UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Dynamic deployment of modular distributed stream processing applications

Diploma Thesis

Angelis Marios

Supervisor: Christos Antonopoulos

Volos 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Dynamic deployment of modular distributed stream processing applications

Diploma Thesis

Angelis Marios

Supervisor: Christos Antonopoulos

Volos 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Δυναμική ανάπτυξη αρθρωτών κατανεμημένων εφαρμογών επεξεργασίας ροών δεδομένων

Διπλωματική Εργασία

Αγγέλης Μάριος

Επιβλέπων: Χρήστος Αντωνόπουλος

Βόλος 2021

Approved by the Examination Committee:

Supervisor	**Christos Antonopoulos** Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Spyros Lalis** Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Dimitrios Katsaros** Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval: 29-9-2021

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Christos D. Antonopoulos. He was always available, feeding me with valuable advice. The guidance, the continuous support and the encouragement he provided will be unforgettable. His willingness to offer me so much of his time and intellect is the major reason this Thesis was completed.

I would also like to show my warmest appreciation to Professor Spyros Lalis who helped me to overcome my virtual limits and inspired me to become a more competent engineer. His active and enthusiastic involvement in the projects we collaborated on, helped me change my view of the engineering world. Also, I am profoundly grateful to Professor Dimitrios Katsaros for being a member of the examination committee of my thesis.

To my dear friends, I want to thank you for your support and for all the amazing moments we lived together during this 5-year wonderful experience. Especially, I am grateful to Thodoris because our collaboration urged us to enrich our personalities and improve our level of engineering. Finally, I would like to thank my family for their endless love, encouragement, and support throughout my studies.

This thesis is heartily dedicated to my uncle who took the lead to heaven before the completion of this work.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Abstract

The rise of the edge/fog/cloud computing model has increased the programming complexity of the designed applications. Trying to deploy such applications in the new distributed and heterogeneous system landscape, latency, scalability, heterogeneity and security issues have arisen. In this Thesis, we examine a structured dataflow approach that simplifies application development and offers great flexibility concerning the deployment of the application across the edge/fog/cloud system. We implemented a fully functional system that is responsible for the deployment of modular applications in a heterogeneous cluster spanning the edge/fog/cloud system. We provide deployment monitoring, scaling and migration functionalities offering flexibility and transparency. We use real-world application examples to illustrate our approach as well as to evaluate the performance trade-offs for various deployment scenarios on a real distributed cluster.

Abstract

Η άνοδος του edge/fog/cloud computing έχει αυξήσει την πολυπλοκότητα προγραμματισμού των εφαρμογών. Προσπαθώντας να αναπτύξουμε τέτοιες εφαρμογές στο νέο τοπίο των κατανεμημένων και ετερογενών συστημάτων, έχουν επέλθει ζητήματα καθυστέρησης, επεκτασιμότητας, ετερογένειας καθώς και ασφάλειας. Σε αυτή τη διπλωματική εργασία, διερευνάμε μια προσέγγιση δομημένης ροής δεδομένων, η οποία απλοποιεί την ανάπτυξη εφαρμογών και προσφέρει μεγάλη ευελιξία όσον αφορά στην ανάπτυξη των εφαρμογών στο σύστημα edge/fog/cloud. Υλοποιήσαμε ένα πλήρως λειτουργικό σύστημα, το οποίο είναι υπεύθυνο για την ανάπτυξη (deployment) αρθωτών εφαρμογών σε ένα ετερογενές σύμπλεγμα που εκτείνεται στο σύστημα edge/fog/cloud. Παρέχουμε λειτουργίες παρακολούθησης της ανάπτυξης, κλιμάκωσης και μετανάστευσης προσφέροντας ευελιξία και διαφάνεια. Χρησιμοποιήσαμε παραδείγματα εφαρμογών του πραγματικού κόσμου για να επεξηγήσουμε την προσέγγισή μας καθώς και για να αξιολογήσουμε τις συνέπειες στην απόδοση για διάφορα σενάρια ανάπτυξης σε μια πραγματική, κατανεμημένη συστάδα υπολογιστικών συστημάτων.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Cloud computing is undeniably one of the most discussed technologies nowadays. Cloud computing is the delivery of computing services including servers, storage, databases, networking, software, and analytics over the Internet to offer faster innovation, flexible resources and economies of scale. It consists of the latest generation of fast and efficient computing hardware, designed to offer effective and reliable solutions. Furthermore, the cloud is flexible and it allows scaling up or down on demand, giving businesses the potential for more cost-effective solutions. The explosive growth and increasing computing power of IoT devices have resulted in unprecedented volumes of data. In addition, data volumes will continue to grow as 5G networks increase the number of connected mobile devices. Sending all that device-generated data to a centralized data center or the cloud causes bandwidth, latency and scalability issues. Therefore, it may not be possible to support control/feedback loops with tight real-time constraints. Last but not least, security issues have arisen because sending all this private/sensitive data to the cloud makes them vulnerable to attacks.

However, writing applications that span this edge-fog-cloud system is not a trivial task. To take advantage of all existing layers, the application should be divided into different parts called components. Each component should be designed to run on a specific target host and the interfaces through which the components are interconnected should be designed and programmed accurately. Finally, each component must be installed on a target host and instantiated properly, for the application to start in the desired way. This process should be repeated from scratch in the case of a different deployment. Also, heterogeneity issues arise especially if the design of a component requires the absolute knowledge of the target host's hardware.

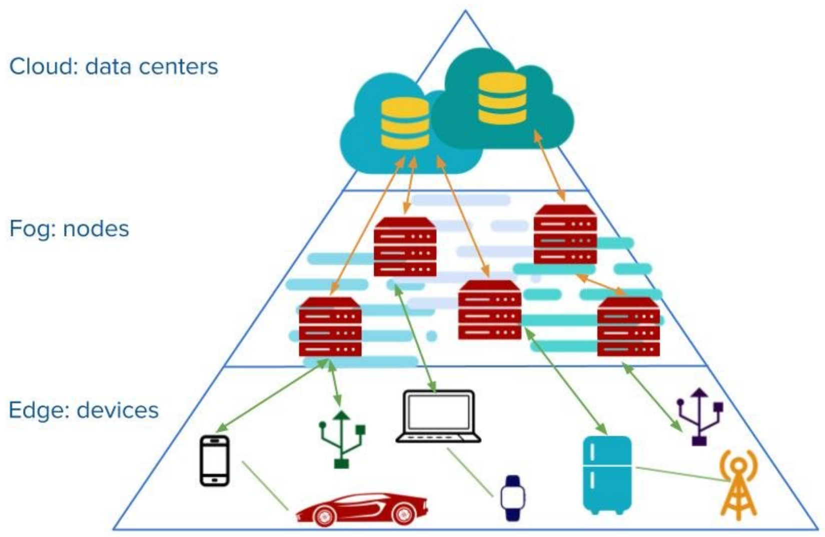


Figure 1.1: Edge-Fog-Cloud architecture [1]

1.2 Contribution

This Thesis introduces a more flexible way to design and deploy such an application, by adopting a combination of component-based and data-flow-oriented programming. The application is described by a graph that consists of nodes, each one representing a component, and from unidirectional links used for data exchange between the components. We have simplified the graph representation process by enabling the developer to describe the components and their interconnections using a specially formatted configuration file. Furthermore, we have designed and implemented a cluster registration service for the dynamic control of a fleet consisting of heterogeneous devices positioned on the cloud-fog-edge system as well as a system responsible for deploying the component-based application based on user deployment preferences and requirements. We provide support for placement preferences, hardware and sensor requirements, link affinities, and other useful features for the desired deployment. At deployment time, the components are instantiated on the target hosts, along with automatically generated connector logic that takes care of component binding and communication over the network. We provide a network API that takes care of the connections' establishment, termination, and control as well as the data transfer between the components, eliminating the complexity of each component's interface programming process. Last but not least, we provide scaling and migration functionalities for improving performance and reducing the communication latency between components.

1.3 Thesis Structure

The rest of the thesis is structured as follows:

- **Chapter 2** provides background information presenting the main tools used in this Thesis.
- **Chapter 3** provides a brief description of the key components of the designed system as well as the functionalities implemented.
- **Chapter 4** provides an extensive analysis of the implementation process of each component of the system of as well as an extended description of the core system's functionalities.
- **Chapter 5** shows the results of our performance experiments.
- **Chapter 6** gives an overview of related work.
- **Chapter 7** provides a conclusion of this Thesis.

Chapter 2

Background

2.1 Docker

Docker [3] is an open platform for developing, shipping, and running applications. It provides the ability to package and run an application in an isolated environment called a container. In the following subsections, we will analyze the basic concepts of the Docker architecture.

2.1.1 Docker Container

A Docker container [4] is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Container images become containers when they run on Docker Engine.

We can compare Docker containers with virtual machines (VMs), in order to identify their differences. As shown in Figure 2.1, unlike Virtual Machines, each container accesses the kernel of the host operating system. Therefore, multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in userspace. On the other hand, each VM includes a full copy of an operating system, called guest OS, which requires more space and longer boot time than a container.

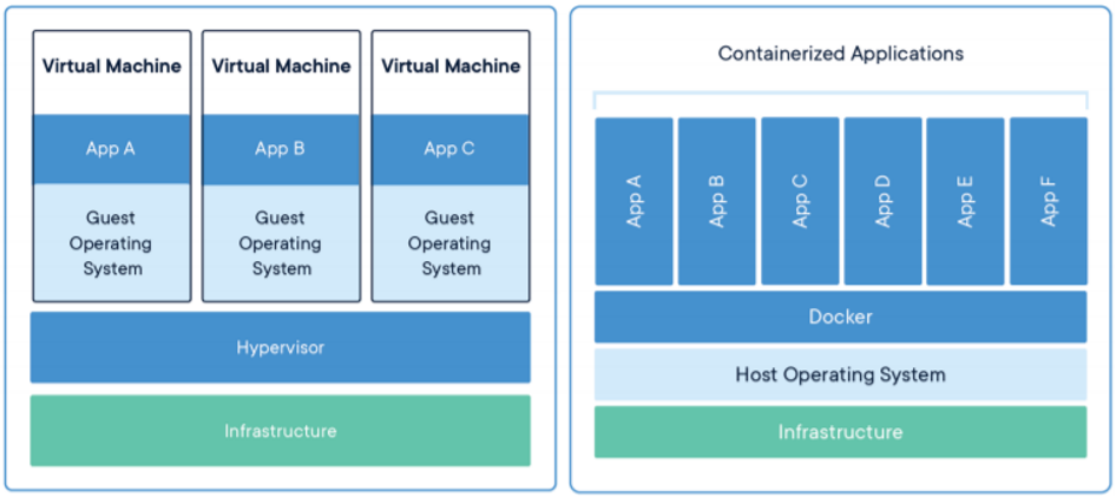


Figure 2.1: Containers VS Virtual Machines

2.1.2 Docker Architecture

As shown in Figure 2.2, Docker uses a client-server architecture consisting of the following components:

- **Docker daemon:** The Docker daemon listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. It is responsible for building, running, and distributing Docker containers.

- **Docker client:** The Docker client is the primary way that Docker users interact with Docker daemon. Using the Docker command line, we can send commands to the Docker daemon which carries them out. The Docker client and daemon can run on the same system, or we can connect a Docker client to a remote Docker daemon.

- **Docker registry:** A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. We can also create our own private registries. When we use the *docker pull* or *docker run* commands, the required images are pulled from our configured registry. When we use the *docker push* command, the image is pushed to the configured registry.

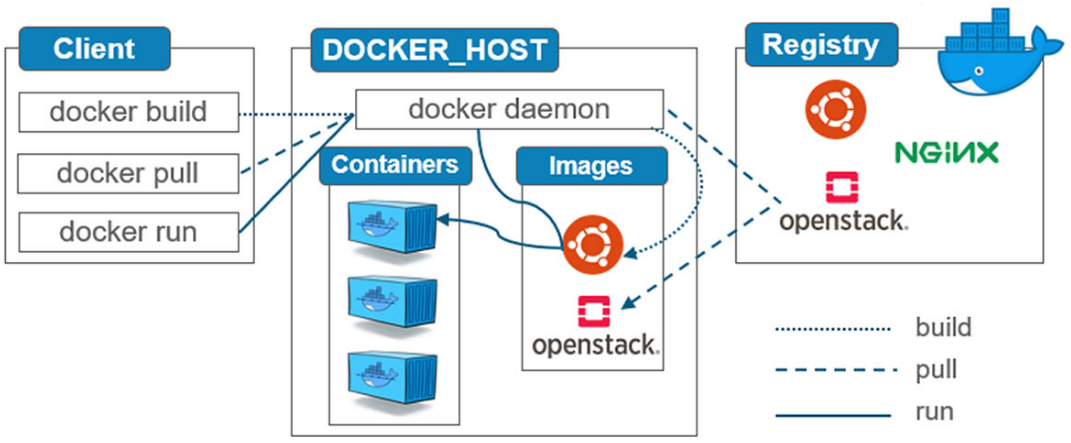


Figure 2.2: Docker Architecture [2]

2.1.3 Docker buildx

Docker Buildx [5] is a CLI plugin that extends the standard Docker commands with the full support of the features provided by Moby Buildkit builder toolkit [6]. It provides the same user experience as Docker build, with many new features, like creating multi-platform images. To obtain this, Buildx uses the QEMU emulation support in the kernel. We can use buildx in order to create Docker images for heterogeneous systems (systems that contain devices of different architectures). For a multi-platform build, we can set the *–platform* flag to specify the target platform for the build output, (for example, linux/amd64, linux/arm64, or darwin/amd64). Moreover, we can specify multiple platforms together. In this case, the builder builds a manifest list that contains images for all specified architectures. When we use this image in Docker run or docker service, Docker picks the correct image based on the node's platform.

Chapter 3

Application Model & System Architecture

3.1 Application Model

3.1.1 Application as a graph

A component-based application is described by a graph that consists of nodes, each one representing a component, and from unidirectional links used for application and control data exchange. During deployment, multiple instances of the same component can be created. For reasons of simplicity, during design time we refer to components, and during deployment time we refer to running instances. Moreover, links between the components of the graph as well as deployment preferences and requirements should be described inside a configuration file given as input to the system.

3.1.2 Network API

At deployment time, the graph is instantiated on the target hosts, along with automatically generated connector logic that takes care of inter-component binding and communication over the network. We provide a network API that takes care of the connections establishment, termination, and control, as well as the data transfer between the running instances. Through function calls, networking information is provided to the application layer helping it distinguish the instances from/to which it can receive/send data respectively.

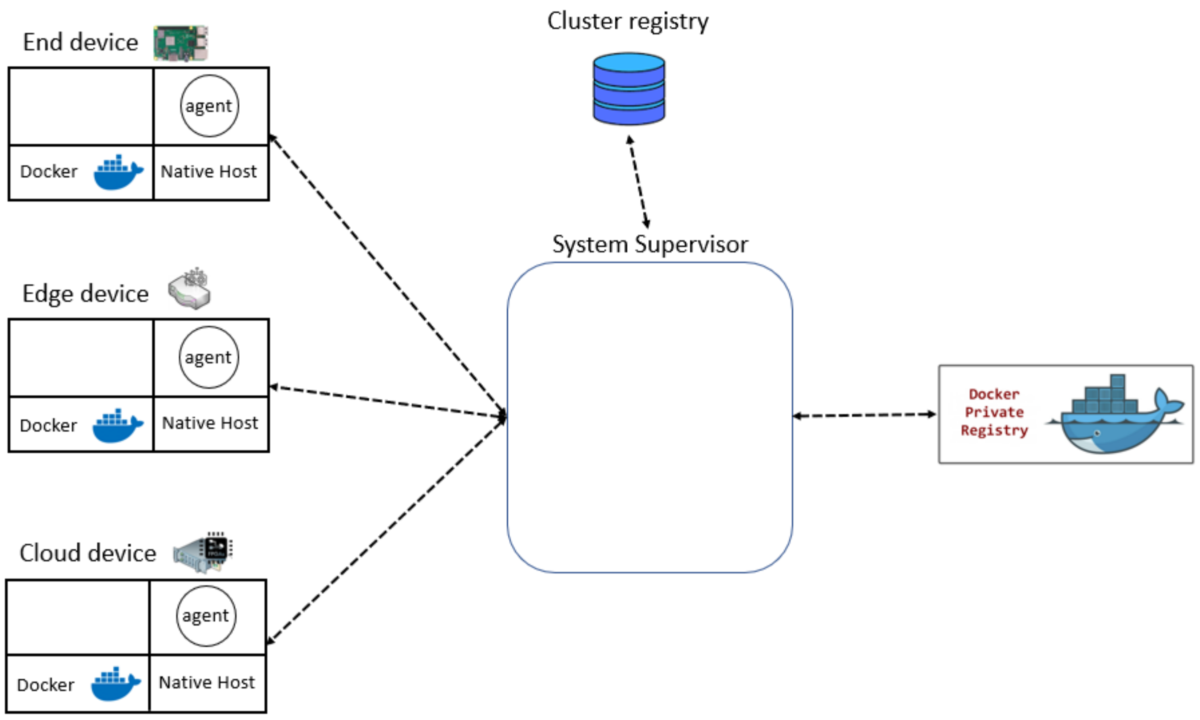


Figure 3.1: System Architecture

3.2 Basic components of the system

As shown in Figure 3.1, the system consists of the following components:

- **Cluster:** It is a set of devices of varying architectures. We provide all the registration logic which is necessary for the management of the cluster. The Docker engine and the python version 3.9 must be installed on each device. In addition, an agent responsible for the communication with the system supervisor is running on each device host.

- **Docker image registry:** A private registry is always available to store and serve Docker images. Each device of the cluster has access to this registry.

- **System Supervisor:** This is a python script in which the logic of the system is applied. It is responsible for the placement, scaling and migration of the components the application described by the user consists of. It needs to be fed with the appropriate input data through which the user defines the application and sets the deployment specifications.

3.3 System Functionality

In this section, we present the core functionalities of our system by providing illustrative examples.

3.3.1 Deployment

For the system to run properly, the user must feed the system supervisor with a directory consisting of the code and the dependencies of each component, along with a configuration file containing the graph representation as well as deployment preferences and requirements. The system supervisor will analyze the configuration file and after being informed about the available devices by polling the cluster registry, it will apply a placement algorithm in order to create the appropriate instances based on the requirements provided by the user.

For a better understanding of the deployment process, we provide an example. At first, we assume that the user describes the graph shown in Figure 3.2. This graph consists of three components named F1, F2, and F3. The user marks the components F2 and F3 as singular which means that only one instance of these components will be created. Instead, the F1 component can be scaled based on the available compatible devices of the cluster. Last but not least, the user sets placement preferences for each component as shown in Figure 3.2. The placement preferences as well as the singularity of each component are defined in the configuration file. In this example, for reasons of simplicity, we present them as a part of the graph.

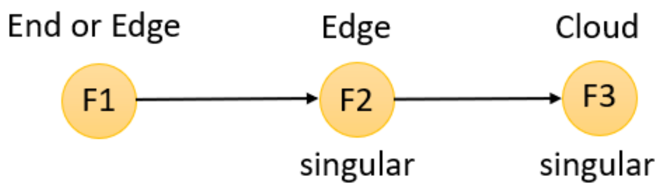


Figure 3.2: Graph representation

Suppose that we have the cluster presented in diagram 3.1 consisting of three devices running on the end, the edge, and the cloud respectively. The actual deployment of the described component-based application is shown in Figure 3.3. We can observe the input given to the system supervisor as well as the running instances on each device. We can also notice that the user-defined placement preferences have been met.

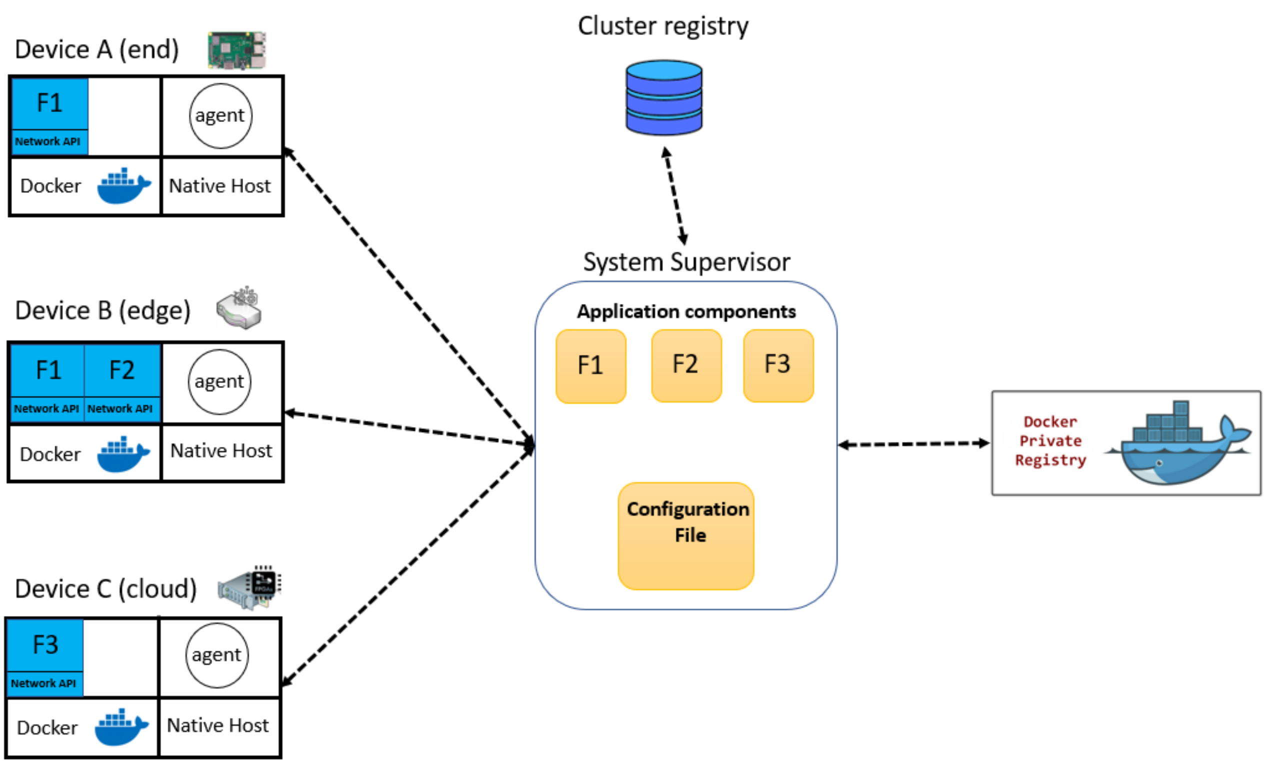


Figure 3.3: Deployment

3.3.2 Scaling

Once all instances are up and running, the system is able to detect the changes in the number of available devices running on the cluster and adapt the deployment. We provide dynamic deployment support, enabling the users to indicate if they want the non-singular instances of the graph to scale depending on the available devices. Specifically, users can declare the scaling factor located in the configuration file either with the ”as_many_as_possible” expression or with a positive integer. The former means that the system will try to scale the non singular components as long as there are compatible devices in the cluster. It is important to mention that at most one instance of each component can be running on a single device.

Having the deployment we described in Figure 3.3 and assuming that the user has set the scaling factor to ”as_many_as_possible”, we add a new device into the cluster (device D) as shown in Figure 3.4. The system will identify this insertion and after checking the device compatibility, it will create a new instance of the component F1 to the new device. Thus, the new deployment is shown in Figure 3.5.

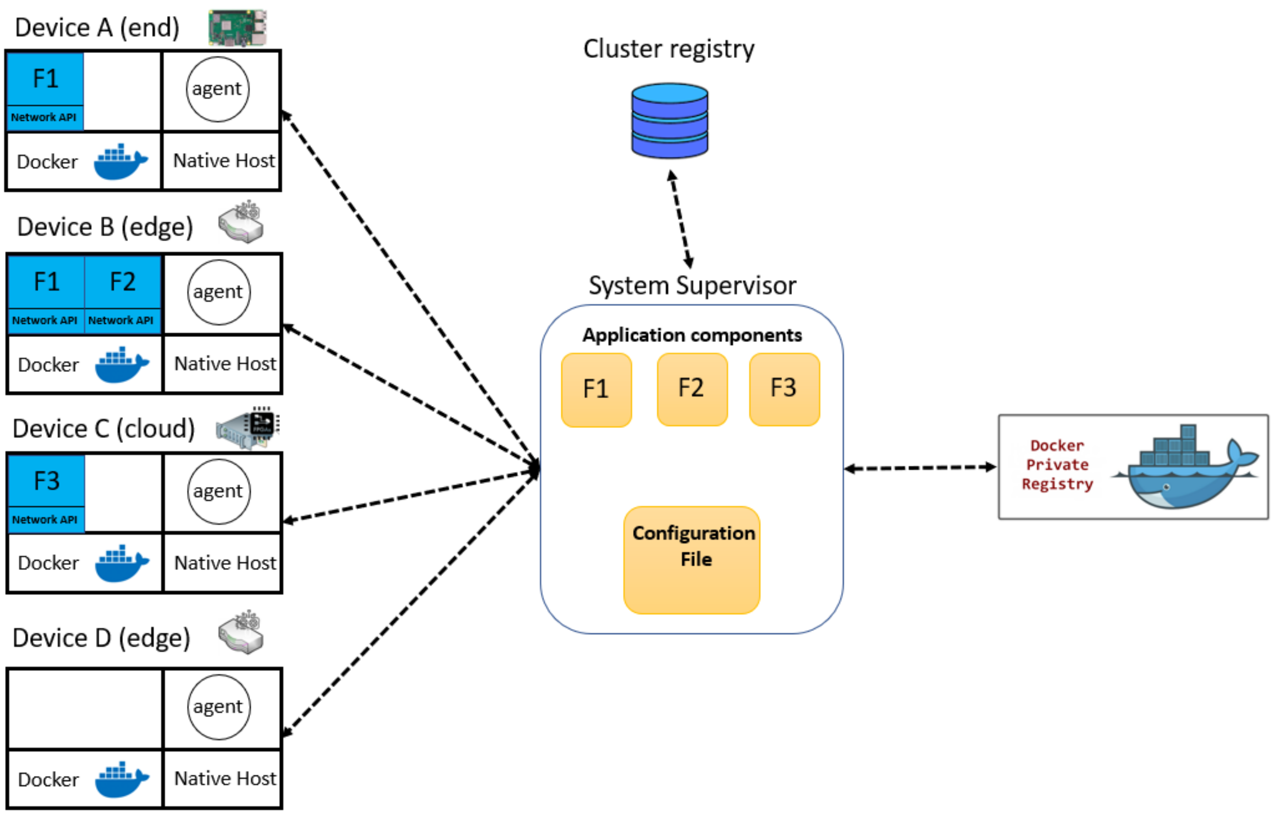


Figure 3.4: Insertion of a new device into the cluster

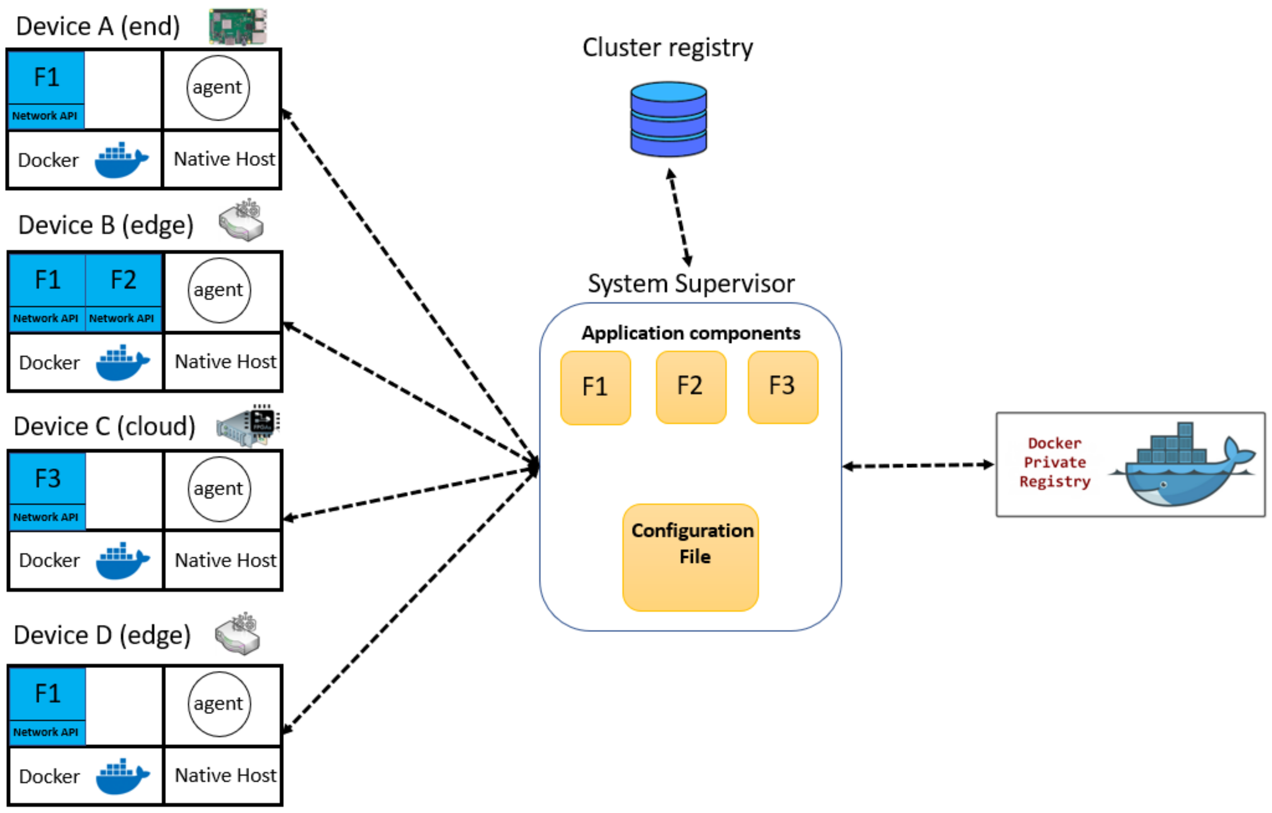


Figure 3.5: Scaled deployment

3.3.3 Migration

We provide a migration service that allows instances to be migrated to a different compatible device. This is done simply by typing a command to the system command prompt. If the destination device is incompatible or if the specific instance can not be migrated at this moment, the system will return an error message.

For further understanding, we provide a migration example. Suppose that we have the deployment presented in Figure 3.6. Assuming that device C meets all the requirements to support an instance of the F2 component, we ask to migrate the running F2 instance from device B to device C. The resulting deployment is shown in Figure 3.7.

3.3.4 Device exit

We have implemented an exit functionality giving each device the ability to exit the cluster normally. If there are running instances on the exiting device, the system guarantees the proper migration of each instance to another compatible device. If at least one of the running instances cannot be migrated, the device that requested to leave remains in the cluster until a compatible device is found.

Having the deployment we described in Figure 3.3 and assuming that the user has set the scaling factor to ”2”, we add a new device (device D) into the cluster as shown in Figure 3.8. The system will notice this insertion, but will not perform any action because the scaling criterion has already been reached. Afterwards, device B in which F1 and F2 instances are running sends an exit request to the cluster registry. The system will try to migrate both instances to the new device. Considering that the added device is compatible with both the components F1 and F2, the new deployment is shown in Figure 3.9.

3.3.5 Monitoring

We have designed and implemented a monitoring service through which the user can observe and/or modify the deployment as well as control each device that belongs to the cluster. Through the system command prompt, the user can interact with the system supervisor in order to monitor or modify the actual deployment. In addition, we provide command line support on each device so that the user can control the status of the respective device.

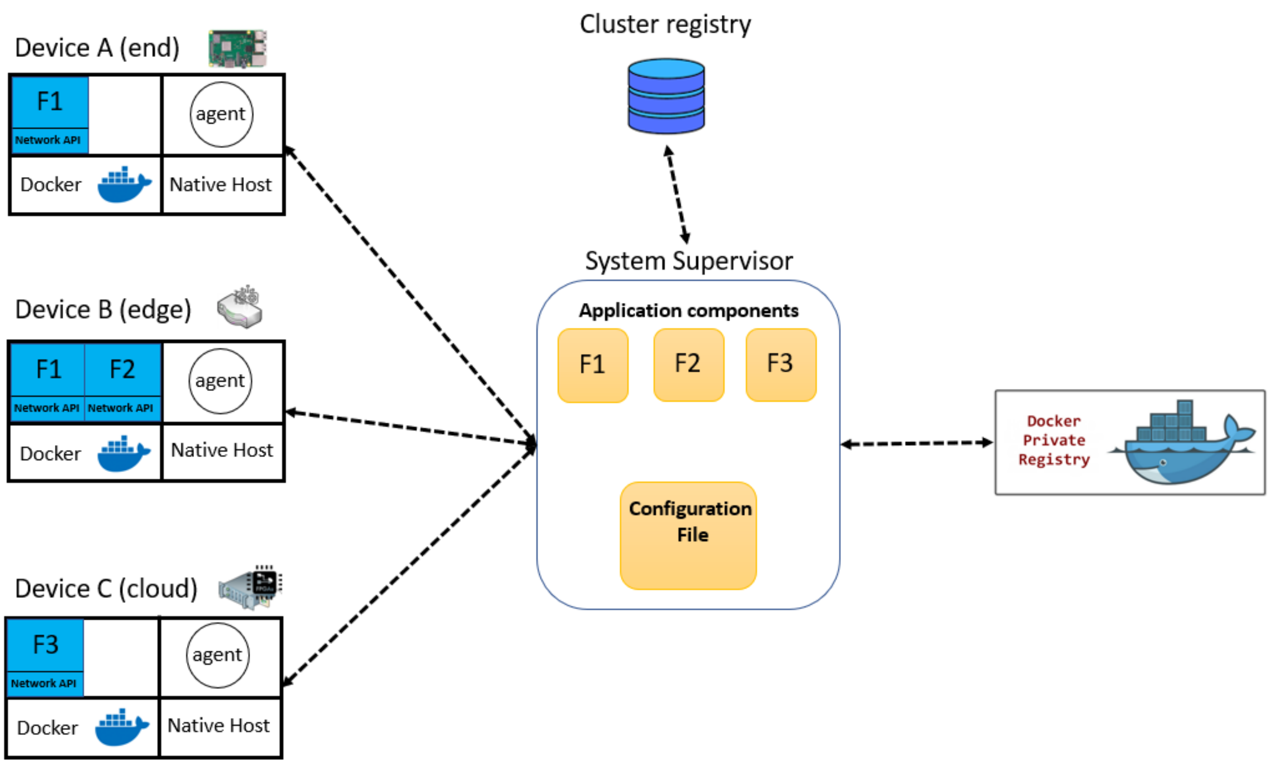


Figure 3.6: Deployment before migration

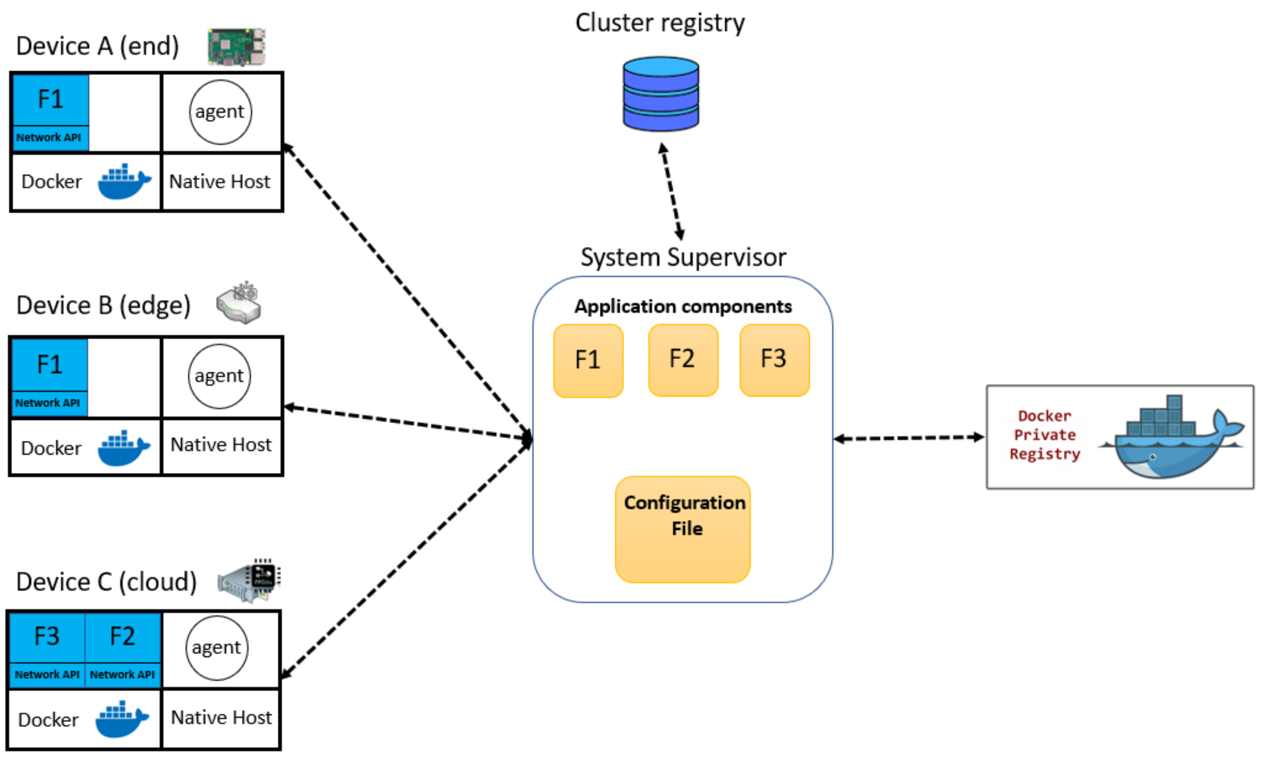


Figure 3.7: Deployment after migration

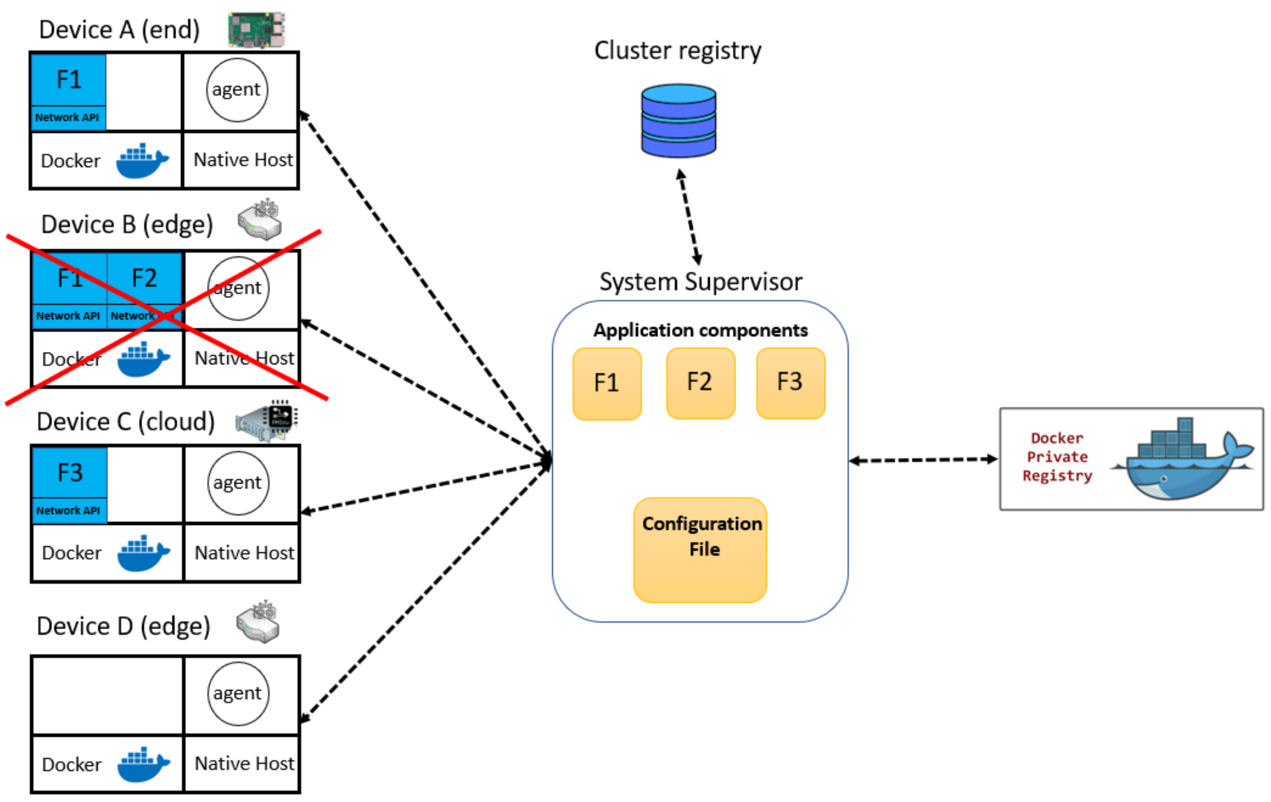


Figure 3.8: Device B requests to exit the cluster

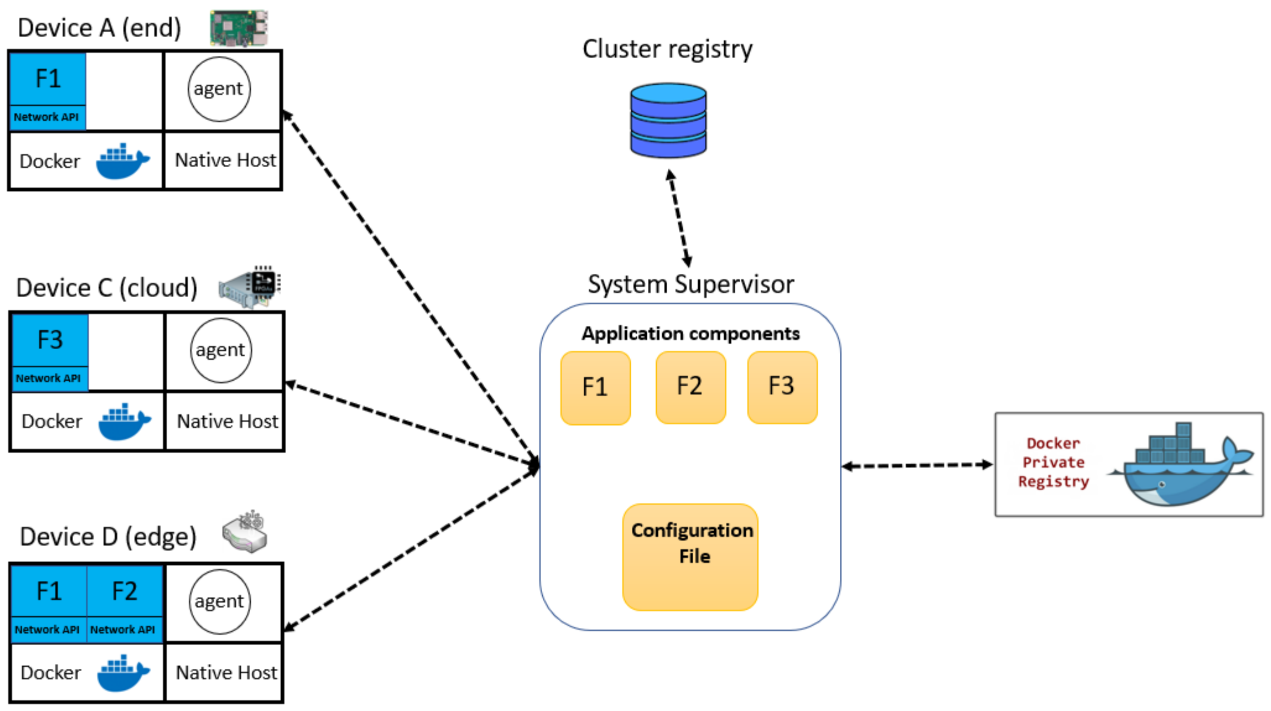


Figure 3.9: Scaled deployment

Chapter 4

Implementation

4.1 Input data

Before proceeding with the communication logic and the explicit analysis of the deployment process, we specify the format of the input data that the user feeds the system supervisor.

4.1.1 Source code and dependencies

The format of the directory provided to the system is presented in Figure 4.1. The code of each component and all its dependencies must be placed in a separate directory that has the name of the component included. In addition, the user should provide a Dockerfile for each component for the Docker images to be built.

4.1.2 Configuration File

The configuration file is a text document in which the representation of the graph is expressed using a description of the links between the components. Also, the developer can provide placement preferences, hardware and sensor requirements, link affinities, and other useful features for the desired deployment. The file is divided into parts, each one representing a component. An indicative configuration file is presented in Figure 4.2. As we can observe, for a specific component, the user can provide the following parameters:

- **Source files directory:** It is a relative path in which the code of this component as well as its dependencies are placed.

```
input
├─configuration.txt
├─F1
│  ├─build.sh
│  ├─system_network_library.py
│  ├─F1.py
│  └─Dockerfile
├─F2
│  ├─build.sh
│  ├─system_network_library.py
│  ├─F2.py
│  ├─Dockerfile
│  └─haarcascade_frontalface_default.xml
└─F3
   ├─build.sh
   ├─system_network_library.py
   ├─F3.py
   ├─Dockerfile
   ├─image_comparison_.py
   └─images
```

Figure 4.1: Input Files Directory

- **Placement requirements**
  - **Singularity:** If a component is marked as singular, only one instance of this component will be created in the cluster. In other words, only one container in a specific device will be generated. Instead, if a component is marked as non-singular, there will be one or more instances of this component in the cluster. This depends on the node availability and the deployment preferences the user has specified.
  - **Position:** The user can provide placement preferences for each component. The possible options are: edge or fog or end or cloud or combinations of them separated with "/" (for example edge/fog). The device in which an instance of this component will be placed must meet this criterion.
- **Hardware requirements**
  - **Processor:** The user can provide processor preferences for each component. The possible options are: aarch64 or x86_64 for ARM and x86 processors respectively or combinations of them separated with "/" (for example aarch64/x86_64). The device in which an instance of this component will be placed must fill this criterion.

  - **Cpu_cores:** The user can provide the preferred number of cpu cores for the device in which an instance of this component will be placed.
  - **Ram:** The user can set the amount of RAM memory (in Gigabytes) required for the device in which an instance of this component will be placed.
  - **Sensors:** The user can provide a list of sensors required to be attached to the device in which an instance of this component will be placed. The possible options are: camera or gpu or combinations of them separated with ”/” (for example camera/gpu).

- **Links:** Each link is represented via an expression of the following form:

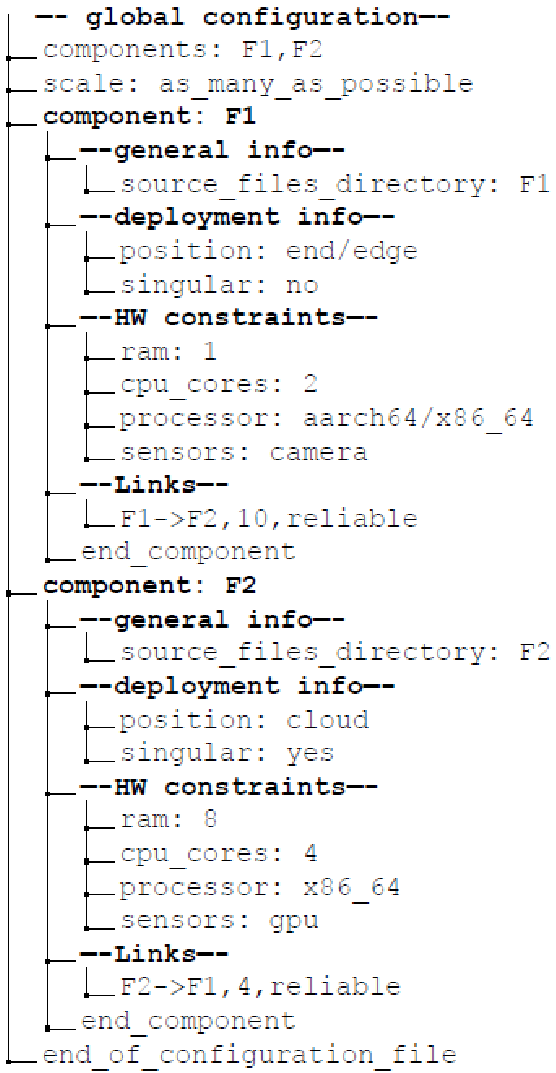
$$source\_component \rightarrow destination\_component, affinity, reliability \tag{4.1}$$

  The affinity of each connection helps the placement algorithm position the instances. Greater affinity means higher priority for the specific link. The user can fill the affinity field with a positive integer in the range [0,10]. During migration, the reliability of data exchanged between two instances is based on the reliability flag of each link connecting them. If the user requests a reliable link, the system provides mechanisms to ensure that all the data have reached their destination securely and orderly. Instead, if the user specifies an unreliable link, there is no guarantee that the data will be transferred reliably. Thus, since we have described the specifications and the singularity of nodes as well as the connections between them, a graph consisting of weighted unidirectional links is instantiated. The user can describe any type of graph by setting the links and the singularity requirements.

In addition to the configuration settings per component, some general settings need to be defined at the beginning of the configuration file. These settings concern the deployment's scalability and are presented below:

- **Components:** It is a list consisting of the names of all the components separated by a comma.
- **Scale:** This parameter specifies the desired number of instances that should be generated per non-singular component. It is important to mention that all non-singular components will be equally scaled so that they have the same number of running

instances during deployment. The user can fill this field either with the expression ”as_many_as_possible” or with a positive integer. The former means that the system will try to scale the non-singular components as long as there are compatible devices in the cluster.



```
│ -- global configuration--
├─components: F1,F2
├─scale: as_many_as_possible
├─component: F1
│  ├─--general info--
│  │  └─source_files_directory: F1
│  ├─--deployment info--
│  │  ├─position: end/edge
│  │  └─singular: no
│  ├─--HW constraints--
│  │  ├─ram: 1
│  │  ├─cpu_cores: 2
│  │  ├─processor: aarch64/x86_64
│  │  └─sensors: camera
│  ├─--Links--
│  │  └─F1->F2,10,reliable
│  └─end_component
├─component: F2
│  ├─--general info--
│  │  └─source_files_directory: F2
│  ├─--deployment info--
│  │  ├─position: cloud
│  │  └─singular: yes
│  ├─--HW constraints--
│  │  ├─ram: 8
│  │  ├─cpu_cores: 4
│  │  ├─processor: x86_64
│  │  └─sensors: gpu
│  ├─--Links--
│  │  └─F2->F1,4,reliable
│  └─end_component
└─end_of_configuration_file
```

Figure 4.2: Configuration File

4.2 Networking

For the deployed instances to communicate, there is a need for a networking module. For this reason, we provide a network API that takes care of the connections establishment, termination, and control as well as the data transfer between the running instances.

4.2.1 Flow-id assignment

Examining a graph before scaling is applied, we define that all traffic exchanged between the instances of the graph belongs to a specific stream. If we scale the graph, in each scaling step, an additional stream (with scaling of the corresponding non-singular components), will be generated. To distinguish streams, we have assigned a flow-id to each of them. The same flow-id is assigned to all non-singular instances belonging to the same stream. In addition, the singular instances contain all the assigned flow-ids. Therefore, any non-singular instance can send/receive data only to/from the stream to which it belongs, using the assigned flow-id. Instead, the singular instances can send/receive data to/from all flow-ids.

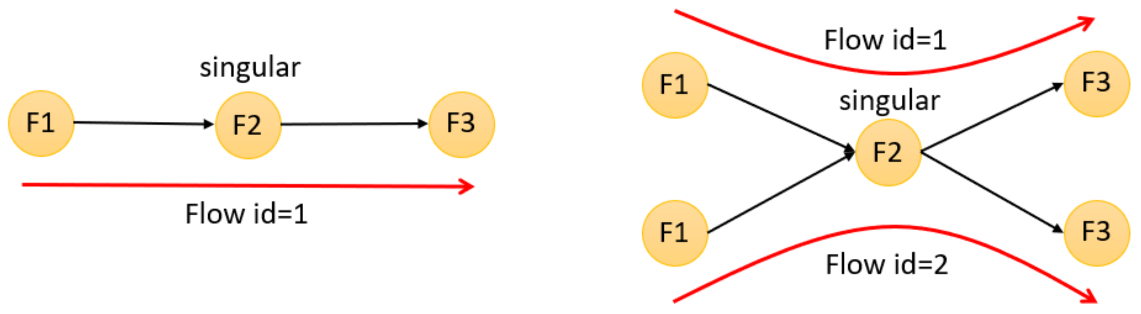


Figure 4.3: Flow-ids in a graph

For further understanding, we provide an example. Suppose a graph consisting of the instances shown in Figure 4.3. As we can observe, all instances belong to the same stream. Therefore, flow-id ”1” is assigned to each of them. If we scale the deployment, an instance for each non-singular component will be generated. Thus, a new stream will be created and the flow-id ”2” will be assigned to each of the new instances.

4.2.2 Network Library API

The interaction between the application and the network library can be achieved using the API summarized in Table 4.1 and described extensively below.

Table 4.1: Network Library Functions

Function Name	**Arguments**
init_system_settings	None
get_dst_components	None
get_dst_flows	None
get_src_components	None
get_src_flows	None
send	<data, destination_component_list[], destination_flow_list[]>
receive	<source_component, source_flow_id, number_of_messages, blocking_mode>
can_move	<status>

- **init_system_settings ()**

  This is the first function that should be called when an instance is starting. It is responsible for the networking API modules establishment.

- **get_dst_components ()**

  Returns a list containing the names of the destination components to which an instance can send data. It can be called from both non-singular and singular components.

- **get_dst_flows ()**

  Returns a list containing the ids of the destination flows to which an instance can send data. It can be called only from singular components.

- **get_src_components ()**

  Returns a list containing the names of the source components from which an instance can receive data. It can be called from both non-singular and singular components.

- **get_src_flows ()**

  Returns a list containing the ids of the source flows from which an instance can receive data. It can be called only from singular components.

- **send (data, destination_component_list[], destination_flow_list[])**

  When there are data to be transmitted, the send function should be called from the application layer. There are three arguments to be specified. The first is a byte array indicating the transferred payload, the second is a list containing the destination components in which the data should be delivered and the third is a list containing the destination flows in which the data should be delivered. The following cases are observed:

  – If all arguments are specified, the network library will transmit the data at each <**destination_component, destination_flow**> pair.

  – If destination_flows_list is not defined, the network library will flood the data to all instances whose name is included in the destination_components_list, regardless of their flow-id.

  – If destination_component_list is not defined, the network library will flood the data to all instances whose flow-id is included in the destination_flow_list regardless of their component name.

- **receive (src_component, src_flow_id, how_much_messages, blocking_mode)**

  When the application layer wants to receive data, the receive function should be called. There are four arguments to be specified. The first argument defines the component name from which the current instance will receive data. The second argument defines the flow-id from which the current instance will receive data. The third argument is a positive integer indicating the number of messages the application layer wants to receive and the fourth argument declares if the receiving process should be blocking or not. If blocking mode has been set to true, the receive call will be blocked until all the required messages become available. The following cases are observed:

  – If all arguments are specified, the network library will receive the data from the <**source_component, source_flow**> pair consisting of the first two arguments.

  – If src_flow_id is not defined, the network library will receive the data from the library's buffer by matching only the component name ignoring the flow-id.

  – If src_component is not defined, the network library will receive the data from the library's buffer by matching only the flow-id ignoring the component name.

- **can_move (status)**

  For the migration functionality to operate properly, the user should set the move status to ”true” using the function above. If the status is marked as ”false”, the instance can not be moved until the status changes again to ”true”.

4.3 Application Layer Representation

In this section, we present two examples that define the structure of the application layer of a real instance. Furthermore, we explicitly analyze the use of the network API functions we described above. In the following examples, we use the deployment shown in Figure 4.4. This application consists of three components, named F1, F2, and F3. The component F2 is marked as singular and the components F1 and F3 are marked as non-singular. As we can observe, there are two streams to which flow-ids 1 and 2 have been assigned respectively.

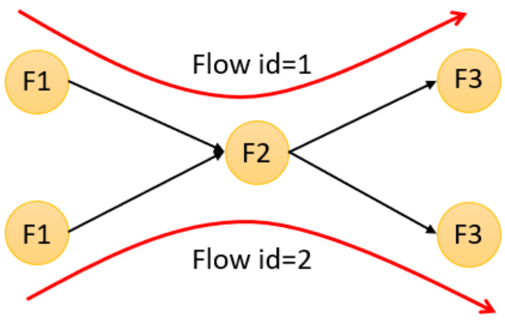


Figure 4.4: Actual Deployment

4.3.1 Transmit to the same flow-id

In this example, each F1 instance transmits a message to the singular F2 instance. The F2 instance expects a message from each source flow in a circular manner and forwards the received message to the F3 instance that belongs to the same stream as the F1 instance that sent the data. The application-level code of the F2 instance is presented in Algorithm 1. Network API function calls are made in the following order:

- The **init_system_settings** function is called first for the networking API modules to be established.

- **get_dst_components()**: This function returns a list containing the names of the destination components to which the F2 instance can send data. In this case, it will return the list: [”F3”].

- **get_dst_flows()**: This function returns a list containing the ids of the destination flows to which the F2 instance can send data. In this case, it will return the list: [1,2].

- **get_src_components()**: This function returns a list containing the names of the source components from which the F2 instance can receive data. In this case, it will return the list: [”F1”].

- **get_src_flows()**: This function returns a list containing the ids of the destination flows from which the F2 instance can receive data. In this case, it will return the list: [1,2].

- **receive(src_components[0],scr_flows[i],1,1)**:
  Inside the for loop, the F2 instance calls the receive function for each source flow in order to receive one message from each F1 instance. We can notice that the blocking mode is enabled.

- **send(data,dst_components,[src_flows[i]])**:
  Finally, the F2 instance calls the send function immediately after the receive function call in order to forward the data received in this iteration to the F3 instance which belongs to the same stream as the F1 instance from which the data was received.

4.3.2 Aggregation

As in the previous example, each F1 instance transmits a message to the singular F2 instance. The F2 instance receives a message from all its source flows and then broadcasts it to all the F3 instances. The application-level code of the F2 instance is presented in Algorithm 2. Network API function calls are identical to those presented in the previous example except for the send function. Specifically, after receiving a message from all the source flows, the F2 instance calls the send function in order to broadcast the received data to all the F3 instances. We can observe that the send function is called outside the for loop and it has the following form: send(rcvmsg,[”F3”],[1,2]).

Algorithm 1 Transmit to the same flow id scenario

```
init_system_settings()
while True do
    dst_components=get_dst_components()
    dst_flows=get_dst_flows()
    src_components=get_src_components()
    src_flows=get_src_flows()
    for i in range(0,len(src_flows)) do
        data_list=receive(src_components[0],src_flows[i],1,1)
        rcvmsg=data_list[0]
        send(rcvmsg,dst_components,[src_flows[i]])
    end for
end while
```

Algorithm 2 Aggregation scenario

```
init_system_settings()
while True do
    dst_components=get_dst_components()
    dst_flows=get_dst_flows()
    src_components=get_src_components()
    src_flows=get_src_flows()
    for i in range(0,len(src_flows)) do
        data_list=receive(src_components[0],src_flows[i],1,1)
        rcvmsg=data_list[0]
    end for
    send(rcvmsg,dst_components,dst_flows)
end while
```

4.4 System Functionalities

In this section, we analyze the deployment process and explicitly describe the core system functionalities.

4.4.1 Deployment process

- **Configuration parsing and Docker images' build**

  At first, the system supervisor analyzes the configuration file in order to adapt the deployment based on the user's preferences and requirements. The necessary data structures for the application components are created and a Docker image is built for each component. Specifically, as mentioned in the Background Chapter [2.1.3], we use the Docker buildx plugin to build multiarchitecture Docker images. The system determines the platforms for which the image of a component will be built based on the platform parameter defined for this component in the configuration file. For each image that has been built, the system pushes it into the private Docker registry.

- **Discovery of the cluster's devices**

  The system polls the cluster registry to find out the active devices. Afterwards, it sends a discovery request to each device to be informed of its status. The agent running on each device is responsible for responding to this discovery message by providing the values of the following parameters:

  – processor type
  – total RAM
  – available RAM
  – position
  – attached sensors

  The position and the attached sensors' information are stored in a file located in the same directory as the agent's executable in each device. Thus, the system computes the compatible devices for each component by comparing the preferences and requirements defined in the configuration file with the information returned from the agent of each device after the discovery process.

- **Meet the desired state**

  The system is now ready to calculate the number of instances that should be created for every non-singular component. It is important to mention that all non-singular components will have the same number of running instances during the deployment. In addition, the system tries to meet the scale factor parameter specified in the configuration file. If the scale factor is a positive integer, the system tries to reach but not exceed this factor. On the other hand, if the scale factor is set to ”as_many_as_possible”, the non-singular component with the least number of compatible devices is a bottleneck to all other components concerning the number of instances that will be created. Assuming that the compatible devices table includes the number of compatible devices per non-singular component, we apply Algorithm 3. If the ”total_instances_per_component” variable is zero, the system will generate an error message stating that this application cannot be deployed.

Algorithm 3 Calculation of the total number of instances for each non singular component

```
if scale_factor == ”as_many_as_possible” then
    total_instances_per_component = min(compatible_devices_table)
else
    total_instances_per_component = min(min(compatible_devices_table),scale_factor)
end if
```

- **Placement algorithm**

  During the placement, the system horizontally generates the graph, applying an iterative algorithm which creates application components’ instances, starting from the leftmost component and ending to the rightmost component. This process is repeated until the ”total_instances_per_component” variable calculated in the previous step is reached. Algorithm 4 summarizes the placement process. As we can observe, the link table is scanned in descending order of affinity. Thus, instances concerning components connected with a higher affinity link will be placed first. In addition, the algorithm optimally places the neighboring instances using a series of priority lists aiming at zero communication latency between them. On the device side, the agent is responsible for receiving and executing system commands. In this case, the agent pulls the specified Docker image from the private Docker registry and creates a Docker container.

Algorithm 4 Placement Algorithm

```
Sort the link table in descending order of affinity
while True do
    for each link $Fx \rightarrow Fy$ of the sorted link table do
        if $Fx$ not checked and there is an instance to be created for component $Fx$ then
            $devices\_list_A = compatible\_devices(Fx) - devices\_with\_running(Fx)$
            $devices\_list_B = devices\_list_A \cap compatible\_devices(Fy)$
            $devices\_list_C = devices\_list_B \cap devices\_with\_running(Fy)$
            $create\_new\_instance\_with\_priority(devices\_list_C, devices\_list_B, devices\_list_A)$
        end if
        if $Fy$ not checked and there is an instance to be created for component $Fy$ then
            $devices\_list_A = compatible\_devices(Fy) - devices\_with\_running(Fy)$
            $devices\_list_B = devices\_list_A \cap compatible\_devices(Fx)$
            $devices\_list_C = devices\_list_B \cap devices\_with\_running(Fx)$
            $create\_new\_instance\_with\_priority(devices\_list_C, devices\_list_B, devices\_list_A)$
        end if
    end for
    if no instance was created in the last iteration then
        break
    end if
end while
```

- **Inform each running instance about its source and destination components and flows**

  After applying the placement algorithm, the system determines the source and destination components and flows for each generated instance. Once each instance boots, the ”init_system_settings” function will be called first. Through this function, each instance sends a message to the system to let it know that all the networking API modules are initialized and ready to receive system messages. When all the instances announce their presence, the system informs each one about its source and destination components and flows by sending a ”NETWORK_INFORMATION” message. Once each instance network library receives this message, the ”init_system_settings” function returns and the instance is ready to start transmitting and receiving data.

- **Cluster monitoring and scaling**

  When the deployment process is completed, the system enters a monitoring mode inspecting the cluster at short intervals. If a new device is inserted into the cluster, the deployment is scaled based on the scaling preference defined in the configuration file and the compatibility of the new device. In addition, we provide command line support by enabling the user to inspect the cluster by typing ”status” into the system command prompt.

4.4.2 Migration

We provide a migration service that allows instances to be migrated to a different compatible device. The user should type a command into the system command prompt that has the following form:
move <component name> from <source device> to <destination device>
The application layer of the instance that wants to be migrated must have set the move status to ”True” using the ”can_move” function of the network API.

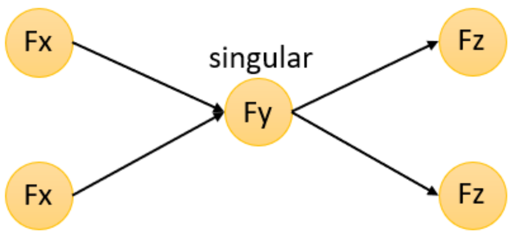


Figure 4.5: Deployment before migration

For a better understanding of the migration process, we analyze the steps through an example. As shown in Figure 4.5, we examine a deployment consisting of three components named Fx, Fy and Fz. We assume that all links of this deployment are defined as reliable in the configuration file. The user decides to migrate the singular Fy instance. At first, the system examines the validity of the command. If the destination device is incompatible or the specific instance can not be migrated at this moment, the system returns an error message. If the instance can be migrated to the new device, the following steps are applied:

- **Create the new instance**

  The system creates a new instance of the same component on the new device. Once the new instance boots, the ”init_system_settings” function is called first. Through this function, the new instance sends a message to the system to let it know that all the networking API modules are initialized and ready to receive system messages. For reasons of simplicity, we refer to the instance which wants to be migrated as the ”old instance” and to the instance created in the destination device as the ”new instance”.

- **Redirect the ingress connections of the old instance**

  The system sends a message to each instance that has established a connection to the old instance over a reliable egress link (this corresponds to an ingress link for the old instance), forcing it to terminate this connection after performing a reliability check. Before each connection is redirected, a reliability check is applied in order to ensure that all the data transmitted on each connection have reached their destination securely and orderly. To perform the reliability test, each instance sends a ”FINAL” message over each TCP connection pointing to the old instance. If an ack is received for each ”FINAL” message sent, the connections can be redirected. In the example shown, both Fx instances redirect their connections pointing to the old Fy instance. When all the Fx instances have completed this process, the system proceeds to the next step. If these links are marked as unreliable, the reliability check is not performed and the connections are redirected without any data protection. The status of the deployment until this step is presented in Figure 4.6.

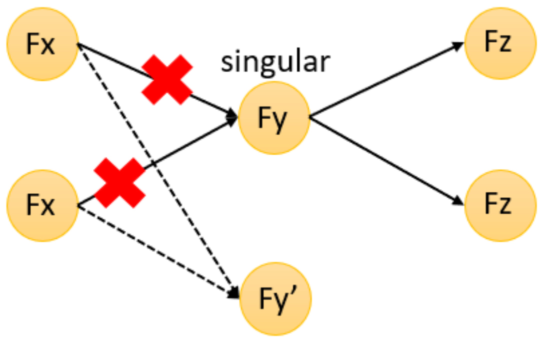


Figure 4.6: Migration deployment [Step 1]

- **Move buffered data from the old instance to the new instance**

  If the ingress links of the old instance are marked as reliable, the data stored in the old instance’s buffers should be migrated to the new instance. Thus, the system sends a

”MOVE_BUFFERS” message to the old instance forcing it to establish a connection with the new instance and transmit all the buffered data. It is important to mention that during the buffers’ data move procedure, the new instance can receive data from the Fx instances. If the ingress links of the old instance are marked as unreliable, this step is skipped. The status of the deployment until this step is presented in Figure 4.7.

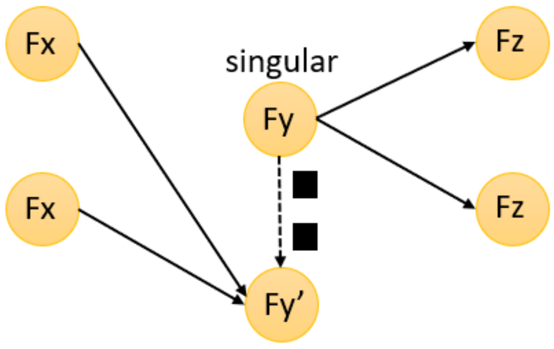


Figure 4.7: Migration deployment [Step 2]

- **Terminate the egress connections of the old instance**
  If the old instance has established connections to other instances over reliable egress links, the system sends a message to the old instance forcing it to terminate all of these connections after performing a reliability check. This is done in order to ensure that all data transmitted on each connection have reached their destination securely and orderly. To perform the reliability test, the old instance sends a ”FINAL” message over each egress TCP connection. If an ack is received for each ”FINAL” message sent, the connections can be closed. In the example shown, the Fy instance terminates its connections with both Fz instances. If the egress links are marked as unreliable, the reliability check is not performed and the connections are terminated without any data protection.

- **Inform the new instance about its destination components and flows**
  Last, the system informs the new instance about its source and destination components and flows by sending a ”NETWORK_INFORMATION” message. Once the new instance’s network library receives this message, the ”init_system_settings” function returns and the new instance is ready to start transmitting and receiving data. The status of the deployment until this step is presented in Figure 4.8.

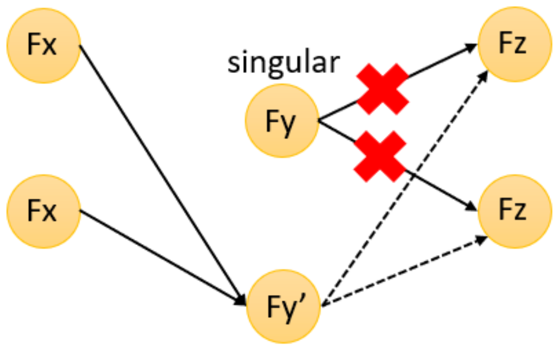


Figure 4.8: Migration deployment [Step 3]

- **Destroy the old instance**

  All the connections are functional and the old instance can be destroyed safely. The deployment after the end of the migration procedure is shown in Figure 4.9.

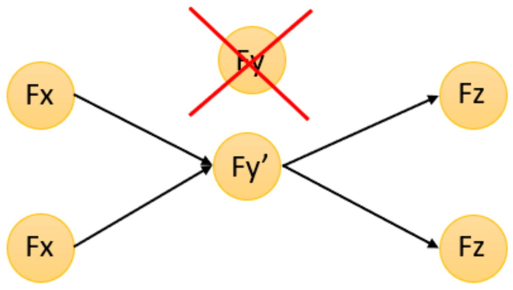


Figure 4.9: Deployment after migration

4.4.3 Device exit

The agent running on each device provides command-line support through which the user can monitor and manage the device. For a device to exit the cluster normally, the user should type the command ”exit” to the device command prompt. The agent will update the cluster registry on the device exit request. When the system identifies this request, if there are running instances on the exiting device, the secured migration of each running instance to another compatible device will be applied. If at least one of the running instances cannot be migrated, the device that requested to leave remains in the cluster until a compatible device is found. Last but not least, for this functionality to work properly, the user should have set the scaling factor of this deployment to a specific positive integer and not to the

”as_many_as_possible” expression in order to prevent the deployment from scaling. If the scaled factor is set to ”as_many_as_possible”, the system will generate an instance of each non-singular component on the newly inserted device. In this case, if the device that requested to exit the cluster has at least one non-singular instance, the migration will fail.

Chapter 5

Validation & Performance

We conducted several experiments in order to demonstrate the functionalities provided by the system as well as to observe the performance of different deployments in a real edge-fog-cloud cluster. In this chapter, we present the respective methodology, as well as validation and evaluation results.

5.1 Placement Algorithm

In this section, we examine the placement algorithm's validity. As we mentioned in the implementation Chapter [4.4.1], the placement algorithm generates the graph by scanning the link table in descending order of affinity. The purpose of the placement algorithm is to place the instances optimally, trying to minimize the communication latency between the instances connected with a high-affinity link.

5.1.1 Placement example with decreasing affinities

In this example, the application presented in figure 5.1 is deployed. The placement preferences and hardware requirements of the components are summarized in Table 5.1. The cluster consists of four devices whose characteristics are summarized in Table 5.2. In addition, the compatible devices per component as calculated by the placement algorithm are presented in the same table.

The results of the placement algorithm are presented in Table 5.3. Assuming that the application's scale factor is set to ”as_many_as_possible”, devices A and B are compatible with both F1 and F2 components. Thus, two instances will be generated for each non-singular

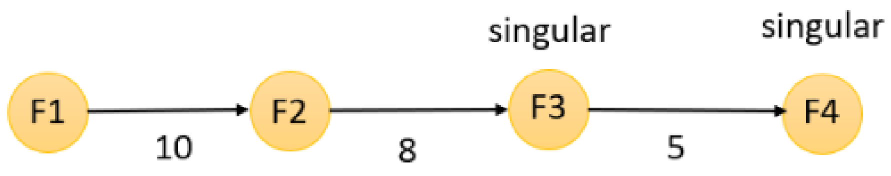


Figure 5.1: Application for deployment

Table 5.1: Component requirements

Component Name	Required sensors	Preferred position
F1	camera	end
F2	camera	end/edge
F3	-	edge
F4	gpu	cloud

Table 5.2: Cluster nodes and compatible devices per component

Cluster nodes

Device Name	Attached sensors	Position
A	camera	end
B	camera	end
C	camera	edge
D	camera, gpu	cloud

Compatible Devices

Component Name	Compatible Devices
F1	A, B
F2	A, B, C
F3	C
F4	D

Table 5.3: Placement results

Device Name	Running Instances
A	F1, F2
B	F1, F2
C	F3
D	F4

component. As we mentioned in Algorithm 4, the placement algorithm scans the link table in descending order of affinity in each iteration. Therefore, the $F1 \rightarrow F2$ link will always be checked first. As a result, each pair of F1 and F2 instances will be placed on the same device, leading to minimal communication delay between them. The singular F3 instance will be placed on device C because it is the only one that meets the position requirements and the singular F4 instance will be placed on device D because it is the only one that meets the position and sensor requirements.

5.1.2 Placement example with increasing affinities

In this example, the application presented in Figure 5.2 is deployed. The placement and sensor requirements of the components as well as the devices' specifications are identical to those of the previous example. The results of the placement algorithm are presented in Table 5.4.

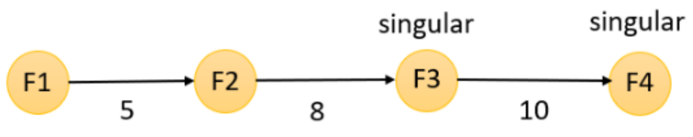


Figure 5.2: Application for deployment

Device Name	Running Instances
A	F1, F2
B	F1
C	F3,F2
D	F4

Table 5.4: Placement results

In each iteration of the placement algorithm, the $F3 \rightarrow F4$ link will be checked first because this is the link with the greatest affinity. However, there is no common device for these instances to be placed together. Therefore, F3 and F4 instances will be placed on devices C and D respectively. The $F2 \rightarrow F3$ link is the next to be checked. Thus, one of the generated F2 instances will be placed on device C where the singular F3 instance has already been placed. The $F1 \rightarrow F2$ link is the last to be checked. As a result, a pair of F1 and F2 instances will be placed on device A.

5.1.3 Infeasible placements

If we change the position of node D from cloud to edge and try to deploy one of the previous examples, the system will return a “Can not deploy this service” error message. This is because the F4 component has zero compatible devices as shown in Table 5.5.

Cluster nodes

Device Name	Attached sensors	Position
A	camera	end
B	camera	end
C	camera	edge
D	camera, gpu	edge

Compatible Devices

Component Name	Compatible Devices
F1	A, B
F2	A, B, C
F3	C
F4	-

Table 5.5: Cluster nodes and compatible devices per component

5.2 Deployment Performance

In this section, we conduct performance measurements in a variety of applications to figure out the latency introduced by the system.

5.2.1 Minimum Round Trip Time

In this experiment, the application presented in Figure 5.3 is deployed. The graph consists of two singular components, named G1 and G2. We measure the round trip time (RTT) until a message sent by the G1 instance completes a cycle in the graph. We perform the measurements in the following clusters:

- A cluster consisting of one device where both instances are running.
- A cluster consisting of two devices, both connected via Ethernet interfaces to the local area network. Each singular instance is running on a separate device.

Table 5.6 summarizes the results obtained. We can observe that, as expected the minimum RTT occurs when both instances are placed on the same device.

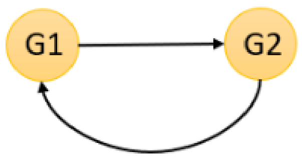


Figure 5.3: Application for deployment

Connection	Elapsed RTT
Loopback Interface	0.044965 sec
Ethernet Interface	0.056756 sec

Table 5.6: Measured RTT results

5.2.2 Transmit to the same flow-id

In this experiment, the application presented in Figure 5.4 is deployed. Table 5.9 summarizes the functionality of each component. We have previously examined the forwarding method each singular component applies in Section 4.3.1. We measure the round trip time (RTT) until a message sent by the G1 instance completes a cycle in the graph. Furthermore, we examine the scaling functionality by inserting new devices into the cluster observing the dynamic scaling of the deployment. Therefore, once the deployment process is complete, if a device inserted into the cluster is compatible with both G1 and G3 components, an instance for each of these components will be created.

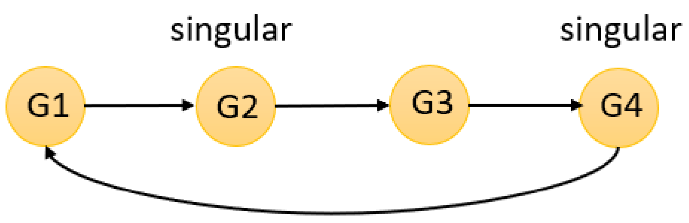


Figure 5.4: Application for deployment

Scaling Step	Elapsed RTT
1	0.085166 sec
2	0,117134 sec
3	0,127304 sec

Table 5.7: Measured RTT results

Component	Functionality
G1	Transmits a message to the singular G2 instance and waits until receiving the same message from the singular G4 instance.
G2	Expects a message from each source flow and forwards it to the G3 instance that belongs to the same stream as the G1 instance that sent the data.
G3	Forwards each incoming message to the singular G4 instance.
G4	Expects a message from each source flow and forwards it to the G3 instance that belongs to the same stream as the G3 instance that sent the data.

Table 5.9: Functionality per component

Table 5.7 summarizes the results obtained from the measurements and is divided into the three following scaling steps:

- The first scaling step is shown in Figure 5.5. The cluster consists of a single device in which all instances are running. We can observe that the measured RTT is twice the minimum RTT measured in the example 5.2.1. The cause of the additional delay is the doubling of the number of links each message traverses until it completes a cycle in the graph.
- The second scaling step is shown in Figure 5.6. We calculate the average RTT by gathering the metrics both G1 instances generate. We can observe that the RTT counted in this experiment is twice the RTT measured in the second scenario of the example 5.2.1 where each singular instance was placed on a different device and both of the devices were connected over Ethernet interfaces to the local area network. The cause of the additional delay is again the doubling of the number of links each message traverses until it completes a cycle in the graph.
- The third scaling step is shown in Figure 5.7. We calculate the average RTT gathering the metrics all G1 instances generate. Henceforth, as we scale the cluster, a constantly increasing additional delay will be introduced in the average RTT due to the forwarding pattern we apply in both singular instances. Specifically, each singular instance serves the incoming streams circularly. As we increase the incoming streams for each singular instance, this round robin process will introduce more latency.

5.2.3 Aggregation scenario

The graph of the deployed application and the scaling process we apply in this example are identical to those presented in the previous one except for the forwarding pattern applied by the application's singular components. The functionality of each component is presented in Table 5.12. We have previously examined this forwarding method in Section 4.3.2. In this experiment, we measure the round trip time (RTT) until a message sent by the G1 instance completes a cycle in the graph. Table 5.10 summarizes the results obtained and is divided into the three following scaling steps:

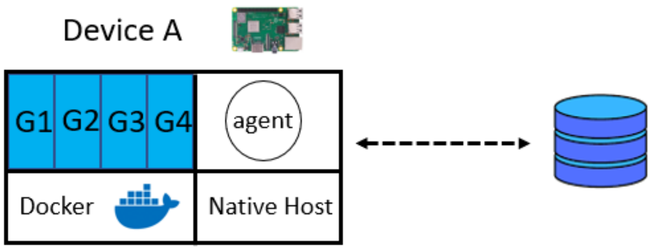


Figure 5.5: Cluster scaling [Step 1]

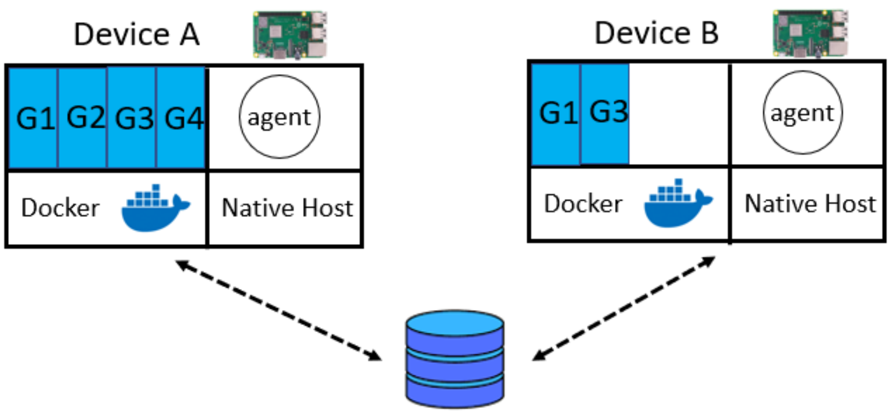


Figure 5.6: Cluster scaling [Step 2]

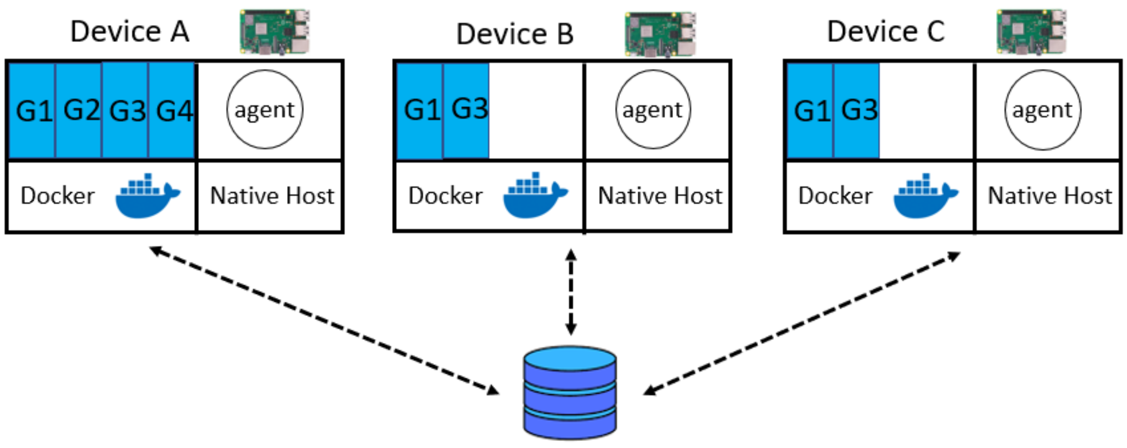


Figure 5.7: Cluster scaling [Step 3]

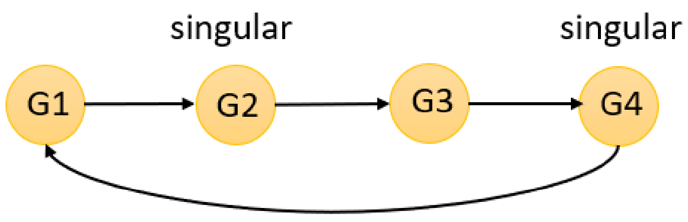


Figure 5.8: Application for deployment

Scaling Step	Elapsed RTT
1	0.084638 sec
2	0,124790 sec
3	0,147473 sec

Table 5.10: Measured RTT results

Component	Functionality
G1	Transmits a message to the singular G2 instance and waits until receiving the same message from the singular G4 instance.
G2	Receives a message from all its source flows and then floods it to all the G3 instances.
G3	Forwards each incoming message to the singular G4 instance.
G4	Receives a message from all its source flows and then floods it to all the G1 instances.

Table 5.12: Functionality per component

- The first scaling step is shown in Figure 5.5. The cluster consists of a single device in which all instances are running. We can observe that the counted RTT is equal to the RTT measured in the previous example [5.2.2]. This is because if only one instance of each component is running, the aggregation forwarding pattern is exactly the same as the transmission to the same flow-id one.

- The second scaling step is shown in Figure 5.6. We calculate the average RTT by gathering the metrics both F1 instances generate. We can observe that the RTT counted in this experiment is larger than the RTT measured in the previous example by 0.01 seconds ($\cong$ 6.53% increase). This additional delay is caused by the aggregation pattern that we apply in both singular components. Specifically, in the aggregation method, each singular instance collects data from all the incoming streams before it broadcasts a message to the right. Instead, in the ”same flow-id transmission” method, each singular instance serves the incoming streams circularly having a performance advantage especially when all the incoming streams send data at the same rate.

- The third scaling step is shown in Figure 5.7. We calculate the average RTT gathering the metrics all G1 instances generate. Henceforth, as we scale the cluster, a constantly increasing additional delay will be introduced in the average RTT due to the aggregation forwarding pattern we apply in both singular instances. As we increase the incoming streams for each singular instance, the aggregation pattern will introduce more latency.

5.2.4 Average Scaling Latency

In this experiment, we calculate the latency introduced from the scaling functionality. Specifically, we deploy the graph presented in the two previous examples, this one shown in Figure 5.4, and we measure the time interval from the moment the system is informed about the insertion of a new device into the cluster until the non-singular G1 and G3 instances are generated on the new device. It is important to mention that we have excluded the components' image fetching time from the counted average time. We examined the same experiment with various types of devices inserted into the cluster each one with different hardware specifications. Table 5.13 summarizes the obtained results.

	Cloud Node	**Edge Node**	**End Node**
CPU	Intel i7-8750H	Cortex-A72 (ARM v8)	Cortex-A53 (ARM v8)
RAM	8 GB	4 GB	1 GB
OS	Ubuntu 20.04.2	Ubuntu 20.04.2	Ubuntu 20.04.2
Cores (threads)	6 (12)	4 (4)	4 (4)
Network Interface	Ethernet 100 Mbps	Ethernet 100 Mbps	Ethernet 100 Mbps
Total Scaling Time	6.71 sec	7.65 sec	49.6 sec
Container Generation Overhead	0.32 [x2] sec	1.05 [x2] sec	19.6 [x2] sec
Average Scaling Latency	**5.5 sec**	**5.5 sec**	**5.5 sec**

Table 5.13: Scaling time results

From what has been presented, we conclude that the scaling latency is about 5.5 seconds. It is important to mention that we subtract the container generation overhead of both the created instances in order to compute the average scaling latency. By improving the hardware features of the inserted device, the total scaling time can be reduced.

5.3 Migration Performance

In this section, we conduct several experiments to figure out the performance of the migration functionality.

5.3.1 Average Migration Latency

In this experiment, we measure the latency introduced by the system for migrating an instance to a new compatible device. Specifically, we have a cluster that consists of two devices as shown in Figure 5.10 and we deploy the application presented in Figure 5.9. It is vital to mention that there is no real traffic between the instances, so the links' reliability factor is not considered. The scaling factor of the application is set to ”1”, therefore all the instances are placed to the same device leaving the second device available to host the migrated instances, as shown in Figure 5.11. We try to migrate the singular G2 instance by typing the command **”move G2 from A to B”** into the system command prompt. We examined the same experiment with various types of devices inserted into the cluster, each one with different hardware specifications. Table 5.14 summarizes the obtained results.

	Cloud Node	**Edge Node**	**End Node**
CPU	Intel i7-8750H	Cortex-A72 (ARM v8)	Cortex-A53 (ARM v8)
RAM	8 GB	4 GB	1 GB
OS	Ubuntu 20.04.2	Ubuntu 20.04.2	Ubuntu 20.04.2
Cores (threads)	6 (12)	4 (4)	4 (4)
Network Interface	Ethernet 100 Mbps	Ethernet 100 Mbps	Ethernet 100 Mbps
Total Migration Time	6.58 sec	7.22 sec	26.94 sec
Container Generation Overhead	0.34 sec	0.98 sec	20.71 sec
Average Migration Latency	**6.24 sec**	**6.24 sec**	**6.23 sec**

Table 5.14: Migration time results

From what has been presented, we reach the conclusion that the migration latency is about 6.24 seconds. It is important to note that we subtract the container generation overhead of the migrated instance in order to compute the average migration latency. By improving the hardware features of the inserted device, the total migration time can be reduced.

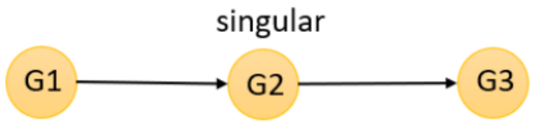


Figure 5.9: Application for deployment

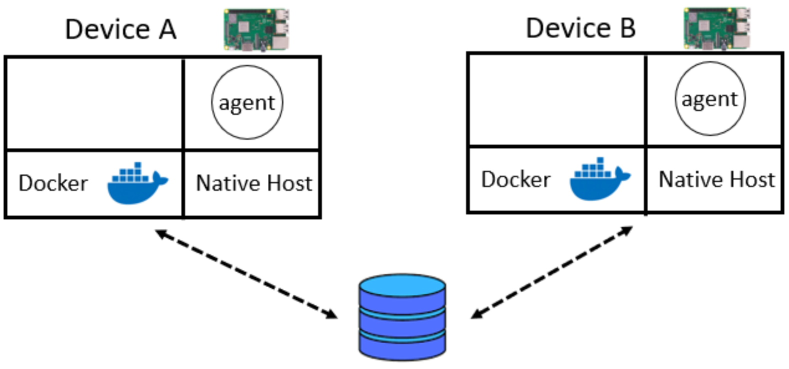


Figure 5.10: Migration [Step 1]

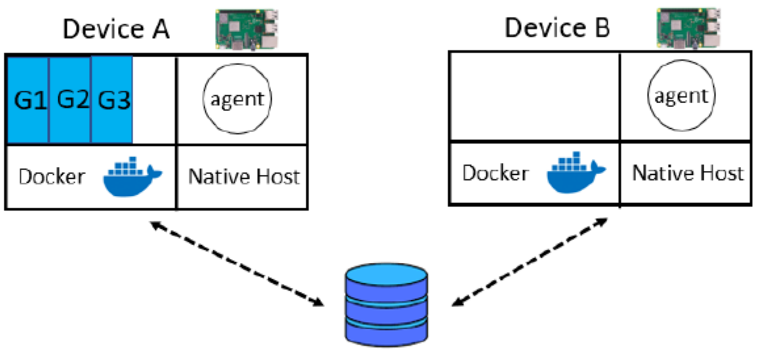


Figure 5.11: Migration [Step 2]

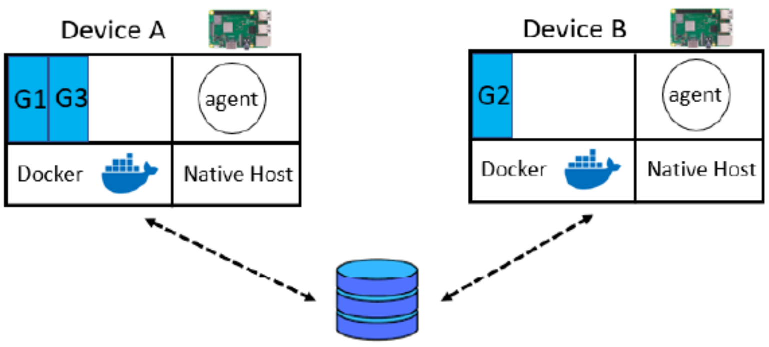


Figure 5.12: Migration [Step 3]

5.3.2 Unreliable Migration Time

In this experiment, the deployed application is identical to that of the previous example [5.3.1]. However, there is real traffic between the components, and all the instances are connected with links that have been marked as unreliable in the configuration file. We measure the average time it takes for the singular G2 instance to be migrated to a new compatible device. We examined the same experiment with numerous incoming and outcoming links connected with the singular instance we try to migrate. The results are exactly the same as those presented in Table 5.14. When a link is marked as unreliable and the system needs to redirect or terminate this connection, there is no need for applying a reliability check to ensure that all the data transmitted have reached their destination securely and orderly. Therefore, there is no additional overhead introduced by the system.

5.3.3 Reliable Migration Time

In this experiment, the deployed application is identical to that of the previous example [5.3.2]. However, all the instances are connected with links that have been marked as reliable in the configuration file. We measure the average time it takes for the singular G2 instance to be migrated to a new compatible device as well as the elapsed time until the buffer's movement process is completed. Figure 5.13 summarizes the obtained results.

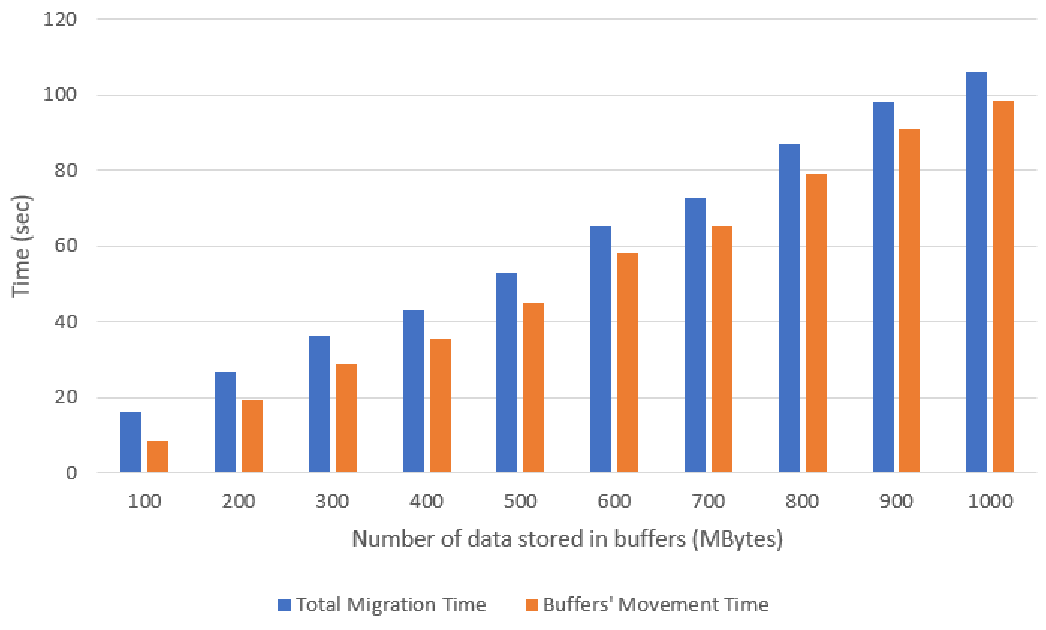


Figure 5.13: Reliable migration time results

We can observe that the total migration time increases linearly as we increase the data stored in the buffers of the migrated instance by a step of 100 MBytes. Therefore, we can conclude that the minimum reliable migration latency is about 7.45 seconds. If we increase the number of ingress and egress links of the migrated instance, the migration latency will be increased respectively.

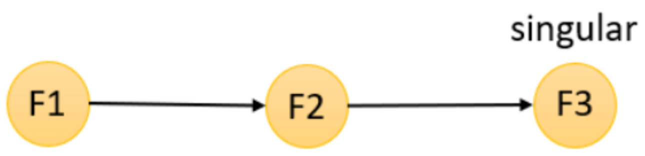
5.4 Application

In this section, we deploy a real-world application whose purpose is to monitor a security-sensitive area by processing camera feeds in order to recognize movement and unwanted persons entering the area. The application presented in Figure 5.14 consists of three components, named F1, F2 and F3. Table 5.16 summarizes the functionality of each component.

Component	**Functionality**
F1	Reads consecutive video frames from an attached camera and sends each frame to the F2 instance that belongs to the same stream without employing any compression technique.
F2	Receives frames from the F1 instance that belongs to the same stream. For each input frame, it applies a face detection algorithm in order to detect individual faces and crops the respective frame areas. Each cropped face is sent to the singular F3 instance. The face detection algorithm uses Haar Feature based Cascade Classifiers [7] on the input frames. Also, we use the OpenCV library [8] to fetch the frames from the attached camera.
F3	Matches each cropped image received from an F2 instance against the images of different faces previously stored in a database using the Local Binary Patterns Histograms face recognition algorithm [9]. If there is a similarity greater than a threshold, it generates an alert message. These algorithms are already implemented as part of the OpenCV library. In addition, we used the MongoDB database [10] for storing the received images.

Table 5.16: Functionality per component

Figure 5.14: Application for deployment



This application can be implemented using a low-cost local infrastructure, deploying the F1 instances in the area of interest. This can be achieved by adapting the F1 component's placement preferences in the configuration file. The core image processing and face recognition functions can be performed on powerful machines in the cloud. While this approach has the well-known advantages of cloud computing, it also comes with important scalability and responsiveness issues. First and foremost, the available network bandwidth of all end devices hosting an F1 instance may not be sufficient to push a high number of video frames to the cloud. Secondly, the latency of the Internet and the relaxed quality of service guarantees of cloud systems can lead to increased end-to-end communication delays. As an alternative approach, the core processing functions of the application can be implemented on the edge or even on the end device itself, leading to better performance.

In this experiment, we used two symmetrically opposite setups, in order to investigate the effects of instances' placement on a cluster of devices spanning the edge/fog/cloud system. Table 5.17 summarizes the deployment scenarios. In all cases, the F1 instance is placed on an end device, located in the area of interest. We used a video dataset as input, with a duration of 22 seconds. The video contains a total of 216 frames, each one having an image resolution of 500x900 pixels. It is important to mention that both the first and the last frames contain a face. We measure the total time until the last face (equivalent to the last frame) is identified from the F3 instance resulting in an alert generation.

Deployment Scenario	**End Device**	**Edge Device**
F1->F2,F3	F1	F2, F3
F1,F2->F3	F1, F2	F3

Table 5.17: Deployment Scenarios

5.4.1 Bandwidth Measurements

First of all, we recorded the amount of data that would travel over each link of the graph. Table 5.18 summarizes the results. We observe that the placement of application components significantly affects the amount of data that travels over the Internet. Specifically, placing an F2 instance on the end device (where an F1 instance is already running) rather than on a remote edge machine, reduces the total amount of data sent over the Internet by 88%.

	Application	**UDP/IP**
F1->F2	291.6 MB	291.67 MB
F2->F3	31.32 MB	32.71 MB

Table 5.18: Transmitted Data

5.4.2 Slow uplink, fast edge machine

In the first set of experiments, we test both deployment scenarios using a setup where the end device has powerful hardware characteristics but is connected via a low capacity link to the edge device ($\cong$ 24 Mbps). Table 5.19 presents the hardware specifications of each device. It is important to mention that the rate at which F1 transmits frames towards F2 is artificially restricted from the TCP protocol's congestion control algorithm. Figure 5.15 shows the obtained results.

	Edge Device	**End Device**
CPU	Intel i7-8750H	Intel i5-4200U
RAM	8 GB	4 GB
OS	Ubuntu 20.04.2	Ubuntu 18.04.5
Cores (threads)	6 (12)	4 (8)

Table 5.19: Device Hardware Specifications

The whole bar shows the total execution time. The orange part shows the delay due to the processing that is performed by the involved application components until the F3 instance receives and processes the last frame which is equivalent to the last detected face. The generated alert for the motion detection includes the processing that is performed from the application components F2 and F3. The blue part shows the communication delay and it is relevant to the link's capacity.

We can observe that when the F2 instance is placed on the end device, where the F1 instance that belongs to the same stream is running (experiment F1,F2->F3), the total execution time is drastically lessened. This is because the F2 instance reduces the number of data that travel upstream towards F3 by 88% leading to performance improvement. Furthermore, when the F2 instance is placed on the end device, the processing time of the face detection algorithm is minimized due to the improved hardware characteristics of the end device. Instead, when the F2 instance is placed on the edge device (experiment F1->F2,F3), the F1 instance transmits all the input frames via the limited capacity link to the F2 instance introducing additional communication delay.

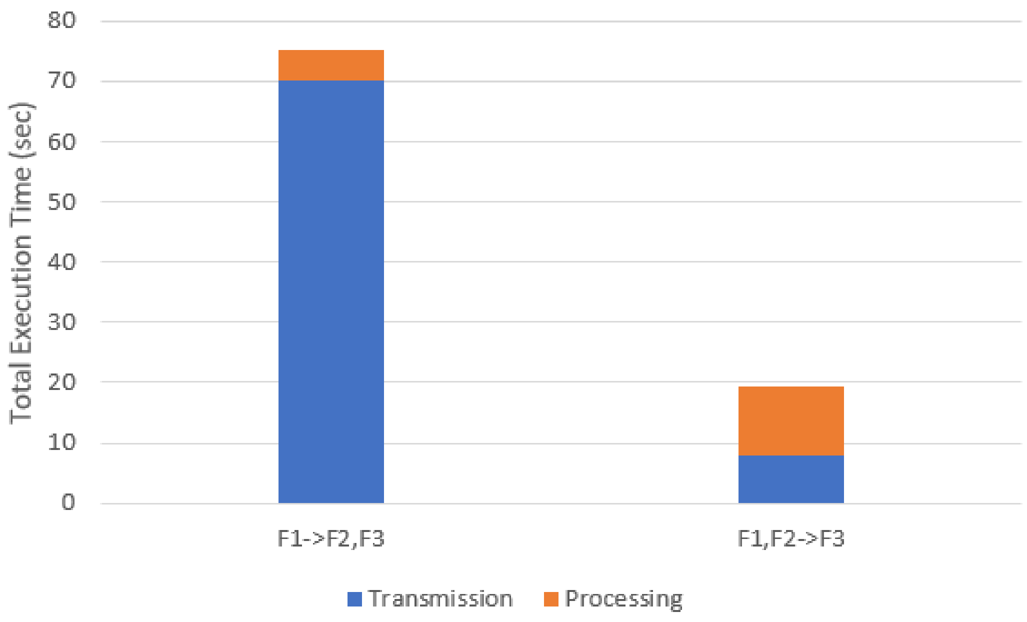


Figure 5.15: Slow uplink, fast edge machine

5.4.3 Fast uplink, slow edge machine

In the second set of experiments, we test both deployment scenarios using a setup where the end device is much weaken than in the first set of experiments but is connected via a high capacity link to the edge device ($\cong$ 95 Mbps). Table 5.20 presents the hardware specifications of each device. It is important to mention that the rate at which F1 transmits frames towards F2 is artificially restricted from the TCP protocol's congestion control algorithm. Figure 5.16 shows the obtained results.

	Edge Device	End Device
CPU	Intel i7-8750H	Cortex-A72 (ARM v8)
RAM	8 GB	4 GB
OS	Ubuntu 20.04.2	Ubuntu 20.04.2
Cores (threads)	6 (12)	4 (4)

Table 5.20: Device Hardware Specifications

We can observe that when the F2 instance is placed on the end device, where the F1 instance that belongs to the same stream is running (experiment F1,F2->F3), the total execution time is drastically reduced, as we mentioned in the previous example [5.4.2]. Furthermore, when the F2 instance is placed on the end device, while the processing time of the face detection algorithm is increased due to the insufficient available hardware resources of the end device, it is small to compete with the gain from reducing the amount of data transmitted upstream towards F3. Instead, when the F2 instance is placed on the edge device (experiment F1->F2,F3), the F1 instance transmits all the input frames to the F2 instance, leading to an increased communication overhead despite the high capacity of the link between the end and the edge device.

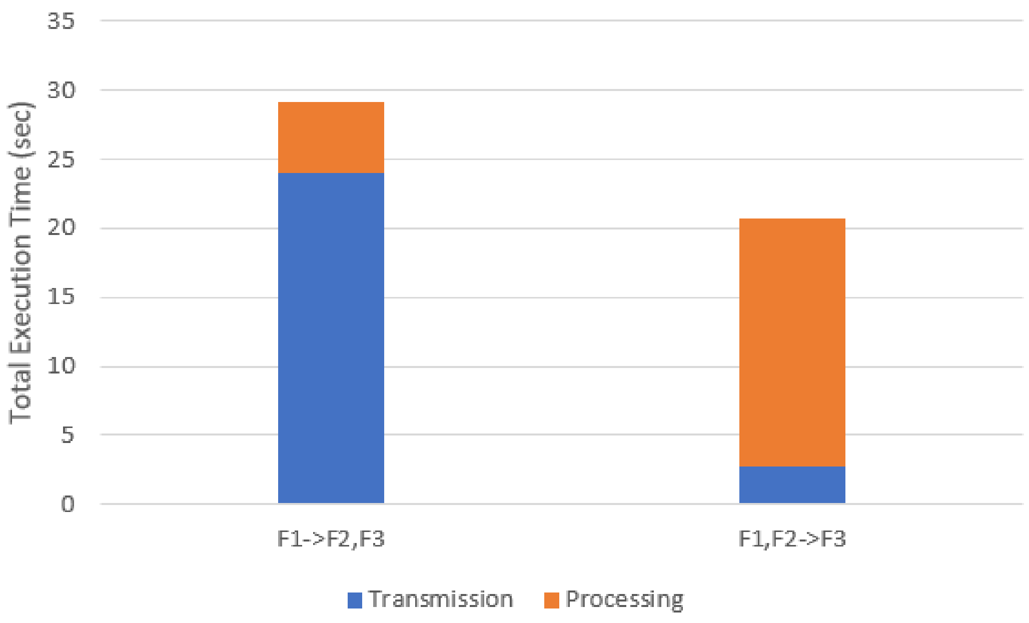


Figure 5.16: Fast uplink, slow edge machine

5.4.4 Summary

The above results show that, even in a relatively simple application scenario, performance only improves by placing the right part of the application at the edge. The hardware characteristics of the low-cost end devices as well as the communication links between the end and the edge devices greatly affect the performance. Blindly placing some application components at the edge can greatly degrade performance, even if the communication link to the edge is very fast. On the other hand, if the link to the edge is slow and the computing infrastructure of the end device is sufficiently powerful, there is potential for significant improvement.

Chapter 6

Related Work

Balena [11] is a complete set of tools for building, deploying and managing fleets of connected Linux devices. The core Balena platform, encompasses device, server, and client-side software, all designed to get the code securely deployed to a fleet of devices. The user can push code to the Balena build servers, where it will be packaged into containers and delivered to the fleet. Therefore, Balena provides static deployment of multi-container applications. Specifically, at deployment time, Balena generates a replicate of the multi-container application on each device of the fleet. However, each replicate runs in an isolated environment without interacting with replicates operating on another device. In addition, there is no support for singular containers as well as for custom deployments based on users' requirements and preferences. Moreover, scaling and migration functionalities are not supported.

Another way for developing applications that can be distributed flexibly is to compose them out of microservices. Individual microservices could then be grouped into larger clusters and be deployed on remote hosts through a suitable container system like Docker and a deployment system like Kubernetes [12] or Docker Swarm [13]. The microservices can be connected via an overlay network. However, there is no support for custom deployments based on users' requirements and preferences.

Flogo [14] is a recent framework, allowing the developer to build an application flow through a graphical interface. The main difference is that the application components are written in the Go programming language. Also, Flogo does not provide deployment support forcing the programmer to take care of the deployment of the components on the respective machines.

Chapter 7

Conclusion

We designed a system that is responsible for the deployment of modular applications in a heterogeneous cluster spanning the edge/fog/cloud. Moreover, we offer flexibility and transparency in the design process of such a component-based application, applying a structured dataflow approach. We have also presented experimental results using realistic test-cases, illustrating the performance of each provided functionality including dynamic scaling, reliable and unreliable migration and cluster monitoring. We examined the experiments using devices with various hardware specifications placed in a heterogeneous cluster.

In the future, we wish to implement a failure detection mechanism in order to detect device failures during runtime and adjust the deployment. Furthermore, an automatic migration service could be implemented for the system to dynamically migrate the running instances based on user-specified requirements such as throughput, latency, or quality of service. Another direction is to investigate more thoroughly the deployment trade-offs between a larger variety of devices spanning the edge/fog/cloud system.

Bibliography

[1] Edge-fog-cloud architecture. https://info.varnish-software.com/blog/edge-cloud-fog-computing.

[2] Docker architecture. https://docs.docker.com/get-started/overview/.

[3] What is docker. https://www.ibm.com/cloud/learn/docker.

[4] What is a docker container. https://www.docker.com/resources/what-container.

[5] Docker buildx. https://docs.docker.com/buildx/working-with-buildx/.

[6] Moby buildkit toolkit. https://github.com/moby/buildkit.

[7] Haar cascade classifiers. https://towardsdatascience.com/computer-vision-detecting-objects-using-haar-cascade-classifier-4585472829a9.

[8] Opencv. https://opencv.org/.

[9] Local binary patterns histograms. https://www.section.io/engineering-education/understanding-facial-recognition-using-local-binary-pattern-histogram-algorithm/.

[10] Mongodb. https://www.mongodb.com/.

[11] Balena. https://www.balena.io/what-is-balena.

[12] Kubernetes. https://kubernetes.io/.

[13] Docker swarm. https://docs.docker.com/engine/swarm/.

[14] Project flogo. https://www.flogo.io/.