Multi-tier program deployment and execution over the cloud, edge and end-devices

Thesis by
Alexandros Patras

For Graduation of Bachelor of Science
Department of Electrical and Computer Engineering

supervised by:
Spyros Lalis, Associate Professor
Christos D. Antonopoulos, Assistant Professor

UNIVERSITY OF THESSALY
Volos, Greece

12th October
2017

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my thesis supervisor, Dr. Spyros Lalis. His guidance was the most valuable and essential for the completion of my thesis as well as for the years of my studies. He was always available for support and advice, not only for academic affairs, but also for my life and my future career. His active and enthusiastic involvement in the projects we collaborated on, encouraged and enabled me to be a better engineer.

I would also like to show my gratitude to my second supervisor, Dr. Christos D. Antonopoulos for his participation and support on my research work. His advice and comments on crucial aspects of the research helped to improve the quality of the results.

I am also grateful to my Department's systems administrator, Mr. Athanasios Fevgas. His support and work allowed me to use the necessary network resources of the University for the purposes of my research.

A very special gratitude goes to Manos Koutsoubelias, Nasos Grigoropoulos and Christos Kalogirou, PhD students of the Computer Systems Lab of my Department. Their support and encouragement to important stages of my research can not be disregarded, and for that, I thank them.

And finally, last but by no means least, i would like to thank the people that were close to me, my family and my friends: My parents that were by my side all the time with their remarkable patience. My siblings for their support with every mean they could. And my friends for the encouragement to go on.

This research work was supported in part by the Horizon 2020 Programme of the European Union, project UniServer, contract number 688540.

ABSTRACT

The emergence of the edge/fog computing paradigm has increased the programming complexity of applications so that they can work seamlessly in the new distributed and heterogeneous system landscape. In this research, we investigate a structured dataflow approach which simplifies application development and offers great flexibility regarding the deployment of the application across end-devices, edge computing infrastructure and remote cloud systems. Our prototype is built on top of the Node-RED framework, with extensions in order to support the transparent deployment and distributed execution of application flows. We use a real-world application example to illustrate our approach as well as to explore the performance trade-offs for different deployment scenarios on real distributed computing setups.

ΠΕΡΙΛΗΨΗ

Η εμφάνιση του παραδείγματος του edge/fog computing έχει αυξήσει την πολυπλοκότητα προγραμματισμού εφαρμογών έτσι ώστε να μπορούν να λειτουργούν άψογα στο νέο τοπίο των κατανεμημένων και ετερογενών συστημάτων. Σε αυτήν την έρευνα, διερευνάμε μια προσέγγιση δομημένης ροής δεδομένων, η οποία απλοποιεί την ανάπτυξη εφαρμογών και προσφέρει μεγάλη ευελιξία όσον αφορά την ανάπτυξη και εκτέλεση της εφαρμογής σε τελικές συσκευές, σε υπολογιστικές υποδομές στα άκρα του δικτύου καθώς και σε απομακρυσμένα συστήματα στο νέφος. Το πρωτότυπο μας είναι χτισμένο πάνω από το εργαλείο Node-RED, κάνοντας χρήση των επεκτάσεων του για να υποστηρίξουμε τη διαφανή ανάπτυξη και τη κατανεμημένη εκτέλεση ροών εφαρμογών. Χρησιμοποιούμε ένα παράδειγμα εφαρμογής που χρησιμποιείται στον πραγματικό κόσμο για να καταδείξουμε την προσέγγισή μας καθώς και για να διερευνήσουμε τους διάφορους συμβιβασμούς που πρέπει να γίνουν για την απόδοση της εφαρμογής σε διαφορετικά σενάρια ανάπτυξης σε μια πραγματική κατανεμημένη υπολογιστική υποδομή.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

A large number of IoT and pervasive computing application scenarios revolve around a rather simple architectural approach, whereby low-end and/or mobile devices send data to and receive actuation/control requests from powerful server machines in the cloud. This approach, though straightforward to implement, has several drawbacks. Firstly, a large amount of low-level data are routed over the Internet to remote machines, leading to scalability issues. Secondly, due to the latency of the Internet, it may not be possible to support control/feedback loops with tight real-time constraints. Last but not least, privacy-sensitive data ends-up in the cloud from where it may leak to third parties, either intentionally for business purposes or unintentionally as a result of attacks.

An alternative approach is to adopt a more complex system architecture, which allows part of the data processing and decision making to be performed at the edge, on machines close to the end-devices, or in part even directly on the end-devices themselves. However, writing applications that span across end-devices, edge computing infrastructure and remote cloud systems is a non-trivial task: the developer has to structure the application in different parts; each part must be written/prepped so that it can run on the target host; the interface between the different parts of the application has to be cleanly defined and implemented so that it can be performed over a network; finally, each part must be installed on its host, and be properly instantiated and linked together with other parts of the application. And this process has to be repeated, in the worst case from scratch, if one wishes to apply / experiment with a different deployment.

This thesis presents the work that we have done to support the programmer in the above task, by adopting a combination of component-based and dataflow-oriented programming. More specifically, we let an application be expressed as a graph, where the nodes represent components and the edges between them represent uni-directional links used for data exchange. The developer also provides hints on how components should be placed on the hosts of the system. At deployment time, the sub-graphs are instantiated on the target hosts, along with automatically generated connector logic that takes care of inter-component binding and communication

over the network. To accelerate prototyping, we have built our support on top of Node-RED [1], which provides several features that are in line with our vision. Our contribution is in the extensions that support transparent component deployment and inter-component binding and communication over the network, without touching the Node-RED core. Moreover, we perform experiments on two real distributed setups and provide performance results that show the benefits and trade-offs of a more flexible application deployment that exploits computing resources at the edge with different computing capabilities and network links.

The rest of the thesis is structured as follows. Chapter 2 describes an indicative application example, where the edge computing paradigm is particularly appropriate. Chapter 3 gives an overview of the Node-RED framework. Chapter 4 presents the extensions made to Node-RED to enable a more flexible application deployment and execution, while Chapter 5 discusses the results of our performance experiments. Chapter 6 outlines related work. Finally, Chapter 7 concludes the thesis and defines directions for future work.

Chapter 2

MOTIVATION

2.1 The transition from cloud to edge/fog computing

The cloud computing industry is currently searching ways to better accomodate the large number of connected devices and to provide more reliable and scalable services to Internet of Things (IoT) applications. However, the current approach, illustrated in Figure 2.1, has significant scalability limitations. The data that is produced at the periphery of the Internet grows rapidly due to the ever increasing number of sensors found in smart cities, smart buildings, smart vehicles and portable/wearable personal devices. The data volume is expected to grow from 1.1 zettabytes per year in 2016 to 2.3 zettabytes per year by 2020 [2]. Sending all this data across the Internet to the cloud is clearly not a viable option. This is not just a bandwidth problem. Some of the applications have rather strict latency/reaction requirements, which may be hard or even impossible to meet if data travels a long way in order to be processed by machines in a distant location.

Figure 2.1: An illustration of the cloud architecture.

The problem can be addressed by adding a layer between the endpoint devices and the cloud, as shown in Figure 2.2. This has two advantages. On the one hand, one can take advantage of high-speed and/or low-latency network links between edge and end-devices to reduce the delay of notifcations or actuation actions. On the other hand, it becomes possible to offload part of the processing that would have been performed on the cloud, to computing infrastructure at the edge.

Figure 2.2: An illustration of the edge architecture.

2.2 Application Example: Camera-based Security

Several applications have been identified in the literature as typical examples where edge/fog computing can be advantageous compared to conventional cloud computing. Here we focus on one such application, which is based on the Airport Visual Security System case study that is described in the OpenFog Reference Architecture [2]. For the purpose of our work we have simplified and adapted the application while keeping its most salient features and performance-critical aspects.

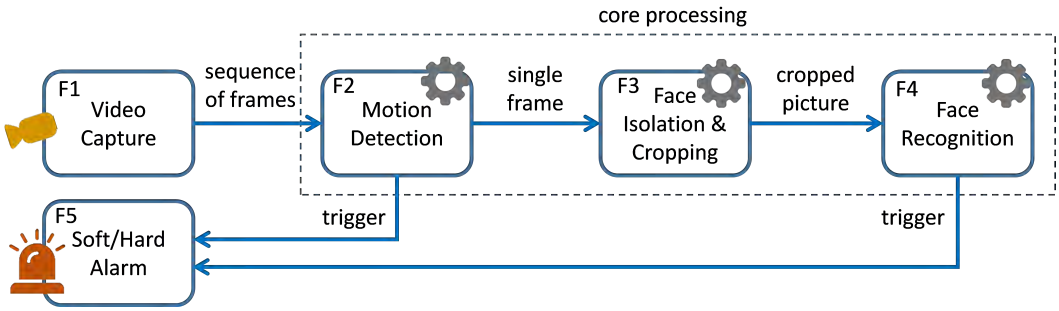


Figure 2.3: High-level functional diagram for the camera-based security application.

In a nutshell, the purpose of the application is to monitor a security-sensitive area by processing camera feeds in order (a) to recognize movement and (b) unwanted persons entering the area, and issue respective notifications to nearby personnel or even take an automated action, such as raising an alarm or locking a door. As shown in Figure 2.3, the application can be partitioned in severel high-level functions:

(F1) take video of the area of interest; (F2) detect motion in the video frames; (F3) detect individual faces and crop the respective frame areas; (F4) match the cropped images against the images of different persons stored in a database; (F5) raise soft/hard alarms in case movement or an unwanted person is detected. Function F1 is achieved using camera sensors, F2, F3 and F4 are implemented using suitable image processing and face recognition algorithms, while F5 will typically trigger an actuator or user interface on some end-device.

This application can be implemented using a low-cost local infrastructure, deploying only the video cameras and alarms in the area of interest. The core image processing and face recognition functions can be performed by employing ready-to-use services offered by cloud providers like [3]–[5], or by running own/custom computer vision software such as [6], [7] on powerful server machines in the cloud. While this approach has the well-known advantages of cloud computing, it also comes with important scalability and responsiveness issues. Firstly, the available network bandwidth may not suffice to push a large number of video streams all the way to the cloud. Secondly, the latency of the Internet and the relaxed quality of service guarantees of cloud systems may introduce large end-to-end actuation and user notification delays.

As an alternative approach, the core processing functions of the application can be implemented in a modular fashion, as independent components that can run, if desired, on different machines —some of them located at the edge of the Internet— leading to better performance and scalability. In this particular case, while F2 takes as input a continuous stream of frames which can arrive at a high rate, it only outputs selected frames when motion is detected. Also note that F3 and F4 are activated only when F2 actually produces some output. Therefore placing F2 close to F1, for instance on a machine with a fast wired/wireless connection to the camera devices, which may also feature special hardware acceleration, might be a better choice in terms of both scalability and responsiveness of the application. To further reduce bandwidth requirements, F3 could be at the edge as well. On the other hand, if the network latency is sufficiently small, F4 could be hosted in the cloud on machines with lots of storage and powerful database support.

2.3 Development issues

In order to develop an application that can span across end-devices, the edge and the cloud, the developer must consider several aspects like software dependencies,

demands in terms of processing power and network latency, as well as the available capacity of the network links.

The developer must then partition the application into smaller pieces, and decide where each piece should be deployed. This typically translates into identifying core functionalality that can be cleanly split into distinct parts, in the spirit of the above application example (Figure 2.2). Moreover, the problem of machine and runtime platform heterogenity has to be addressed, depending on the desired partioning and deployment. A programming language or runtime framework that might be suitable on a cloud machine, is not necessarily suitable or even available for an edge machine, let alone an end-device. In the worst case, parts of the application may have to be re-written to be able to execute on the target platform.

Finally, communication logic must be developed to enable the interactions between the parts of the application that may be deployed and run on different machines. For each such interaction, one should consider the requirements in terms of reliability, and choose coresponding transport mechanisms/protocols. For instance, if the communication between two parts of the application is in the form of a stream (the communication between F1-F2 as well as between F2-F3 in the above example), this can be implemented using an unreliable datagram transport such as UDP/IP. In contrast, if two parts require reliable communication (the communication between F2-F5 and F4-F5), one would need to employ a reliable transport such as TCP/IP.

Chapter 3

AN IOT PROGRAMMING TOOL: NODE-RED

3.1 Node-RED Overview

Node-RED [1] is built on top of NodeJS [8], with the purpose of enabling rapid and simple development of IoT applications. It consists of (i) a collection of runnable software components called *nodes*, (ii) a graphical user interface (GUI) for picking nodes and linking them together into a flow graph, and (iii) a runtime environment for deploying and executing such flows. The collection of nodes can be enriched in an open-ended manner, and developers are free to write their own nodes that implement missing functions or custom glue-logic needed for the target applications. A large number of nodes embodying a wide range of different functions have already been contributed by the community, making Node-RED a very powerful component ecosystem.

The GUI allows the developer to interactively browse the list of available nodes, also referred to as *node pallete*, choose the ones to be used in the application, drag and drop them on the so-called *canvas*, and link them together into a flow that achieves the desired functionality. The linking between nodes is done by drawing a line that connects an output port of the source node with the input port of the destination node. When the application flow graph is finalized, it can be deployed by pressing a button. To give a concrete example, Figure 3.1 shows a Node-RED flow for the camera-based security application, following the component structure of Figure 2.3.

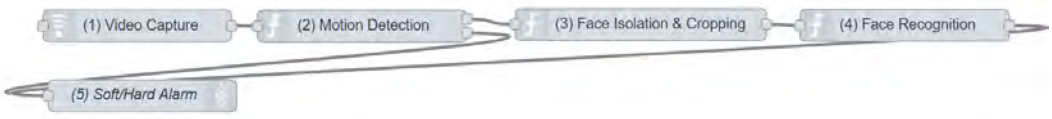


Figure 3.1: Node-RED flow graph for the camera-based security application.

Node components are written in JavaScript. According to the conventions of the Node-RED framework, nodes can have at most one input port and zero or more output ports; they may also communicate through shared objects stored in a global or flow-specific context. The actual node logic resides in a handler function, which is invoked by the Node-RED runtime environment when a message arrives at the input port in order to processes it and produce messages for the output ports. Apart

```
// Sample Node-RED node file
module.exports = function(RED) {
    "use strict";
    // require any external libraries we may need.
    //var foo = require("foo-library");

    // The main node definition - most things happen in here
    function SampleNode(n) {
        // Create a RED node
        RED.nodes.createNode(this,n);

        var node = this;

        // Do whatever you need to do in here - declare callbacks etc
        // Note: this sample doesn't do anything much - it will only send
        // this message once at startup.
        var msg = {};
        msg.topic = this.topic;
        msg.payload = "Hello world !"


        // respond to inputs....
        this.on('input', function (msg) {
            node.warn("I saw a payload: "+msg.payload);
            // in this example just send it straight on. Should process it here really
            node.send(msg);
        });
    }

    // Register the node by name. This must be called before overriding any of the
    // Node functions.
    RED.nodes.registerType("sample",SampleNode);

}
```

Figure 3.2: JavaScript sample file that implements a custom node.

from simple functions, nodes can implement more complex data processing directly in JavaScript or by proxying other programs that are invoked through suitable middleware. One may also exploit the node-gyp [9] tool of NodeJS to create bindings between external libraries and nodes, and the NodeJS Package Manager to install contributed modules. Meta-information on the functionality, input/output ports and configuration options of a node is provided in a separate HTML file, which is accessed by the GUI in order to guide the user during the interactive node selection, linking and configuration process.

Finally, the Node-RED runtime environment takes as input a flow graph and deploys it on the local machine. More specifically, it creates an instance for every node in the graph, and routes messages between the nodes as dictated via the respective links.

3.2 Custom node implementation

The implementation of custom nodes is simple, and Node-RED [1] provides samples of the two necessary files that are shown in Figure 3.2 and Figure 3.3. The user needs to implement the message handler to respond to inputs, and configure the HTML file for the needed configuration settings.

```
<!-- First, the content of the edit dialog is defined.                    -->
<script type="text/x-red" data-template-name="sample">
    <div class="form-row">
        <label for="node-input-topic"><i class="fa fa-tasks"></i> Topic</label>
        <input type="text" id="node-input-topic" placeholder="Topic">
    </div>

    <br/>
    <!-- By convention, most nodes have a 'name' property. The following div -->
    <!-- provides the necessary field. Should always be the last option      -->
    <div class="form-row">
        <label for="node-input-name"><i class="fa fa-tag"></i> Name</label>
        <input type="text" id="node-input-name" placeholder="Name">
    </div>
</script>

<!-- Next, some simple help text is provided for the node.                -->
<script type="text/x-red" data-help-name="sample">
   <p>Simple sample input node. Just sends a single message when it starts up.
   This is not very useful.</p>
   <p>Outputs an object called <code>msg</code> containing <code>msg.topic</code> and
   <code>msg.payload</code>. msg.payload is a String.</p>
</script>
<!-- Finally, the node type is registered along with all of its properties   -->
<!-- The example below shows a small subset of the properties that can be set-->
<script type="text/javascript">
    RED.nodes.registerType('sample',{
        category: 'input',      // the palette category
        defaults: {             // defines the editable properties of the node
            name: {value:""},   //  along with default values.
            topic: {value:"", required:true}
        },
        inputs:1,               // set the number of inputs - only 0 or 1
        outputs:1,              // set the number of outputs - 0 to n
        // set the icon (held in icons dir below where you save the node)
        icon: "myicon.png",     // saved in  icons/myicon.png
        label: function() {     // sets the default label contents
            return this.name||this.topic||"sample";
        },
        labelStyle: function() { // sets the class to apply to the label
            return this.name?"node_label_italic":"";
        }});
</script>
```

Figure 3.3: HTML sample file that implements the graphical configuration settings of a custom node.

3.3 User Interaction

We illustrate the capabilities and usage of Node-RED, through a simple example of a "Hello world" flow, shown in the canvas area of the GUI in Figure 3.4.

This flow is created step by step. First, the user must drag and drop each node into the canvas, and configure each node according to the options dictated by the corresponding HTML file, as shown in Figure 3.5a. Then, the nodes need to be linked, by connecting an output port of the source/sender with the input port of the destination/receiver, as shown in Figure 3.5b. Of course, in order for this to work, the ports must be compatible, i.e., the messages sent over the output port must be of the same type as the messages that are expected to be received via the input port.

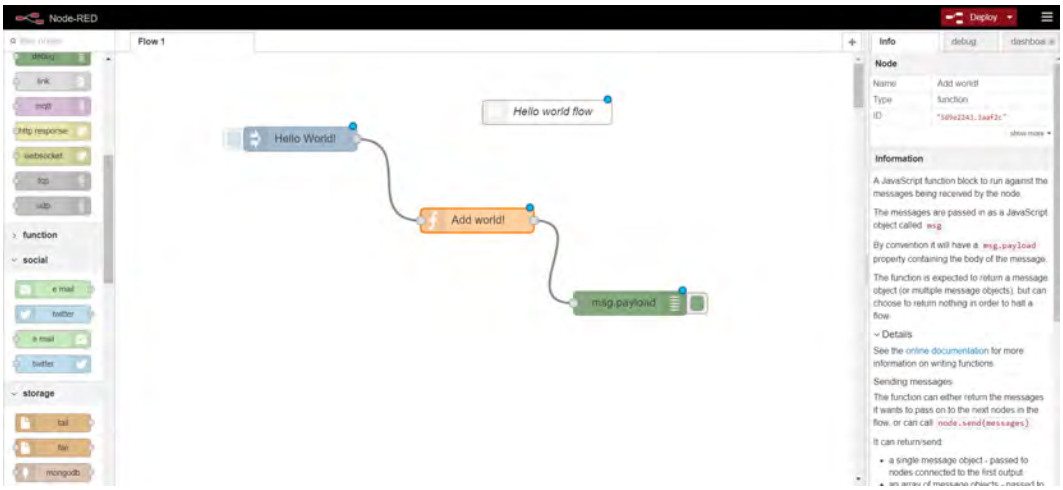
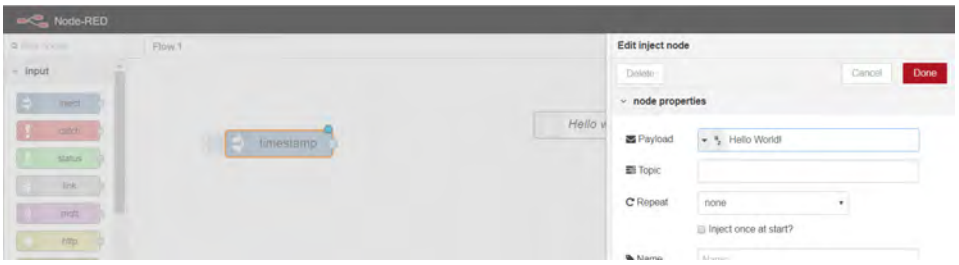
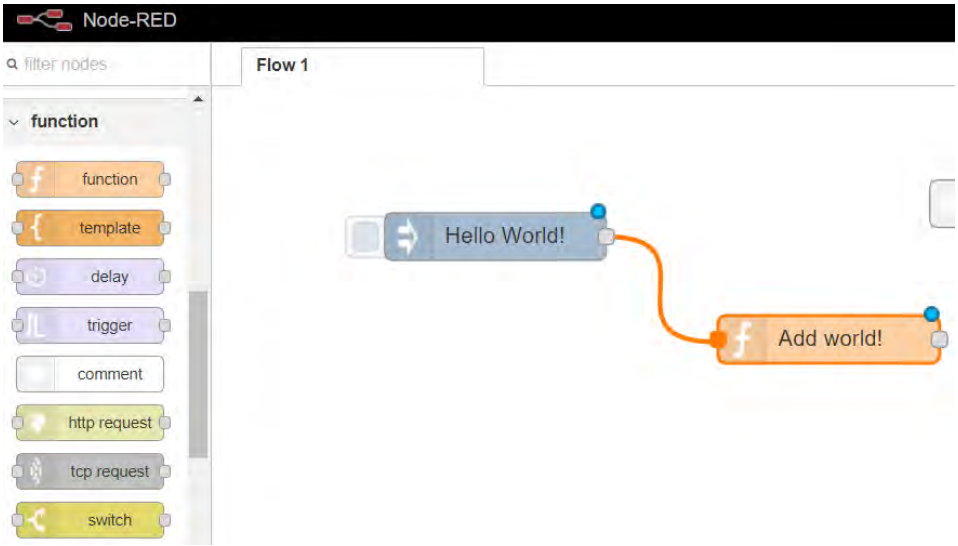


Figure 3.4: Screenshot from the GUI for a Hello World flow.



(a) Dropping and configuring nodes.



(b) Linking the nodes.

Figure 3.5: Step-by-step flow construction.

(a) Flow on process 1. (b) Flow on process 2.

Figure 3.6: Two flows that communicate with each other through suitable transport nodes.

Finally, when the flow has been constructed, the user presses the Deploy button, and the Node-RED runtime system handles the instantation and the running of each node as well as the message routing between connected nodes. Note that Node-RED is single-threded, so all nodes of a flow runs within a single instance of the Node-RED process.

To facilitate an interaction between nodes that reside on different Node-RED processes, on the same or on different machines, Node-RED offers built-in communication / data transport components. As an example, Figure 3.6 shows two distinct flows that can run on different processes/machines, with their respective UDP communication nodes and the appropriated configuration settings.

However, these communication components have to be manually added into each flow. They must also be appropriately configured by hand, in order to be able to communicate with each other; since they do not belong to the same flow, the usual drag-drop-configure process cannot be applied to them. Furthermore, Node-RED communication nodes only support the transfer of *String*, *Buffer* and *Base64 encoded Strings* objects. If two flows that run on separate processes/machines need to exchange any other type of data, the developer has to introduce corresponding data types and provide suitable data serialization code/methods. Last but not least, the two flows have to be instantiated separately on the target machines. So, while in principle it is possible to write distributed applications/flows, the process for doing this is not very smooth and introduces some hassle. What is worse, this needs to be repeated if it is desired to support/investigate a different deployment scenario.

Chapter 4

EXTENDING THE NODE-RED

4.1 Extending Node-RED for Flexible Distributed Computing

Node-RED is intended primarily for running application logic on the same machine, and the runtime environment takes care of component instantiation and message forwarding at a local scope only. While it is possible for an application to exploit functions running on remote machines, the respective software management and communication becomes the responsibility of the developer, and has to be done in a manual way, separately from the application flow that is built/run using Node-RED.

We have extended Node-RED so that it can be used to build applications that can span across machine boundaries in a transparent way. The main difference compared to regular Node-RED flows is that the developer has to specify the placement of nodes on hosts according to the desired deployment. Doing this is trivial, and one can explore different deployments scenarios with practically zero effort. For our prototyping purposes, we leave the core of the Node-RED framework untouched, and introduce our extensions on top of it while exploiting the out-of-the-box functionality as much as possible.

4.2 Extension Overview

For the orchestration of the system we adopt a centralized approach whereby a distinguished machine, called the *master*, acts as the application manager and coordinator for the distributed execution environment. The machines that can be used to host/run one or more application nodes, called *slaves*, run the conventional Node-RED environment. Slaves register with the master using a simple directory protocol, and accept requests for configuring and executing flows locally. This master-slave interaction is performed using the HTTP management API of the Node-RED framework. Figure 4.1 shows the high-level system architecture.

4.3 User Interaction

The developer builds the application flow on the master environment, by dragging-dropping nodes on the canvas and linking them together as usual. The user can find the machines that are available for hosting nodes and inspect their properties by interactively browsing the master directory. The placement of the nodes on hosts

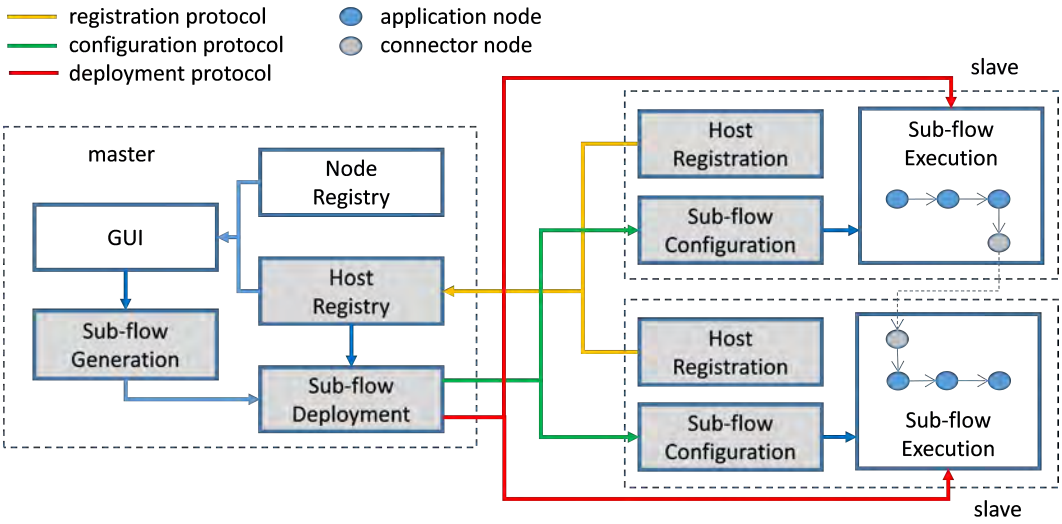


Figure 4.1: High-level view of the system architecture. Our extensions to the Node-RED framework are marked in grey. The protocol for the deployment of node flows is already supported by Node-RED.

is specified via a syntactical convention, namely by appending the host identifier as a suffix in the name of the node. For example, if node *Node* should run on host *Host*, the full name of the node component should be *Node_Host*. It is thus easy to define and change node placement, by setting these suffixes accordingly. Once the application flow is defined, the user can deploy and run it by pressing a button of the GUI.

As an example, Figure 4.2 shows the flow for the camera-based security application example (Figure 2.3). In this case, the developer has decided to place the nodes for F1 and F5 on the same endpoint device. Similarly, the nodes for F2 and F3 are also to be placed on the same edge machine, whereas the node for F4 on the cloud (the host names are intentionally kept simple).

4.4 Sub-flow Deployment

Based on the node placement information, the master splits the global application flow into sub-flows each comprising only nodes that should be co-located on the same host. For each link that spans across hosts, additional connector logic is generated, which comprises a communication endpoint and suitable data serialization for each side. This is implemented via proper nodes, re-using the networking and serialization support of Node-RED (plus application-specific serialization add-ons for custom data types, if any), which are then linked to the corresponding application-

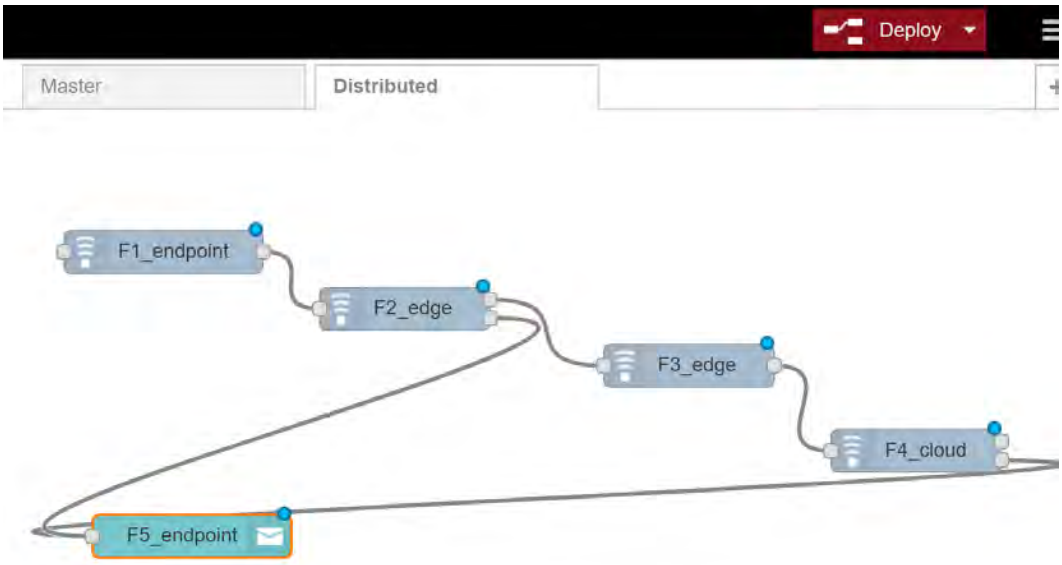


Figure 4.2: A distributed flow on the canvas of the master Node-RED environment.

level nodes in the respective local sub-flows. Finally, the master describes each sub-flow independently, in the form of a JSON file that follows the standard Node-RED flow specification format, and sends it for execution to the respective host. The allocation of the UDP/TCP ports for the communication endpoints on the hosts is done by the master via a suitable configuration protocol, before activating the individual sub-flows. The entire process, including the generation of the *connector* logic and required nodes installation, is automated and does not require any involvement of the application developer. Figure 4.3 shows the deployment/orchestration process between the master and the salves, in the form of a message sequence diagram.

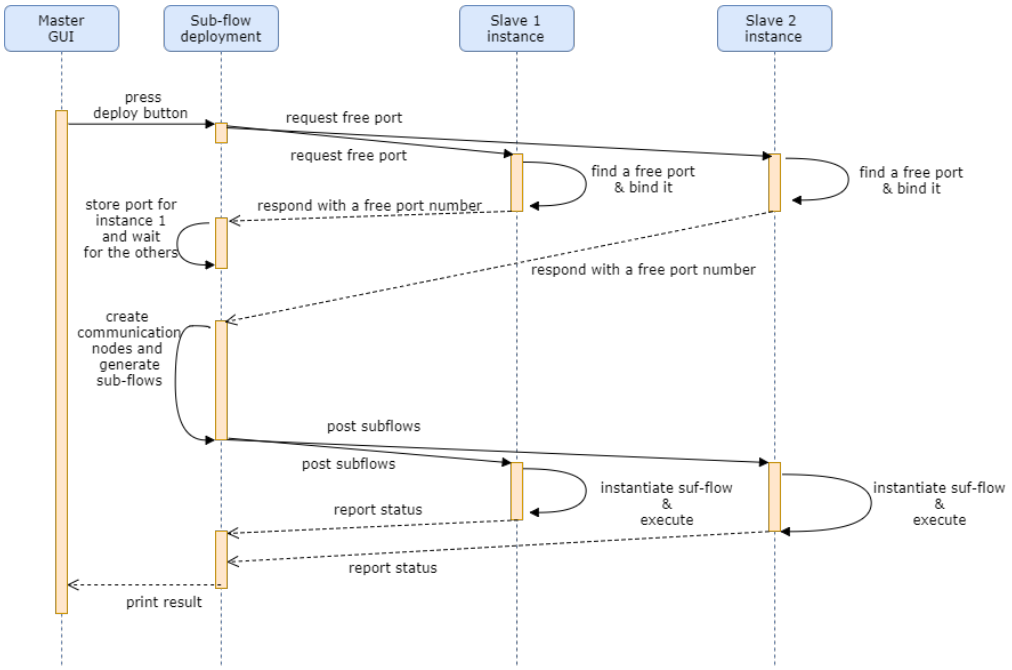


Figure 4.3: Sequence diagram for the sub-flow deployment/orchestration process.

Chapter 5

EVALUATION

5.1 Purpose

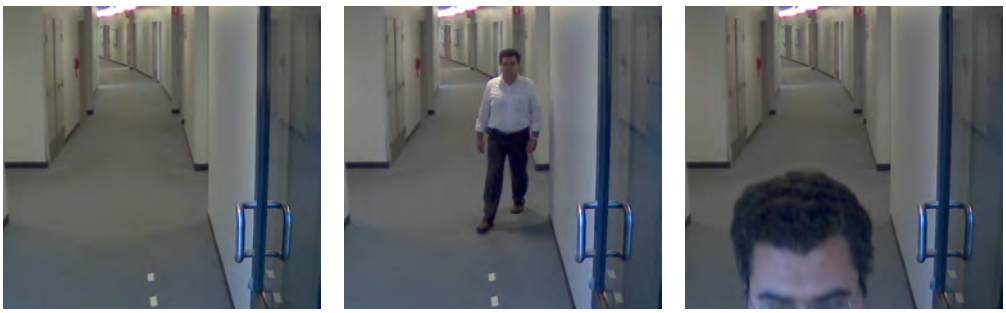
We conducted several experiments to check the robustness of our prototype, as well as to observe the performance of different deployments that combine computing resources at the edge and remote cloud infrastructures. In particular, we wanted to investigate the effects of edge environments with different communication links and different hardware platforms. To this end, we used two symmetrically opposite setups, which clearly demonstrate the importance of both factors.

5.2 Application implementation & datasets

Our experiments are performed for the camera-based security application described in Chapter 2. The application is structured as a Node-RED flow, as already discussed in Chapter 4.

We implement function F1 using a node that reads a sequence of video frames that are stored in JPEG format, simulating a video stream. The individual frames are transmitted "as is" without employing any compression technique. The node for F2 is based on existing software that compares subsequent frames [10]. If the difference between them is above a threshold, the node outputs the last frame and a corresponding notification towards F5. F3 runs a face detection algorithm that uses Haar Feature-based Cascade Classifiers on the input frames and produces cropped frames that contains a face in gray-scale color. F4 is implemented using the Local Binary Patterns Histograms face recognition algorithm on the incoming cropped frames, generating a notification in case a positive match is found. These algorithms are already implemented as part of the OpenCV library [6], and are accessed via respective NodeJS bindings [11]. Finally, F5 is a simple node that consumes the notifications coming from F2/F4 and prints a warning.

We use a small part of the ChokePoint [12] video dataset as input, with a duration of 22 seconds: the first 5.5 seconds show an empty corridor without any movement; during the next 5.5 seconds a person passes through the corridor facing the camera, but this face is not in the face database; the following 5.5 seconds are identical to the first phase; the last 5.5 seconds show another person whose face is included in the



(a) No motion. (b) Person entering the scene. (c) Person leaving the scene.

Figure 5.1: Indicative frames of the input video footage.

(a) A (familiar) face from the database. (b) Cropped frame generated by F3.

Figure 5.2: Faces samples from the database.

face database. The original video dataset has an image resolution of 800x600 pixels at 30 frames per second. To let each frame fit into a single UDP/IP packet and avoid frame fragmentation and reassembly at the application level, the video was cropped to 550x500 pixels. The cropped frames produced by F3 vary in size. We train the face recognition model using the Labeled Faces in the Wild face database that contains faces of 5,749 people in 13,233 images [13]. Figure 5.1 shows indicative frames of the video footage used. Figure 5.2 shows samples from the face database and the cropped frames generated by F3.

5.3 Bandwidth Measurements

In a single run that is performed on a single machine, we record the amount of data that would travel over the network if each of the application components were deployed on a different host. This is done by adding serialization and de-serialization nodes for each link between two components, along with the corresponding communication/transport nodes. We measure the size of application-level messages as

Table 5.1: Data traffic between the components of the camera-based security application. The bandwidth is estimated for the frame rate of the original video footage without employing compression.

	Application	**UDP/IP**	**Bandwidth @ 30 fps**
F1 -> F2	27.5 MB	55 MB	20 Mbps
F2 -> F3	10.3 MB	20.6 MB	3.7 Mbps
F3 -> F4	0.5 MB	1.1 MB	0.2 Mbps

well as the traffic at the level of the transport (in our case, UDP/IP) using the *iptraf* Linux tool [14]. Table 5.1 presents the results. We observe that the placement of application components significantly affects the amount of data that travels over the Internet. More specifically, placing F2 at the edge rather than on a remote cloud reduces the number of frames and total amount of data sent over the Internet by 60%. If F3 is placed at the edge as well, the outbound traffic drops by more than 95% (fewer frames, which are also much smaller in size due to face cropping).

These numbers are for uncompressed video, but also for just one camera. In a real-world setting, there could be numerous such video streams that need to be processed concurrently, leading to very high bandwidth requirements even if compression is employed. For instance, a H.264 or H.265 IP camera generates 12 Mbps at 30 fps [2]. This still amounts to approximately 1 TB/day, and to support just 10 such cameras with a pure cloud-based approach one would already require more than 100 Mbps of bandwidth. According to a recently published Akamai report on the state of the Internet, the global average peak speed of Internet connectivity stands at 44.6 Mbps at an average speed of just 7.2 Mbps [15]. Clearly, to support such an application at scale, one would have to run F2 and probably also F3 at the edge.

5.4 Notification delay

We measure the end-to-end delay for the motion detection notification issued by F2 and the face recognition notification issued by F4. Both delays are recorded at F5, when the respective notification messages arrive, using as a reference the time when F1 outputs frames with motion (we know the sequence numbers of those frames through manual inspection).

The deployment scenarios used in our tests are summarized in Table 5.2. In all cases, the nodes for F1 and F5 are co-located on a laptop computer that represents an end-device. The nodes for F2, F3 and F4 run either on a local personal computer that stands for the edge computing infrastructure, or on a remote cloud system, depending

Table 5.2: Deployment scenarios.

	End-Device	**Edge Computer**	**Remote Cloud**
Cloud	F1, F5	none	F2, F3, F4
Edge/Cloud 1	F1, F5	F2	F3, F4
Edge/Cloud 2	F1, F5	F2, F3	F4
Edge	F1, F5	F2, F3, F4	none

on the deployment scenario. The laptop (end-device) and edge host are connected to a LAN with a small ping latency, less than 3 milliseconds. The communication latency between the edge and cloud host depends on the configuration (see next).

To see how much time of the notification delay is due to processing, we also measure the computing time of each application component by recording the time that is spent in the respective message handler. Recall that Node-RED runs all application nodes of a local flow within a single thread, and thus all nodes that reside on the same host run within the same process. As a consequence, our experiments do not exploit/investigate the multi-core capability of the host machines.

To get a feeling of the communication overhead between application components that reside on different hosts, we measure the round-trip-time (RTT) for respective application-level ping-pongs using messages of the same size as the ones that are exchanged during the actual experiments. As an estimate for the one-way communication latency, we use RTT/2.

Slow uplink, fast edge machine

In a first set of experiments, we test all deployment configurations of the application using setup where the edge has an ADSL link to the Internet but features a powerful machine as a host for application components. Table 5.3 shows the key characteristics of the platforms used for the end-device, the edge host and the cloud host, while Table 5.4 reports the characteristics of the ADSL link and the latency between the edge and the remote cloud host. In order not to over-stress the (slow) uplink, the rate at which F1 produces frames towards F2 is artificially reduced to 1.25 frames per second.

Figure 5.3 shows the results (median over ten runs for each deployment). The whole bar shows the recorded notification delays. The blue part shows the delay due to the processing that is performed by the involved application components for each notification. The notification for the motion detection includes the processing that is performed from the application component F2, whereas the notification for the face

Table 5.3: Host platform characteristics for the setup with the ADSL link and a fast edge machine.

End-Device	Edge	Remote Cloud
Linux Atom CPU, 2C@1.66 Ghz 512 KB Cache, 1 GB RAM	Linux Image in VM I7 CPU, 4C@4.2 Ghz 8 MB Cache, 8 GB RAM	Linux via Docker I7 CPU, 8C@3.4 Ghz 8 MB Cache, 32 GB RAM

Table 5.4: Communication characteristics for the setup with the ADSL link to the Internet.

Download/Upload bandwidth	Ping latency 60 B	Ping latency 60 KB
24/1 Mbps	70 ms	750 ms

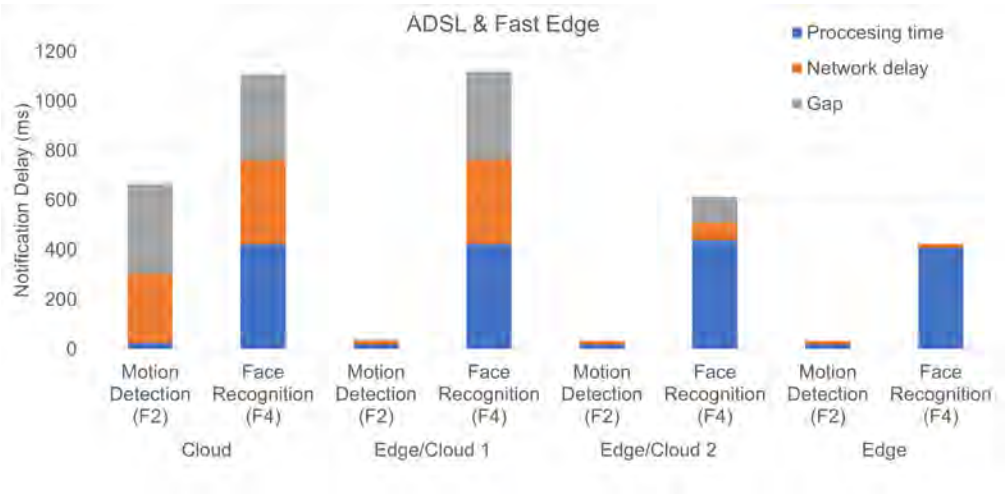


Figure 5.3: Notification delay for motion detection and face recognition for the setup with the ADSL link and a fast edge machine.

recognition includes the aggregated processing time of application components F2, F3 and F4. In either case, in the processing time is also included the aggregated time that is spent inside the serialization/de-serialization components. The orange part shows the aggregated end-to-end communication delay for the notifications, which is estimated based on the RTTs for the respective application-level ping-pong. The network delay for the motion detection includes the delay for the link F1-F2 and the link F2-F5, whereas the network delay for the face recognition includes the network delays for all the links that are between F1 and F5.

The notification for motion detection is much faster in all cases where F2 is hosted at the edge (Edge and both Edge/Cloud variants) vs. on the remote cloud system (Cloud). This is expected, because in all deployments F2 is hosted on equally fast

machines, but in the Cloud deployment it is further away from the end-device where F5 resides.

However, the results for the face recognition notification delay are not that intuitive. On the one hand, as expected, the Edge deployment has the lowest delay. On the other hand though, the face notification delay for the Edge/Cloud 1 deployment is slightly larger than that of the Cloud deployment, despite the fact part of the application (F2) runs at the edge on a host machine that is as powerful as the cloud host. This is due to the extra overhead of passing the data flow through the host at the edge before going to the cloud. While F2 reduces the number of frames that travel upstream towards F3, the amount of processing done by F2 (just 5 milliseconds per frame, in all configurations) is too small to outweigh the time it takes to relay a frame via the application layer (about 7 ms). This overhead is not visible in the Edge/Cloud 2 deployment because of the heavier aggregated processing of F2 and F3 on the edge host (a total of 55 ms per frame) but also due to the very significant reduction by F3 in the number and size of frames that flow out of the edge host to the cloud, leading to a lower face notification delay compared to the Cloud deployment.

Note that for the Cloud and both Edge/Cloud variants, the measured notification delays are significantly larger than the sum of the recorded application-level processing time and the estimated communication latency. This difference, which corresponds to the gray part of the bars in Figure 5.3, is attributed to the asymmetry of the ADSL link, which presumably results in a an uplink latency that is much larger than the RTT/2 that was measured for the application-level communication between the end-device and the cloud. No such gap exists for the Edge measurements, as in this case no communication takes place over the ADSL link.

Fast uplink, slow edge machine

In a second set of experiments, we test all deployment configurations of the application using setup where the edge has fast, fiber-optic link to the Internet but features a much weaker machine as a host for application components. Table 5.5 shows the key characteristics of the platforms used for the end-device, the edge host and the cloud host, while Table 5.6 reports the characteristics of the fiber link and the latency between the edge and the remote cloud host. To avoid over-stressing the (slow) edge machine, in this experiments the frame rate of F1 was artificially reduced to 1 frames per second (25% lower that in the previous experiments).

Figure 5.4 shows the results (median over ten runs for each deployment). As in

Table 5.5: Host platform characteristics for the setup with the fiber link and a slow edge machine.

End-Device	Edge Computer	Remote Cloud
Linux Image in VM Core2 Duo CPU, 2C@2.20 Ghz 2 MB Cache, 1 GB RAM	Linux Image in VM I5 CPU, 4C@2.2 Ghz 3 MB Cache, 4 GB RAM	Linux via Docker I7 CPU, 8C@3.4 Ghz 8 MB Cache, 32 GB RAM

Table 5.6: Communication characteristics for the setup with the fiber link to the Internet.

Download/Upload bandwidth	Ping latency 60 B	Ping latency 60 KB
100/100 Mbps	59 ms	71 ms

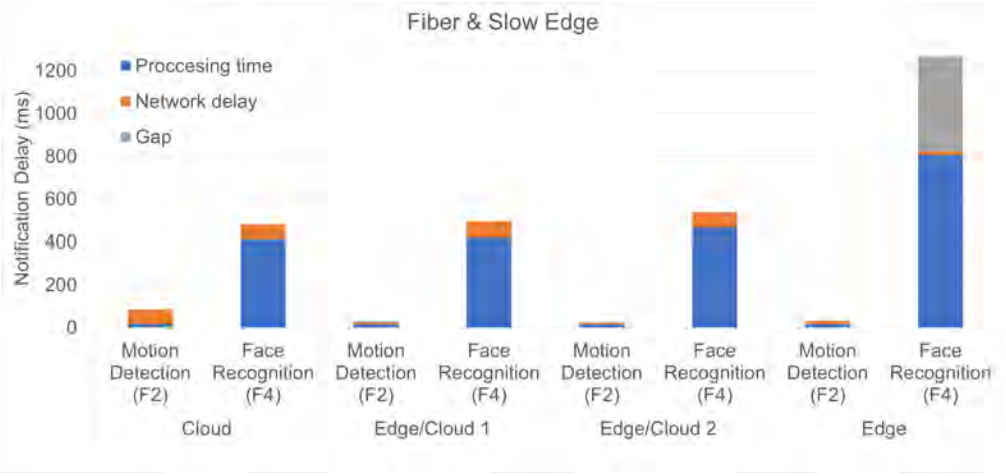


Figure 5.4: Notification delay for motion detection and face recognition for the Fiber link and the slow edge machine.

the previous experiments, we see that the notification for motion detection is lower for every configuration where F2 resides on the edge machine (Edge, Edge/Cloud 1 and Edge/Cloud 2). Even though the fiber link is fast, it still introduces a small delay, which in turn leads to a slower notification for motion detection in the Cloud configuration.

The notification delay for the face recognition shows a different pattern. A first, general observation is that for all configurations, expect Edge, this is significantly lower than in the previous setup, even though the edge machine is slower. This can be explained due to the much faster link to the Internet. Further, we see that in those configurations the sum of the recorded processing delay (blue) and estimated communication delay (orange) is practically the same with the recorded notification delay. This is because in this case the network link between the edge and the cloud

hosts is symmetrical and thus the RTT/2 estimate for the one-way latency is much more accurate than for the ADSL link.

Moreover, in contrast to the previous setup, the lowest face recognition notification delay is now achieved in the Cloud configuration, whereas the Edge/Cloud 1 and Edge/Cloud 2 configurations show a slightly higher notification delay. For the Edge/Cloud 1 configuration, the reasoning is in part the same as in the previous setup (the application-level relay overhead that is introduced by placing F2 on the edge machine is higher than the gains). On top of that, since the edge host is slower than the cloud host, F2 now takes (slightly) more processing time. The latter also explains the increase of the notification delay for the Edge/Cloud 2 configuration, where the additional 45 milliseconds are spent due to the longer time required by F3 to process an incoming frame. Additionally, given the very fast uplink, the reduction of the data that flows out of F3 to the cloud does not translate into any tangible benefit in terms of notification latency.

Note that the Edge configuration has the highest face detection notification latency, which is also much higher than in the previous setup. This is because the edge machine is just too slow to handle F4 as well (together with F2 and F3), which introduces an additional time of approximately 580 milliseconds due to heavy processing, as opposed to approximately 355 milliseconds when using the faster edge machine in the previous setup. Of course, this overload situation also affects the rest of the application components that run on this host, and in fact leads to significant frame loss (recall that we transport frames over UDP/IP).

Figure 5.5 and Figure 5.6 show the CPU load for the fast and slow edge machines in the Edge configuration. Given that Node-RED runs on a single thread, the reported load is given using as baseline the capacity of single CPU core (the fact that some values exceed 100% indicate that parts of the Node-RED framework do use the other cores, but this does not apply to the part that runs the application nodes). We observe that the slow edge machine is overwhelmed as soon as F4 receives and attempts to processes the first frame with a face (182th second in Figure 5.5). It exceeds the 100% value and remains above that for a long period of time, leading to a significant performance degradation. In contrast, Figure 5.6 shows that the fast edge machine can handle the load (even though in this setup the frame rate is higher by 25%) as it rarely exceeds the 100% value and then only for a short time.

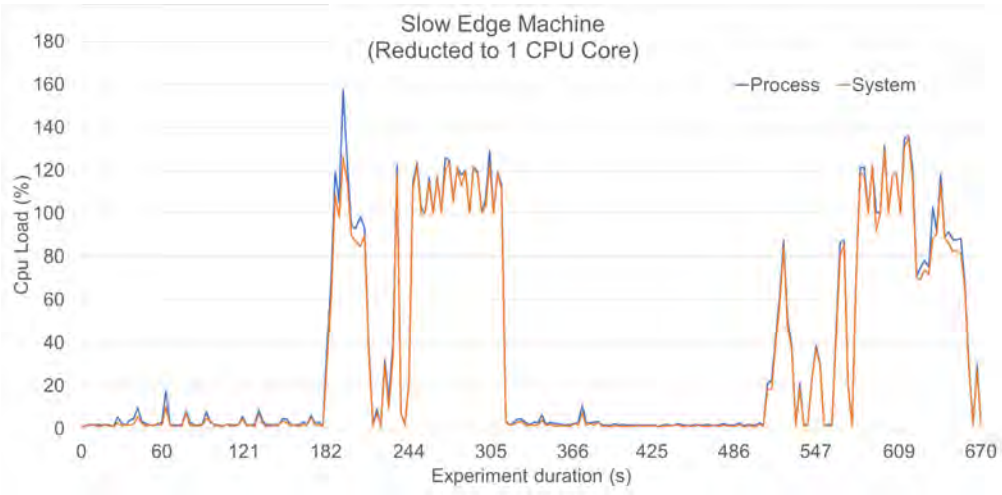


Figure 5.5: Slow edge machine CPU load.

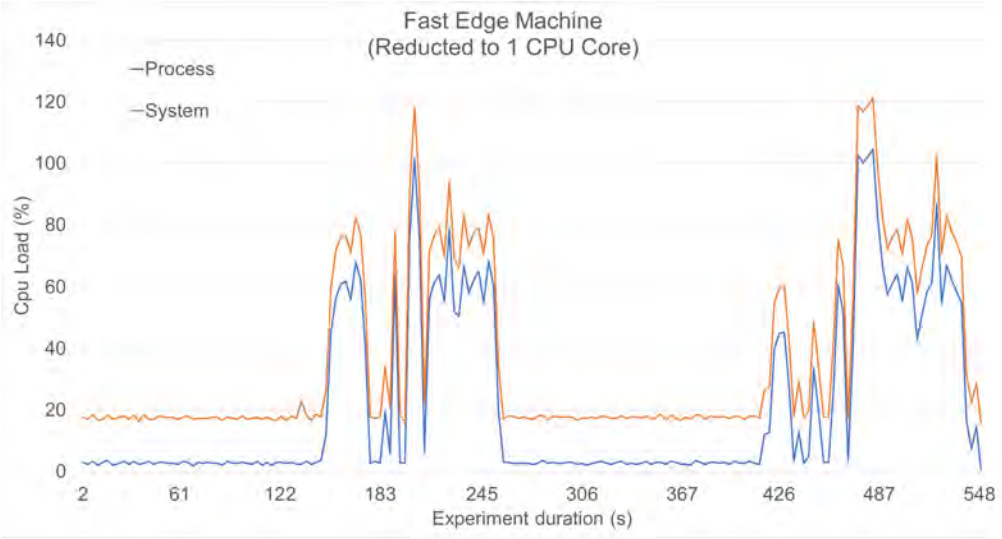


Figure 5.6: Fast edge machine CPU load.

5.5 Summary

The above results above show that, even in a relatively simple application scenario, performance only improves by placing the *right part* of the application at the edge. The machines used as hosts and the communication links between them also greatly affect the performance, and in order to estimate the trade-offs for every configuration, the developer must take in consideration many factors. Blindly placing *some* application components at the edge can greatly degrade performance, even if the communication link to the cloud is very fast. On the other hand, if the link to the Internet is slow and the edge computing infrastructure is sufficiently powerful, there is potential for significant improvement.

Chapter 6

RELATED WORK

Another effort to support distributed computing based on Node-RED is Distributed Node-RED (DNR) [16]. The ideas behind DNR are described in [17], but the system was released and made available for testing just recently, at the same time when we developed our own support. DNR follows a similar high-level approach to ours. An important difference is that in order to support more flexible linking patterns the communication between nodes running on different hosts revolves around a pub/sub scheme via MQTT [18], with the master node acting as the broker. Given the results of our experiments, we believe that this indirection is likely to be too costly in terms of latency and perhaps even sheer throughput for streaming applications like the one discussed here. DNR also changes to the core of the Node-RED framework, whereas our functionality is introduced via extensions that leave the core untouched. DNR offers some additional functionality for authentication, data encryption and the formulation of node placement constraints (which call all be easily added in our system too). We could not find any information about experimental deployments and performance measurements for DNR.

Flogo [19] is a recent framework that is similar to Node-RED, 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, the programmer has to take care of the deployment of the components on the respective machines, but it is easy to do the wiring via the GUI. The creators of Flogo claim that it is more lightweight and faster compared to other technologies like Node-RED.

There are several other frameworks that support programming with data streams and dataflow graphs, like Apache Storm [20], Apache NiFi [21] and Cask [22]. These are mainly visualization and management tools intended for data analysis and aggregation from many different data sources. Kura [23] and Fiware [24] are centralized IoT middleware frameworks, with the end-devices running an agent that communicates through a protocol gateway/adapter with the main application data processing and control logic which runs on a remote server. The programmer is offered a high-level API which hides the underlying communication with the end-devices. However. there is no support for distributing the application logic across

different host environments.

TensorFlow [25] adopts a distributed dataflow computing model, but it is primarily geared towards supporting the time-consuming training of neural networks. The developer writes the application using a custom programming notation, while a graphical representation is used to visualize (rather than specify) the program structure in a user-friendly way. However, the application can have a large number of primitive function/operator components, as opposed to the much coarser software components we target in our work. TensorFlow also enables the transparent exploitation of heterogeneous computing resources (GPUs) via abstract functions and corresponding runtime support.

Yet another way for developing applications that can be distributed in a flexible way is to compose them out of microservices [26]. Individual microservices could then be grouped into larger clusters and be deployed on remote hosts through a suitable container system like Docker [27] and deployment system like Kubernetes [28]. An issue that would have to be addressed in this case is how to link together microservices without going through some centralized glue logic or broker as in [29], [30], for instance by using a decentralized message bus [31]. In our work inter-component links are implemented without any intermediate broker, through direct UDP/IP or TCP/IP transport channels. There are also proposals for new programming languages that ease the development of microservices and their communication via a custom runtime environment [32]. We use the well known JavaScript language on a fully supported runtime.

Chapter 7

CONCLUSION

We have presented extensions made to the Node-RED framework, enabling the seamless distributed placement and execution of application flows on different machines. This, in turn, makes it possible to exploit computing resources at the edge and remote cloud systems in a flexible way. We have also presented experimental results using a realistic application, illustrating the functionality of our prototype and the performance trade-offs for different deployment scenarios as well as with different communication links and edge platforms.

In the future we wish to investigate the introduction of a logical layer via abstract components, in order to support even more flexible deployment scenarios, such as the exploitation of pre-installed components, the sharing of components among different application flows, and the transparent scaling-out of components in order to support deployment at different edges of the Internet. Another direction is to investigate more thorougly the deployment trade-offs between a larger variety of edge machines. Moreover, one could explore online component replacement and migration schemes to support dynamic/context-sensitive workloads.

BIBLIOGRAPHY

[1] *Node-RED*. [Online]. Available: http://nodered.org.

[2] O. C. OpenFog Consortium Architecture Working Group, *OpenFog Reference Architecture for Fog Computing*. 2017.

[3] *Microsoft Azure cognitive services*. [Online]. Available: https://azure.microsoft.com/en-us/services/cognitive-services.

[4] *Google Cloud Vision API*. [Online]. Available: https://cloud.google.com/vision.

[5] *IBM Bluemix visual recogntion API*. [Online]. Available: https://www.ibm.com/watson/developercloud/visual-recognition.html.

[6] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek, "A brief introduction to OpenCV", in *2012 Proceedings of the 35th International Convention MIPRO*, 2012, pp. 1725–1730.

[7] D. E. King, "Dlib-ml: A machine learning toolkit", *J. Mach. Learn. Res.*, vol. 10, pp. 1755–1758, Dec. 2009, ISSN: 1532-4435. [Online]. Available: http://dl.acm.org/citation.cfm?id=1577069.1755843.

[8] *NodeJS runtime*. [Online]. Available: https://nodejs.org.

[9] *Node-gyp tool*. [Online]. Available: https://github.com/nodejs/node-gyp.

[10] *Motion detection for NodeJS using OpenCV*. [Online]. Available: https://github.com/Daan-Grashoff/Motion-detection-node-opencv.

[11] *NodeJS OpenCV bindings*. [Online]. Available: https://github.com/peterbraden/node-opencv.

[12] Y. Wong, S. Chen, S. Mau, C. Sanderson, and B. C. Lovell, "Patch-based probabilistic image quality assessment for face selection and improved video-based face recognition", in *IEEE Biometrics Workshop, Computer Vision and Pattern Recognition (CVPR) Workshops*, IEEE, 2011, pp. 81–88.

[13] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller, "Labeled faces in the wild: A database for studying face recognition in unconstrained environments", University of Massachusetts, Amherst, Tech. Rep. 07-49, 2007.

[14] *IPTraf*. [Online]. Available: http://iptraf.seul.org.

[15] *Akamai state of the internet report*. [Online]. Available: https://www.akamai.com/us/en/about/our-thinking/state-of-the-internet-report.

[16] *Distributed Node-RED*. [Online]. Available: https://github.com/namgk/dnr-editor.

[17] N. K. Giang, M. Blackstock, R. Lea, and V. C. M. Leung, "Developing iot applications in the Fog: A distributed dataflow approach", in *2015 5th International Conference on the Internet of Things (IOT)*, 2015, pp. 155–162. DOI: 10.1109/IOT.2015.7356560.

[18] *Message Queue Telemetry Transport (MQTT)*. [Online]. Available: http://mqtt.org.

[19] *Project Flogo*. [Online]. Available: http://www.flogo.io.

[20] *Apache Storm*. [Online]. Available: http://storm.apache.org.

[21] *Apache NiFi*. [Online]. Available: https://nifi.apache.org.

[22] *Cask*. [Online]. Available: http://cask.co.

[23] *Kura*. [Online]. Available: http://www.eclipse.org/kura.

[24] *Fiware connection to the Internet of Things*. [Online]. Available: https://www.fiware.org.

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

[26] K. Bakshi, "Microservices-based software architecture and approaches", in *2017 IEEE Aerospace Conference*, 2017, pp. 1–8. DOI: 10.1109/AERO.2017.7943959.

[27] *Docker*. [Online]. Available: https://www.docker.com.

[28] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015, All. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp.

[29] J. Innerbichler, S. Gonul, V. Damjanovic-Behrendt, B. Mandler, and F. Strohmeier, "NIMBLE collaborative platform: Microservice architectural approach to federated IoT", in *2017 Global Internet of Things Summit (GIoTS)*, 2017, pp. 1–6. DOI: 10.1109/GIOTS.2017.8016216.

[30] L. Sun, Y. Li, and R. A. Memon, "An open IoT framework based on microservices architecture", *China Communications*, vol. 14, no. 2, pp. 154–162, 2017, ISSN: 1673-5447. DOI: 10.1109/CC.2017.7868163.

[31] P. Kookarinrat and Y. Temtanapat, “Design and implementation of a decentralized message bus for microservices”, in *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2016, pp. 1–6. DOI: 10.1109/JCSSE.2016.7748869.

[32] C. Xu, H. Zhu, I. Bayley, D. Lightfoot, M. Green, and P. Marshall, “CAOPLE: A programming language for microservices SaaS”, in *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2016, pp. 34–43. DOI: 10.1109/SOSE.2016.46.