widely known with the rise of the Docker container platform. Docker was the first container system that made containers easily portable across different machines. It simplifies the process of packaging up not only the application but also all its libraries and other dependencies, even the whole OS file system, into a simple, portable package that can be used to provision the application to any other machine running Docker. When you run an application packaged with Docker, it sees the exact filesystem contents that you have bundled with it. It sees the same files whether it is running on your development machine or a production machine, even if it the production server is running a completely different Linux OS. The application will not see anything from the server it is running on, so it does not matter if the server has a completely different set of installed libraries compared to your development machine. This is similar to creating a VM image by installing an operating system into a VM, installing the app inside it, and then distributing the whole VM image around and running it. Docker achieves the same effect, but instead of using Virtual Machines (VMs) to achieve app isolation, it uses Linux container technologies to provide (almost) the same level of isolation that VMs do. Instead of using big monolithic VM images, it uses container images, which are usually smaller.

2.4.1 Understanding Docker Concepts

Docker is a platform for packaging, distributing, and running applications. It allows you to package your application together with its whole environment. This can be either a few libraries that the app requires or even all the files that are usually available on the filesystem of an installed operating system. Docker makes it possible to transfer this package to a central repository from which it can then be transferred to any computer running Docker and executed there. Three main concepts in Docker comprise this scenario:

- **Images**: A Docker-based container image is something you package your application and its environment into. It contains the filesystem that will be available to the application and other metadata, such as the path to the executable that should be executed when the image is

run.

- **Registries**: A Docker Registry is a repository that stores your Docker images and facilitates easy sharing of those images between different people and computers. When you build your image, you can either run it on the computer you have built it on, or you can push (upload) the image to a registry and then pull (download) it on another computer and run it there. Certain registries are public, allowing anyone to pull images from it, while others are private, only accessible to certain people or machines.

- **Containers**: A Docker-based container is a regular Linux container created from a Docker-based container image. A running container is a process of the host on which Docker is running, but it is completely isolated from both the host and all other processes running on it. The process is also resource-constrained, meaning that it can only access and use the amount of resources (CPU, RAM, and so on) that are allocated to it.

Figure 2.5 shows all three concepts and how they relate to each other. The developer first builds an image and then pushes it to a registry. The image is thus available to anyone who can access the registry. They can then pull the image to any other machine running Docker and run the image. Docker creates an isolated container based on the image and runs the binary executable specified as part of the image. Docker images are composed of layers. Different images can contain the exact same layers because every Docker image is built on top of another image and two different images can both use the same parent image as their base. This speeds up the distribution of images across the network, because layers that have already been transferred as part of the first image do not need to be transferred again when transferring the other image. But layers do not only make distribution more efficient, they also help reduce the storage footprint of images. Each layer is only stored once. Two containers created from two images based on the same base layers can therefore read the same files, but if one of them writes over those files, the other one does not see those changes. Therefore, even if they share files, they are still isolated from each other. This works because container image layers are read-only. When a container is run, a new writeable layer is created on top of the layers in the image. When the

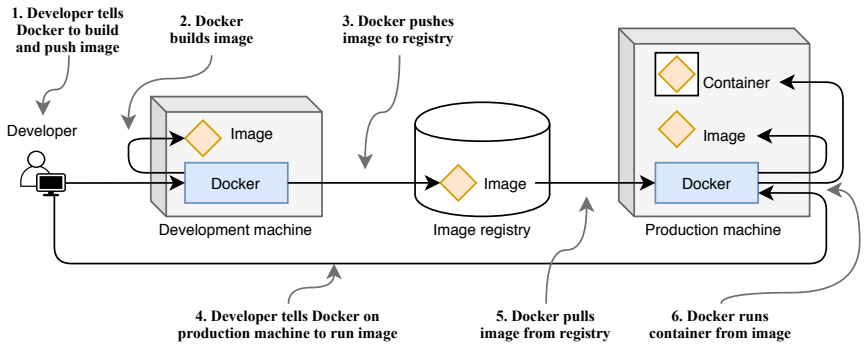


Figure 2.5

Docker images, registries, and containers.

process in the container writes to a file located in one of the underlying layers, a copy of the whole file is created in the top-most layer and the process writes to the copy.

2.4.2 Portability Limitations of Container Images

In theory, a container image can be run on any Linux machine running Docker, but one small caveat exists, one related to the fact that all containers running on a host use the host's Linux kernel. If a containerized application requires a specific kernel version, it may not work on every machine. If a machine runs a different version of the Linux kernel or does not have the same kernel modules available, the app cannot run on it.

While containers are much more lightweight compared to VMs, they impose certain constraints on the apps running inside them. VMs have no such constraints, because each VM runs its own kernel. And it is not only about the kernel. It should also be clear that a containerized app built for a specific hardware architecture can only run on other machines that have the same architecture. It is not possible containerize an application built for the x86 architecture and expect it to run on an ARM-based machine because it also runs Docker. You still need a Virtual Machine for that. It is very important to understand that Docker itself does not provide process isolation. The actual isolation of containers is done at the Linux kernel level using kernel features such as Linux Namespaces and cgroups. Docker only makes it easy to use those features.

2.5 Kubernetes

As the number of deployable application components in your system grows, it becomes harder to manage them all. Google was probably the first company that realized the need of a much better way to deploy and manage their software components and their infrastructure to scale globally. It is one of only a few companies in the world that runs hundreds of thousands of servers and has had to deal with managing deployments on such a massive scale. This has forced them to develop solutions for making the development and deployment of thousands of software components manageable and cost-efficient.

2.5.1 What is Kubernetes

Kubernetes is a software system that allows you to easily deploy and manage containerized applications on top of it. It relies on the features of Linux containers to run heterogeneous applications without having to know any internal details of these applications and without having to manually deploy these applications on each host. Because these apps run in containers, they do not affect other apps running on the same server, which is critical when you run applications for completely different organizations on the same hardware. Kubernetes enables you to run your software applications on thousands of computer nodes as if all those nodes were a single, enormous computer. Deploying applications through Kubernetes is always the same, whether your cluster contains only a couple of nodes or thousands of them. The size of the cluster makes no difference at all. Additional cluster nodes simply represent an additional amount of resources available to deployed apps.

Figure 2.6 shows the simplest possible view of a Kubernetes system. The system is composed of a master node and any number of worker nodes. When the developer submits a list of apps to the master, Kubernetes deploys them to the cluster of worker nodes. What node a component lands on does not (and should not) matter neither to the developer nor to the system administrator.

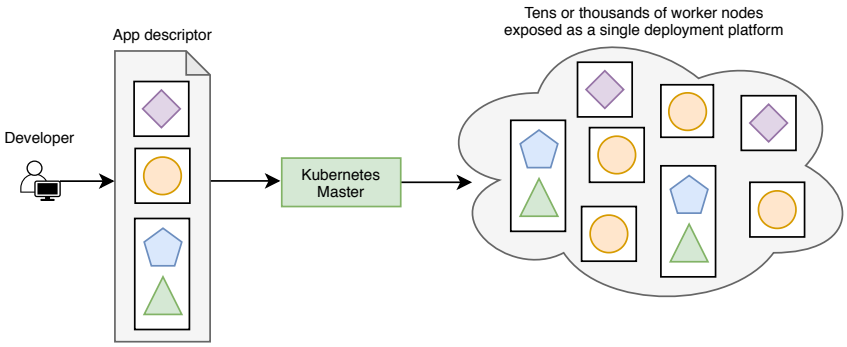


Figure 2.6

Kubernetes exposes the whole datacenter as a single deployment platform.

2.5.2 Architecture of a Kubernetes cluster

A Kubernetes cluster is composed of many nodes, which can be split into two types.

- The *master* node, which host the *Kubernetes Control Plane* that controls and manages the whole Kubernetes system.
- Worker *nodes* that run the actual applications you deploy.

The *Control Plane* is what controls the cluster and makes it function. It consists of multiple components that can run on a single master node or be split across multiple nodes and replicated to ensure high availability. These components are:

- The Kubernetes *API Server*, which you and the other Control Plane components communicate with.
- The *Scheduler*, which schedules your apps (assigns a worker node to each deployable component of your application).
- The *Controller Manager*, which performs cluster-level functions, such as replicating components, keeping track of worker nodes, handling node failures, and so on.
- *etcd*, a reliable distributed data store that persistently stores the cluster configuration.

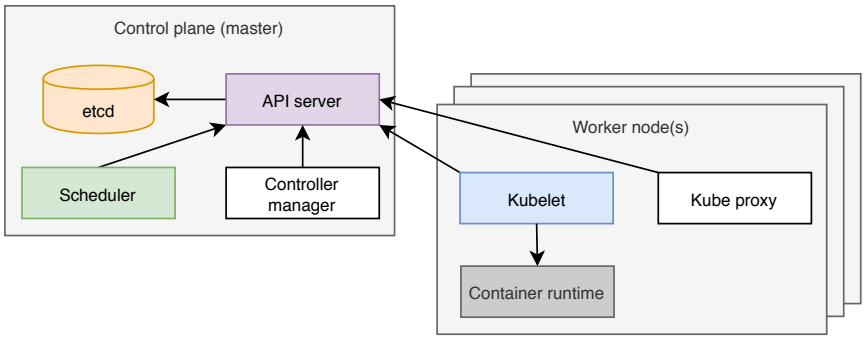


Figure 2.7

The components that make up a Kubernetes cluster.

The components of the Control Plane hold and control the state of the cluster, but they do not run your applications. This is done by the (worker) nodes.

The worker nodes are the machines that run your containerized applications. The task of running, monitoring, and providing services to your applications is done by the following components:

- Docker, rkt, or another container runtime, which runs your containers.
- The Kubelet, which talks to the API server and manages containers on its node.
- The Kubernetes Service Proxy (kube-proxy), which load-balances network traffic between application components.

Figure 2.7 shows the components running on these two sets of nodes.

2.5.3 Running an application in Kubernetes

To run an application in Kubernetes, you first need to package it up into one or more container images, push those images to an image registry, and then post a description of your app to the Kubernetes API server.

The description includes information such as the container image or images that contain your application components, how those components are related to each other, and which ones need to be run co-located (together on the same node) and which do not. For each component, you can also specify how many copies (or replicas) you want to run. Additionally, the description also includes which of those components provide a service to either internal

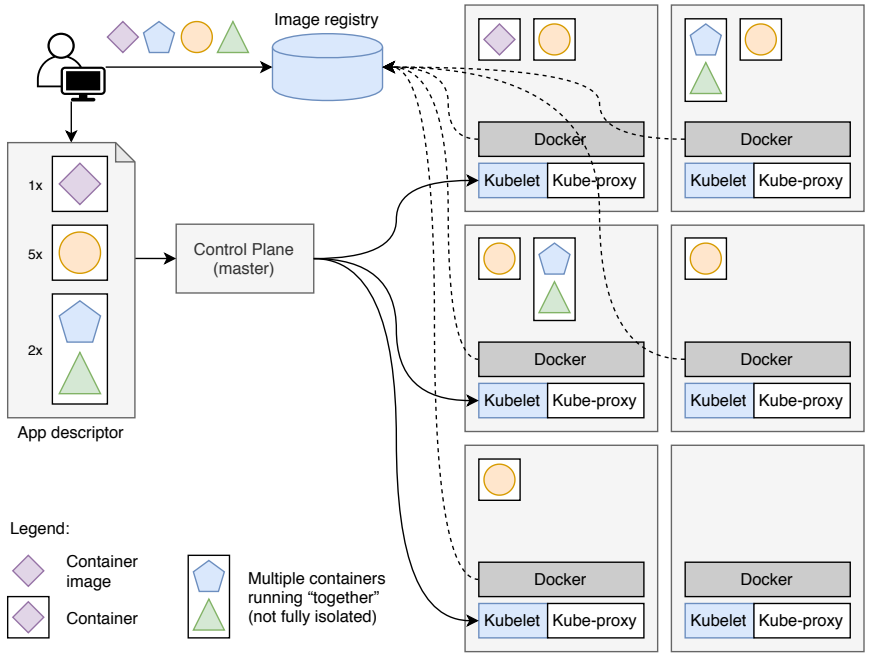


Figure 2.8

A basic overview of the Kubernetes architecture and an application running on top of it.

or external clients and should be exposed through a single IP address and made discoverable to the other components. When the API server processes your app's description, the Scheduler schedules the specified groups of containers onto the available worker nodes based on computational resources required by each group and the unallocated resources on each node at that moment. The Kubelet on those nodes then instructs the Container Runtime (Docker, for example) to pull the required container images and run the containers.

Figure 2.8 gives a better understanding of how applications are deployed in Kubernetes. The app descriptor lists four containers, grouped into three sets (these sets are called pods). The first two pods each contain only a single container, whereas the last one contains two containers. This means both containers need to run co-located and shouldn't be isolated from each other. Next to each pod, you also see a number representing the number of replicas of each pod that need to run in parallel. After submitting the descriptor to Kubernetes, it will schedule the specified number of replicas of each pod to the available worker nodes. The Kubelets on the nodes will

then tell Docker to pull the container images from the image registry and run the containers.

2.5.4 Monitoring of Performance

Properly setting resource requests and limits within a Pod, is crucial for getting the most out of your Kubernetes cluster. If requests are set too high, your cluster nodes will be underutilized and you will be throwing money away. If you set them too low, your apps will be CPU-starved or even killed by the Out Of Memory (OOM) Killer. How do you find the sweet spot for requests and limits? You find it by monitoring the actual resource usage of your containers under the expected load levels. Once the application is exposed to the public, you should keep monitoring it and adjust the resource requests and limits if required. Since Kubernetes v1.18 monitoring is split into two pipelines:

- A **core metrics pipeline** consisting of Kubelet, a resource estimator, a slimmed-down Heapster called metrics-server, and the API server serving the master metrics API. These metrics are used by core system components, such as scheduling logic (e.g. scheduler and horizontal pod autoscaling based on system metrics) and simple out-of-the-box UI components (e.g. kubectl top).

- A **monitoring pipeline** used for collecting various metrics from the system and exposing them to end-users, as well as to the Horizontal Pod Autoscaler (for custom metrics) and Infrastore via adapters. Users can choose from many monitoring system vendors, or run none at all. In open-source, Kubernetes will not ship with a monitoring pipeline, but third-party options will be easy to install. Pipelines will typically consist of a per-node agent and a cluster-level aggregator.

The architecture of the core metrics pipeline is illustrated in figure 2.9. The interaction with these API can be performed through the command line tool or into graphical web user interfaces web dashboard provided by Google. The dashboard allows you to list all the Pods, Replication Controllers, Services, and other objects deployed in your cluster, as well as to create, modify, and delete them. Figure 2.10 shows the dashboard that we are talking about.

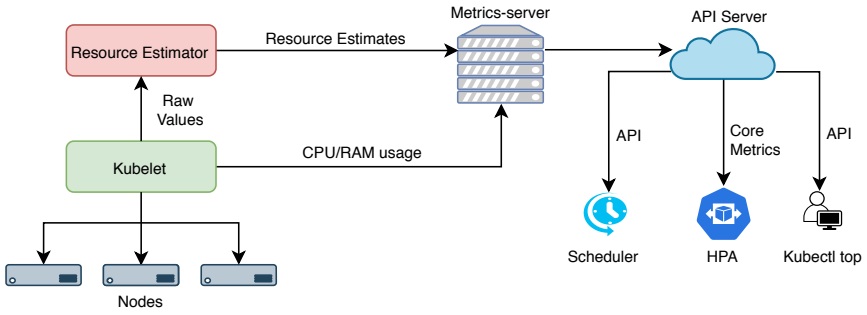


Figure 2.9

Core Metrics pipeline.

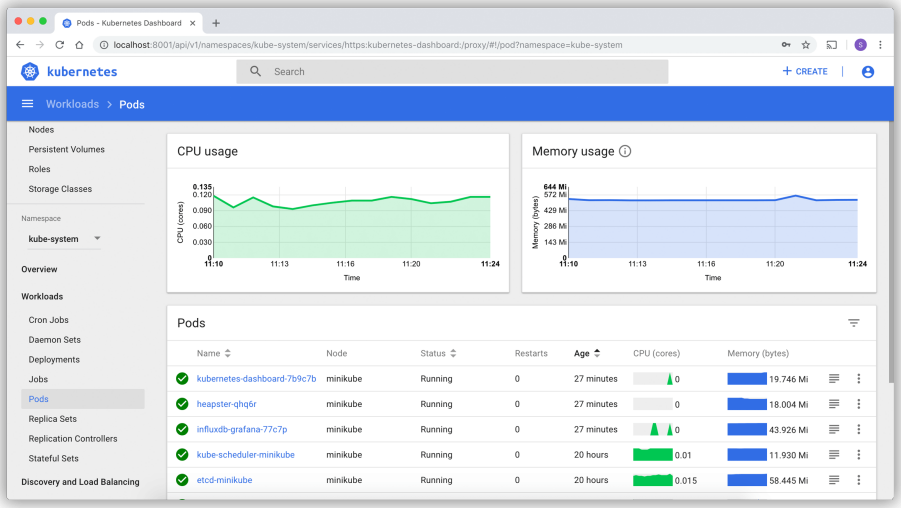


Figure 2.10

Kubernetes Dashboard.

2.6 KubeEdge

KubeEdge is an open source system extending native containerized application orchestration and device management to hosts at the Edge. It is built upon Kubernetes and provides core infrastructure support for networking, application deployment and metadata synchronization between cloud and edge. It also supports MQTT and allows developers to author custom logic and enable resource constrained device communication at the Edge.

2.6.1 Motivation

With KubeEdge it is easy to get and deploy existing complicated machine learning, image recognition, event processing and other high level applications to the Edge. With business logic running at the Edge, much larger volumes of data can be secured and processed locally where the data is produced. With data processed at the Edge, the responsiveness is increased dramatically and data privacy is protected. The advantages of KubeEdge include mainly:

- **Edge Computing**: with business logic running at the Edge, much larger volumes of data can be secured and processed locally where the data is produced. This reduces the network bandwidth requirements and consumption between Edge and Cloud. This increases responsiveness, decreases costs, and protects customer's data privacy.
- **Simplified development**: Developers can write regular http or mqtt based applications, containerize these, and run them anywhere - either at the Edge or in the Cloud - whichever is more appropriate.
- **Kubernetes-native support**: With KubeEdge, users can orchestrate apps, manage devices and monitor app and device status on Edge nodes just like a traditional Kubernetes cluster in the Cloud.
- **Abundant applications**: It is easy to get and deploy existing complicated machine learning, image recognition, event processing and other high level applications to the Edge.

In the era of Internet of Thing (IoT), billions of sensors and actuators are deployed worldwide. To manage the IoT devices and process data with cloud computing resource, Cloud providers such as Amazon AWS and Microsoft

Azure are developing the IoT platform and are providing services or solutions on their respective cloud environments.

Most IoT platforms employ a Publisher/Subscriber (Pub/Sub) brokers such as MQTT [27] or AMQP to provide the communication channel between IoT devices and Cloud services, like Azure IoT Hub. Pub/Sub protocol such as MQTT is suitable for asynchronous communication between edge devices and cloud services. However it does not support synchronous RPC based communication, for the increased need as more and more computation tasks [28][29] move to the edges and tightly integrate with services in the cloud. One common scenario for RPC based communication is the cloud native micro-service based application. With micro-service architecture, an application is designed into multiple micro services, each of which is deployed and managed independently. These micro services communicate each other usually through REST/HTTP protocol. When some of the micro services run on the edge nodes and need to communicate with those in the cloud, it requires the one network address space for both edge nodes and server instances in the cloud. This is where current edge computing solutions break down, partly due to the asynchronous Pub/Sub based MQTT protocol [30]. KubeEdge leverages Kubernetes container platform to provide RPC based communication channel between edge and cloud, the runtime execution environment of containers and Serverless functions, as well as a mechanism to sync and store metadata to support self-management of an application running on the edge in an offline scenario.

Furthermore, since KubeEdge is built upon Kubernetes all the tools available in Kubernetes are also available in KubeEdge. Therefore we can benefit of all the monitoring platforms such as the dashboard and metric-server shown in the previous section.

2.6.2 Kubeedge Architecture

As shown in Figure 2.11, KubeEdge is a multi-tenant infrastructure platform for edge computing. The platform includes the following components, excluding Kubernetes.

- **Edged**: an agent that runs on edge nodes and manages containerized applications.

- **EdgeHub**: a web socket client responsible for interacting with Cloud

Service for edge computing (like Edge Controller as in the KubeEdge Architecture). This includes syncing cloud-side resource updates to the edge and reporting edge-side host and device status changes to the cloud.

- **CloudHub**: A web socket server responsible for watching changes at the cloud side, caching and sending messages to EdgeHub.
- **EdgeController**: an extended kubernetes controller which manages edge nodes and pods metadata so that the data can be targeted to a specific edge node.
- **EventBus**: an MQTT client to interact with MQTT servers (mosquitto), offering publish and subscribe capabilities to other components.
- **DeviceTwin**: responsible for storing device status and syncing device status to the cloud. It also provides query interfaces for applications.
- **MetaManager**: the message processor between edged and edgehub. It is also responsible for storing/retrieving metadata to/from a lightweight database (SQLite).

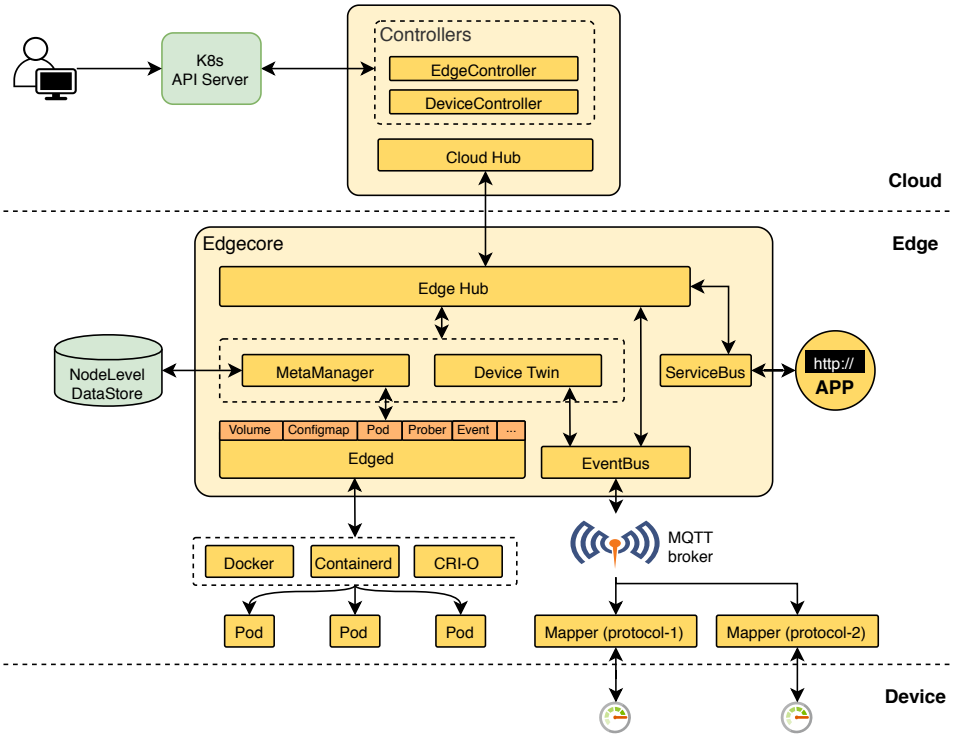


Figure 2.11

KubeEdge Architecture.

Chapter 3

ORB-SLAM at the edge

This chapter will focus on description, verification and customization of ORB-SLAM2 [31] for execution on a low-power heterogeneous embedded device (i.e. NVIDIA Jetson TX2). Primary it will describe how ORB-SLAM2 works. Secondly it will show an accurate inspection necessary to find some potential bugs that cause an unexpected behaviour of the algorithm.

3.1 System description

As described in [32], ORB-SLAM2 is a Simultaneous Localization And Mapping system that can works with data coming from monocular, stereo, and RGB-D cameras. A system of this kind has the purpose to allow both map reconstruction and navigation in the most common environment without the support of a GPS. The system consists of the following three main blocks (see figure 3.1):

Tracking and localization This block is in charge of computing visual features, localizing the robot in the environment, and, in case of significant discrepancies between an already saved map and the input stream, communicating updated map information to the mapping block. The frames per second (FPS) that can be computed by the whole system strongly depends on the performance of this block.

Mapping It updates the environment map by using the information (map changes) sent by the localization block. It is a computational time consuming block and its execution rate strictly depends on the agent speed. However, considering the actual agent speed of the KITTI

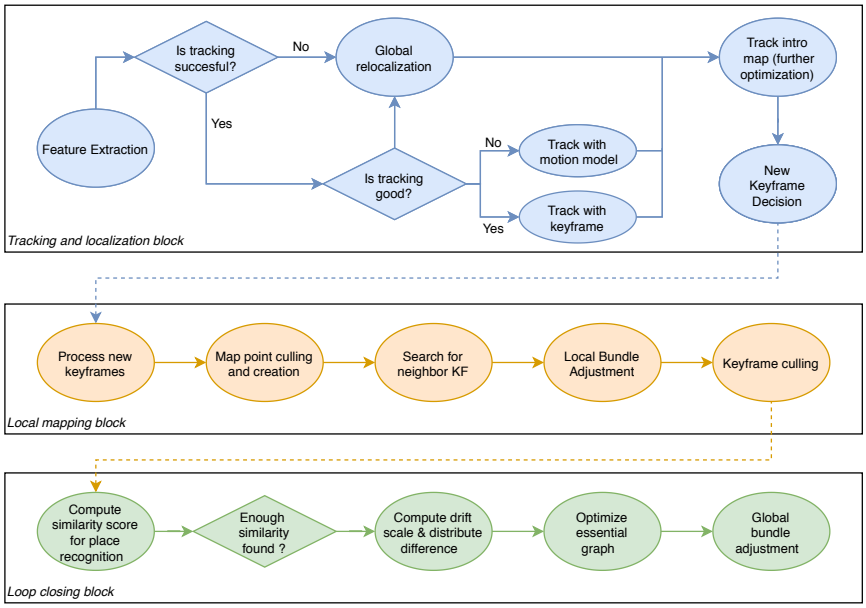


Figure 3.1

Main blocks of the ORB-SLAM2 algorithm.

datasets analysed in this work [33], it does not represent a system bottleneck.

Loop closing It aims at adjusting the scale drift error accumulated during the input analysis. When a loop in the robot pathway is detected, this block updates the mapped information through a high latency heavy computation, during which the first two blocks are suspended. This can lead the robot to loose tracking and localization information and, as a consequence, the robot to get temporary lost. The computation efficiency of this block (running on-demand) is crucial for the quality of the final results.

The system is organized on three parallel threads, one per block. The use of parallel threads allows for obtaining real-time processing on an Intel Core i7 desktop PC with 16GB RAM [31]. The original open source version of ORB-SLAM2 provides two level of parallelism. The first level is given by the three main algorithm blocks (see Fig. 3.1), which are implemented to be run as parallel PThreads on shared-memory multi-core CPUs. The second level is given by the automatic parallel implementation (i.e., through OpenMP directives) of the *bundle adjustment* sub-block, which is part both

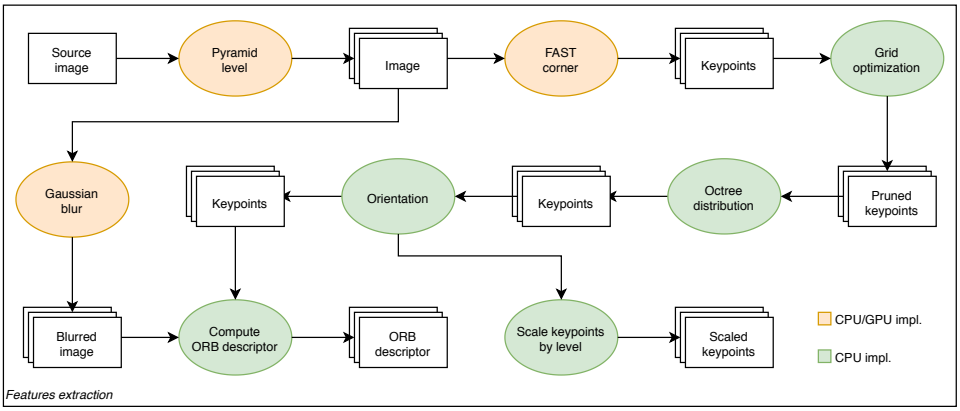


Figure 3.2

DAG of the feature extraction block and the corresponding sub-block implementations (GPU vs. CPU).

of the local mapping and loop closing blocks. This allows the parallel computation of such a long latency task on multi-core CPUs. To fully exploit the heterogeneous nature of the target NVIDIA Jetson TX2 board, the work done in [32] added two further levels of parallelism. The first is given by the parallel implementation for GPU of a set of tracking sub-blocks (see Fig. 3.2). This is because the most important bottleneck that characterizes the processing rate in terms of supported FPS, was detected on the feature extraction block. The second is given by the implementation of a 8-stage pipeline of such sub-blocks in order to benefit of an overlapped computation.

3.2 Unexpected behaviour

After the porting and the optimizations of the ORB-SLAM2 algorithm on the NVIDIA Jetson TX2 to achieve real-time performance, the experimental results conducted in [32] shown a massive improvement of the execution times on some KITTI dataset sequences. Despite these performance improvements, the algorithm seems to exhibit some unexpected behaviours too frequently, especially with sequences that contain a higher space-temporal variation between one frame and the next one. After processing a frame the algorithm can be in three state: *Not Initialized*, *Track*, *Lost*. The problem is that it get into the Track Lost state (fig. 3.3a) after the initialization in a totally random way. Another Track Lost state is reached when less then 5 Keyframes are computed and it loses its position (see fig: 3.3b).

(a) Track lost state

(b) Track lost soon after initialization, system resetting.

Figure 3.3

LOST states of ORB-SLAM2 in sequence 03 of KITTY dataset.

Due to the randomicity with which the problem occurs, to identify the cause it's necessary isolate the problem as much as possible. For this purpose the first step has been sequentialize the execution between Tracking and Mapping blocks. Once the two blocks are running sequentially, we proceeded to inspect the code in more detail founding in such way some potential bugs. The main strategy we adopted to find the possible problems within the code, was looking at the difference among the produced results after two o more sequential execution. To achieve this goal we built a custom logger with different levels of debugging.

An example of the problems we found is located in the KeyFrame source code, where a `vector<pair<int, KeyFrame*>> vPairs` has been applied to the C++ standard `sort` function. The problem here is the sort procedure use the less operator compare the address of the pointers instead of the id of the KeyFrame. To solve this issue we override the less operator in order the achieve the expected behaviour (see listing 1). Another problem encountered during the code inspection that could introduce a non deterministic execution, is related to the members variables initialization. Unlike

Listing 1 Overriding the less operator.

```
namespace std {
    bool less<ORB_SLAM2::KeyFrame*>::operator()(
    const ORB_SLAM2::KeyFrame* k1,
    const ORB_SLAM2::KeyFrame* k2) const {
        return k1->mnId < k2->mnId;
    }
    bool operator<(const std::pair<int, ORB_SLAM2::KeyFrame*> lhs, const
    ↪  std::pair<int, ORB_SLAM2::KeyFrame*> rhs) {
        return lhs.first < rhs.first || ((lhs.first == rhs.first) &&
        ↪  lhs.second->mnId < rhs.second->mnId);
    }
}
```

some programming languages, C/C++ does not initialize most variables to a given value (such as zero) automatically. When a variable is assigned a memory location by the compiler, the default value of that variable is whatever (garbage) value happens to already be in that memory location! A variable that has not been given a known value (usually through initialization or assignment) is called an uninitialized variable.

Once these kind of problems have been resolved, the algorithm now is running with a deterministic behaviour. If it get into Track Lost state it happen every time and vice-versa. Unfortunately, when we restore the concurrency between Tracking and Mapping blocks, the problem of unexpected behaviour rise again.

To solve the problem we propose to revise in detail the implementation of the main algorithm for the localization. Finally it is also important note that the strategy adopted to achieve the goal can be applied in every application that runs in a concurrent and non-concurrent environment.

Chapter 4

Methodology

This chapter presents a multi-level model design flow for the integration of ROS-based heterogeneous applications and their deployment at the edge on embedded low power devices like NVIDIA Jetson TX2.

4.1 A ROS based robotic system

The choice of the most suitable integration platform is one of the main problems to be faced when integrating heterogeneous applications such as robotic software systems. This choice will have implications on performance and communication of the whole system. Furthermore it will be a key factor during the design phase of the entire architecture.

After an accurate comparison among all available robotics platforms, for the purpose of the specific application developed in this thesis it has been decided to use ROS [23] in favour of its standard on communication among other robotic applications.

The design flow proposed in this work is composed of three bottom-up architecture levels, but the same methodology can be applied with more levels. The purpose of this methodology is to bring an heterogeneous application within an orchestration system such as Kubernetes, and then extend it to the edge computing. The architecture are explained in the next sections.

4.2 First level architecture

This is the first architecture to start with and the most common used by robotics researches and developers. At this architecture level the purpose is

to verify the functionality of the whole system with the only ROS capabilities on a single powerful machine. A simple scheme of this architecture is shown in figure 4.1.

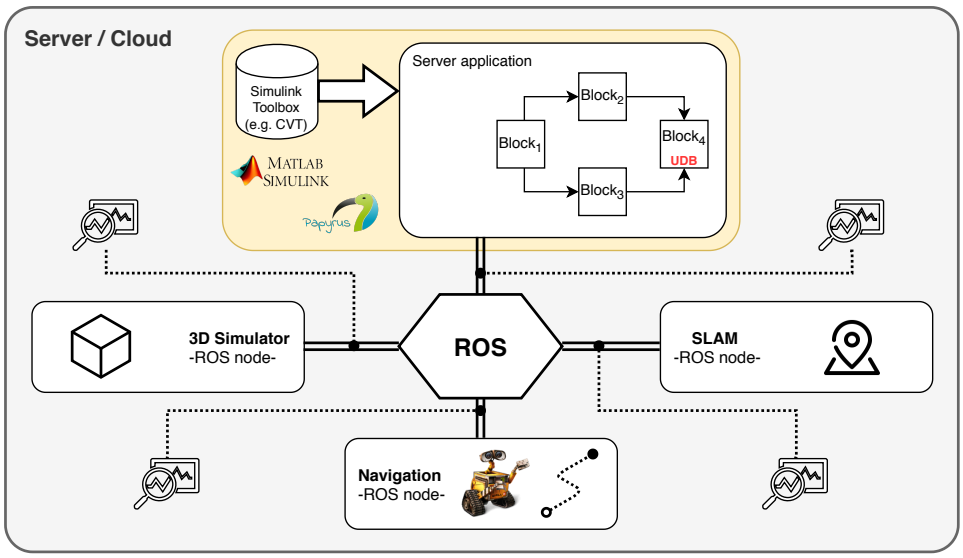


Figure 4.1

A general architecture at L1

The first step to realize this architecture is to wrap your (heterogeneous) applications with the ROS conformance protocol advertises and callbacks. An example of this operation is provided in listings 4.1 and 4.2. In this way, all the applications that compose your system will be able to interact following the standard communication based on ROS. The ROS nodes (publishers and subscribers) communicate between them through virtual channels called topics, thus it is necessary choice the right names of these topics and who will read/write on them. Note that with reference to the scheme presented in figure 4.1, among the other components, there is a robot simulator which have to simulate an industrial robot controller that adheres to the ROS-Industrial driver specification. It means that, for the simulator, we shouldn't choose names and properties of his topics different than the topics published from the robot we are going to develop on. Furthermore all the properties configured in the project within the simulator should be follow the technical specification of the real robot we will use.

Once the applications are ready to run on ROS, they can be executed using a launchfile, where the user can specify a set of nodes to launch and

their parameters. Another possibility is invoke the `rosrun` command for each node that have to be executed.

```
#include "ros/ros.h"
...
int main(int argc, char **argv) {
ros::init(argc, argv, "nodeA");
ros::NodeHandle n;
ros::Publisher pub;
pub=n.advertise<T>("topicT", 10);
while (ros::ok()) {
<put here your application>
...
pub.publish(msg);
...
}
return 0;
}
```

Listing 4.1
Example of publisher.

```
#include "ros/ros.h"
...
void subCallback(const <T> msg) {
<do stuff with received message>
}

int main(int argc, char **argv) {
ros::init(argc, argv, "nodeB");
ros::NodeHandle n;
ros::Subscriber sub;
sub=n.subscribe("topicT", 10,
↪  subCallback);
...
return 0;
}
```

Listing 4.2
Example of subscriber.

Commands such as `rostopic nodes` and `rostopic list` can be used to verify the correct communication between the nodes that compose the system. There exist more other tools to better inspect an application based on ROS, and they are all available online on ROS documentation. A good practise for performance measurement in ROS is to use profilers such as: *perf*, *gprof* and *callgrind*. Another profiler tool available in ROS is `rqt_top`, but it doesn't seem very stable and precise yet. Nevertheless, the best choice still put some time points in your code and write the measured intervals on files.

4.3 Second level architecture

Once the system is working properly on a single x86 host machine, it is possible to split it deploying manually the ROS nodes on a low power embedded device. Again, it is necessary to pay attention on communication between nodes, because now they are on different machines and they need to communicate between them through a physical network layer.

It would be preferable to have the control over which ROS nodes run on which machines, for load-balancing and bandwidth management. Intuitively a user would duplicate the roslaunch files and comment them in a mutually exclusive way, such that the nodes that run on a machine, do not run on the other. Even if it could work correctly, this kind of hard-coding is not the best practice for this purpose, especially for large projects. The recommanded way is to use machine tags to balance load and control which nodes run on the same machine, and consider having the machine file name depend on an environment variable for reusability.

There are some situations, where that's inconvenient or impossible to use the `env` substitution `arg` in order to modifying the behaviour without changing any launch files. For example, this is the case when we need for a package that contains a version of the 2D navigation app, but for use in the Gazebo simulator. For navigation, the only thing that changes is actually that the Gazebo environment is based on a different static map, so the *map_server* node must be loaded with a different argument. It could be use another `env` substitution here. But that would require the user to set a bunch of environment variables just to be able to roslaunch. Instead, it is possible to modify a "top-level" aspect of an application, copying the top level launch file and changing the portions you need.

ROS is designed with distributed computing in mind. A well-written node makes no assumptions about where in the network it runs, allowing computation to be relocated at run-time to match the available resources. Deploying a ROS system across multiple machines require the following:

- You only need one master. Select one machine to run it on.
- All nodes must be configured to use the same master, setting the ROS_MASTER_URI environment variable.
- There must be complete, bi-directional connectivity between all pairs of machines, on all ports.
- Each machine must advertise itself by a name that all other machines can resolve.

Finally, if the roslaunch files have been properly defined as described above, the only thing to do is copy them as they are on all involved devices, and then run the `roslaunch` command.

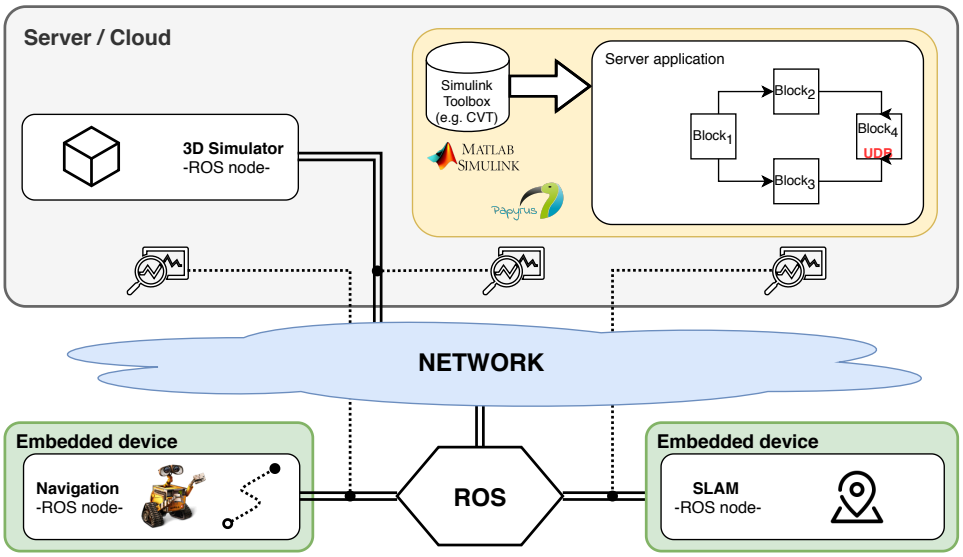


Figure 4.2

A general architecture at L2

Once again it is very important measure the performance of our system, because we will in the next sections that they will be critical for a correct orchestration of the system.

4.4 Third level architecture

This is the final architecture level. As shown in figure 4.3 the simulator has been removed and it has been replaced by the real robot.

The difference respect to the previous architecture level is that there there is one or more driver node that communicate with a piece of hardware and it must run on the machine to which the hardware is physically connected. If the simulator at L1 has been properly designed and configured, in order to have the system running on the real robot it's enough remove the simulator from the ROS netowrk and attach the robot instead. This operation depend to the physical infrastructure of the netwrok we are on.

4.5 Deploying the system at the edge

The next step is to prepare the system for automatic deployment at the edge. For this purpose we need a container platform with which containerize

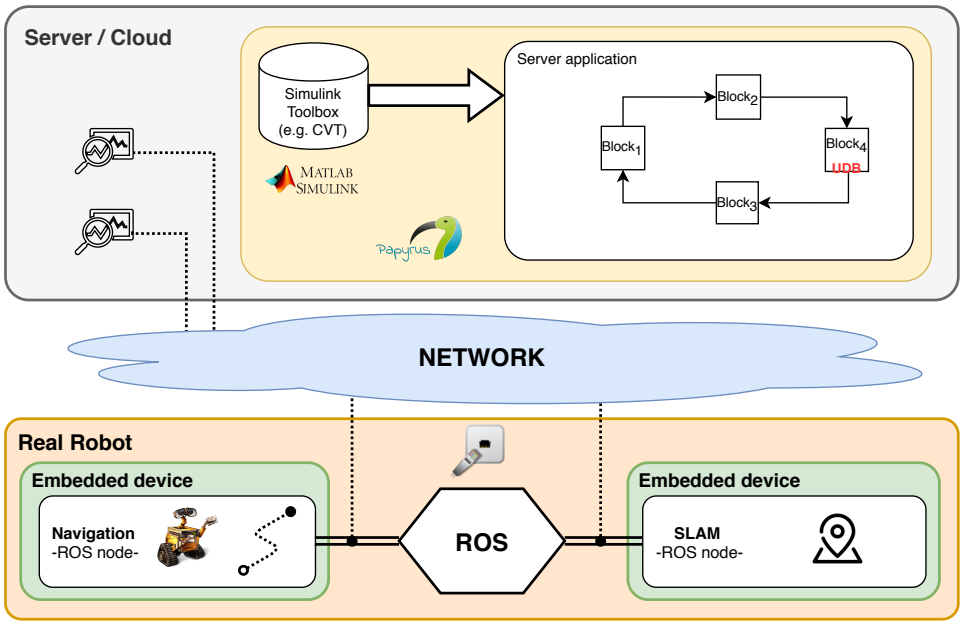


Figure 4.3

A general architecture at L3

the nodes that compose the system, and second, an orchestration platform to automatize the deployment over the cluster of embedded devices.

4.5.1 Containerization with Docker

The use of containers to deploy applications is called containerization. Containers are not new, but their use for easily deploying applications is. Fundamentally, a container is nothing but a running process, with some added encapsulation features applied to it in order to keep it isolated from the host and from other containers. One of the most important aspects of container isolation is that each container interacts with its own private filesystem. This filesystem is provided by a Docker image. Images include everything needed to run an application - the code or binary, runtimes, dependencies, and any other filesystem objects required.

To develop containerized applications, in general, the development workflow looks like this:

- Create and test individual containers for each component of your application by first creating Docker images.

- Assemble your containers and supporting infrastructure into a complete application.
- Test, share, and deploy your complete containerized application.

An image is created only after the definition of a *Dockefile*. Dockerfiles describe how to assemble a private filesystem for a container, and can also contain some metadata describing how to run a container based on this image. An example of Dockerfile is shown in listing 2.

Listing 2 A minimal example of a Dockerfile.

```
# Use the official image as a parent image
FROM node:current-slim

# Set the working directory
WORKDIR /usr/src/app

# Copy the file from your host to your current location
COPY package.json .

# Run the command inside your image filesystem
RUN npm install

# Inform Docker that the container is listening on the specified port at
↪  runtime.
EXPOSE 8080

# Run the specified command within the container.
CMD [ "npm", "start" ]

# Copy the rest of your app's source code from your host to your image
↪  filesystem.
COPY . .
```

The main server provider of images is Docker Hub [34] from which it is possible to search for Docker images created for different systems and configurations. ROS Docker images are listed in the official ROS repository on Docker Hub, therefore this is a good starting point for create an image of a “simple” ROS application. In some cases, when the applications are very heterogeneous and they make use of gpus, a CUDA containerized image (in case of NVIDIA gpus) could be a better starting point. This is because a CUDA proprietary library could be more tedious to install within the image.

In both cases when we containerize a ROS application, we need to pay attention on two critical points, especially if we a system that must be run on different machines. The first is that in order to properly source the ROS packages for our application within the container (once it has been created), we must to put the following command at the end of the Dockerfile:

```
RUN . "/opt/ros/$ROS_DISTRO/setup.sh" && sed -i -e '$isource
↪  "$CATKIN_WS/devel/setup.bash"' /ros_entrypoint.sh
```

Some official ROS images available on the Docker Hub, come already provided of the *ros_entrypoint* script indicated above. In case it is missing, it is very simple to create it in the root of the file system's image. In this way, when a container is starting, before run the `CMD` command, it will source all the necessary ROS packages previously installed within the image.

Before proceeding, we need to spend so words about Docker network. Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

- **User-defined bridge networks** are best when you need multiple containers to communicate on the same Docker host.
- **Host networks** are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
- **Overlay networks** are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- **Macvlan networks** are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
- **Third-party network plugins** allow you to integrate Docker with specialized network stacks.

Looking at the above descriptions provided from Docker documentation, when containers will run on different machines it should used the *Overlay networks*. Nevertheless, in case of ROS applications running on multiple host/devices within a container, there is a further problem related to the

Listing 3 Configuration of daemon.json for NVIDIA runtime.

```
{
        "runtimes": {
                "nvidia": {
                        "path": "nvidia-container-runtime",
                        "runtimeArgs": []
                }
        },
        "default-runtime": "nvidia"
}
```

ports that the ROS system opens. This mean that we cannot statically choice what ports open when the container is running, therefore we have to open all of them using the *Host network*. When we are building the docker image, to enable the communication over the *Host network* we need to run the `build` command with the flag `--network=host`.

Another issue is coming out, when we want to benefit of the hardware resources other than CPU, such as GPUs and tensor cores, provided by the heterogeneous platforms on which we will deploy the applications that compose our system. Since Docker 19.03, it has been added the support to the NVIDIA-container-runtime. Provided by NVIDIA, it is a modified version of runc adding a custom pre-start hook to all containers. If environment variable `NVIDIA_VISIBLE_DEVICES` is set in the Open Container Initiative (OCI) specifications, the hook will configure GPU access for the container by leveraging nvidia-container-cli from libnvidia-container project. There are various methods to setup the docker engine for this purpose. The one proposed by NVIDIA is to reconfiguring the docker daemon configuration file as shown in listing 3.

Because of a NVIDIA proprietary library could be more tedious to install within a Docker image, the choice to start from an image already provided with gpu (CUDA) libraries would be recommanded. This kind of images are available on the NVIDIA NGC website [35]. Once a gpu capable image has been pulled, we can create a container from it simply adding the `--gpus all` flag to the `docker run` or `docker build` commands.

Setting up the Docker platform following the above procedure, will force the system to load some NVIDIA runtime libraries from the host. In particular we could have some problem even we are building our (CUDA) appli-

cation during the build phase of the image, or trying to build them within a running container previously created. Most of the available images on the NVIDIA NGC website are missing development versions of the CUDA libraries. A workaround solution for this problem is the manually installation of the these missing libraries ignoring its dependencies. Otherwise the package manager tool will fail the installation.

Moreover, it is important to highlight that the automatic loading of the runtime libraries, performed by the nvidia-container-runtime, involve a strong dependence with the physical device where the these libraries are installed on. Therefore, if we want to deploy the image created in such way, we have to repeat the manually installation of the missing libraries on all devices on which it will be deployed.

4.5.2 Deployment

Finally, the system is ready to be distributed on heterogeneous platforms, but it isn't yet ready for an automatic smart deployment and orchestration. For this purpose we need to do one more thing. First of all, we have to push to the Docker Hub all the images created as described in the previous section. In this way we can pull them from any device wherever they are. Then we need for an orchestration capable platform like Kubernetes (see section 2.5). Moreover with the support of KubeEdge 2.6 we can extend the Kubernetes capabilities from the cloud to the edge.

To address all these requirements, we can follow the standard procedure described in KubeEdge documentation [36]. KubeEdge is composed of cloud and edge sides. It is built upon Kubernetes and provides core infrastructure support for networking, application deployment and metadata synchronization between cloud and edge. So if we want to setup KubeEdge, we need to setup Kubernetes cluster (existing cluster can be used), cloud side and edge side. On cloud side, we need to install Docker, Kubernetes cluster and cloudcore. On edge side instead, we need to install Docker, MQTT (We can also use internal MQTT broker) and edgecore. After the installation of KubeEdge, setting up cloud side requires two steps:

- Modification of the configuration files.
- Edge node manually registration (this is optional because will be auto registered by default).

Listing 4 Cloudcore configuration.

```
1  apiVersion: cloudcore.config.kubeedge.io/v1alpha1
2  kind: CloudCore
3  kubeAPIConfig:
4      kubeConfig: /path/to/.kube/config
5      master: ""
6  modules:
7      cloudhub:
8          nodeLimit: 10
9          tlsCAFile: /etc/kubeedge/ca/rootCA.crt
10         tlsCertFile: /etc/kubeedge/certs/edge.crt
11         tlsPrivateKeyFile: /etc/kubeedge/certs/edge.key
12         unixsocket:
13             address: unix:///var/lib/kubeedge/kubeedge.sock
14             enable: true
15         websocket:
16         address: <master node ip-address>
17         enable: true
18         port: 10000
```

The *cloudconfig.yaml* configuration file can be generated specifying the `--minconfig` argument to the cloudcore binary (see listing 4). This configuration file is suitable for beginners, but a more advanced configuration can be generate using `--defaultconfig` flag instead. The most important lines to pay attention in the listing 4, are the line 4 and 16. In line 4 we have to type the right path to the configuration file generated during the kubernetes cluster creation. In line 16 we have to write the (static) ip address of the master node where the cloudcore will run.

To the edge side, the procedure to generate the *edgeconfig.yaml* configuration file is the same of the cloud side: either create a minimal configuration with command `edgecore --minconfig` or a full configuration with command `edgecore --defaultconfig`. As we can see in listing 5 its content is slightly different than the cloudcore configuration file. In this case the lines to pay attention are the line 13 and 15. In line 13 we have to write the node name (necessary for deployments). In line 15 we have to type (static) ip address of the edge node. Finally the lines 16 and 18 are the ip address and port where the server socket of the master node is listening on, and must match those configured in the cloudcore configuration file.

Listing 5 Edgecore configuration.

```
apiVersion: edgecore.config.kubeedge.io/v1alpha1
database:
    dataSource: /var/lib/kubeedge/edgecore.db
kind: EdgeCore
modules:
    edged:
        cgroupDriver: cgroupfs
        clusterDNS: ""
        clusterDomain: ""
        devicePluginEnabled: false
        dockerAddress: unix:///var/run/docker.sock
        gpuPluginEnabled: true
        hostnameOverride: <edge node name>
        interfaceName: eth0
        nodeIP: <edge node ip-address>
        podSandboxImage: kubeedge/pause-arm64:3.1
        remoteImageEndpoint: unix:///var/run/dockershim.sock
        remoteRuntimeEndpoint: unix:///var/run/dockershim.sock
        runtimeType: docker
    edgehub:
        heartbeat: 15
        tlsCaFile: /etc/kubeedge/ca/rootCA.crt
        tlsCertFile: /etc/kubeedge/certs/edge.crt
        tlsPrivateKeyFile: /etc/kubeedge/certs/edge.key
        websocket:
            enable: true
            handshakeTimeout: 30
            readDeadline: 15
            server: <master node ip-address>:10000
            writeDeadline: 15
    eventbus:
        mqttMode: 2
        mqttQOS: 0
        mqttRetain: false
        mqttServerExternal: tcp://127.0.0.1:1883
        mqttServerInternal: tcp://127.0.0.1:1884
```

Once the cluster is set up, we are ready to write the last three configuration files necessary for the device controller in order to automatically deploy our containerized applications to the edge nodes. The device controller is the cloud component of KubeEdge which is responsible for device manage-

ment. Device management in KubeEdge is implemented by making use of Kubernetes Custom Resource Definitions (CRDs) to describe device metadata/status and device controller to synchronize these device updates between edge and cloud. The device controller starts two separate go-routines called *upstream controller* and *downstream controller*. These are not separate controllers as such but named here for clarity. The device controller makes use of device model and device instance to implement device management : A device model describes the device properties exposed by the device and property visitors to access these properties. A device model is like a reusable template using which many devices can be created and managed. A device instance represents an actual device object. It is like an instantiation of the device model and references properties defined in the model. The device spec is static while the device status contains dynamically changing data like the desired state of a device property and the state reported by the device. Generally the steps to address these requirements are the following:

1. Create a device model in the cloud node.
2. Create a device instance in the cloud node.
3. Run the mapper application corresponding to your protocol.
4. Edit the status section of the device instance yaml created in step 2 and apply the yaml to change the state of device twin. This change will be reflected at the edge, through the device controller and device twin modules. Based on the updated value of device twin at the edge the mapper will be able to perform its operation on the device.
5. The reported values of the device twin are updated by the mapper application at the edge and this data is synced back to the cloud by the device controller. User can view the update at the cloud by checking his device instance object.

Note: Creation of device instance will also lead to the creation of a config map which will contain information about the devices which are required by the mapper applications. The name of the config map will be as follows: device-profile-config-<edge-node-name>. The updation of the config map is handled internally by the device controller. Further details on both device model and device instance definition can be found on Kubeedge documentation.

An example of device model, device instance and deployment configuration files are respectively shown below:

```yaml
apiVersion: devices.kubeedge.io/v1alpha1
kind: DeviceModel
metadata:
  name: led-light
  namespace: default
spec:
  properties:
    - name: power-status
      type:
        string:
          accessMode: ReadWrite
          defaultValue: 'OFF'
    - name: gpio-pin-number
      type:
        int:
          accessMode: ReadOnly
          defaultValue: 18
```

```yaml
apiVersion: devices.kubeedge.io/v1alpha1
kind: Device
metadata:
  name: led-light-instance-01
  labels:
    model: led-light
spec:
  deviceModelRef:
    name: led-light
  nodeSelector:
    nodeSelectorTerms:
      - matchExpressions:
          - key: ''
            operator: In
            values:
              - edge-node1
status:
  twins:
    - propertyName: power-status
      desired:
        metadata:
          type: string
        value: 'OFF'
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: led-light-mapper-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: led-light-mapper
  template:
    metadata:
      labels:
        app: led-light-mapper
    spec:
      hostNetwork: true
      containers:
        - name: led-light-mapper-container
          image: '<your_dockerhub_username>/led-light-mapper:v1.1'
          imagePullPolicy: Always
          securityContext:
            privileged: true
          volumeMounts:
            - name: config-volume
              mountPath: /opt/kubeedge/
      volumes:
        - name: config-volume
          configMap:
            name: device-profile-config-<edge_node_name>
      restartPolicy: Always
```

Finally we can create/apply these configuration files files using the `kubectl` commands and leave KubeEdge the creation, distribution and management of the pods related to the our applications.

Once again the are several tools we can use to measure health and performance of the running system. As explained in section 2.5 we can also benefit of the user friendly dashboard provided by Google.

Chapter 5

Experimental Results

This chapter presents the experimental results obtained during the customization, debugging and verification of the ORB-SLAM algorithm for edge computing and the use of the methodology for ROS-based robotics application based on Docker and Kubernetes. First the chapter describes the Hardware and Software setup used during the experiments. Then, it explains how the presented methodology has been applied to merge ORB-SLAM and a typical robotic system developed for mobile robot navigation in a warehouse scenario.

The main technical specifications of the *Host* and the embedded device used during the experiments are shown in table 5.1 and in table 5.2 respectively.

Hardware	Model
Processor	Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
Storage	SATA 500GiB
Memory	16GiB
GPU	NVIDIA GeForce GTX 980
Software	**Version**
Operating System	Ubuntu 18.04
OpenCV	3.4.7
CUDA	10.0
ROS	Melodic Moreina

Table 5.1
Technical specifications of the Host

Hardware	Model
Processor	Dual-core Denver 2 64-bit CPU and quad-core ARM A57 complex
Storage	eMMC 5.1 32 GB
Memory	LPDDR4 8 GB 128-bit
GPU	NVIDIA Pascal™ architecture with 256 NVIDIA CUDA cores 1.3 TFLOPS (FP16)
Software	**Version**
Operating System	18.04.1-Ubuntu
OpenCV	3.4.7
CUDA	10.0
ROS	Melodic Moreina

Table 5.2
Technical specifications of NVIDIA Jetson TX2.

5.1 Mobile robots application

5.1.1 Architecture at L1

At this architecture level we put a whole ROS-based system on a single Intel x86 powered machine that we will conventionally call *Host*. As shown in figure 5.1, in this case there are two main ROS nodes that communicate: the ORB-SLAM2, described in chapter 3, and a hybrid motion planner (VOVD [37]). Both applications act as ROS nodes to perform a mobile robot 2D navigation task.

Because of the most used robotics platforms used today support the Melodic Morenia [38] version of ROS, and because VOVD was written with the Kinetic version, it has been decided to convert VOVD in the relative Melodic ROS version. This task required a redefinition of some functions related to the ROS `tf2` package [39] due to the deprecation of the `tf` package functions previously used. Another modification done is related to the 3D map used by Gazebo [40] simulator. The problem was that it came with a featureless map generated from an ".pgm" file (see fig. 5.2), and because of ORB-SLAM2 properly works only if some features are detected during its execution, it has been necessary to add a material to the 3D model of the

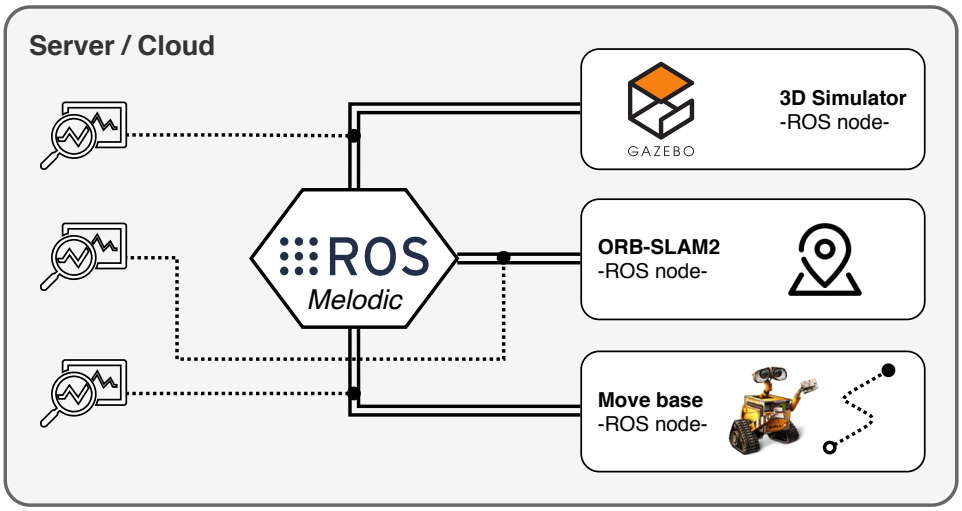


Figure 5.1

L1 architecture used during the experiments

map. The results obtained before and after applying the material on the 3D map are visible in figure 5.3.

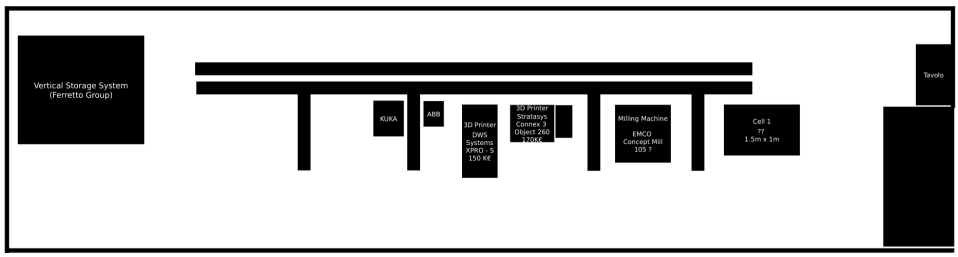
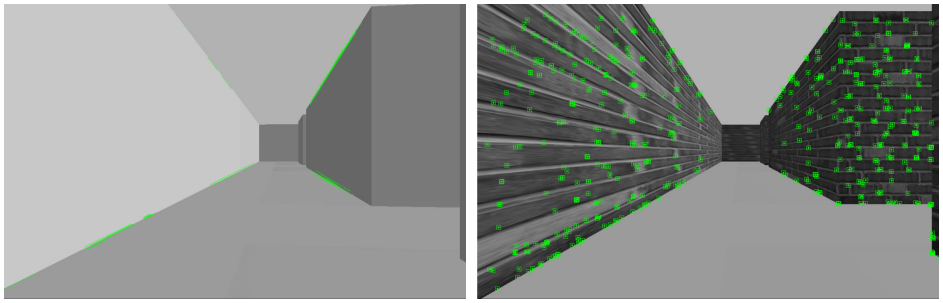


Figure 5.2

pgm file map description of IceLab.

Talking about ORB-SLAM2, it comes already provided with the necessaries functions to run in a ROS environment, thus no others operations have been required to complete the first architecture level.

Thanks to the modularity of ROS we were able to verify the functional behaviour of the system implementing a custom ROS node that acts as a monitor. It listens on all topics of the system and captures the informations of interest. To obtain a more accurate performance measurement, we added some time points within the source code of both VOVD and ORB-SLAM2. Specifically speaking for VOVD we measure the time to perform the local planner computation. To the ORB-SLAM2 side instead we measure the time



(a) Map with material **(b)** Map without material

Figure 5.3
3D maps material comparison

to elaborate each frame coming from the simulator camera. The performance measurement obtained at this architecture level are shown in table 5.3.

Setup	Odom (ms)	Track Pipe (ms)	Frame Elab. (ms)	Feature Ext.(ms)	Supp. (FPS)
orbslam	-	18.52	24.09	22.04	41.45
vovd	1.20	-	-	-	-
(orb;vovd)	2.33	38.99	39.93	36.75	25.03

Table 5.3
Performance measurement at L1 architecture.

5.1.2 Architecture at L2

Another common architecture in robotics involves the distribution of ROS nodes on different hardware platforms. In our case we have two NVIDIA Jetson TX2. This step requires the split of our system in two parts: the *Host*, on which runs Gazebo simulator, and the *Device/s* that will perform either the robot navigation and SLAM. Figure 5.4 shows the configuration we setup during the experiment.

Due to the several interconnected nodes, each of which having many parameters, it is recommanded to create a roslaunch file using the *machine tag*. As explained in chapter 4, these tags allow to control on which machines the node run.

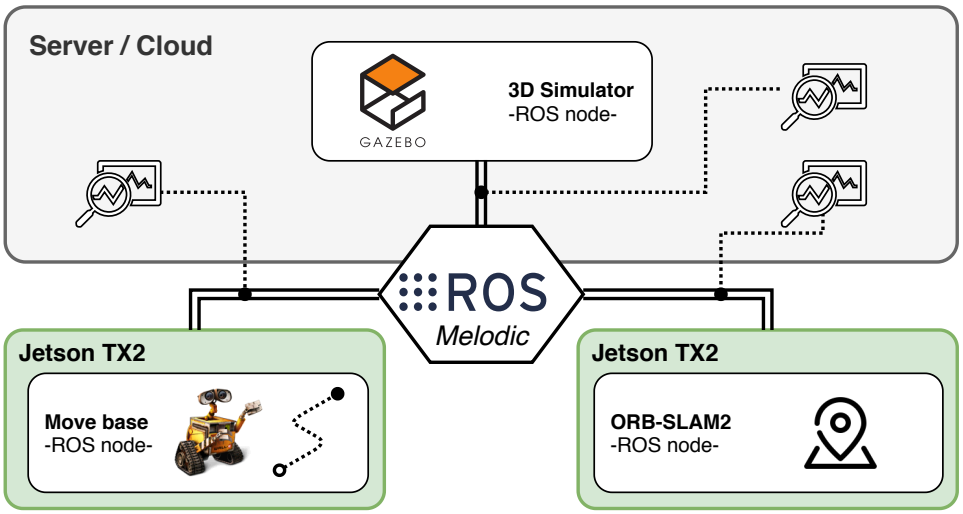


Figure 5.4

L2 architecture used during the experiments

At this architecture level the main focus is on the communication between the *Host*, that acts as Master, and the other external nodes that represent the slaves. It is important to verify that the system continues working properly and that the Network doesn't become a bottleneck. For example, if we need to transfer a stream of images from one ROS node to another that is running on different machine, we should compress each frame before send it over the network. In this way, we avoid to saturate the bandwidth of the network. In our case the compression task is performed to the Host side from the robot's camera within Gazebo simulator. The decompression instead is computed on Jetson by the ORB-SLAM2 node whenever it receives a new frame to elaborate. We measured the performance of the system in the same way we have done in the previous architecture level. Note that, as expected, the performance of ORB-SLAM2 are dropped down due to the limited computing capability of the Jetson. This is more highlighted when we run both VOVD and ORB-SLAM2 on the same device because they are competing for the hardware resources. The new performance measurements are presented in table 5.4. Note that, due to the modularity of ROS, we were able to fine tune the system without making any change to the code. For example, we can easily modify either frequency of the camera output or the number of key-points extracted for each incoming frame. Another parameter to consider is the bandwidth of the network. The virtual Asus Xtion camera

used by Gazebo publish images with a resolution of 1280*720*3 at 20 frames per second. Computers hooked up to LANs are connected using Cat5 cable. Although the Cat5 ethernet cable can handle up to 10/100 Mbps at a 100 MHz bandwidth, it is not sufficient to transfer the data coming from the camera because it requires a bandwidth of 442 Mps. This is the reason why we measured the data decompression time performed by the ORB-SLAM2 ROS node.

Setup	Odom (ms)	Track Pipe (ms)	Dec. (ms)	Frame Elab. (ms)	Feature Ext.(ms)	Supp. (FPS)
Performance Measurement: camera 10 Hz						
orb(j1)	-	43.41	13.08	49.21	43.62	20.32
vovd(j1)	1.70	-	-	-	-	-
(orb;vovd)(j1)	2.58	73.49	18.69	62.70	54.76	13.61
(orb(j1);vovd(j2))	1.60	49.81	13.53	49.25	43.01	20.08
Performance Measurement: camera 20 Hz						
orb(j1)	-	54.82	14.47	49.40	44.35	18.24
vovd(j1)	1.70	-	-	-	-	-
(orb;vovd)(j1)	2.85	93.15	23.96	67.99	60.41	10.74
(orb(j1);vovd(j2))	1.60	56.20	14.00	49.32	43.95	17.79

Table 5.4
Performance measurement at L2 architecture.

5.1.3 Architecture at L3

Finally we reached the last architecture level of the design flow. It is composed by the *Host*, *Device/s* and the real robot *Robot/s*. In our case the real robot is the KUKA YouBot [41] provided by the robotics laboratory ALTAIR from University of Verona.

The key point here is that if the simulator has been properly configured at L1 architecture (as described in chapter 4), the operation to detach the simulator and attach back the real robot is completely transparent with respect to the entire system. This is one of the main advantage we had benefit due to the modularity of ROS.

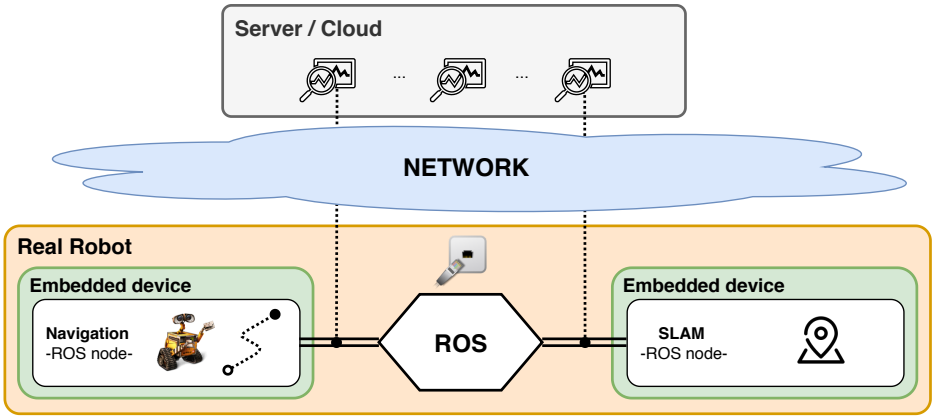


Figure 5.5

L3 architecture

As we can see in Fig. 5.5, the simulator has been removed from the *Host* and it has been replaced by the real robot, on which one or more embedded devices can be attached.

5.2 Deployment from the cloud to the edge

As mentioned in section 2.6, KubeEdge is built upon Kubernetes and provides fundamental infrastructure support for network, app. deployment and metadata synchronization between cloud and edge. As we know, Kubernetes uses containers to run isolated, packaged applications across its cluster nodes. To run on Kubernetes, our applications must be encapsulated in one or more container images and executed using a container runtime like Docker. While containerizing components is a requirement for Kubernetes, it also allows for scaling and management. For instance, containers provide isolation between the application environment and the external host system, support a networked, service-oriented approach to inter-application communication, and typically take configuration through environmental variables and expose logs written to standard error and standard out. Containers themselves encourage process-based concurrency and help maintain dev/prod parity by being independently scalable and bundling the process's runtime environment. These characteristics made possible to package our applications so that they run smoothly on Kubernetes. In this describes the process used in our project to prepare both VOVD and ORB-SLAM2

applications for the edge computing through the support of KubeEdge.

5.2.1 Dockerization

At this point the main goal of the project was to reach the edge devices from the host/cloud server wherever they are. For this purpose we need first containerize both VOVD and ORB-SLAM2 for NVIDIA Jetsons. Since both are wrapped as ROS nodes, we have to include the ROS (Melodic) base distro and the required dependencies to run our applications. Listing 6 shows the Dockerfile used to containerize VOVD during the experiments.

Listing 6 Dockerfile used to create the Docker image of VOVD.

```
1  FROM ros:melodic-ros-base
2
3  # set args
4  ARG ip=192.168.254.3
5  ARG master_uri=http://192.168.254.2:11311
6
7  # set enviroment
8  ENV ROS_MASTER_URI=$master_uri ROS_IP=$ip ROS_WS=/home/catkin_ws
9
10 COPY VOVD install_dependencies.sh ./
11
12 # install ros packages
13 RUN mkdir -p $ROS_WS/src && mv VOVD $ROS_WS/src && \
14 chmod +x install_dependencies.sh && sh install_dependencies.sh && \
15 . "/opt/ros/$ROS_DISTRO/setup.sh" && cd $ROS_WS && catkin_make && \
16 sed --in-place --expression '$isource "$ROS_WS/devel/setup.bash"' \
17 /ros_entrypoint.sh
18
19 # set the working directory
20 WORKDIR $ROS_WS
21
22 # run ros package launch file
23 ENTRYPOINT ["roslaunch", "vo_controller"]
24 CMD ["test_edge.launch"]
```

In phase of image's building we set the `--network=host` flag in order to enable the communication over the ROS network among containers running on different machines. This is all about the containerization of a "simple" ROS node such as VOVD.

To the other side ORB-SLAM2 also needs to access to the GPU of the device. On NVIDIA NGC web portal, Linux4Tegra (L4T) package provides the bootloader, kernel, necessary firmwares, NVIDIA drivers, flashing utilities and a sample filesystem to be used on Jetson systems. The software packages contained in L4T provide the functionality necessary to run Linux on Jetson devices (Xavier, TX2, and Nano). Thus in this case the L4t image provided by NVIDIA is a more affordable starting point for the containerization of the ORB-SLAM2. Furthermore we know that ORB-SLAM2 is an heterogeneous application made a wide variety of libraries such OpenCV, OpenVX, CUDA, TensorRT and so on. Consequently we had to provide all of these libraries within the dockerfile in order to build our application.

5.3 The whole system on KubeEdge

After a functional verification at container's level, the system is almost ready to be deployed from the cloud to the edge using kubeEdge. Following the procedure described in section 4.5.2, first we setup our cluster in order to enable the physical and virtual communication between the Host (to the cloudside) and edge nodes (to the edgeside). Then we create three configuration files for each application we want to deploy to the edge. These configuration files contain the informations for device model, device instance and deployment respectively.

During our experiments we tested the system in various conditions: we forced the deployment either to a single device or across multiple devices. The performance measurements done with the system running on KubeEdge are shown in table 5.5. The results show that KubeEdge platform does not contribute to the normal network latency in a significant way (the container image download time is not included in the test).

Setup	Odom (ms)	Track Pipe (ms)	Dec. (ms)	Frame Elab. (ms)	Feature Ext.(ms)	Supp. (FPS)
KubeEdge - Performance Measurement: camera 10 Hz						
orb(j1)	-	47.28	12.78	40.52	36.02	21.15
vovd(j1)	1.30	-	-	-	-	-
(orb;vovd)(j1)	1.40	82.47	15.10	53.54	48.09	12.12
(orb(j1);vovd(j2))	1.38	50.83	13.98	43.80	38.04	19.67
KubeEdge - Performance Measurement: camera 20 Hz						
orb(j1)	-	50.20	16.20	42.16	34.82	19.92
vovd(j1)	1.21	-	-	-	-	-
(orb;vovd)(j1)	1.43	70.54	15.02	55.60	50.24	14.18
(orb(j1);vovd(j2))	1.26	58.14	13.52	42.34	37.17	17.20

Table 5.5

Conclusions And Future Works

In this thesis, we presented a design flow based on Kubernetes platform and ROS for edge computing. We showed that it is possible for KubeEdge to manage remote edge nodes and deploy and manage applications into the edge with the same API.

...

This work is a preliminary experience and we intend to optimize and enhance KubeEdge as future work.

Bibliography

[1] H. Abbas, I. Saha, Y. Shoukry, R. Ehlers, G. Fainekos, R. Gupta, R. Majumdar, and D. Ulus, "Embedded software for robotics: Challenges and future directions: Special session," in *Proceedings of the International Conference on Embedded Software*, EMSOFT '18, IEEE Press, 2018.

[2] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. PP, pp. 1–20, 07 2019.

[3] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[5] S. Ruder, "An overview of gradient descent optimization algorithms.," 2016. cite arxiv:1609.04747Comment: Added derivations of AdaMax and Nadam.

[6] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, (USA), p. 311–318, Association for Computational Linguistics, 2002.

[7] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy, "Speed/accuracy trade-offs for modern convolutional object detectors," 2016.

[8] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6517–6525, 2017.

[9] Z. Lu, S. Rallapalli, K. Chan, and T. La Porta, "Modeling the resource requirements of convolutional neural networks on mobile devices," in *Proceedings of the 25th ACM International Conference on Multimedia*, MM '17, (New York, NY, USA), p. 1663–1671, Association for Computing Machinery, 2017.

[10] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "Deepdecision: A mobile deep learning framework for edge video analytics," pp. 1421–1429, 04 2018.

[11] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[13] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2016.

[14] "Tensorflow lite." `https://www.tensorflow.org/lite`.

[15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, (New York, NY, USA), p. 675–678, Association for Computing Machinery, 2014.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. W. et al., ed.), pp. 8024–8035, Curran Associates, Inc., 2019.

[17] N. Corporation, "Cuda zone." `https://developer.nvidia.com/cuda-zone`.

[18] N. Corporation, "cuDNN deep neural network library." `https://developer.nvidia.com/cudnn`.

[19] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," 2018.

[20] B. Taylor, V. S. Marco, W. Wolff, Y. Elkhatib, and Z. Wang, "Adaptive selection of deep learning models on embedded systems," 2018.

[21] Q. Z. Y. L. W. Shi, J. Cao and L. Xu, "Edge computing: Vision and challenges," *in IEEE Internet of Things*, vol. "3", no. "5", pp. 637–646, 2016.

[22] O. C. A. W. Group, "Openfog reference architecture for fog computing." `https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf`.

[23] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," vol. 3, 01 2009.

[24] N. Corporation, "Isaac sdk." `https://developer.nvidia.com/isaac-sd`.

[25] M. Freese, S. Singh, F. Ozaki, and N. Matsuhira, "Virtual robot experimentation platform v-rep: A versatile 3d robot simulator," in *Proceedings of the Second International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, SIMPAR'10, (Berlin, Heidelberg), p. 51–62, Springer-Verlag, 2010.

[26] E. Rohmer, S. P. N. Singh, and M. Freese, "Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework," in *"Proc. of The International Conference on Intelligent Robots and Systems (IROS)"*, 2013. "www.coppeliarobotics.com".

[27] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s — a publish/subscribe protocol for wireless sensor networks," in *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*, pp. 791–798, Jan 2008.

[28] Q. Zhang, X. Zhang, and W. Shi, "Poster abstract: Firework: Big data processing in collaborative edge environment," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 81–82, Oct 2016.

[29] H. Cao, M. Wachowicz, and S. Cha, "Developing an edge computing platform for real-time descriptive analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 4546–4554, Dec 2017.

[30] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 373–377, Oct 2018.

[31] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[32] D. D. B. S. Aldegheri, N. Bombieri and A. Farinelli, "Data flow orb-slam for real-time performance on embedded gpu boards," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5370–5375, 2019.

[33] A. Geiger, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR '12, (USA), p. 3354–3361, IEEE Computer Society, 2012.

[34] "Docker hub." `https://hub.docker.com/`.

[35] "Nvidia ncg." `https://ngc.nvidia.com/catalog/containers`.

[36] "Kubeedge documentation." `http://docs.kubeedge.io/en/latest/`.

[37] N. Piccinelli and R. Muradore, "Hybrid motion planner integrating global voronoi diagrams and local velocity obstacle method," pp. 26–31, 06 2018.

[38] "Ros melodic morenia." `http://wiki.ros.org/melodic`.

[39] W. M. Tully Foote, Eitan Marder-Eppstein, "tf ros package." `http://wiki.ros.org/tf`.

[40] "Gazebo robot simulation." `http://gazebosim.org/`.

[41] U. H. R. Bischoff and E. Prassler, "Kuka youbot - a mobile manipulator for research and education,," pp. 1–4, 2011.