Technische Universität Berlin

Fakultät IV
Institut für Wirtschaftsinformatik und Quantitative Methoden
Fachgebiet Agententechnologien in betrieblichen Anwendungen und der Telekommunikation

Bachelor Thesis

Enabling QoS-Aware Task Execution on Distributed Node-RED Clusters for Fog Computing Environments

Daniel Sebastian Lienau

Matriculation Number: 378170

30.09.2019

Supervised by
Prof. Dr.-Ing. habil. Dr. h.c. Sahin Albayrak

Assistant Supervisor
Cem Akpolat

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 30.09.2019

. . . . . .
Unterschrift

Zusammenfassung

Fog-Architekturen werden aufgrund der steigenden Anzahl vernetzter Geräte immer häufiger eingesetzt, insbesondere im Bereich von Internet of Things. Hierbei werden Dienste wie Aufgabenausführung oder Datenspeicherung näher am Benutzer platziert, was im Vergleich zu Cloud-Architekturen zu einer höheren Servicequalität führt, da geringere Latenzzeiten und höhere Bandbreiten möglich sind. Eine Herausforderung dabei ist, Services bestmöglich auf einer Infrastruktur zu verteilen.

In dieser Arbeit wird ein Algorithmus konzipiert und umgesetzt, welcher eine QoS-bewusste Verteilung von Aufgaben in einer Fog-Umgebung ermöglicht. Darüber hinaus wird ein System entwickelt, welches mit Hilfe des Algorithmus die optimale Verteilungsstrategie eines vorgegebenen Services findet und diese auf ein Node-RED Cluster anwendet. Dies geschieht dynamisch, da die Infrastruktur überwacht wird und das System regelmäßig überprüft, ob die aktuelle Verteilungsstrategie optimal ist oder ob durch Veränderung der Netzwerkbedingungen eine andere Verteilung angewendet werden kann, die geringere Ausführungszeiten ermöglicht.

Abstract

Fog architectures are used more and more due to the increasing number of connected devices through networks, particularly with regard to the Internet of Things domain. In a fog architecture, services such as task execution or data storage are placed closer to the user, resulting in a higher quality of service compared to cloud architectures because lower latencies and higher bandwidths can be achieved. One challenge is to distribute services on a fog infrastructure in the best possible way.

In this work, an algorithm is designed and implemented that enables QoS-aware distribution of tasks in a fog environment. Furthermore, a system is developed that uses the algorithm to find and deploy the optimal distribution strategy for a given service to a Node-RED cluster. This is done dynamically since the environment is monitored and the system regularly checks if the current task distribution is still optimal or if another distribution enabling shorter latencies can be applied due to changed network conditions.

Contents

1. Introduction

The number of interconnected devices on the internet is constantly increasing, thus more and more data is exchanged between different participants. While the internet initially experienced a broad distribution during the 80s thanks to the world wide web service, where mainly personal computers and enterprise servers were involved in the communication, nowadays the number of offered services and online devices is countless. Over the last few years, *Internet of Things (IoT)* has become very popular and an end to this trend is not predictable. Therefore, the amount of exchanged data over the internet and between network nodes is constantly increasing as well, leading to higher requirements on the overall network infrastructure to ensure a high connection quality with little delays as well as the avoidance of network congestion.

A common scenario is devices communicating with each other via a cloud server in a big data center, where the physical distance between devices might be relatively far. Often though, and this is especially the case for IoT scenarios, a lot of data is produced and consumed locally, on the edge of networks. While this is not a big deal for services which do not have a high real-time requirement, a service that has a high real-time requirement relies on the network connection quality and low delay times between participating devices. If this cannot be ensured, the service is unusable. This becomes even more important in applications where personal safety plays a role, for example, a device at the side of the road communicating with an autonomous car and giving instructions to brake. If these devices were interconnected via a server in the cloud, the delay time would depend on the current network congestion level, resulting in a varying time in delay, thus not ensuring the *quality of service (QoS)*. This issue is addressed by fog computing.

Fog computing provides a distributed infrastructure at the edge of the network, resulting in low-latency access and faster response to application requests [Bit+17]. In a fog infrastructure, there is a variable number of so called fog nodes, which can provide resources like computation or storage. Fog nodes are usually small devices with limited resources compared to servers in the cloud. Furthermore, they are not necessarily of the same kind, thus have a different amount of resources available. For example, one fog node can have a relatively low computation power compared to another fog node. The challenge here is to distribute the services among different fog nodes while ensuring the QoS requirements are met. While some services cannot be executed at all on certain nodes due to the lack of resources, other services are competing for resources on other fog nodes. One solution here is to *decompose* services into smaller *tasks*, so that these tasks can run on different fog nodes. However, these tasks need to be recomposed thereafter to provide the service as a whole. This

part is called *service composition* and is illustrated in figure 1.1.

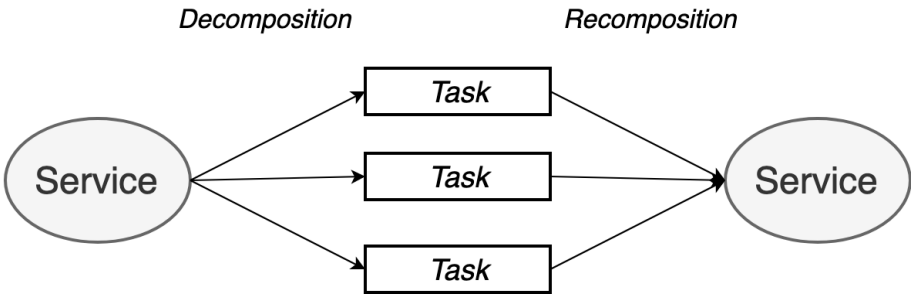

Figure 1.1.: Service composition

The challenge here is though, that the conditions inside of the fog infrastructure can change at any time. First of all, the demand for the offered services can increase or decrease at any point in time. Second, fog nodes can join or leave the network at any point in time. Simply said, tasks have to be distributed *dynamically*. To ensure that each service meets the QoS requirements, each node has to be constantly monitored. If the demand for services increases, the average load on the fog nodes will be high. To still provide an acceptable service quality for each service, tasks of services which have lower QoS requirements have to be moved to another fog device. A task could always be executed in the cloud, but this would lead to longer delay times.

2. Related Work

In this section, relevant topics for this work are discussed based on related work.

2.1. Fog Infrastructure

Fog is an architecture that distributes computation, communication, control, and storage closer to the end users along the cloud-to-things continuum [CZ16]. Although cloud computing has established itself over the last few years, it cannot be used for certain use-cases, especially time-critical and bandwidth-intensive ones. Because of the closer physical distance to a fog node, fog computing can reach a much lower end-to-end latency than cloud computing [Bit+17][He+16]. This plays a vital role in applications such as vehicle-to-vehicle communication [He+16]. Figure 2.1 shows a typical fog environment.

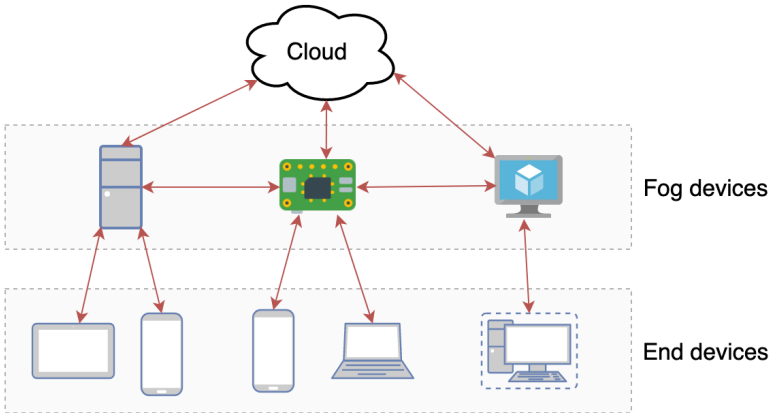

Figure 2.1.: Fog architecture

A network of fog nodes is a distributed heterogeneous network. Unlike in cloud computing, where hardware capabilities are virtually unlimited [BF17], resources on a fog device are usually constrained. They cannot execute every task because their computation power is limited. Furthermore, the available resources vary from node to node, making the whole network *heterogeneous*. Instead of executing a task on a central computer, it is *distributed* between several nodes in the network. Firstly, one must determine which fog node can execute which part of the task in terms of resource-constraints. In the end, all partial results are combined into the final result.

2.2. Flow-Based Programming

Flow-based programming (FBP) allows the developer to structurally define the informational flow within a system. As it is not important *where* a specific task is physically executed, FBP focuses on the *path* the data takes from one process to another. Any number of further processes can be involved in between, while every process manipulates or just reads the data to trigger further actions. Each process has a specific input and output. For the developer, all execution units are *black box processes*, which can consume and create data. The processes can be executed and recomposed in any order, as long as the output and input formats match each other. It is also possible to split the data flow at a particular point and combine it back together at a later point, making parallel computation possible.

Data between processes is transferred in *Information Packets (IPs)*. IPs belong to a single process or are in a transfer state, where they are owned by no process. As soon as a process receives an IP, it can start processing it without the need to communicate with other processes, because the IP contains all necessary information the process needs to fulfill the task. This makes FBP highly attractive for fog computing environments, where each fog node can be used to execute one or more process types, depending on the process requirements like computation power, bandwidth or latency.

Node-RED is a development tool that provides a browser-based flow editor where a flow can be programmed and deployed on the device which runs the Node-RED software.

2.3. QoS-Aware Resource Allocation and Scheduling Approaches

In this section, related work on QoS, load-balancing, and resource allocation is discussed. Although fog is considered to be a cloud close to the ground, load balancing strategies of cloud computing cannot be directly adopted in the fog network because of the heterogeneity of fog [He+16].

FogTorch

In [BF17] a model to support QoS-aware deployment of multi-component IoT applications to fog infrastructures is proposed. Furthermore, a Java tool called `FogTorch` has been prototyped which implements that model. The model allows to define QoS profiles, fog infrastructures and IoT applications, which are used to determine eligible deployments.

A *QoS profile* defines bandwidth and average latency required for an application,

or offered by a communication link. A *fog infrastructure* includes IoT devices, fog nodes, cloud data centers, and communication links, while each link is associated to its QoS profile. Cloud computing is modeled according to the hypothesis that it can offer a virtually unlimited amount of hardware capabilities. An IoT *application* is a set of independently deployable components that are working together and must meet some QoS constraints. For this, software components as well as required interactions among components, including the desired QoS profile, are defined. In the end, an *eligible deployment* for the IoT application is calculated by an algorithm.

An algorithm selects where a component is to be deployed within the cloud to things continuum. For this, a preprocessing procedure which reduces the search space for eligible deployments runs before a backtracking procedure and heuristics are used to determine a single eligible deployment. Because the proposed backtracking algorithm follows a heuristic approach to get a solution faster, it shows greedy behavior.

[BFI17] presents FogTorchπ, an open source prototype based on ForTorch [BF17]. Compared to FogTorch, FogTorchπ additionally allows for expressing *processing capabilities* and *average QoS attributes* of a fog infrastructure, as well as processing and QoS requirements of an application. It determines deployments of the application over the fog infrastructure that meet all such requirements [BFI17]. The QoS of communication links are modeled by using probability distributions repeatedly (based on historical data) to simulate different runtime behaviors. In the end, it aggregates the results for deployments generated over a large number of runs. The output of FogTorchπ contains eligible deployments (like FogTorch), but additionally outputs QoS assurance and resource consumption over fog nodes which allows to compare possible deployments and evaluate the impact of possible changes.

MPSO-CO

[He+16] proposes the *modified constrained optimization particle swarm optimization* (MPSO-CO) load balancing algorithm, which is *software defined networking* (SDN)-based. It is able to effectively decrease latency and improve the QoS in a *software defined cloud/fog networking* (SDCFN) architecture compared to other algorithms. It was developed for Internet of Vehicles (IoV) applications which still suffers high processing latency. SDN is used for centralized control and to get the required information before load balancing. The key technology of SDN is decoupling data and control plane. The controller collects real-time information of the network including load, processing speed, and communication latency. Based on that, it can formulate optimal load balancing strategies for the network.

iFogSim

In [Gup+17] a tool called *iFogSim* is designed and implemented. It is used to *simulate* a fog computing environment by using two different placement strategies (*cloud-only* and *edge-ward*). It was found that the average latency of a control loop

is much lower in an *edge-ward* than in a *cloud-only* placement strategy.

In this work, an algorithm for a *real* fog environment is developed and implemented in chapter 5, which is inspired by *iFogSim* and adopts some classes of its class model. For this reason, the relevant classes for this work are briefly summarized in the following.

- `FogDevice`: Specifies hardware characteristics of a node like CPU power, available RAM, available disk storage, as well as network communication capabilities (uplink and downlink).
- `Sensor`: Represents an IoT sensor, is connected to a `FogDevice`, has an *output characteristic*.
- `Tuple`: Fundamental communication unit. It is characterized by its *type*, *source* and *destination* application module. Processing requirements (in *million instructions*) and *length of data* are defined as well.
- `Application`: An application consists of several modules, each module processing incoming data.
  - `AppModule`: Represents processing elements of an application. An instance of an *AppModule* produces an output tuple for every incoming tuple.
  - `AppEdge`: Models the *data-dependency* between a pair of application modules. It is characterized by the *type* of a tuple, *processing requirements* as well as the *length of data* the tuple carries.
  - `AppLoop`: Specifies *process-control loops*. A loop has a *starting* and *terminating* module (as well as any number of modules in between). Figure 2.2 shows an application containing two loops:
    1. `Sensor -> Analyzer -> User Interface`
    2. `Sensor -> Analyzer -> Actuator`

    For every loop an *end-to-end latency* can be specified. For instance, the second loop is time critical if an actuator changes the environment based on the sensors value and thus has a lower latency requirement.

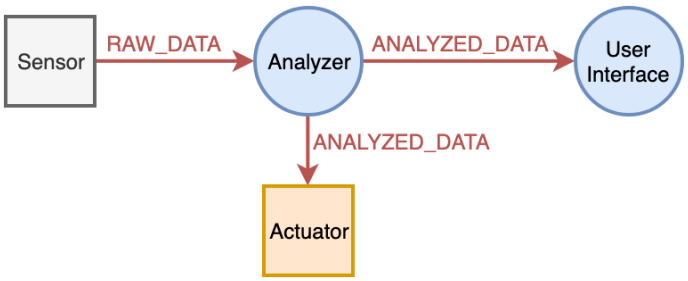

Figure 2.2.: Application containing four AppModule, three AppEdge, and two AppLoop instances

2.4. MAPE-K

A self-adaptive system consists of the two layers *managed subsystem* and *managing subsystem*, whereas the managing subsystem resides on top of the managed subsystem and monitors the managed subsystem as well as the environment. It realizes a feedback loop which adapts to changes, e.g. goal changes or environmental changes like congestion or failures [ARS15].

The *Monitor-Analyze-Plan-Execute over a Knowledge base (MAPE-K)* [KC03] reference control model is the most influential reference control model for autonomic and self-adaptive systems. It is commonly used to realize feedback loops [ARS15].

The component *Knowledge (K)* is responsible for storing and providing data from or to other components. The component *Monitor (M)* collects data through probes or sensors from the environment as well as data from the managed subsystem. The collected data is saved in K. The component *Analyze (A)* analyzes the collected data to check if the system needs to adapt. If this is the case, the component *Plan (P)* will determine which actions are required to put the system in the desired state. The component *Execution (E)* then carries out these actions [ARS15].

3. Use Cases

This section describes two different use cases which would benefit from QoS-Aware deployment.

3.1. Notify a Device Using Temperature Sensor Data

Sensor networks are used to monitor an (industrial) environment for various measurable parameters. In a *wireless sensor network* (WSN) different types of sensors (e.g. microphones, CO2, pressure, humidity, thermometers) can be used. The measured values are used to decide whether an action should take place or not. Since the sensors themselves usually have few resources for the calculation, this calculation takes place externally. This can be done either centralized or decentralized by distributing the calculation between different nodes.

A task execution on a local node rather than on a cloud server makes sense especially for time-critical or data-intensive applications: *Data-intensive* because the stream of data remains in the local network and does not rely on an internet connection. Therefore, no bandwidth of the internet connection is occupied. *Time critical* because the round trip time to a cloud server is usually higher compared to a local server. This work focuses on the time-critical aspect of task distribution and execution.

To distribute services between different nodes, a service must first be split into different modules or tasks, which can then be executed on different fog nodes. In the end, the partial results must be merged to an overall result. The fog nodes must therefore be able to communicate with each other and forward the final result to another unit, which then takes further action or not based on the result. The possible actions can be of different priority, e.g. a temperature adjustment might be less important than an emergency stop or emergency braking of a machine.

It is very important to use a distributed deployment here because the computational resources of the measurement taking node are limited. However, the final result of a calculation must be available within a specified time window if time-critical actions have to be executed. The node itself cannot guarantee to calculate the result in time, so it has to offload the task to another node.

Use case: A factory worker takes products from the conveyor belt. These could still be hot due to previous processing steps. A sensor measures the temperature

of the products and stops the conveyor belt if necessary to ensure the worker's safety.

Task: Notify a device using sensor data

- Collect raw data from devices. Devices can be simulated. If a dataset that can be used by these services is found, devices can only forward the data to a specified cloud server.
- A cloud service should operate some processes. Based on the data structure provided by those sensors, the cloud service should evaluate different parameters and end up with a result that will be used by another local service or device in the smart factory.

Challenge: The number of the devices sending data to the cloud service will increase, while the network will be dynamically manipulated through the `DITG` tool at the same time in order to enforce the Orchestrator to make a decision. Possible decisions are: Moving the service from a cloud to a fog device, or from one fog to another fog device.

3.2. Object Detection

Object detection is used in most of the fields in daily life. It is also relevant for a smart factory and factory devices where the information has to be extracted from the recorded picture. This use case aims at reflecting a simple object detection process that will be operated using an object detection service running in cloud and fog network. The essential goal is to compare the total delay for the object recognition process in different locations of the network.

Use case: Factory prints the products, worker takes the product from the conveyor belt and stores it in one of the container locating in front of the worker. An IoT-enabled camera tracks the worker object placement to figure out whether the products are placed correctly. The object detection service should analyze the object and inform the user/robot if there is a mistake while placing the objects.

Alternative use case: A factory worker does not know where to place the object, since the objects differ from each other. Therefore, the printed object should be recognized by the object detection service beforehand and then inform the worker in which container the object should be placed.

Task: Video processing

- Video recording: A local computer sends a video stream to the cloud or to a fog node running an object detection service.
- Object detection: The service returns the name of the object in text format as well as the correct container for the object.

Challenge: The task should first be executed in the cloud. Afterwards, the system should decide if the service requirements are satisfied. Due to varying network conditions which are created by the same tools mentioned in 3.1, the system should decide to move the service from the cloud or a fog node to another fog node which can satisfy the service requirements.

4. Problem Definition & Challenges

This chapter describes the technical and theoretical challenges of this thesis. The problem is mainly resource allocation and its usage.

4.1. Load Balancing Among Fog Nodes

In a Node-RED context, a service can consist of several *flows*, while each flow provides certain functionality. These flows are able to communicate with each other by using the *HTTP* or *TCP* nodes available in Node-RED, for instance. Hence, a service can be *distributed* by deploying its flows on different devices. The challenge here is to decide which *flow* should be deployed to which *node* (see figure 4.1).

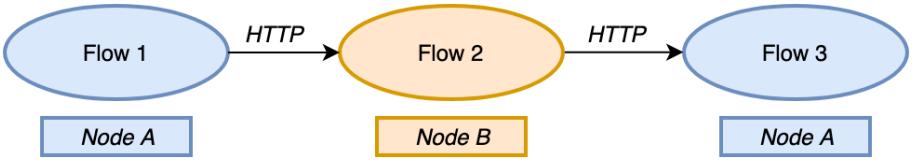

Figure 4.1.: Distributed service in a Node-RED context

The goal is to meet the service's QoS requirements. For the WSN use cases it has to be ensured that a result is available within the defined maximum latency timespan, e.g. within 10 milliseconds for condition monitoring for safety.

Fog infrastructure is characterized by different nodes offering various hardware and software resources. In order to assign tasks to nodes, it must first be checked whether a node is able to execute the task with given QoS requirements. The challenge here is to implement a resource allocation algorithm for the given use cases.

4.2. Dynamically Varying Network Conditions

The network conditions might change at any point in time. Increased network traffic plays a major role here, but there may also be increased demand for services. For instance, in the WSN scenario, the amount of sensors requesting services might increase, which leads to a higher load on the fog nodes hardware until the point of congestion. To avoid congestion and ensure QoS, tasks of services with a lower priority have to be moved elsewhere to free resources for high priority services. Apart from that, the network traffic might increase due to other network services which are not part of the fog infrastructure but are using the same link.

The challenge here is to adapt to the mentioned changes, for which they must be identified first before the load balancing algorithm is executed to eventually calculate a differentiated, optimized deployment.

4.3. Fog Node Failures

A fog infrastructure is not as reliable as a cloud infrastructure. For instance, fog nodes can join or leave the network at any point in time, which leads to a dynamic change in the available resources. Since fog nodes can be any type of hardware, it is not unlikely that small and relatively cheap devices (e.g. Raspberry Pi) are used. Thus, it is more likely that they might fail since the hardware components are not of high quality compared to enterprise server hardware. Furthermore, the environment is neither monitored nor controlled like a cloud server data center, which could lead to node failures (e.g. network connection errors or power supply failures).

The challenge here is to adapt to those dynamic changes. If a node leaves the network, fewer resources are available and therefore another deployment might be the optimum. Similarly, when a node joins the network, more resources are available, making another deployment ideal.

5. Design and Implementation of a QoS-Aware Scheduling Algorithm

This section describes the design and implementation of the *QoS Scheduler* which is used in chapter 6. UML diagrams can be found in appendix A.

5.1. Algorithm Architecture

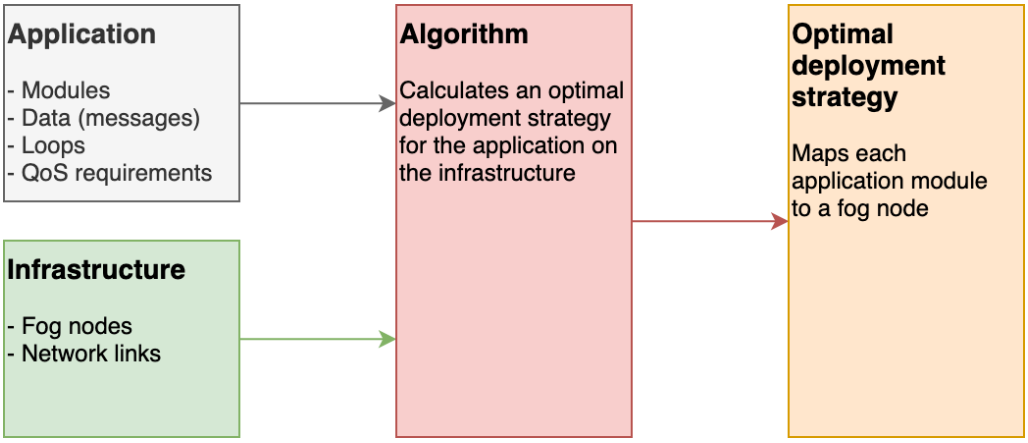

Figure 5.1.: QoS scheduling algorithm architecture

The algorithm takes an instance of `Infrastructure` and `Application` as inputs, and outputs an optimal deployment strategy (instance of `AppDeployment`). The optimal deployment strategy differs depending on the type of scheduler. For example, the goal of a scheduler can be to minimize the total *energy consumption*, the total *network utilization* or the *computing power* used. Since the focus here is on *QoS*, the deployment that minimizes *total latency* should be the optimal deployment. Figure 5.1 illustrates the architecture of the algorithm.

5.1.1. Infrastructure

The infrastructure consists of *fog nodes* and *network connections* between those nodes. Network connections are modeled as uplinks only, because data is sent forward in the application loop from one module to another, while every module is deployed on one fog node. Between two nodes (node A and node B), the downlink of node B from node A's point of view is the same as the uplink of node A from node B's point of view. Therefore, each fog node has a set of uplinks to all other

reachable fog nodes. A node also has an uplink to itself, since the next module in the loop could be executed on the same node. This uplink is modeled with *infinite speed* and *zero latency*. Hence, there are n^2 uplinks in the infrastructure in total ($n \triangleq number\ of\ fog\ nodes$). Figure 5.2 shows a fog infrastructure with three nodes and the corresponding nine uplinks.

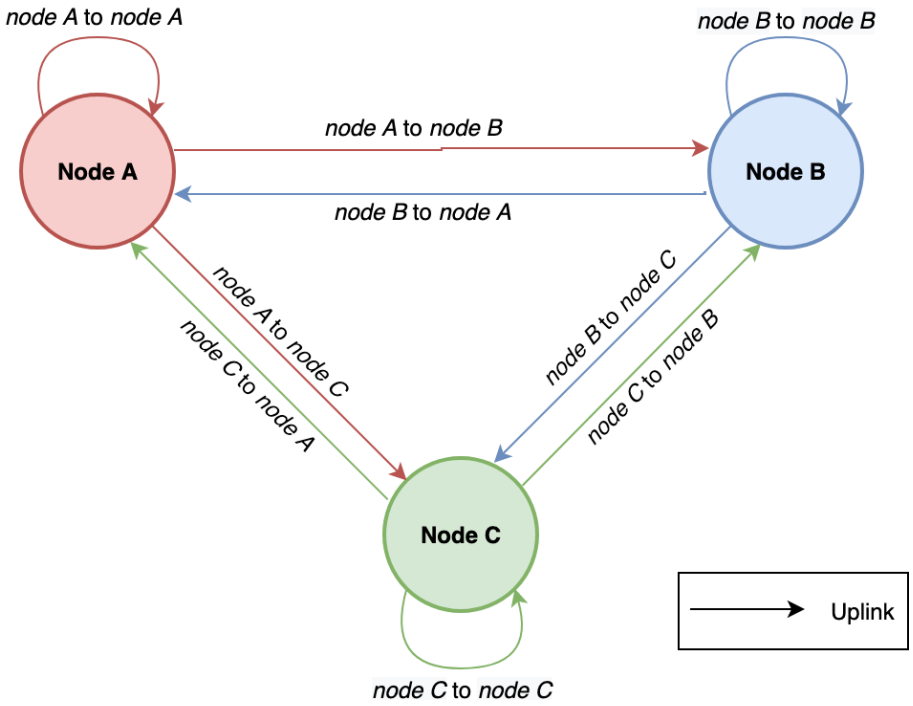

Figure 5.2.: Fog infrastructure with three nodes and nine uplinks

Next, the relevant classes for the infrastructure are described. The corresponding class diagram is depicted in figure A.1.

- `Infrastructure`: Contains a list of `FogNode` instances.
- `NetworkUplink`: Every instance of `NetworkUplink` has the following *fields*:
  - `source`: The source `FogNode` of the link.
  - `destination`: The destination `FogNode` of the link.
  - `latency`: The RTT of the link.
  - `bitPerSecond`: The link speed (bandwidth) in bit per second.
- `FogNode`: Every instance of `FogNode` has the following *fields*:
  - `id`: The identifier of the device, e.g. its hostname.
  - `ramTotal`: The total available ram.
  - `storageTotal`: The total available disk space.
  - `cpuCores`: The amount of available CPU cores.
  - `cpuMips`: The power of a CPU measured in MIPS (million instructions per second).
  - `connectedHardware`: Hardware connected to the node (e.g. camera, tem-

perature sensor)

- `uplinks`: Uplinks to other fog nodes (instances of `NetworkUplink`)

Every instance of `FogNode` has the following *methods*:

- `deployModule(AppSoftwareModule module): boolean`
  Tries to deploy an `AppSoftwareModule` (see section 5.1.2) on this node. Checks whether this node can fulfill the modules requirements (RAM, storage, hardware modules). Modules are *not* automatically undeployed. The method can be called multiple times and thus more than one module can be deployed. In this case, sufficient resources must be available to execute *all* modules. The method returns `false` for the first module that cannot be deployed, otherwise `true`.
- `undeployModule(AppSoftwareModule module): void`
  Undeploys the `AppSoftwareModule` from this node.
- `getUplinkTo(FogNode destination): NetworkUplink`
  Returns the `NetworkUplink` from this node to the `destination` node.
- `calculateProcessingTime(AppSoftwareModule module): double`
  Calculates the execution time of an `AppSoftwareModule` (see section 5.1.2) on this node. Returns the value in milliseconds. The result is calculated using the following formula:

$$\text{execution time [ms]} = \frac{\texttt{requiredMi}}{\texttt{cpuMips}} \cdot 1,000$$

5.1.2. Application

An application consists of *modules*, *messages* and *loops*. A module can either be a *hardware module* (e.g. a camera or a temperature sensor) or a *software module* (e.g. data processor, object detection engine).

Next, the relevant classes for an application are described. The corresponding class diagram is shown in figure A.2.

- `AppModule`: This abstract class is extended by the classes `AppHardwareModule` and `AppSoftwareModule`. It contains the following *fields*:
  - `id`: The type of this module, e.g. `CAMERA` (hardware) or `OBJECT_DETECTOR` (software).
  - `inputType`: The modules input type, e.g. `IMAGE_RAW` from a camera (can be `null` for the starting module of a loop).
  - `outputType`: The modules output type, e.g. `IMAGE_PROCESSED` (can be `null` for the terminating module of a loop).
- `AppHardwareModule`: A hardware module extends `AppModule` and solely consists of the three fields of `AppModule`. If it has an `outputType` but no `inputType`, it is a *sensor* (e.g. temperature sensor, outputs the current temperature). If it has an `inputType` but no `outputType`, it is an *actuator* (e.g. air conditioner, takes a target value as input, outputs nothing).

- `AppSoftwareModule`: A software module extends `AppModule` and additionally has the following *fields*:
  - `requiredRam`: Memory required by the module.
  - `requiredStorage`: Disk storage required by the module.
  - `requiredHardwareModules`: Hardware modules that must be connected to the node that executes the software module. For instance, the hardware module `CAMERA` must be connected to a node which executes the software module `CAMERA_CONTROLLER` in order to instruct the camera to take a picture first and output that picture to another module for further processing in the next step.
  - `requiredMi`: This attribute determines how many CPU instructions (defined in million instructions) are needed approximately for processing a single message. This allows the execution time to be estimated.
- `AppMessage`: A message has the following *fields*:
  - `contentType`: The content of a message, e.g. `IMAGE_RAW`, `IMAGE_PROCESSED`.
  - `dataPerMessage`: The data size per message in KByte, e.g. 500 KB for an image.

  A message has the following *methods*:
  - `transferTime(source: FogNode, destination: FogNode): double` Calculates the transfer time of the message based on the `NetworkUplink` from `source` node to `destination` node. Returns the value in milliseconds. The result is calculated using the following formula:

$$\text{transfer time [ms]} = \text{latency [ms]} + \frac{\text{message size [bit]}}{\text{link speed}[\frac{\text{bit}}{\text{ms}}]}$$

- `AppLoop`: A loop defines the connection between the different app modules. An application can consist of one or more loops, while each loop has a unique *start*, *end*, and *maximum latency*. The loop determines how or in which *order* the modules are *connected* to each other. The maximum latency attribute defines the maximum allowed duration for one cycle. For instance, the loop's last module `IMAGE_VIEWER` should view an image not later than *1 second* after the loop's first module `CAMERA` took the image.

  In figure 5.3, the order is as follows:
  `CAMERA` → `camera-controller` → `object-detector` → `image-viewer`

  A loop has the following *fields*:
  - `application`: The application this loop belongs to.
  - `name`: The name of the loop, e.g. *object-detection*.
  - `maxLatency`: Time (in milliseconds) in which the result must be available, i.e. the time required to process all modules from the first to the last.
  - `modules`: A `LinkedList` containing all modules belonging to this loop. The order of this list determines the execution order of the loop.

  A loop has the following *methods*:

– `totalLatency(AppDeployment d): double`
Calculates the total latency for the loop for a given `AppDeployment` (see section 5.1.3). Returns the value in milliseconds. The total latency is the sum of the loop's task executions and its message transfers between modules. How these two values are calculated is depicted in figures A.3 and A.4.

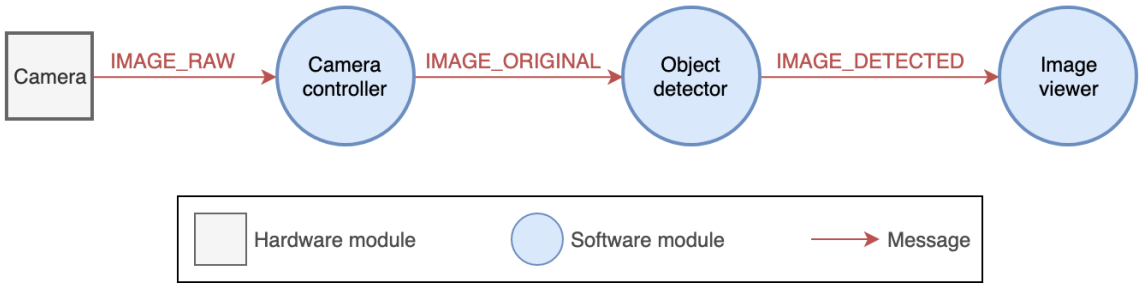

Figure 5.3.: Object detection application containing four modules and three messages

5.1.3. Scheduler

The architecture includes the interface `Scheduler`, which has the two methods `getOptimalDeployment()` and `getValidDeployments()`. The former method returns the one optimal deployment (`AppDeployment`), while the latter one returns a list of all possible deployments that meet the requirements. Class relationships are shown in figure A.5.

The class `AppDeployment` is used to represent a deployment for a given `Application`. The field `moduleToNodeMap` is used to specify which module (`AppSoftwareModule`) should be executed on which node (`FogNode`).

In section 5.2 the concrete class `QosScheduler` is described, which implements the `Scheduler` interface.

5.2. QoS Scheduler

The job of the *QoS Scheduler* is to find all deployments which satisfy the latency requirements of all application loops. It outputs the one deployment with the lowest latency. This section describes how this task is solved.

The `QosScheduler` takes an `Application` and an `Infrastructure` in the constructor. It tries to find all valid deployments as well as the optimal deployment for the `Application` over the `Infrastructure`.

In the first step, the scheduling algorithm creates a combination of all possible deployment strategies without checking if the modules could be executed on the assigned nodes (hardware requirements) and if latency requirements are met (app loop

requirements). In a mathematical sense, a permutation with repetition is created in this step.

Table 5.1 shows all nine possible combinations of an application consisting of *two modules* and an infrastructure consisting of *three fog nodes*.

#	**module A**	**module B**
1.	node 1	node 1
2.	node 1	node 2
3.	node 1	node 3
4.	node 2	node 1
5.	node 2	node 2
6.	node 2	node 3
7.	node 3	node 1
8.	node 3	node 2
9.	node 3	node 3

Table 5.1.: Combination of all possible deployments for an application with two modules and an infrastructure with three nodes

Each line represents a *deployment strategy candidate*. However, it has not yet been checked whether the deployment strategy is valid because the requirements have not been examined yet. In order to do so, the *QoS Scheduler* goes through several steps to analyze if a deployment strategy fulfills the application requirements. If the answer to any step is negative, this deployment strategy candidate is not valid and the next one will be analyzed. How the algorithm creates a set of valid deployments is depicted in 5.4, which is also the implementation of the method `getValidDeployments()`.

The result is a set of deployments which fulfill the application requirements. The *optimal deployment* is the one with the *lowest total latency* and will be returned by calling the method `getOptimalDeployment()`. The corresponding activity diagram is shown in figure 5.5.

How the hardware and latency requirements are validated is shown in figures A.6 and A.7. To validate latency requirements it is necessary to calculate the *total latency* of a loop, which is the sum of the processing times of all modules and all message transfers between the modules of the loop. The following formula shows the calculation of the total latency with t_n = execution time of module n, and t_m = transfer time of message m:

$$\text{total latency} = \sum_{n=0}^{N} t_n + \sum_{m=0}^{M} t_m$$

The calculations of *total processing time* and *total transfer time* are depicted in figures A.3 and A.4.

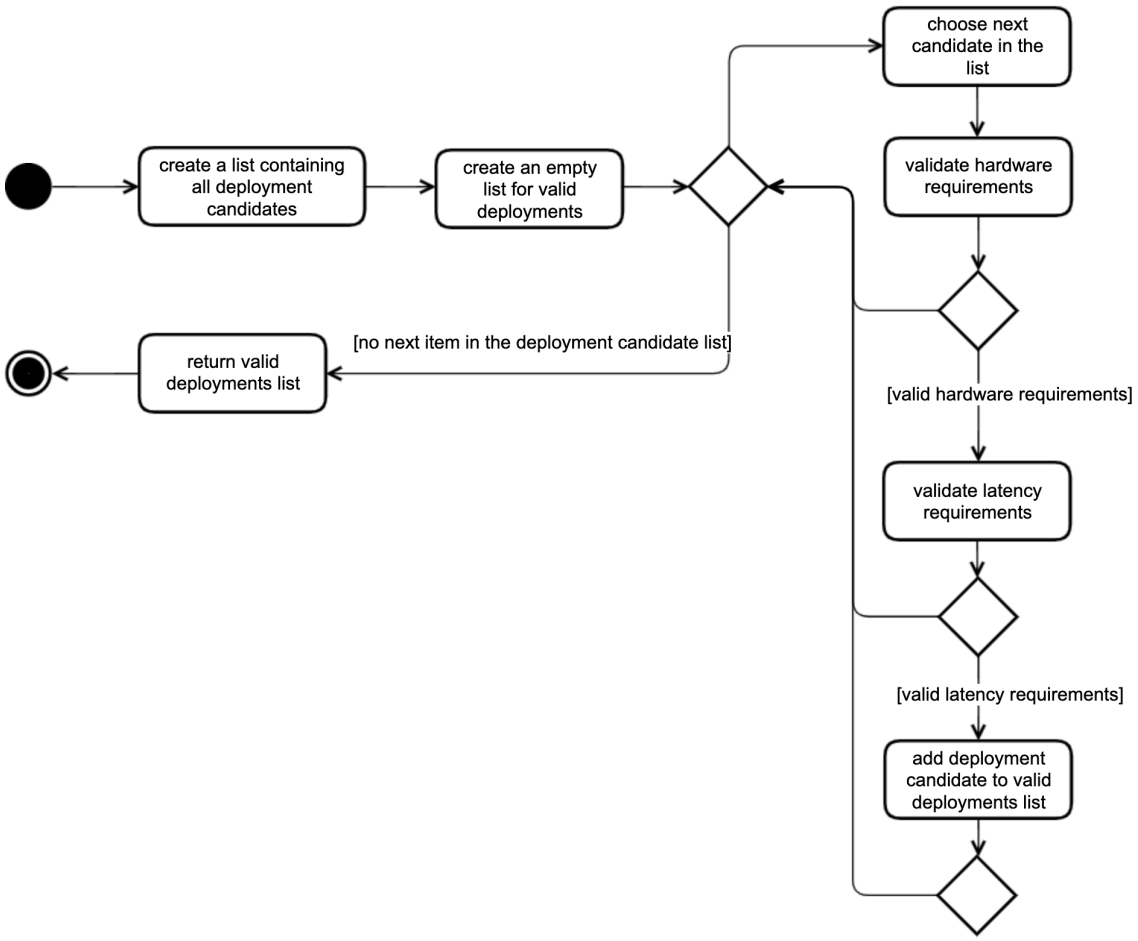

Figure 5.4.: Activity diagram for finding possible deployments

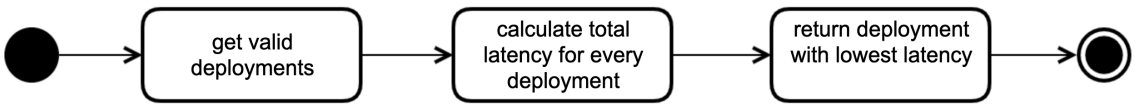

Figure 5.5.: Activity diagram for finding the optimal deployment

6. QoS-Monitor & -Orchestrator

The algorithm described in the previous chapter works so far in a static context. However, the task is to apply the optimal deployment to a Node-RED cluster, where the network conditions can change at any time. While the application characteristics remain unchanged, the infrastructure can change mainly in two ways. Firstly, fog nodes may join or leave the network, and secondly, network connection quality between nodes can improve or deteriorate. These changes must be detected by the Orchestrator and, if the current deployment strategy no longer meets the application requirements, a new optimal deployment must be found and applied. Figure 6.1 depicts the system architecture of the QoS-Monitor & Orchestrator.

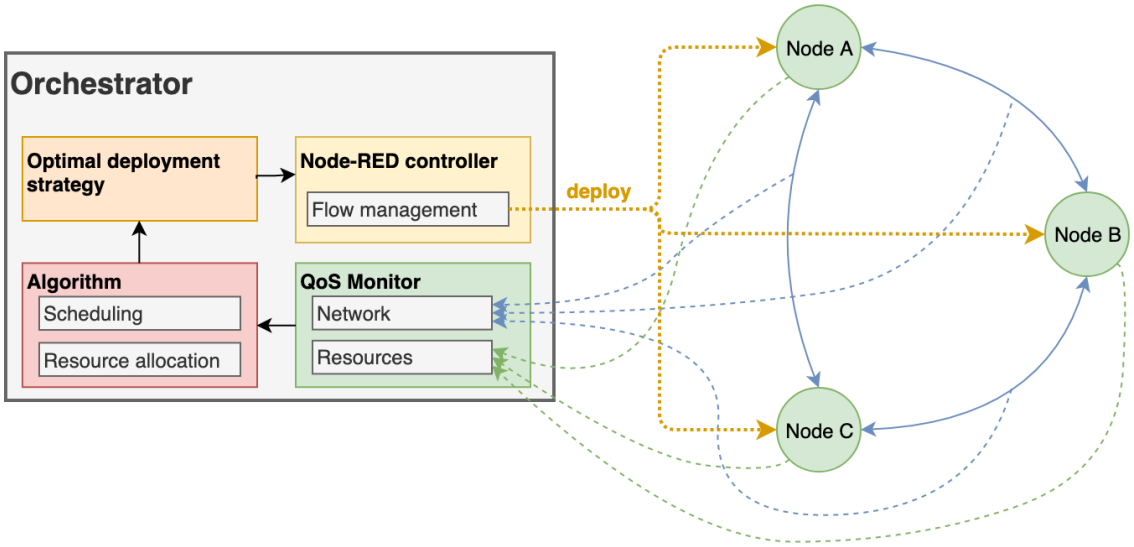

Figure 6.1.: System architecture of the QoS-Monitor & Orchestrator

The class `NodeRedOrchestrator` is the implementation of the described Orchestrator. It monitors the infrastructure and deploys the optimal deployment to it by using the *QoS-Scheduler*. To make use of the *QoS-Scheduler*, an `Infrastructure` and `Application` instance must be available. The former one is actively maintained by the Orchestrator (see section 6.1), while the latter one is statically created upon start-up of the `NodeRedOrchestrator`. The class `FogNode` of the algorithm in chapter 5 is extended by the class `NodeRedFogNode` in order to add additional functionality for measuring the nodes hardware and network capabilities before it is added to the infrastructure. The corresponding class diagram is shown in figure A.8.

6.1. Monitoring the Infrastructure

Directly after starting the Orchestrator, it is only aware of the applications that should be deployed to the infrastructure. However, the infrastructure is empty at the beginning. Fog nodes send their availability status via a *heartbeat message* to a *MQTT broker* at a given interval. The Orchestrator receives those messages and handles them according to the following cases:

1. The node sending the heartbeat is not registered in the current infrastructure
   → Handle new fog node (see section 6.2)
2. The node sending the heartbeat is registered
   → Update timestamp of last received heartbeat
3. Orchestrator is expecting but missing a heartbeat from a node
   → Check if node is still up and if not, remove it from infrastructure

6.2. Handling New Fog Nodes

In the case of receiving a heartbeat from a node which has not been registered by the Orchestrator yet, the hardware capabilities as well as the network connection of the node must be checked before it can be added to the infrastructure. To achieve this, the node is able to receive several commands via `MQTT`.

- `sysinfo`: Sends basic information about the nodes hardware (*RAM*, *storage*, *cpu cores*, *connected hardware*).
- `ping`: Measures the *RTT* from this node to another node by using the command line tool `ping`. Returns the latency in milliseconds.
- `bandwidth`: Measures the network connection from this node to another node. Returns the *bandwidth in Mbit/s*.
- `benchmark_cpu`: Runs the tool `sysbench` which is a lightweight benchmark tool made for linux servers. The result is the *execution time* for a given (but always the same) task. A CPU score is calculated and returned based on the execution time (see section 6.2.2).

Figure 6.2 shows how the Orchestrator handles a heartbeat message from an unknown fog node. Note that `ping` and `bandwidth` are measured between this node and *all other existing nodes* in the infrastructure. These commands can also be used to check the current state of a node or network connection at a later point in time when the node is already registered by the Orchestrator.

After the Orchestrator has collected all required information about the node, it is added to the infrastructure and the scheduling algorithm is run to (eventually) find a new optimal deployment.

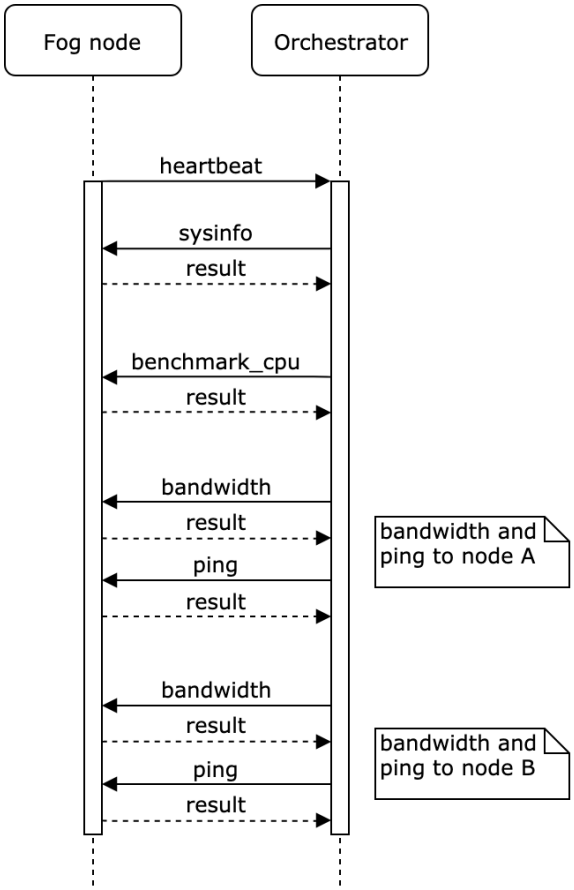

Figure 6.2.: Sequence diagram of the Orchestrator handling a new heartbeat message

6.2.1. Bandwidth Measuring

To measure the bandwidth, a tool called `iperf3` was used first, which measures the transfer rate via `TCP`. However, these measured values could not be achieved in practice, since `HTTP` is used for message transmission between the modules. Because `HTTP` is built on top of `TCP`, it introduces additional protocol overhead leading to a lower transfer rate compared to `TCP`. For this reason, `iperf3` was replaced by a measurement method using `HTTP`:

To measure the uplink from *node A* to *node B*, *node A* sends a `HTTP POST` request to a predefined endpoint on *node B*. The `HTTP body` initially contains `1MB` of data. After *node B* has received the request, it responds with an empty body. *Node A* can then calculate the *transfer rate* by using the *data size* and *transfer time*. If the response from node B is received fairly rapidly (less than 500 milliseconds), node A repeats the measurement with a larger data size to get a more accurate result.

6.2.2. CPU Benchmarking

The algorithm uses the unit *MIPS* to identify the speed of a CPU. The execution time of a module on a node is calculated by using that value in addition to the required CPU instructions of the module (see function `calculateProcessingTime()`

of the `FogNode` object). However, *MIPS* cannot be read from the system like total RAM or amount of CPU cores. For this reason, a *CPU score* is used instead. The Orchestrator calculates this score according to the following formula:

$$\text{CPU score} = \frac{10,000}{\text{sysbench result [time in ms]}}$$

Thus, the lower the execution time of `sysbench`, the higher the CPU score. This score is used for the field `cpuMips` of a `FogNode` instance.

However, the field `requiredMi` of an `AppSoftwareModule` instance must also be set accordingly. To determine this value, the corresponding software module is deployed on a node where the CPU score is known. Since the execution time of processing one message on that node can be measured, the required value can be calculated as follows:

$$\texttt{requiredMi} = \text{CPU score} \cdot \frac{\text{execution time [ms]}}{1,000\ \text{[ms]}}$$

6.3. Using the QoS-Scheduler in the Orchestrator

The output of the algorithm determines which module should be deployed on which node. In the example shown in table 6.1, the algorithm has decided to distribute the three modules of the object detection application (see figure 5.3) between two nodes (*node A* and *node B*).

Order	Module	Executing node
1	`camera-controller`	node A
2	`object-detector`	node B
3	`image-viewer`	node A

Table 6.1.: Sample deployment strategy for object detection application

Node-RED Controller

Every running Node-RED instance can be controlled via the *Node-RED Admin API*. In order to make use of it, the `NodeRedController` was implemented (see figure A.8). It is able to call all necessary API functions, so that the Orchestrator can control the Node-RED instances (create, update and delete flows in particular).

Flow database

A Node-RED flow can be manually exported, saved, and imported via the Node-RED web interface so that every flow could be transferred from one Node-RED instance to another. However, in this architecture, nothing is manually exported and stored in a database. Instead, there is a *dedicated Node-RED instance* which serves exclusively as the *flow database* and is *not used for task execution*. The Orchestrator can

then query a specific flow from this instance via the `NodeRedController` by calling the method `getFlowByName(flowName)`. In a Node-RED context, each application *module* is a Node-RED *flow* which is stored in the database.

The flow database is implemented in the class `NodeRedFlowDatabase` (see figure A.8).

Deploying flows

The Orchestrator obtains each flow from the database and deploys it to the Node-RED instance running on the scheduled fog node. The Orchestrator manages the flows on the nodes by using the `NodeRedController`. The process of deploying a deployment strategy (instance of `AppDeployment`) to the infrastructure is shown in figure 6.3.

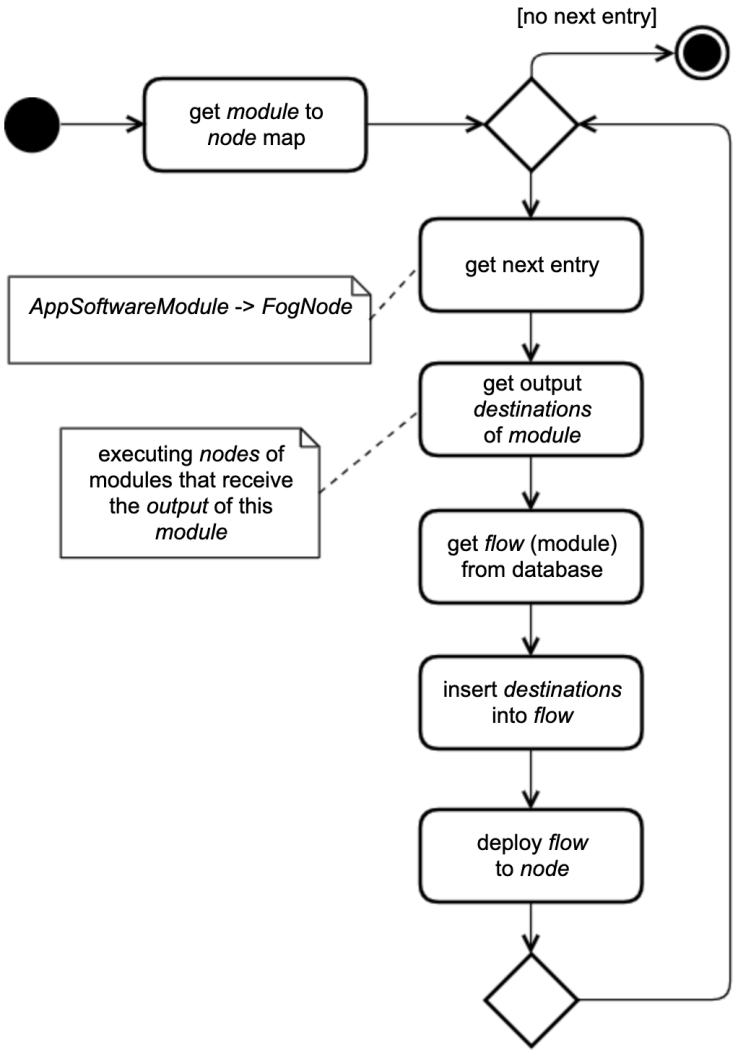

Figure 6.3.: Activity diagram for applying a deployment strategy to a Node-RED infrastructure

Communication

The communication between two application modules (Node-RED flows) is done by using the *HTTP nodes* available in Node-RED. For instance, the module `object-detector` accepts HTTP requests at the endpoint `/object-detection/object-detector` (module *input*), so that the module `camera-controller` (the previous module in the loop) can send its *output* via an `HTTP POST` request to that endpoint. However, *node A* must know that the output of `camera-controller` must be sent to *node B* to achieve this, so the Orchestrator places the address of *node B* in the `camera-controller` flow which it received from the database before it deploys this flow to *node A* (see figure 6.3).

6.4. Monitoring the Current Deployment Strategy

The Orchestrator must verify that the selected deployment strategy really meets the application requirements. Since the goal here is to get the result within a certain time, the last module in the loop must inform the Orchestrator how long it took to process the application loop. If the latency requirements cannot be fulfilled, the Orchestrator must be able to identify the problem and react to environmental changes.

To achieve this, every application module attaches the current time *before* and *after* processing to the message object (in other words: *after* receiving the message, and *before* sending a new message to the next module/node). The actual processing time on a node, as well as the message transfer time between two nodes can be calculated using these timestamps. Figure 6.4 illustrates where these timestamps are taken and how these are used to calculate the individual processing and transfer times.

Figure 6.4.: Timeline for executing the loop of the object detection application

The calculated values are stored in a `statistics` object which is attached to the message and hence passed through to the next module. The last module in the loop sends this object to the Orchestrator via `MQTT`, which can then further analyze this object.

The `statistics` object contains two arrays:

1. `transfers` for all message transfers between nodes
2. `processes` for all task executions on the nodes

An example is shown in listing 6.1.

```
{
  "transfers": [
    {
      "messageType": "IMAGE_ORIGINAL",
      "size": 1057046,
      "sourceNode": "nodeA",
      "time": 173,
      "destinationNode": "nodeB"
    }
  ],
  "processes": [
    {
      "process": "camera-controller",
      "node": "nodeA",
      "time": 10
    },
    {
      "process": "object-detector",
      "node": "nodeB",
      "time": 200
    }
  ],
  "totalTransferTime": 173,
  "totalProcessingTime": 210,
  "totalTime": 383
}
```

Listing 6.1: Sample statistics JSON object

Not only the entire execution time of a loop can be extracted from that object, but also the duration of every single message transfer and task execution. In the case of a worsening of a network connection that is used in the current deployment strategy, it is possible to determine between which nodes the connection has declined. Since the *size*, *transfer time*, *source node*, and *destination node* are known, the current bandwidth between those nodes can be calculated and the corresponding `NetworkUplink` instance can be updated accordingly. The Orchestrator can then run the algorithm with the updated values to see if there is a new optimal deployment.

7. Evaluation

In this chapter, the Orchestrator introduced in the previous chapter is tested and evaluated. Although two use cases were presented in chapter 3, only the object detection application is evaluated here. The sensor network use case was also tested with the system but is not described here because no additional findings were revealed.

In the first experiment, the nodes join and leave the network one after the other. The second experiment investigates how the system behaves upon network quality changes. For both experiments, the decisions the Orchestrator makes and the effect these decisions have on the total latency of the application are examined.

7.1. Evaluation Setup

7.1.1. Infrastructure

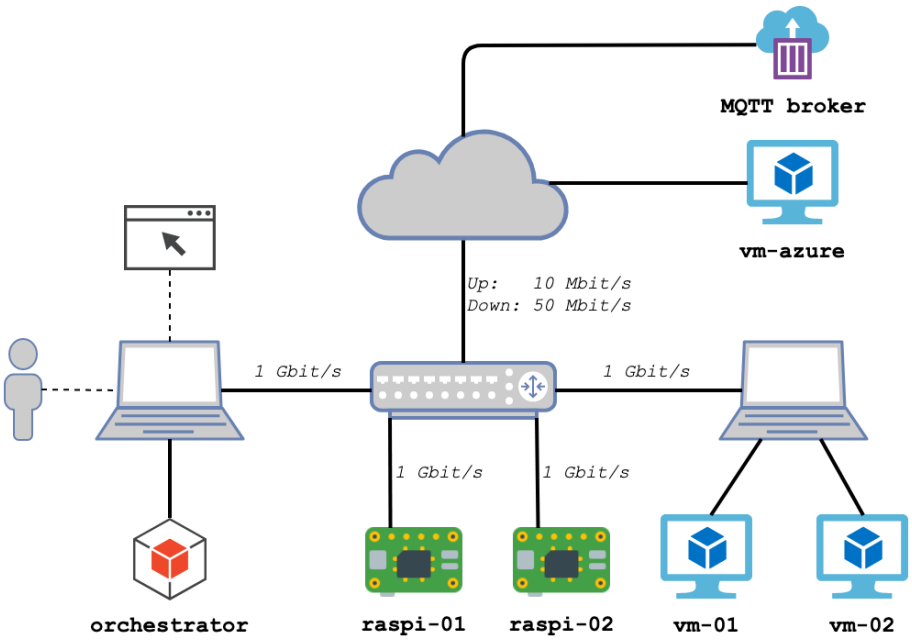

Figure 7.1.: Test infrastructure for evaluation

The test bed consists of the five nodes presented in figure 7.1. The hardware characteristics of the devices are shown in table 7.1. The two Raspberry Pis run Raspbian 10, the three virtual machines run Debian 10. All nodes have Docker version 19.03.2 installed. Another device on the local network runs the Orchestrator and a web

browser which is used to access the *Object Detection Web Application*. Communication between the nodes and the Orchestrator is done via a MQTT broker running in the Azure Cloud.

Node	Device Type	CPU	RAM
`raspi-01`	Raspberry Pi 3B+	ARM Cortex-A53 1.4 GHz, 4 cores	1 GB
`raspi-02`	Raspberry Pi 4B	ARM Cortex-A72 1.5 GHz, 4 cores	4 GB
`vm-01`	Hyper-V VM	Intel i7-8550U 1.8 GHz, 2 vCPUs	2 GB
`vm-02`	Hyper-V VM	Intel i7-8550U 1.8 GHz, 4 vCPUs	4 GB
`vm-azure`	Azure VM `F4s_v2`	Intel Xeon 2.7 GHz, 4 vCPUs	8 GB

Table 7.1.: Hardware characteristics of test devices used for evaluation

7.1.2. Application

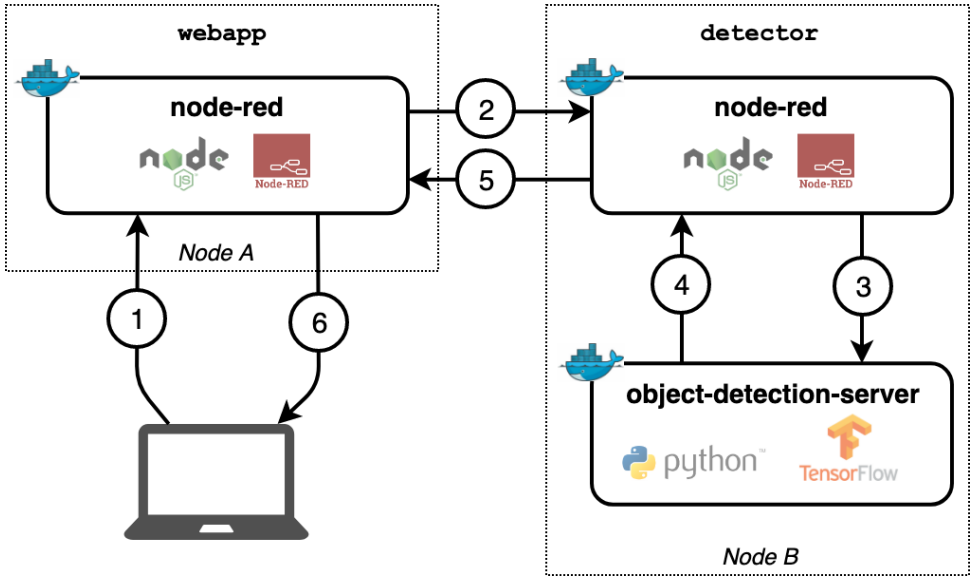

Figure 7.2.: System architecture of the Object Detection Web Application

The *Object Detection Web Application* consists of the two modules `webapp` and `detector` as illustrated in figure 7.2. A screenshot of the web application is shown in figure B.1.

In the first step, a user sends an image for object detection to the `webapp` module which is deployed as a Node-RED flow on `Node A`. This module should always be deployed on the same node and not be moved, so that the web application is always available at the same address. The user calls the web application on that fixed node while it is irrelevant for them which node actually executes the object detection task. The module `webapp` forwards the image to the module `detector` for object detection which is deployed on another node (`Node B`). This node can vary and is chosen by the Orchestrator. If a new node is selected by the Orchestrator, the flow

`webapp` on `Node A` will then be updated so that subsequent requests are forwarded to the new node.

The object detection engine cannot be executed as a regular Node-RED flow since Node-RED handles JavaScript functions only and the engine requires Python. Even if Node-RED could execute Python code (there are extensions enabling this), the execution of the object detection engine in Node-RED would not be efficient because the engine would have to be initialized every time a new object detection request arrives. However, the engine has a certain initialization time to load the detection graph. Therefore it is more efficient to initialize the engine only once at the beginning so that the following object detection tasks can be executed without additional initialization time. Because Node-RED cannot initialize and hold Python objects, a separate Docker container is used for this purpose. The flow `detector` sends an HTTP request containing the undetected image to the `object-detection-server` running on the same node. The server responds with the detected image which the `detector` then sends back to the `webapp` module.

The following three constraints are defined for the experiment:

1. The flow `webapp` must be deployed on `raspi-01` because this is the only node known to the user
2. The flow `detector` needs an `object-detection-server` Docker container running on the same node which is the case for all nodes except for `raspi-01`
3. The detected image must be available within 5 seconds

7.2. Preparing the Orchestrator

For evaluating the Orchestrator, nodes with different CPU's performance are necessary. However, one vCPU in `vm-01` is equally as fast as in `vm-02` because they both run on the same host. Since the object detection engine uses only one core, the execution time on these two nodes is approximately the same, although `vm-01` has two cores and `vm-02` has four cores. For this reason, the Docker containers on `vm-01` run with the additional option `--cpus="0.5"` which limits the available CPU resources a container can use to a half vCPU.

7.2.1. Possible Deployment Strategies

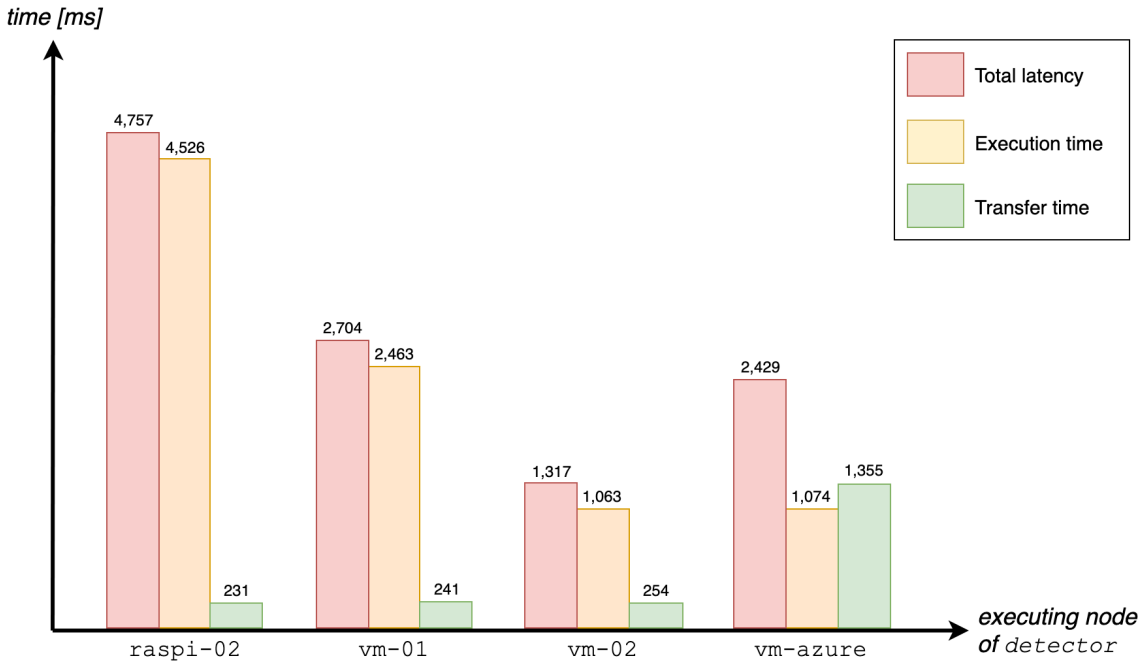

Figure 7.3.: Total latencies, execution times, and transfer times of the object detection application regarding the placement of module detector

To find the actual execution times of all possible deployment strategies, the application was deployed to the infrastructure without using the Orchestrator. The module `webapp` was always deployed on node `raspi-01` while the module `detector` was switched through the remaining nodes. The object detection application was executed 10 times on each deployment strategy by using a 1,383 KB sized image. The mean values for *total latency*, *processing time* and *transfer time* were calculated thereafter. These are compared in figure 7.3. Based on the measurement results, a prioritization of all possible deployments was created where deployments with a lower latency were preferred (see table 7.2).

Priority	Placement of `webapp`	Placement of `detector`	Latency
1	`raspi-01`	`vm-02`	1,317 ms
2	`raspi-01`	`vm-azure`	2,429 ms
3	`raspi-01`	`vm-01`	2,704 ms
4	`raspi-01`	`raspi-02`	4,757 ms

Table 7.2.: Possible deployment strategies for the object detection application

7.2.2. Defining the Application

Although the Orchestrator handles the infrastructure dynamically, an application must be defined. Therefore, an `Application` object is created during the initializa-

tion of the Orchestrator. It contains all modules, loops and messages of the object detection application described in section 7.1.2.

Setting required RAM and disk storage

The fields `requiredRam` and `requiredStorage` of the two modules `webapp` and `detector` are set to `0` because both Node-RED flows are relatively small and thus consume neither significant RAM nor storage. Although the `object-detection-server` container requires 1GB RAM, this memory is not allocated/released by deploying/removing the flow to/from Node-RED because the container runs even if the Node-RED flow is not deployed on a node. Furthermore, the container does not require any additional disk storage while running the engine.

Setting required hardware modules

The field `requiredHardwareModules` of the module `webapp` contains one string: `"CAMERA"`. The `node-red` Docker container on `raspi-01` is started with the additional option `-e CONNECTED_HARDWARE=["CAMERA"]` so that `webapp` can only be deployed on `raspi-01`. The same approach is used for the module `detector`, but here the string `"OD-DOCKER-CONTAINER"` is used and passed to the Docker container on the remaining nodes instead.

Setting required instructions

The field `requiredMi` of the module `webapp` is set to `0` because the Node-RED flow simply forwards messages and therefore does not require any significant CPU power. However, CPU resources consumed by the module `detector` are substantial. Since the Orchestrator uses a CPU score instead of MIPS, the value for `requiredMi` has to be calculated using the formula introduced in section 6.2.2. Values for *CPU score* and *execution time* are taken from `vm-02`, which are *11,661* and *1,317 ms*, respectively. Therefore:

$$\texttt{requiredMi}_{\texttt{detector}} = \text{CPU score} \cdot \frac{\text{execution time [ms]}}{1,000\ \text{[ms]}} = 11,661 \cdot \frac{1,317}{1,000} = 15,357$$

Setting the loop and messages

The application has exactly one loop (instance of `AppLoop`) where the field `modules` is populated with a list containing three elements in the following order: `"webapp"`, `"detector"`, `"webapp"`. The field `maxLatency` is set to `5000`.

Furthermore, the application contains two messages (instances of `AppMessage`). The first one contains the original image, which in this case has a file size of 1,383 KB. Therefore, the fields `contentType` and `dataPerMessage` are set to `"IMAGE_UNDETECTED"` and `1383`, respectively. This is the input message to the module `detector`. The second message contains the detected image, which has a file size of

142 KB. Therefore, the fields are set to `"IMAGE_DETECTED"` and `142`, respectively. This is the output message of the module `detector`.

7.3. Experiment 1: Consistent Network Quality

In this experiment, the nodes were started in the following order:

1. `raspi-01`
2. `raspi-02`
3. `vm-01`
4. `vm-azure`
5. `vm-02`

This order allows to apply the deployment strategy of lowest priority (priority four) in step two, while a strategy of higher priority is possible with each following step (see table 7.2).

It was expected that the Orchestrator finds no valid deployment in step one because solely `raspi-01` is online which is unable to execute the module `detector`. Another expectation was that the Orchestrator deploys the module `webapp` to `raspi-01` and `detector` to `raspi-02` in the second step and then moves the `detector` to `vm-01`, `vm-azure` and finally to `vm-02` in the subsequent steps, while `webapp` remains on `raspi-01`.

Results

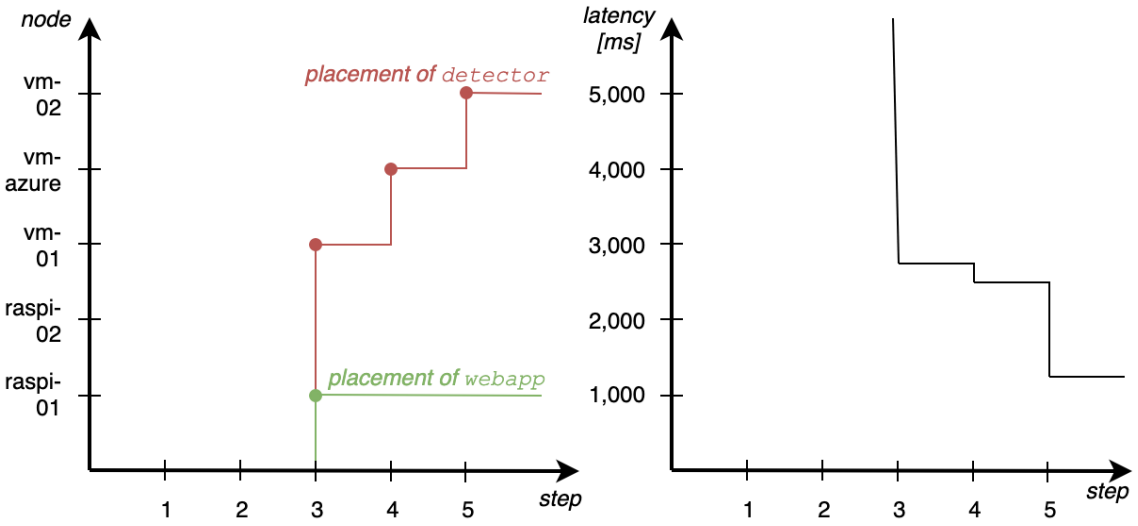

Figure 7.4.: Result of Experiment 1: Orchestrator's decisions and their impact on total latency

The Orchestrator's decisions are depicted in figure 7.4. The figure shows that the Orchestrator behaved slightly different than expected. The `detector` was expected

to be deployed to `raspi-02` in the second step. However, this did not happen. Instead, no possible deployment was found. In steps 1, 3, 4 and 5, however, the Orchestrator made the expected decisions and the latency is minimized continuously from step 3 onwards.

Moreover, it was examined how the Orchestrator behaves when a node leaves the network. If a node that has *no* module deployed leaves the network, the Orchestrator detects this and removes it from the infrastructure. The deployment strategy remains unchanged as it should be. If a node that *has* a module deployed leaves the network, the Orchestrator detects this and removes it from the infrastructure first before it then finds and applies the next best possible deployment strategy (next best deployment strategy is the one of highest priority in the updated infrastructure). This complies with the desired behaviour.

Investigation of the Orchestrator's incorrect decision

The reason for the wrong decision in step two is that the execution time of module `detector` on node `raspi-02` calculated by the Orchestrator is incorrect. The field `cpuMips` of the `FogNode` object representing this node has the value 973. This value is the CPU score returned by the `benchmark_cpu` command which the Orchestrator executed on the node after the first heartbeat has been received (see section 6.2 and 6.2.2 in particular). To calculate the execution time of module M on node N, the Orchestrator uses `cpuMips` of N and `requiredMi` of M. While preparing the Orchestrator in section 7.2, $\texttt{requiredMi}_{\texttt{detector}}$ was set to 15,357. Thus, the execution time is calculated as follows:

$$\text{execution time [ms]} = \frac{\texttt{requiredMi}}{\texttt{cpuMips}} \cdot 1,000 = \frac{15,357}{973} \cdot 1,000 = 15,783 \text{ [ms]}$$

The calculated execution time is approximately 3.5 times greater than actual execution time of 4,526 milliseconds (see figure 7.3). Furthermore, it is greater than 5,000 milliseconds, which is the `maxLatency` of the opject detection loop. Therefore, the Orchestrator finds no valid deployment strategy.

The `benchmark_cpu` command internally uses the CPU benchmarking tool `sysbench` which searches for all prime numbers between 1 and a certain upper limit (2000 in this case) and returns the time needed to execute this task. Apparently, this time cannot be used to deduce the execution time of other types of tasks like object detection. To further investigate this, the simple JavaScript function shown in listing 7.1 was executed in Node-RED on `raspi-02` and `vm-02` while no other CPU intensive processes were running.

```
const timeStart = new Date();
let result = 0;
for (let i=0; i < Math.pow(10,7); i++) {
    result = result + i;
}
const timeEnd = new Date();
```

```
const time = timeEnd - timeStart;
```

Listing 7.1: JavaScript function which executes simple mathematical tasks while measuring the total execution time

The field `time` of that function contains the execution time in milliseconds which is approximately 2,500 on `vm-02` and 7,500 on `raspi-02`, respectively. Thus, the execution of the function is about *3 times faster* on `vm-02` than on `raspi-02`, even though the CPU score of `vm-02` (11,848) is about *12 times greater* than the score of `raspi-02` (973). Additionally, the object detection task takes about 1,063 milliseconds on `vm-02` and 4,757 milliseconds on `raspi-02`, which is a *4,2-fold difference* (see figure 7.3). Therefore, a correlation between the execution times of different tasks on various nodes cannot be recognized.

However, there might be a correlation between CPUs that share the same architecture, but this is not the case either: Executing the function shown in listing 7.1 takes about 17,800 milliseconds on `raspi-01`, which is *2.37 times slower* than on `raspi-02`, although the CPU score of `raspi-01` (612) is only *1.59 times smaller* than on `raspi-02`. Both nodes own a CPU of type `arm`.

In conclusion, the execution time of a given task cannot be used as the basis to compare CPU performances of different devices. There seem to be a number of relevant factors affecting the execution time of a task, but this goes beyond the scope of this work.

7.4. Experiment 2: Volatile Network Conditions

In this experiment, the three nodes `raspi-01`, `vm-01` and `vm-azure` were online. Therefore, deployment strategy of *priority two* allowed the lowest latency (see table 7.2). Here, the `detector` module is deployed on `vm-azure`. In further course, the router's internet connection was utilized by other hosts on the network that are not part of the fog infrastructure. After a certain period, the bandwidth was freed again.

It was expected that the Orchestrator moves the `detector` module from `vm-azure` to `vm-01` after the usage of the internet connection exceeds a certain threshold. This is the case as soon as the application no longer has enough bandwidth to meet its latency requirements. After the full bandwidth is available again, the `detector` should be moved back to `vm-azure` by the Orchestrator as soon as possible.

Results

The Orchestrator's decisions are depicted in figure 7.5. Initially, it applied deployment strategy of priority two as expected. The object detection task in request one was finished after 2,370 milliseconds, while the `detector` was executed on `vm-azure`.

Between request one and two, network traffic between a host in the local network and another host on the internet was generated. Both hosts were not part of the fog infrastructure, but they did utilize the same internet connection that was used for

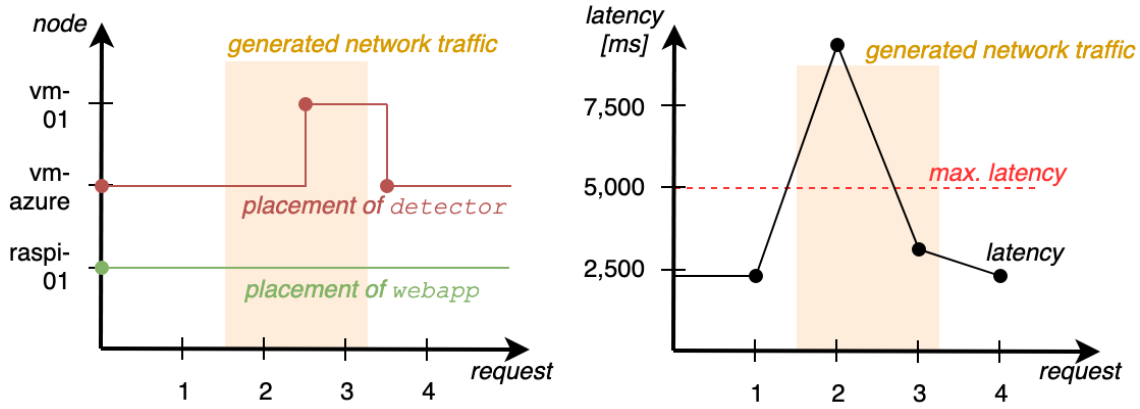

Figure 7.5.: Result of Experiment 2: Orchestrator's decisions and their impact on total latency

transferring the original image from `webapp` (deployed on `raspi-01`) to `detector` (deployed on `vm-azure`) as well. That is why the maximum latency requirement of 5,000 milliseconds could not be met in request two. Although the object detection task on `vm-azure` took approximately the same time in both requests, the time for transferring the original image (1,382 KB) took 7,641 milliseconds instead of 1,192 milliseconds in request one, so that the total latency added up to 8,921 milliseconds.

The Orchestrator recognized this by evaluating the `statistics` object (see listing 6.1) introduced in section 6.4. Based on the image size and transfer time, it calculated the actual bandwidth between both nodes, which is *1.48 Mbit/s* in this case. The respective `NetworkUplink` object was updated and the scheduling algorithm was executed to find a new optimal deployment strategy. This is what happened between request two and three, before the `detector` was moved to `vm-01`. The latency requirement was fulfilled in request three, although the total latency of 2,671 milliseconds is a bit higher than in request one.

As said before, the scheduling algorithm was triggered during the evaluation of the `statistics` object in this case. However, the Orchestrator executes the algorithm in a predefined interval as well. Since the uplinks are also remeasured in a predefined interval by the Orchestrator, the scheduling algorithm might find a new optimal deployment strategy. This is what happened between request three and four: The network traffic generation has been stopped and the uplink between `raspi-01` and `vm-azure` has been remeasured. The updated bandwidth was taken into account during the subsequent execution of the scheduling algorithm, so that the Orchestrator switched back to the deployment strategy of priority two which led to a lower latency in request four.

It should be added that there might be a new optimal deployment strategy in case of a downgrade of an uplink, even though the new strategy might have a higher latency than the previous one. This could be the case if an uplink that is involved in the current deployment strategy deteriorates. Nevertheless, the Orchestrator always applies the deployment strategy which allows the lowest latency to the best of his knowledge.

7.5. Discussion

The evaluation showed that the Orchestrator developed in this thesis is fundamentally able to dynamically apply different deployment strategies to a Node-RED environment resulting in an optimized quality of service.

The measurements of the network connections between the nodes produced realistic results, so the Orchestrator could estimate transfer times of messages fairly well. Calculating the execution time of tasks has partly delivered unrealistic results, so this estimation does not work reliably and can lead to wrong decisions as happened in the first experiment. The reason for this is the method used by the Orchestrator to measure and compare the CPU performance of different devices. A program is run on every new fog node joining the infrastructure, while the execution time is measured. However, the execution time of one task cannot be set in relation to the execution time of another task.

In the first experiment, the values of `vm-02` were used as the basis for calculating $\texttt{requiredMi}_{\texttt{detector}}$. Therefore, the execution time of the `detector` module could be calculated relatively precisely on `vm-02`. The same was true for `vm-01`, since both nodes shared the same CPU, although the CPU of `vm-01` was limited. If the values of `raspi-02` had been used to calculate $\texttt{requiredMi}_{\texttt{detector}}$ instead, the execution time of `detector` on `raspi-02` would have been estimated correctly, but the estimated execution time on other nodes would have been wrong.

In the second experiment, the deployment strategy changed *after* failing to process a request within the required time. This could have been avoided by setting lower intervals for checking uplinks and deployment strategies, hence the increased network traffic after request one would have been detected earlier. Therefore, the Orchestrator would have changed the deployment strategy *before* request two, so the maximum allowed latency would not have been exceeded. However, the uplink measurements generate network traffic themselves, so this interval has to be chosen with caution. If set too low, the measurements would constantly use bandwidth, leading to higher latency of the deployed services.

Although the system enables the QoS-aware task execution on distributed Node-RED clusters, there are some expansion and improvement capabilities of the system which are discussed in the next and last chapter.

8. Conclusion and Future Work

The purpose of this work was to enable QoS-aware task execution on distributed Node-RED clusters for fog computing environments. Therefore, a system that dynamically reacts to changes in the fog network had to be designed and implemented in order to achieve the lowest possible service latencies.

Initially, a literature research was conducted to obtain the current state of research about fog computing, QoS, resource allocation and self-adaptive systems. In the next step, use cases which would benefit from such a system were introduced before describing the technical and theoretical challenges.

Based on the preceding work, a QoS scheduling and resource allocation algorithm was designed and implemented. However, this algorithm had to be used in a dynamic fog environment. Therefore, an infrastructure monitoring system was implemented next. To apply the deployment strategy determined by the algorithm, functionalities for controlling Node-RED instances have been added to the system.

The evaluation has shown that the system is able to dynamically apply varying deployment strategies to the infrastructure depending on the fog environment's current state. Overall, the results are satisfying, although improvements are possible.

The task execution time could not always be estimated reliably, so incorrect decisions by the system are possible. It was found that the technique used to compare CPUs of different devices is not suitable for this purpose. The system would benefit from further research on this topic: How to reliably predict the execution time of different tasks on different devices? For instance, this could be achieved by implementing a machine learning algorithm which predicts the execution time of a specific module on a specific node based on historical data.

While evaluating the system, the same image was always sent to the object detection service, so the incoming data size was equal throughout all experiments. However, incoming data may vary in size and content, thus affecting the execution time as well as the transfer time. The system's decision making process could be supported if metadata of incoming data could be predicted by a machine learning algorithm.

Another interesting question is how the system behaves with an increasing amount of nodes. Since any network connection from one node to another is remeasured at a certain interval, the network may congest if the amount of measurements exceed a certain point because each measurement takes time and utilizes bandwidth that cannot be used by users' service requests during the measurement. The system could be improved here by checking only certain connections, skipping the ones that are unlikely to be used in a deployment strategy.

Ideally, no network measurements are performed. This could be realized by another

layer on top of the network. For instance, information about the network's current load could be provided by the routers (gateway routers in particular) that are part of the infrastructure.

The algorithm developed in this thesis is not limited to be used in a Node-RED infrastructure. For instance, it could also be used to deploy and move Docker containers instead of Node-RED flows.

Bibliography

[KC03] J O Kephart and D M Chess. "The vision of autonomic computing". In: *Computer* 36.1 (2003), pp. 41–50.

[ARS15] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. "Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation". In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (2015).

[CZ16] Mung Chiang and Tao Zhang. "Fog and IoT: An Overview of Research Opportunities". In: *IEEE Internet of Things Journal* 3.6 (2016), pp. 854–864.

[He+16] Xiuli He et al. "A novel load balancing strategy of software-defined cloud/fog networking in the Internet of Vehicles". In: *China Communications* 13.2 (2016), pp. 140–149.

[Bit+17] Luiz F Bittencourt et al. "Mobility-Aware Application Scheduling in Fog Computing". In: *IEEE Cloud Computing* 4.2 (2017), pp. 26–35.

[BF17] Antonio Brogi and Stefano Forti. "QoS-Aware Deployment of IoT Applications Through the Fog". In: *IEEE Internet of Things Journal* 4.5 (2017), pp. 1185–1192.

[BFI17] Antonio Brogi, Stefano Forti, and Ahmad Ibrahim. "How to Best Deploy Your Fog Applications, Probably". In: *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)* (2017).

[Gup+17] Harshit Gupta et al. "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments". In: *Software: Practice and Experience* 47.9 (2017), pp. 1275–1296.

List of Figures

List of Tables

Listings

A. UML diagrams

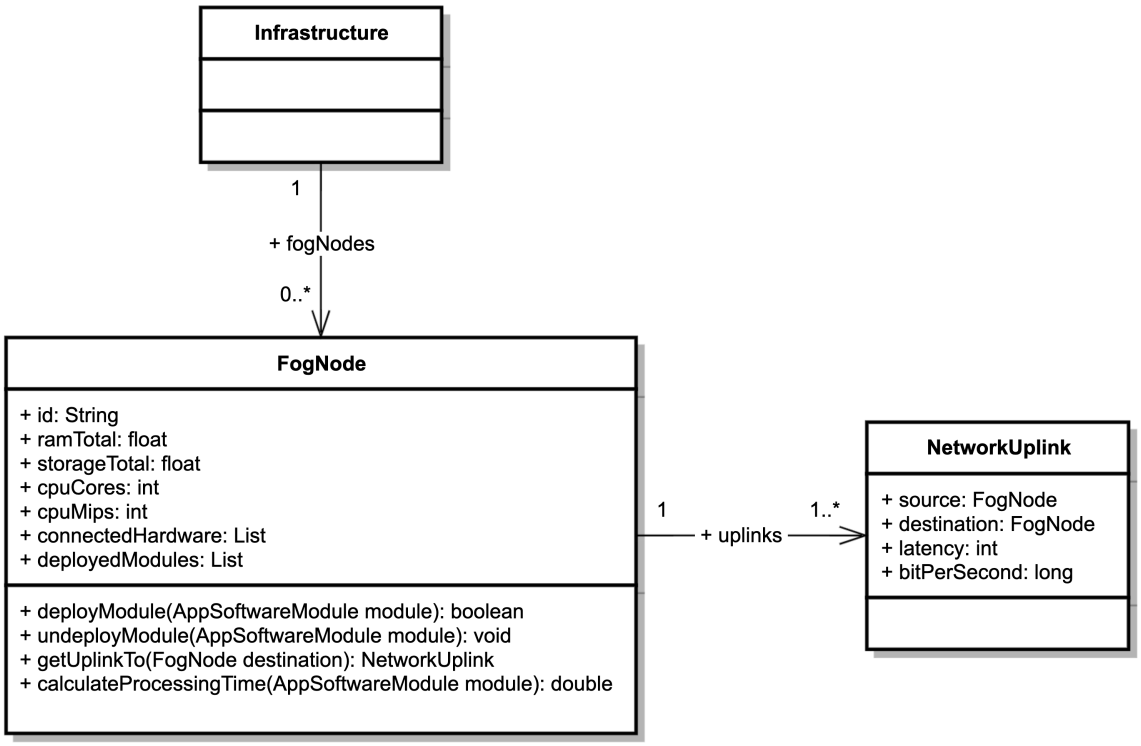

Figure A.1.: Class diagram representing the scheduling algorithm's infrastructure

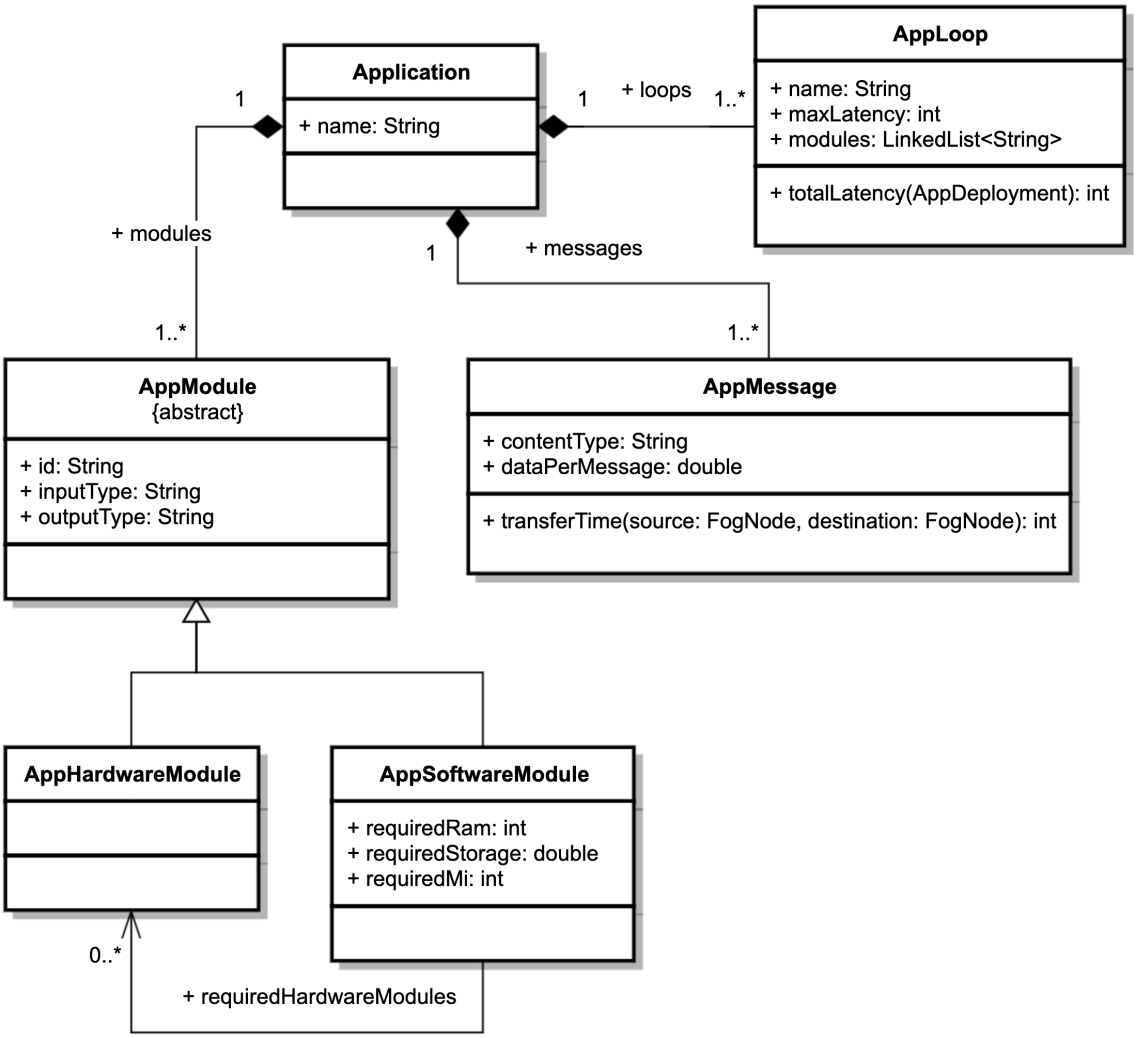

Figure A.2.: Class diagram representing the scheduling algorithm's application

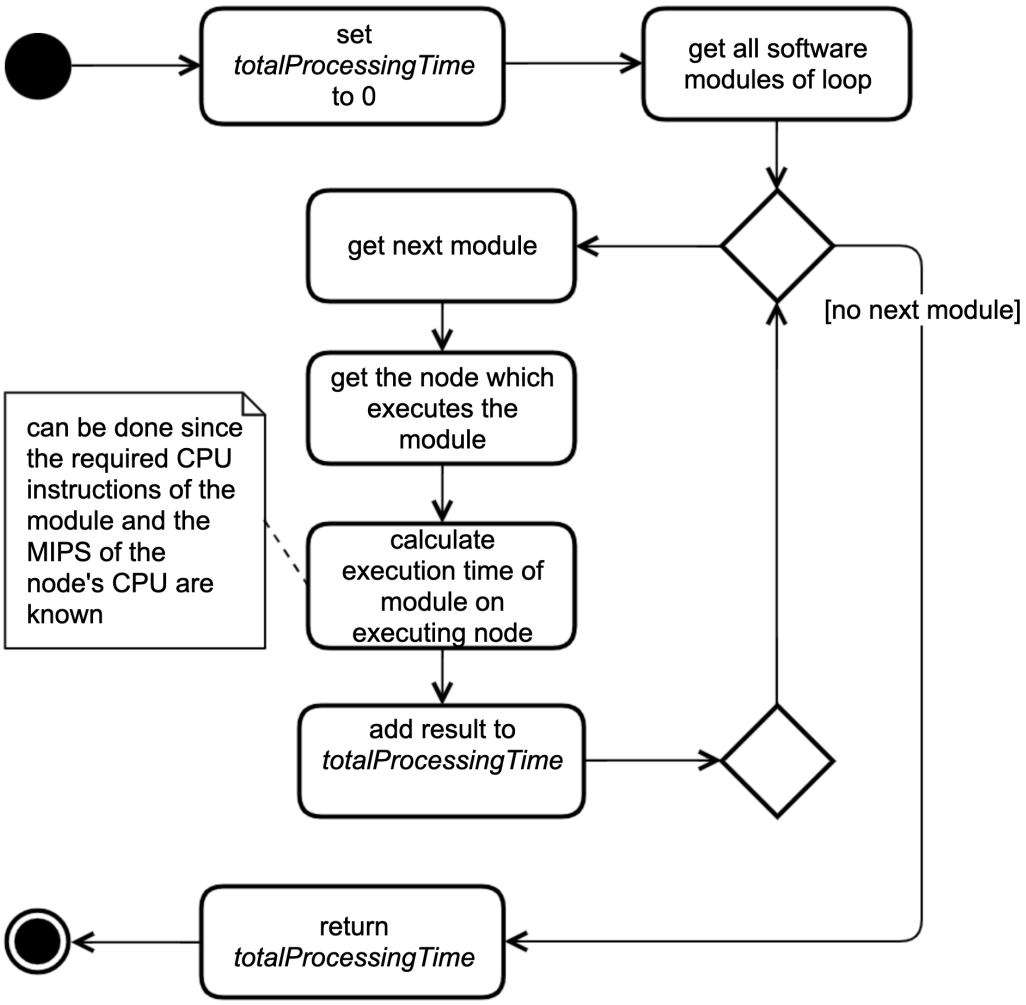

Figure A.3.: Activity diagram for calculating the total task execution time of a loop

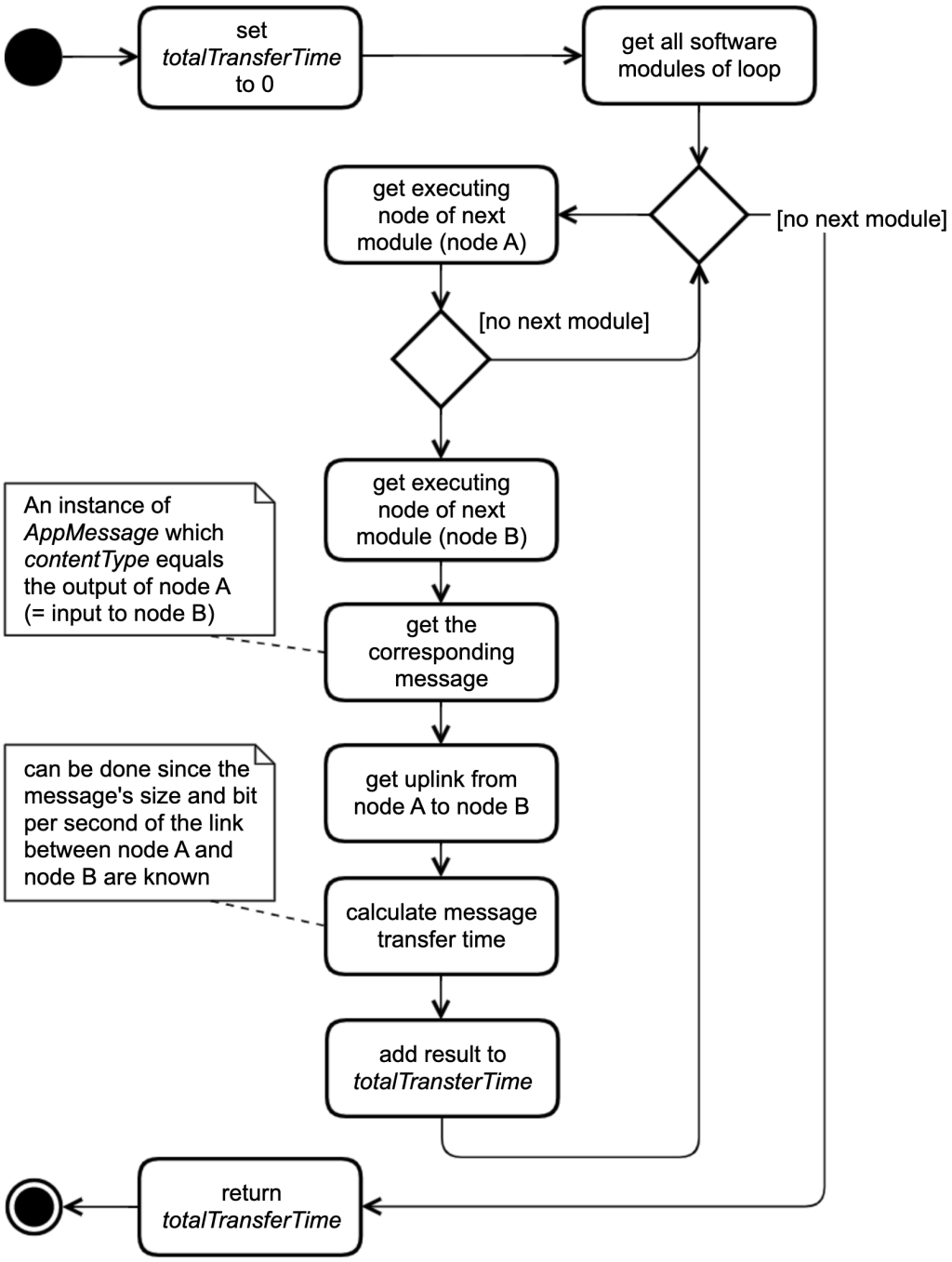

Figure A.4.: Activity diagram for calculating the total transfer time of messages between nodes of a loop

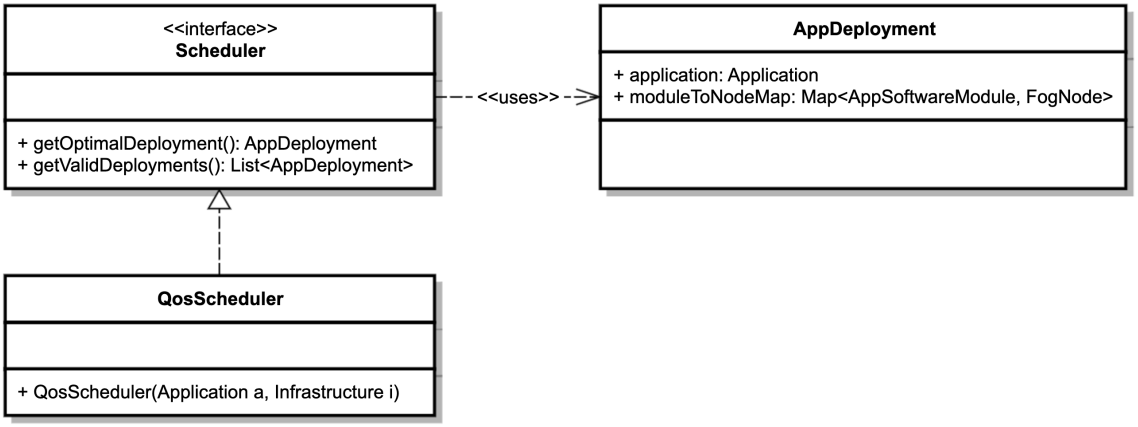

Figure A.5.: Class diagram representing the QoS scheduler

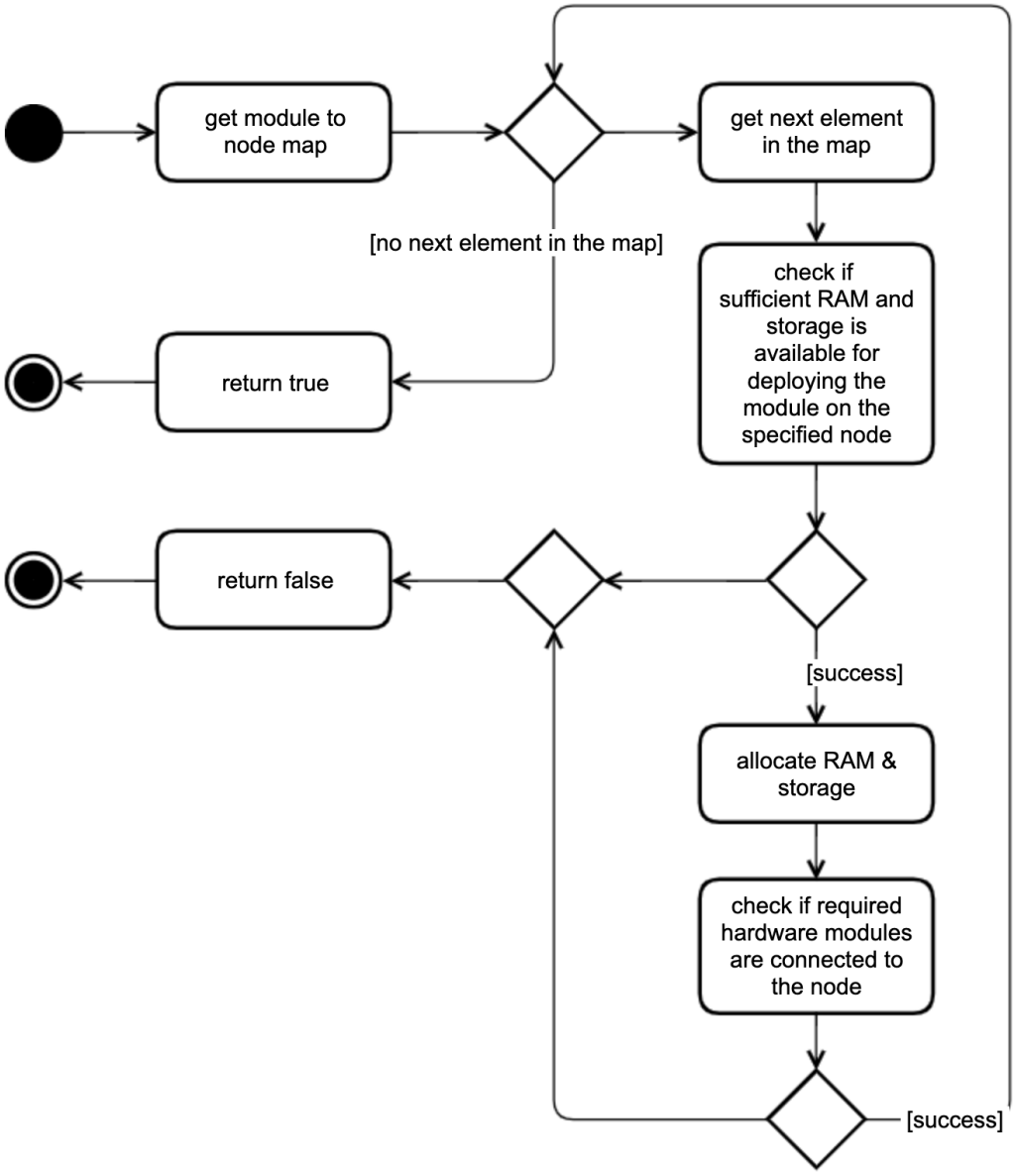

Figure A.6.: Activity diagram for validating hardware requirements

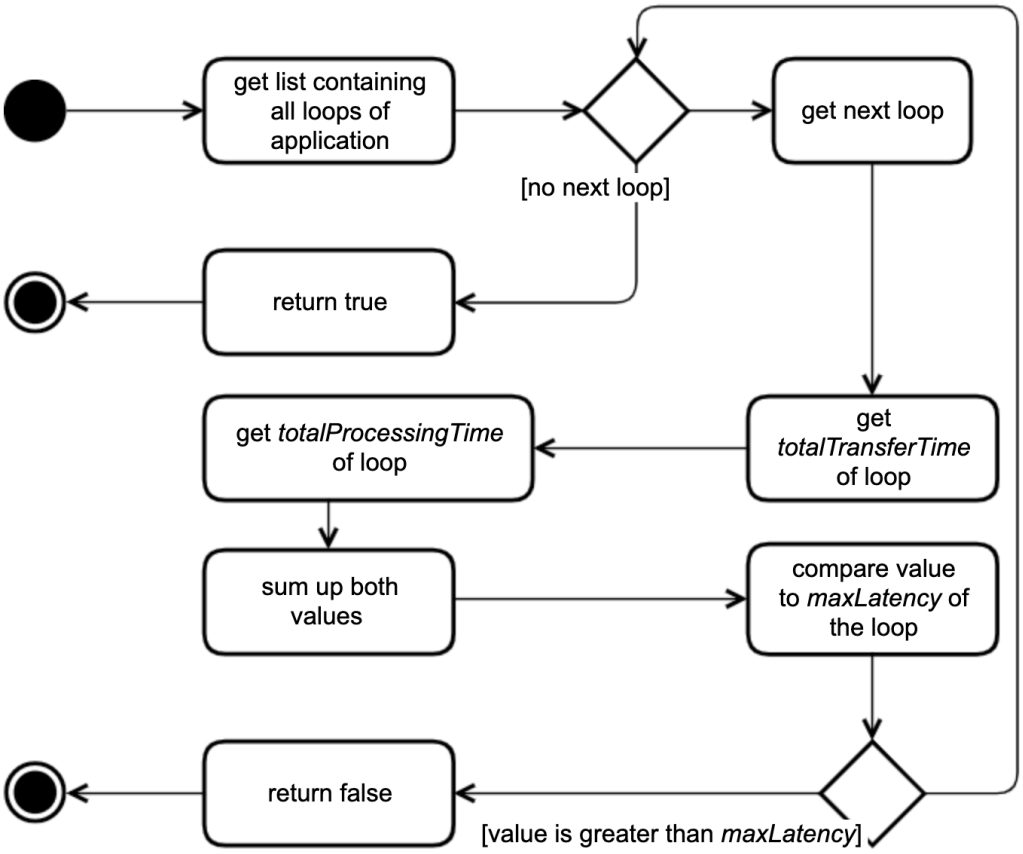

Figure A.7.: Activity diagram for validating latency requirements

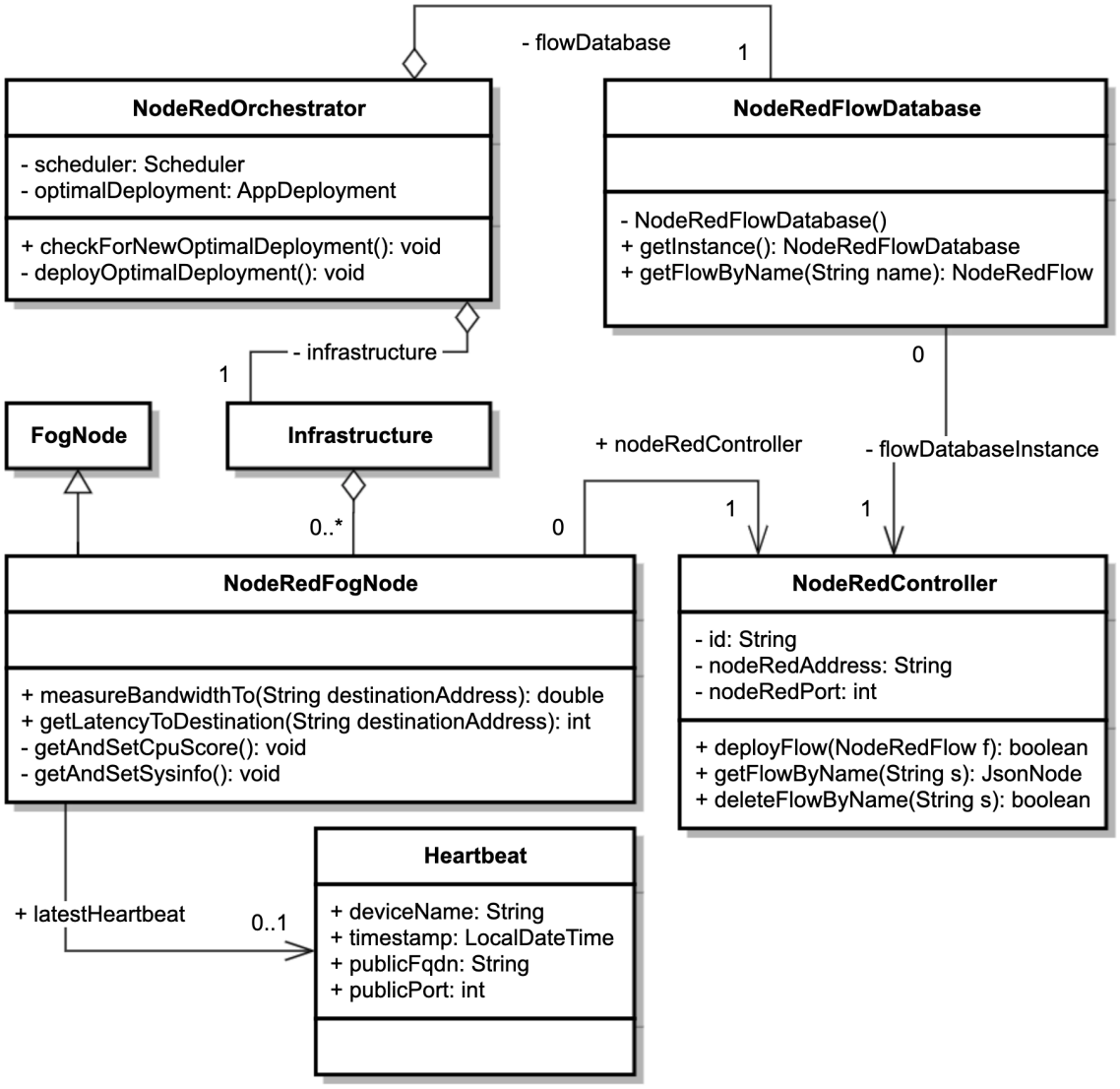

Figure A.8.: Class diagram of the Orchestrator

B. Screenshots

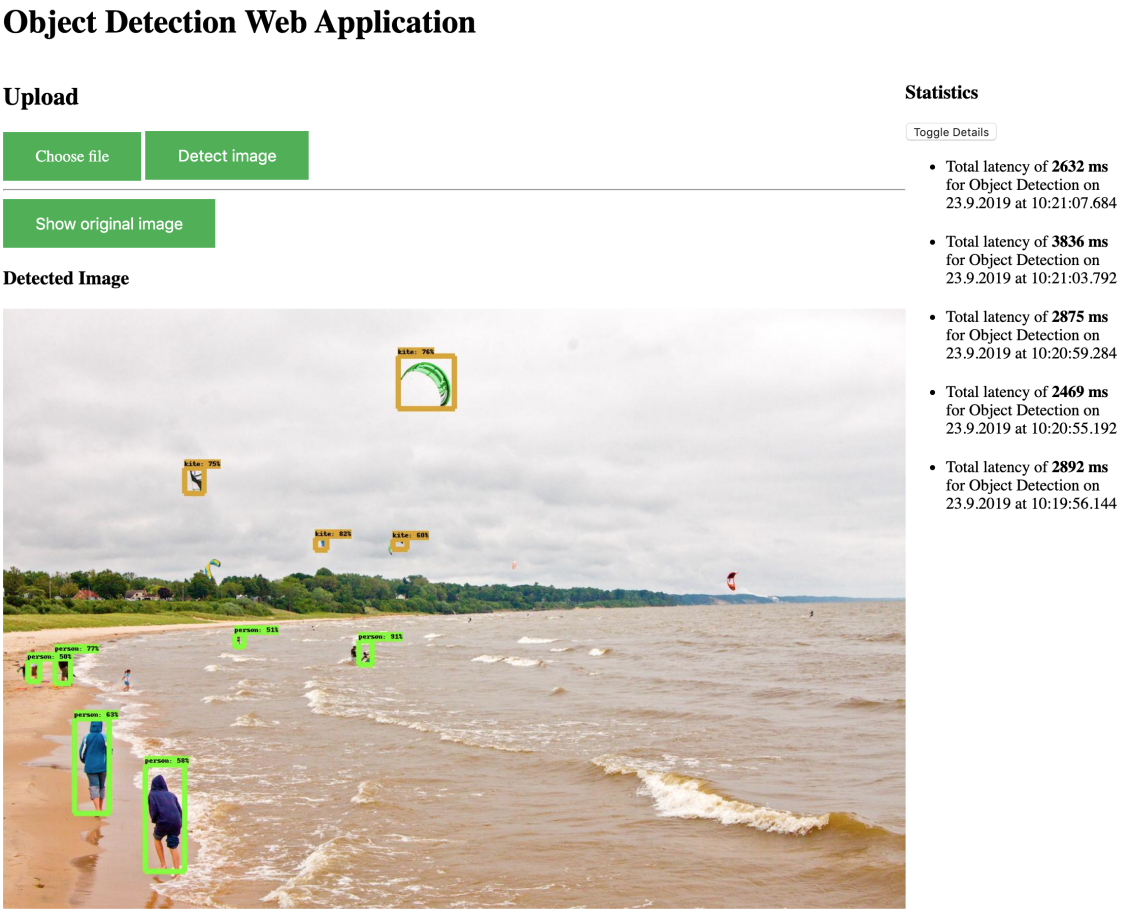

Figure B.1.: Screenshot of the Object Detection Web Application