FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Thesis title

Aly Saleh

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Thesis title

Titel der Abschlussarbeit

Author:	Aly Saleh
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	M.Sc. Teemu Kärkkäinen
Submission Date:	Submission date

I confirm that this master's thesis in informatik is my own work and I have documented all sources and material used.

Munich, Submission date Aly Saleh

Acknowledgments

Abstract

Contents

Acknowledgments iii

Abstract iv

List of Figures 1

List of Tables 3

1 Introduction 4
- 1.1 IoT & Distributed Sensor Networks . . . 4
  - 1.1.1 Show how Iot is being currently used, its pros and cons . . . 4
  - 1.1.2 Give an idea about the devices used to make a distributed sensor network . . . 4
- 1.2 Motivation . . . 4
  - 1.2.1 Show the need to explore Pervasive Computing . . . 4
  - 1.2.2 Illustrate why it might be better to distribute the data in some cases rather than accumulating it in a single server . . . 4
  - 1.2.3 Explain why Cloud Computing is not always the right solution in some cases . . . 4
  - 1.2.4 Explain the need to find IoT devices capabilities and limitations when used for data computation . . . 4

2 Background & Related Work 5
- 2.1 Internet of Things . . . 5
  - 2.1.1 Pervasive Computing . . . 6
  - 2.1.2 Fog Computing . . . 8
- 2.2 Messaging Protocols . . . 9
- 2.3 Networking . . . 9
  - 2.3.1 Information Centric Networking . . . 10
    - 2.3.1.1 Content Centric Networking . . . 11
    - 2.3.1.2 Networking of Information . . . 13
  - 2.3.2 Delay Tolerant Networking . . . 14
- 2.4 Used Platforms . . . 15
  - 2.4.1 SCAMPI . . . 16
  - 2.4.2 Raspberry Pi . . . 16

2.4.3 Node-RED . . . 17
2.4.4 Time-series Databases . . . 19

3 Framework in Theory **21**
3.1 Foundation . . . 21
3.2 Computational Model . . . 24
3.2.1 Distributed Nodes & Flows . . . 24
3.2.2 Software Dependencies . . . 25
3.2.3 Resources " Physical Dependencies" . . . 26
3.2.3.1 Sensors and Actuators . . . 27
3.2.3.2 Hardware Resources . . . 27
3.2.4 Pub-Sub Messaging Queues . . . 28
3.3 Data Model . . . 30
3.3.1 Data Types . . . 30
3.3.2 Moving Data . . . 31
3.3.3 IO Specification . . . 31
3.4 Summary . . . 32

4 Approach **33**
4.1 Use Cases . . . 33
4.2 Requirements . . . 33
4.3 Framework Architecture . . . 33
4.3.1 SCAMPI . . . 34
4.3.2 Node-RED . . . 35
4.3.3 Middleware . . . 36
4.3.4 Architecture Usage . . . 36
4.4 Summary . . . 38

5 Implementation **39**
5.1 Middleware . . . 39
5.2 Flows . . . 43
5.2.1 Send Computations . . . 43
5.2.2 Temperature Sensor Alert . . . 44
5.2.3 Detect Movement and Store Image Responses . . . 44
5.2.4 Show Recognized images . . . 45
5.2.5 TensorFlow Water Bottle Recognition . . . 45
5.3 Starting the framework . . . 46
5.4 Summary . . . 46

6 Evaluation **48**
6.1 Basic IoT Usage . . . 49
6.2 Recognizing Water Bottles . . . 50

6.3 Local Composability 54
6.4 Challenged Networks 55
6.5 Summary 58

7 Conclusion **59**
7.1 Summary 59
7.2 Future Work 59
7.2.1 Streaming API 59

Bibliography **60**

List of Figures

2.1 Pervasive Computing environment architecture. *Adapted from [SM03]* . 7
2.2 ICN architecture by Dirk Trossen et al., *Adapted from* [TSS10] 10
2.3 Hierarchal namespace example for CCN 12
2.4 Content centric networking architecture and flow *Adapted from [Xyl+14]* 13
2.5 *Adapted from [Dan+13]* 14
2.6 Raspberry Pi model 3 model B design. *Adapted from [EFA]* 17
2.7 Node-Red user interface to create flows for IoT applications 18

3.1 A node-red flow that stores an image in a database every time interval 21
3.2 A node containing two composable flows 23
3.3 Two separate nodes having distributed composability 23
3.4 Distributing flows approaches 25
3.5 Common message queues 28
3.6 De-centralized message queues 29
3.7 Two separate computational flows describing the IO through a database. 32
3.8 Two composable flows exchanging data via the messaging system. . . . 32

4.1 The framework architecture stack. 34
4.2 SCAMPI synchronization even without an end-to-end path. 35
4.3 Software framework architecture summary. 37

5.1 Class diagram for the api package. 40
5.2 Class diagram for the domain package. 42
5.3 Class diagram for the model package. 42
5.4 A flow that publishes computations to the middleware dn thus to SCAMPI. 43
5.5 The HTML page used to publish computations conveyed in *html template* element. 43
5.6 A flow that reads temperature and stores it, also start a red lamp if temperature is above 30 degree Celsius. 44
5.7 A flow that detects motion, take an image, publish message and store response. 45
5.8 A flow that creates an endpoint for stored database images. 45
5.9 A flow that uses tensorflow to recognize a water bottle. 46

6.1 Testbed setup for temperature sensing flow. 49

6.2 Testbed setup for recognizing water bottles. 51
6.3 Sequence diagram for recognizing water bottles. 52
6.4 Testbed setup for challenged and delay tolerant networks. 56

List of Tables

6.1 Devices used for the implementation evaluation. 48
6.2 Results, mean and standard deviation of temperature flow delays. . . . 50
6.3 Results, mean and standard deviation of the image recognition flow delays. 53
6.4 Results, mean and standard deviation of the motion detection flow delays. 53
6.5 Results, mean and standard deviation for sending and receiving data delays. 54
6.6 Results, mean and standard deviation for retrieving recognized images flow delays. 55
6.7 The delays for sending flows to the network devices including the disconnected PI. 57
6.8 Messages delays sent from the disconnected Raspberry Pi to the NUC. . 57
6.9 Messages delays sent from the NUC to disconnected Raspberry Pi having successfully recognized water bottles. 58

1 Introduction

1.1 IoT & Distributed Sensor Networks

1.1.1 Show how Iot is being currently used, its pros and cons

1.1.2 Give an idea about the devices used to make a distributed sensor network

1.2 Motivation

1.2.1 Show the need to explore Pervasive Computing

1.2.2 Illustrate why it might be better to distribute the data in some cases rather than accumulating it in a single server

1.2.3 Explain why Cloud Computing is not always the right solution in some cases

1.2.4 Explain the need to find IoT devices capabilities and limitations when used for data computation

2 Background & Related Work

This chapter describes the concepts and background information that this thesis uses and relies on. It gives a brief introduction about Internet of things and other concepts such as pervasive and fog computing in addition to explaining Delay-Tolerant and Information-Centric Networking as they play an important role in this thesis. Further, we explain the software platforms and hardware used to implement the system framework.

2.1 Internet of Things

In general terms, IoT refers to a highly dynamic and scalable distributed network of connected devices equipped with context-aware gadgets that enables them to see, hear and think[Xia+12]. Then, transform theses senses to a stream of information allowing them to digest the data and act intelligently through actuators if needed. They are also allowed to communicate and share knowledge, which make them smart, powerful and capable of acting independently. Smart devices in an IoT network are heterogeneous in terms of computation capabilities, also each device is energy optimized and able to communicate. Additionally, to qualify for being smart, devices must have a unique global identifier, name, address and can sense the environment. However, the IoT network may also contain devices that are not "smart" which act upon receiving orders triggered through certain circumstances in the network, for example, a lamp post that is set on and off according to network signals.

Since smart devices have unique identifiers and are context-aware, they can be tracked and localized, which is very helpful when performing geospatial computations [Mio+12]. The huge demand on IoT has triggered the development small-scale, high-performance, low-cost computers, in addition, sensors and actuators are getting cheaper, smaller and more powerful which in turn increased the interest even more.

The IoT concept can be viewed from different perspectives, it is very elastic and provides a large scale of opportunities in many areas. Currently the number of connected smart devices are estimated in billions, they aim to automate everything around us and are mainly targeted to increase life quality. The broad range of IoT applications include:

- Smart homes which tend to use sensors and actuators to monitor and optimize home resource consumption and control home devices in a way that increases humans satisfaction. Further, expenses generated from resource usage such as gas, power, water and telecommunications can be sent directly to related authorities without any human intervention [Cha+08].
- Smart factories also known as "Industry 4.0" the fourth industrial revolution which are optimized machines that communicate together in order to improve the manufacturing process and gather data to analyze factories logistics, pipeline and product availability It also creates intelligent products that can be located and identified at all times in the process [Gil16].
- Smart cities is one of the most adopted applications in the IoT field, it comprises smart parking, traffic congestion monitoring and control, real time noise analysis, waste management and others. All this applications need enhanced communication and data infrastructure. It aims to increasing quality of living for individuals[Zan+14].
- There are also applications in health care, environmental monitoring, security and surveillance.

IoT is very diverse, one way of applying it is to gather data from the smart devices, then process data in the cloud via *Cloud Computing*. Afterwards, results could be sent back to smart devices in order to act somehow. Nevertheless, there are approaches for pushing computations to the smart devices "Edges" such as *Edge Computing*, *Pervasive Computing* and *Fog Computing* emerged.

2.1.1 Pervasive Computing

Pervasive computing, also known as *Ubiquitous Computing*, is a concept in which software devices and agents are expected to support and act upon human needs anytime and anywhere without their interference [CFJ03]. It is usually integrated with intelligent agents and smart devices which keep learning from human actions and the decisions taken previously to be even more helpful every time. Also, pervasive software agents are context-aware in most of the cases, in which they know what changes are happening around them at a specific point in time even hold a history of what has happened in the environment. They also communicate seamlessly in order to share knowledge and help each other take better decisions. Moreover, pervasive devices can be relocated from one place to another, thus changing the network and possibly environment. Therefore, devices can not be addressed with their respective networked addresses because they might eventually change.

In 1991 Mark Wieser said in the paper describing his vision of ubiquitous computing "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it" [Wei91]. Since then, computing has evolved from using only desktop personal computers to the current phase of wireless sensor networks, small computational devices and distributed systems. Imagine the large scale of applications that could incorporate the computational power, artificial intelligence, machine learning and context-awareness to serve human beings without them even noticing that it exists. In the same paper Wieser also concluded "Most important, ubiquitous computers will help overcome the problem of information overload. There is more information available at our fingertips during a walk in the woods than in any computer system, yet people find a walk among trees relaxing and computers frustrating. Machines that fit the human environment, instead of forcing humans to enter theirs, will make using a computer as refreshing as taking a walk in the woods."

Figure 2.1 shows the architecture of a pervasive environment, in which devices are connected together through a pervasive network which should be lenient to relocating. In addition, each pervasive device has several applications that depend on environment and context. The pervasive middleware is an abstraction of the core software to the end-user applications.

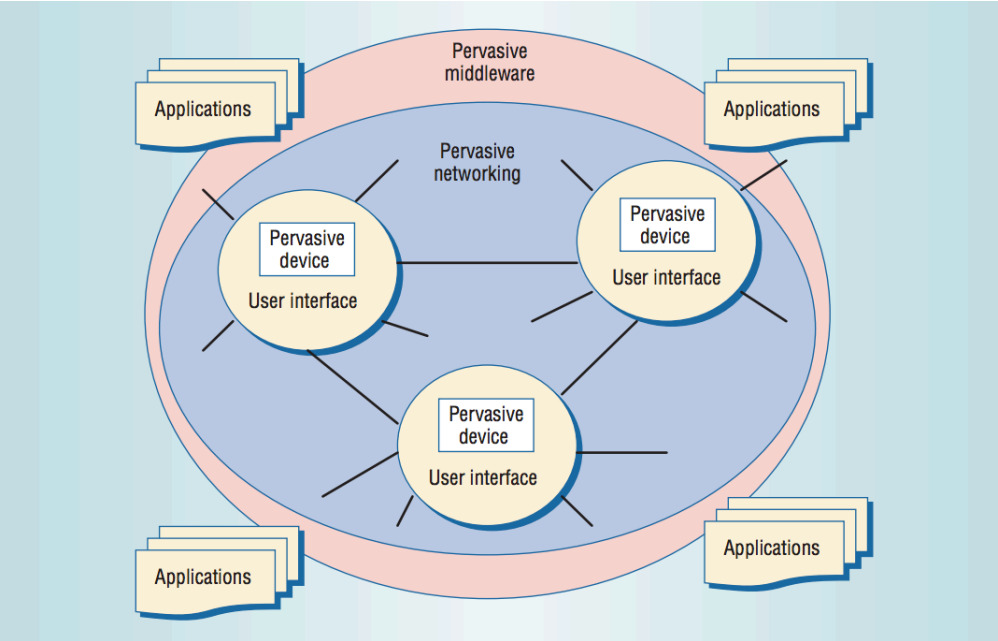

Figure 2.1: Pervasive Computing environment architecture. *Adapted from [SM03]*

The road to pervasiveness is not paved with gold, there are many challenges that faces the design and implementation of pervasive applications. Some of these challenges

are [SHB10]:

1. Devices have become more heterogeneous and the middleware must be able to execute on each of them, therefore, the use of self-contained software environments is advised. For example, using docker as an execution environment for all the heterogeneous devices.

2. Communication reliability is often questioned, in addition, environments are highly dynamic thus devices are only known at run time. Therefore, service discovery is a must, either peer-based in which all nodes take part in the discovery or mediator-bases in which some special devices are promoted to perform service discovery.

3. Sensor availability, readings uncertainty and continuous update of user requirements.

4. Communication and cooperation between devices requires interoperability. There are three different ways that allows them to cooperate:

   - Fixed standardized protocol, in which we set some technologies, protocols and data formats in order to be used across the system.
   - Dynamically negotiated, in which devices are allowed to negotiate on which protocols and data formats to use at run time.
   - Using interaction bridges that map between different approaches and protocols.

2.1.2 Fog Computing

The fog is an extension to cloud computing at the edge of the network. It provides computation, storage, networking and application services to end-users. Fog and cloud are independent, in fact, cloud can be used to manage the fog. They are also mutually beneficial, some use cases are better deployed in the fog and vice versa. Research yet to determine which applications should go where. The fog is characterized by having lower latency than the cloud, thus are more useful in time critical applications. Also, fog devices have location awareness with a better geographical distribution than the centralized cloud approach. It can distribute the computations and storage between the cloud, itself and idling devices on the network edge [CZ16]. However, it remains a challenge to deal with all the heterogeneous devices in the fog. Connecting various components with different nature ensuring quality of service is not easy. Moreover, a unified programming model should be used in all fog devices in order to help programmers make use of the fog model. Other issues are also being researched such as security and privacy of the fog network[YLL15].

2.2 Messaging Protocols

With the rise of IoT and the need for interoperability and seamless communication between smart devices, researchers and professionals have been working on developing messaging protocols that aim to aid in Machine-To-Machine (M2M) communication without any human interaction. The M2M messaging approaches can be divided into two main categories, broker and broker-less.

The broker architecture means that there is a server in the middle of all communication acting as a "broker" . Every machine in this architecture is connected to this broker and every message whether a publish or subscribe goes through it. The advantages of this model is that, machines do not need service discovery for peers, the only thing they need to know is the broker's address. Further, if one machine published a message to the broker and died, it can still reach a receiver ("even if not yet online") through the broker, it can also provide a delivery guarantee. However it has some disadvantages, there is an extensive network communication that goes through the broker, thus it becomes a bottleneck, though there is a possibility to have multiple brokers in a single network. Also, in a dynamic environment the broker or brokers addresses might not be known beforehand therefore it becomes very hard to set up broker architecture in this environment. MQTT [MQT] and AMQP [AMQ] are examples of this approach protocols.

The broker-less architecture means that machines communicate directly to each other or through multiple hops, thus it relies heavily on peer discovery. It tackles the bottleneck of broker network communication, since messages only have to go from publishing peers to interested ones. Furthermore, since machines are not always available, therefore the architecture needs to deal with unavailable machines which may not have started. In addition, it has to handle message delivery to machines with no end-to-end path or direct connectivity.

There is a third category which is endpoint centric such as RESTful services, web sockets and protocols like CoAP [Z S14] that uses the REST architectural style and is built over UDP.

2.3 Networking

In this section, we introduce the main networking grounds that are used in this thesis. Since, the software framework that is proposed in this thesis focuses on exchanging data and computation between nodes even without an end-to-end path, its imminent that we shed light on Information-Centric Networking which proposes replacing cur-

rent host-centric Internet architecture with a content-centric one. Additionally, having no end-to-end paths between some of our nodes, guides us to leverage the concept of Delay-Tolerant Networking that can store messages and carry it forward even without a connection through mobile devices.

2.3.1 Information Centric Networking

Information-Centric Networking (ICN) is an architecture that focuses on *WHAT* information is being exchanged rather than *WHO* are exchanging it. ICN was mainly described by Dirk Trossen et al. as a networking architecture that aim to replace the current Internet inter-networking layer using publish-subscribe model as an underlying service [TSS10]. Trossen introduced four main challenges that faces the architecture which are namely information-centrism of applications, supporting and exposing tussles, increasing accountability, and addressing attention scarcity.

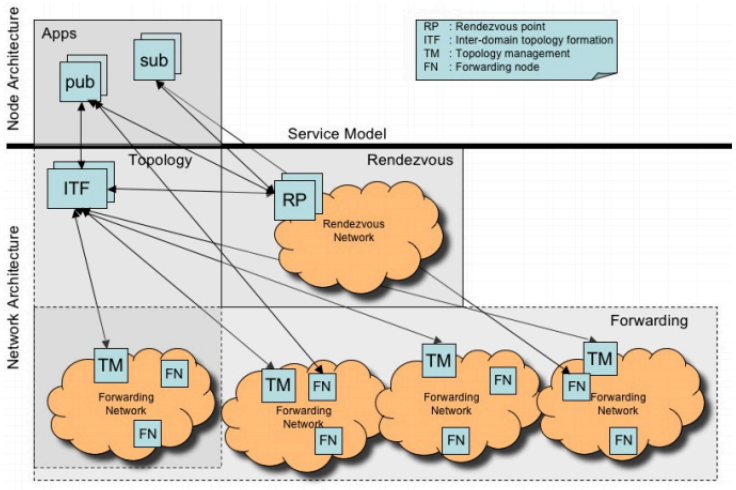

Figure 2.2: ICN architecture by Dirk Trossen et al., *Adapted from* [TSS10]

The architecture design has three main functions:

- Rendezvous which are used to match publishers and subscribers of information, each of them is identified by a globally unique identifier called Rendezvous Identifier (RI). The information items required to perform this matching exists in the Rendezvous Points (RP).
- Forwarding Topology which is created once there is a match between publications and subscriptions in cooperation with the inter-domain topology formation (ITF). It depends on the publishers and subscribers location on the level of Autonomous Systems (AS).

- Topology manager which resides in each AS and is used as a transfer link between ASes. Also, it is used to guide Forwarding Nodes (FN) to create a route between local publishers and subscribers.

ICN is content-centric in contrast to current network approaches which are host-centric, wherein communication takes place between hosts such as servers, personal computers, etc. ICN was brought to light as a result of the increasing demand on content sharing in highly scalable, distributed and efficient fashion. ICN compromises network caching, replication across entities and resilience to failure. The content types includes web-pages, videos, images, documents, streaming and others which are titled Named Data Objects (NDO). The NDO is only concerned by its name and data. As long as name identity is preserved, it does not matter where the NDO is going to be persisted, what is the storage method or which type of transport procedure is used. Therefore, copies of NDO are equivalent and can be supplied from any location or replica across an ICN network. However, since the name represents its identity, ICN requires unique naming for individual NDOs.

ICN also provides an Application Programming interface (API) that is responsible for sending and receiving NDOs. The two main roles in this API are the producer who produces content to a specific name and the consumer who asks if an NDO is available by its name. There is also the publish-subscribe approach in which a consumer registers for a subscription to a certain name and gets notified whenever new content is available. This caters for decoupling between producers and consumers.

To ensure that an NDO goes from one entity to another, a consumer request must go through two different routing phases. The first is to find a node that holds a copy of the NDO and deliver the request to that node. The second is to find a routing path back from the receiving node to requester carrying the required NDO. This can be achieved in two different ways: i) *name-resolution* in which a resolution service is queried in order to find a way to locate a source node, ii) *name-based routing* in which the request is forwarded to another entity on the network based routing algorithms, policies and cached information.

ICN caches are available on all nodes, requests to an NDO can be served from any node having a copy in the cache. An NDO can be cached on-path from sender to receiver or off-path through routing protocols or by registering it into a name-resolution service[Ahl+12].

2.3.1.1 Content Centric Networking

Content-centric networking (CCN) was first introduced by Van Jacobson et al., in order to tackle current network architecture issues such as availability, security and location independence. The main communication model relies on two main CCN

packets namely `interest` and `data` packets. It works as follows, a consumer broadcasts its interest of information to all connected nodes. If a node posses the desired data and received an interest packet, it should respond with a data packet[Jac+09].

CCN is based on the ICN concept, namespace in CCN is hierarchal, for instance, `/tum.de/connected-mobility/iot` matches the figure 2.3. Names do not have to be meaningful or readable, they can include hashes, timestamps, ids, etc. A request matches an NDO if its name is a prefix of any named object, for example, a request with the name `/tum.de/connected-mobility/iot` matches an NDO with the name `/tum.de/connected-mobility/iot/pervasive-computing`. CCN natively support on-path caching with name-based routing, however, off-path can also be supported[Xyl+14].

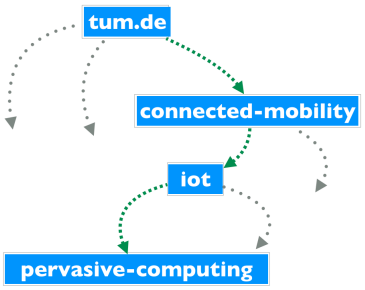

Figure 2.3: Hierarchal namespace example for CCN

Each node in the network contains a *Content Router* which includes three data structures [Xyl+14][Ahl+12].

- Pending Interest Table (PIT) which stores the subscriptions and interests of NDOs. The subscription does have to originate from the node itself, rather can be a forwarded from another node. Once an interest reaches a content source and the data is retrieved, the PIT entries serves as a trail to the original subscriber and is removed afterwards.
- Forwarding Information base (FIB) stores a mapping that indicates which node should the request be forwarded to. The FIB uses longest common prefix in order to determine the next hop. Multiple entries are allowed and can be queried in parallel.
- Content Store (CS) which is basically the cache that stores the NDOs and uses *least recently used* (LRU) eviction strategy. Caches with high hierarchy posses a larger storage to be able to store popular NDOs which might get evicted due to lower storage down in the hierarchy

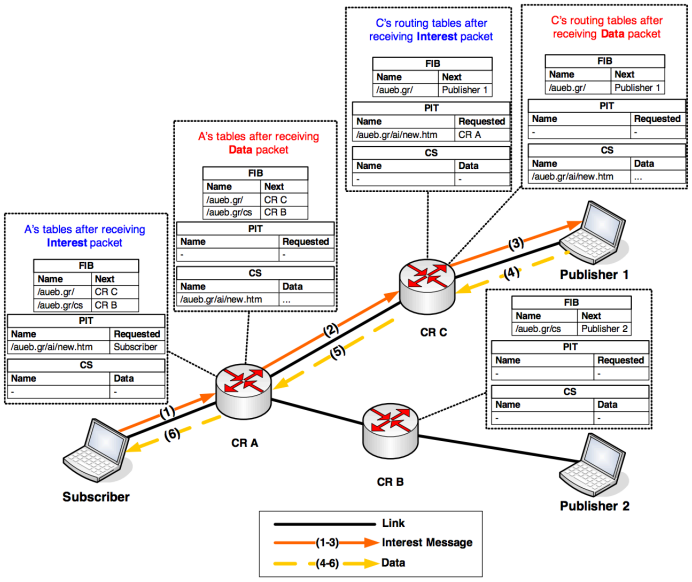

Figure 2.4: Content centric networking architecture and flow *Adapted from [Xyl+14]*

Figure 2.4 describes the flow and state of PIT, FIB and CS of network nodes when receiving the interest message and after acquiring the data. Notice that all the PIT entries have been erased after acquiring the data and CSs have been updated.

2.3.1.2 Networking of Information

Network of Information (NetInf) is an architecture based on the ICN concept. Unlike CCN, routing in NetInf is a hybrid of name-based and the name-resolution scheme, also, NDO names are not human readable. Namespace is not hierarchal rather flat, however, there is one common naming format for all NDOs across all nodes. Moreover, it supports on-path and off-path caching[Dan+13]. In figure 2.5, there are two requests namely **A** and **B**. NetInf used name-resolution service (NRS) to get the source location of **B**. Alternatively, it used name-based routing (NBR) in combination with NRS to find the request source of **A**.

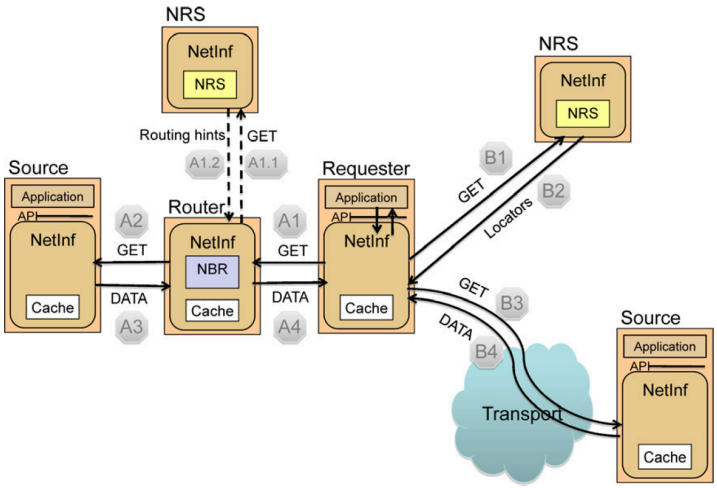

Figure 2.5: *Adapted from [Dan+13]*

2.3.2 Delay Tolerant Networking

Delay-Tolerant Networking (DTN) is an overlay architecture proposed to overcome unreliable connections between nodes through asynchronous message forwarding. In some challenged networks, there might not exist an end-to-end path between nodes either wired or not. Further, valid connections might exhibit extensive delays which might not be acceptable and thus cause the message transfer to drop. By an overlay architecture it is meant that DTN functions above the existing protocol stacks achieving interoperability and providing the store-carry-forward functionality [Fal03].

The DTN research group (DTNRG) has proposed an architecture which was published as Request For Comments (RFC) to the Internet Engineering Task Force (IETF). The architecture defines an end-to-end message-oriented overlay called "bundle layer" which is located between application layer and transport layer. The bundle layer encompasses a persistent storage, basic security model and store-and-forward to overcome disconnected and disrupted communications. Devices implementing the bundle layer are called DTN nodes and are bound to the Bundle Protocol [SB07]. Data units exchanged between these layers are named "bundles", which are messages containing a header, user application's data and control information such as how to handle, store and dispose of user data. Bundles can also be divided into fragments to increase delivery performance which are assembled again at the destination. Fragments are also bundles themselves, two or more of them can be re-assembled anywhere in the network to create a bundle. Bundles contain identifiers which distinguish the original sender and final destination. DTN nodes can store and persist bundles over longer time periods until a connection is regained. Persistence allows DTN nodes to recover

from system restarts[Fal+07].

There are different implementations of the DTN architecture. DTN2 is a reference implementation by the DTNRG to demonstrate the basic functionalities thus the performance is not optimized. There are also ION and DTNLite, however, they do not allow the use of common programming languages. IBR-DTN is yet another implementation designed for embedded systems, it also has discovery via broadcasting[Doe+08]. Also, SCAMPI application platform which will be explained in details.

2.4 Used Platforms

In this section, we describe the platforms that we used in order to create our framework. The section includes a hardware component which is the Raspberry Pi and all the other components are software oriented. This includes SCAMPI publish-subscribe platform for message passing in delay tolerant networks, node-red project for wiring IoT applications and time-series databases.

2.4.1 SCAMPI

SCAMPI is a delay tolerant platform based on the DTNRG architecture that hides networking from the application user [Kär+12]. It enables communication between peers even without and end-to-end path. The store-carry-forward router implemented by SCAMPI empower peer discovery via broadcast, mutlicast, TCP unicast and subnet scanning for known ports. In addition, it offers caching and multi-hop message transfer. Unlike DTN2 and IBR-DTN which exchanges payloads as blobs of data, SCAMPI supports structured data messages as map where arbitrary string keys map to binary buffers, strings, numbers or file pointers. SCAMPI also provides the entity SCAMPIMessage that maps to the Bundle Protocol in the DTN architecture. Moreover, the SCAMPIMessage entity can carry meta-data that describes the content.

SCAMPI is also based on the information-centric architecture in which it provides an API (over TCP) that allows broker-less publish-subscribe service of messages using NDO names that can be human readable or hashes. Additionally, it supports automatic framing of structured messages, searching for content based on message meta-data and peer discovery of nearby nodes. The (TCP) API works as an interface that can be implemented by any application written in any programming language. SCAMPI is Java based, therefore, the Java Virtual Machine (JVM) is the only requirement to have SCAMPI up and running. Furthermore, there is an android application that runs router as a persistent background process, allowing android phones to be used as a router through the native TCP API. Having SCAMPI running on the android phone allow it carry messages from one endpoint to another without even having neither wired connection or a wireless one, simply, the phone is used to carry the data from one endpoint to another.

Extending DTN and ICN, we think that SCAMPI is the best platform to use for this thesis. The are several reasons for that, firstly being a DTN, SCAMPI gives reliability of delivering messages even when there is no end-to-end path or connection is disrupted. Secondly, using peer discovery allow us to add or remove pervasive agents at-will. Thirdly, as our approach is information-centric having a delay tolerant architecture with publish-subscribe service comes really handy.

2.4.2 Raspberry Pi

Raspberry Pi is a very small, low-cost and single-board computer that was originally developed to promote computer science education in schools by the Raspberry Pi Foundation. It has a rather low processing power and random access memory compared to todays computers and mobile devices. Nevertheless, its graphical processing units equals the latest of todays mobile devices that it can stream high definition

videos and play 3D games. Raspberry Pi also has 40 General Purpose Input/Output (gpio) pins that act like switches in order to send signals to devices such as LED lamps, sensors and actuators which makes it suitable for the IoT applications. Further, there are lightweight operating systems deigned especially to cope with the edge devices and Raspberry Pi such as *Ubuntu MATE*, *Windows 10 IoT Core* and *Raspbian* which is officially supported by the foundation.

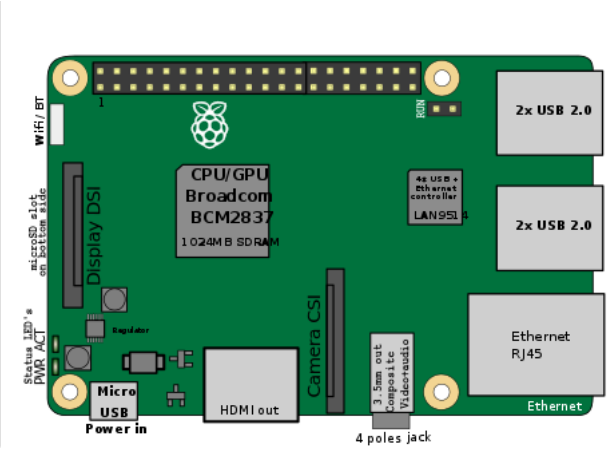

Figure 2.6: Raspberry Pi model 3 model B design. *Adapted from [EFA]*

In this thesis we use the Raspberry PI model 3 as shown in figure 2.6 with the following specifications:

- 1.2 GHz processor designed by ARM and 1 GB of random access memory.
- Ethernet port, wireless LAN, Bluetooth 4.1 and Bluetooth low energy to enable wide range of connectivity.
- Micro-SD card slot for storage.
- Audio jack, HDMI port and 4 USB ports.
- Camera, display interface and 40 gpio pins.

There are other models of the Raspberry Pi such as Pi 2 Model B, the Pi Zero, and the Pi 1 Model B+ and A+, they differ in size and specification. However, they all share the same concept of being low-cost and single-board computers [Ras].

2.4.3 Node-RED

Node-RED [Nod] is a powerful, open-source and flow-based programming project originally developed by IBM Emerging Technology Services and now a part of the JS foundation designed for IoT. Flow-based programming describes an application as a

network of black-box nodes that exchange data together. Node-RED is widely used as a visual tool for wiring IoT applications that can be deployed locally, on the cloud or on edge devices such as (Raspberry Pi, Android, Arduino). nodes are the building blocks of node-RED, each element has its own defined purpose and can be given a certain input which in turn can give an output. Further, these nodes can be re-used and have different visuals which makes them easy to use and more handy for a wider range of users. The function of these nodes varies from digital, social or even physical events that include sensors. The network of nodes is called a *flow*. Flows can be serialized into JSON objects and thus simplifying importing, exporting and sharing process. Since node-RED is an open-source project which is very well adopted. Therefore, the community is allowed to create new flows and extend current flows for their own use cases and make them available to the public, they are also allowed to report potential problems and bugs to the contributers. This results in a very fast development, fixes, new features and releases.

Node-RED user interface is available through the browser via `localhost:1880` where `1880` is the default port, however, there is an API to import and export flows through node-RED engine without using the user interface. Figure 2.7 shows the browser window with node-RED UI. On the left side exists the nodes which represent the building blocks of the flow, they can be dragged and dropped inside the plane canvas. Afterwards, nodes can be wired together and deployed. In the figure, a simple flow including a time-stamp triggered every five seconds and debug node are wired together and deployed. On the right side of the figure, printed time-stamps can be seen in the debug pane as a result of wiring the debug element into the flow and connecting it to a configured time-stamp element. The Info tab on the right side, explains each element once it was selected and shows how to use it.

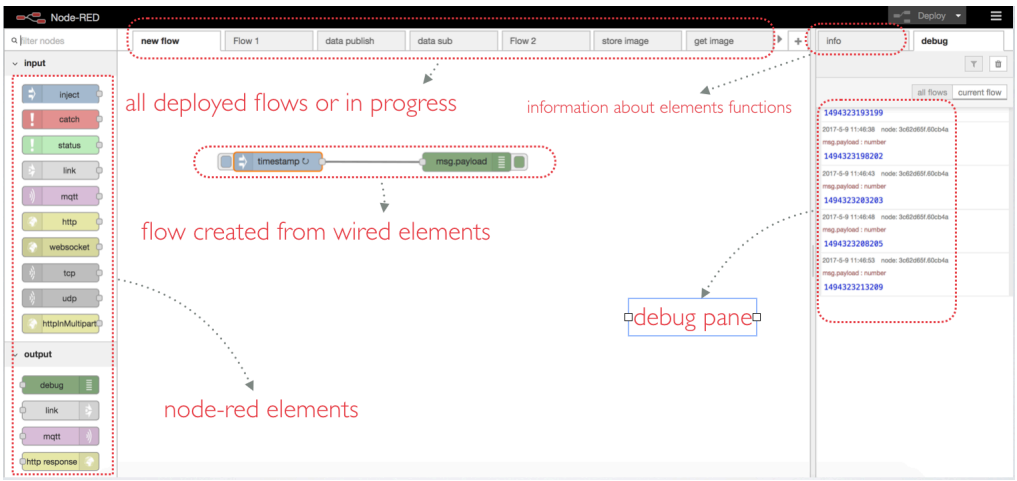

Figure 2.7: Node-Red user interface to create flows for IoT applications

Node-RED also allows global context, meaning, we can set custom variables that live across all nodes. This is really helpful when an application should keep a certain shared state. The main programming language of node-RED is JavaScript, it is also possible to write a custom node which executes code or script of any language through the execute node and the result can be wired inside the flow. Moreover, node-RED has a built-in node to control gpio pins of a Raspberry Pi , that can send low or high signals whenever desired to a certain pin. This is powerful because it hides the abstraction behind controlling the Raspberry Pi pins.

Given its features and specifications, our proposed framework uses node-RED as an environment for deploying computations, in other words, each smart device in our architecture must have a node-RED instance. Hence, we could use flows and wired nodes to express computations and serialize it via the export feature. Thereafter, we could send serialized computations to other nodes having also node-RED instances along with their dependencies, meta-data and import/deploy the computation on any number of nodes.

2.4.4 Time-series Databases

Time-series databases are optimized for storing and fetching data which is collected in a timely based manner, in other words, data which is timestamped. They are intended to handle huge volume of data especially for monitoring, real-time analytics and continuous measurement of IoT sensor data. Manipulating time-series data like aggregating, creating subsets and re-sampling can be a tricky task [Lei+15]. Time-series databases can be based on either SQL or NoSQL, moreover, some NoSQL based time-series databases offer SQL-like query language in order to simply fetching the data. Multiple implementations of time-series databases exist such as *InfluxDB* [Inf], *CrateDB* [Cra], *OpenTSDB* [Ope]. Some general purpose NoSQL databases are used as time-series like *MongoDB* [Mon] and *Elasticsearch* [Ela].

Since time-series databases are mostly used to ask questions related to time, therefore precision is of key importance, some time-series databases can support time-stamps up to nanosecond precision. Further, query languages are developed and optimized to facilitate grouping by time intervals and selecting ranges. Efficiency and performance are a must when querying months of data over millisecond or nanosecond interval as it requires huge amount of processing. Also, time-series databases are required to have high write performance in order to cope with the continuous collections of measurements and sensor data.

In this thesis we use a time-series database to demonstrate that our framework behaves as expected with the traditional use of IoT applications through gathering sen-

sors data and storing them in a time-series database in order to get insights over this data.

3 Framework in Theory

In this chapter we explain the framework model in theory, the key concepts behind it, challenges facing the design and their possible solutions.

3.1 Foundation

The fundamental core element of this framework is the computational unit derived from the use case. One possible abstraction of the computational unit is the *flow*, which is a purposeful unit of computation that contains groups of sequential instructions "*elements*" whose input/output are connected together. These elements could have a significant meaning on their own such as snapping a photo or making simple data transformation as shown in figure 3.1. Also, A flow can not only be a standalone self-contained computation, but can interact with other flows in which they collaborate for data gathering, sharing and processing as well.

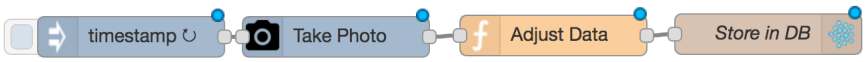

Figure 3.1: A node-red flow that stores an image in a database every time interval

After having defined flows, the next step is ti execute them. To begin with, we must address the challenge that flows are distributed in the sense that each flow could reside on a different node. In addition, as previously mentioned, flows may interact, therefore, they need a way to communicate. Moreover, since nodes might be disconnected, the communication mechanism must not require end-to-end paths. Furthermore, it should handle sending the computations themselves from one node to another, at the end we are designing a pervasive framework that should manage sending computations everywhere.

Another challenge that faces the execution of flows, are the dependencies and resources needed to carry out the execution. They might vary from one use case to another, thus need to be orchestrated across nodes though the messaging system by ensuring the delivery of relevant dependencies along with their respective computa-

tions.

Now assuming that we can send flows to the nodes, make them communicate and satisfy their dependencies and provide their resources, one aspect remains, which is triggering the execution of flows. There are multiple ways to start an execution, one simple example is a time interval trigger. Other ways include starting the execution when new data has been received or other events have been triggered e.g. via physical sensors.

A flow should be modular having a specific functionality with defined interfaces that reduce the complexity, allowing re-use and re-assembly. Moreover, since flows should be composable, they need to interact and exchange data. Think of composability as LEGO parts that need to be assembled in their correct positions in order to create a figure, however in contrast to individual LEGO parts which do not have a meaning on their own, individual flow elements could serve a specific purpose besides their global one.

To establish flow composability in our context, we need to be able to match the output data of one flow to the input data of another, no matter whether the flows are on the same node or distributed; connected or disconnected. For instance in general terms, if we have a flow f_1 that takes A as input and gives B as an output

$$f_1 : A \to B$$

and another flow f_2 that takes B as an input and gives a new output C

$$f_2 : B \to C$$

we should be able to compose a new flow by taking f_1's output and giving f_2's input, resulting in flow f_3 which is a composite of both:

$$f_3 : A \to C = f_2 \circ f_1.$$

Composability ensures that regardless the use case, logic or implementation of a flow, it still can be composable if it matched the input/output of another flow. Composability should be valid in both local and distributed environments. Thus, in the case of local flow composability, there should be a way to connect the output of a flow to the input of another locally as shown in 3.2. In the case of distributed composability, the messaging system should connect the flows and serve as a broker to deliver the data as shown in 3.3.

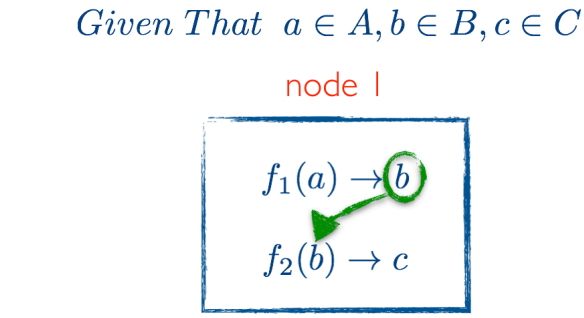

Figure 3.2: A node containing two composable flows

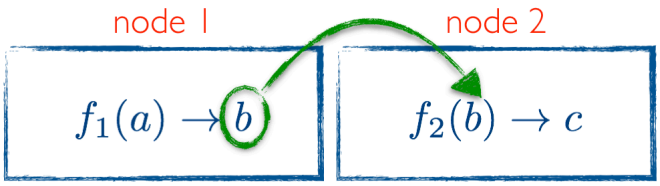

Figure 3.3: Two separate nodes having distributed composability

To sum up, flows are distributed and modular units of computation derived from a use case which require dependencies and resources. They communicate with each other and can be composed both locally and in a distributed manner. By achieving modularity and composability, flows can be assembled in different combinations, thus allowing re-use and extensibility.

3.2 Computational Model

Below we present the computational model as an abstraction for the framework design. It explains the components, challenges and the possible solutions that could be implemented to overcome these challenges.

3.2.1 Distributed Nodes & Flows

In order to start with the framework explanation we must understand the idea behind pervasiveness. Pervasive computing relies on the idea of pushing flows to the edge "nodes" and thus it is fundamentally distributed. A system is distributed if its components are on networked computers which communicate only by sending and receiving of messages [CDK01] which is the case here. Now in our model, each node should be capable of executing flows and producing results as long as it has the required dependencies and resources. Moreover, to ensure that flows are composable, nodes should be able to communicate seamlessly even though nodes hosting this flows might be disconnected. Moreover, the model should be delay tolerant and subject to connection failures.

Turning to flows, in essence every challenge related to making the nodes distributed also applies to flows because nodes host flows. However, there are more to flows, distributing them across nodes could have different approaches depending on the use case. Let us explain this with figure 3.4, to start with, lets take the set of all nodes in the system and call it $S(t)$ which is a function of time since nodes can be removed or added to the system dynamically at any instant of time. Moreover, nodes in Then comes the candidate set $S_C(t)$, which are nodes that satisfy the required dependencies and resources of a certain computation. Note that, nodes inside any of the these sets might not have end-to-end path. Given $S_C(t)$, the flows could be either sent to a random set of nodes or to a specific set of nodes. This provides flexibility in applying the use case without wasting resources. In addition, it magnifies the effect of locational context, meaning that if we want to compute a certain computation or measurement in a specific location and we know the general identifiers of the nodes residing in this location, we can send a flow to this exact set of nodes with our desired computation. Continuing to explain flows approaches with figure 3.4, a flow could be distributed across nodes of the candidate set $S_C(t)$ in multiple ways explained as follows: (i) flows are sent to all nodes in $S_C(t)$, (ii) flows are sent to a set of n nodes where $n > 1$, whether they are selected as case *C #2* in the figure or picked at random, (iii) choosing only one node to execute a specific flow *C #1*. As a result, the communication model is one of the most crucial parts to guarantee a distributed system, it should have the flexibility to provide these approaches and overcome the hurdle of disconnected nodes.

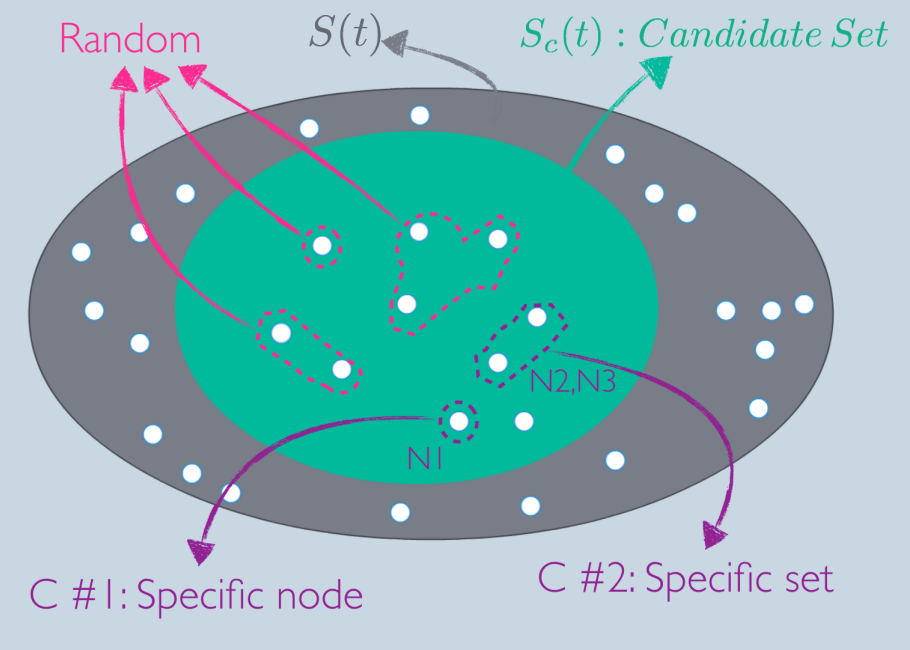

Figure 3.4: Distributing flows approaches

Another main challenge is to actually find the connected nodes. Distributed and pervasive environments are dynamic, their components are not known to be live or dead at compile-time. Thus the framework should be able to run service discovery at runtime in order to find the connected nodes or it should be able to broadcast its message to all the other nodes and receive them as well. Otherwise, the approach would not qualify to be a distributed system.

There are possible ways to achieve that, one could set up a static DNS server to resolve the domain name of a node, however, this requires having static IP's for all the nodes. Another solution is to use DDNS or MDNS to update the domain name whenever an ip changes dynamically.

3.2.2 Software Dependencies

Dependencies are one of the main requirements of computation execution, dropping one or more dependencies would stop the execution from proceeding. Thus, we need to deal with them and make sure that all dependencies are satisfied. There are two types of dependencies; the static software frameworks that the whole design relies on and must exist on each node, and the dynamic dependencies that are specific to each computation.

First, are the static dependencies are mainly the common libraries and software that most of the computations would require, they represent the base of framework.

That is why these dependencies are installed to each node in our design, examples of these dependencies include the operating system, data store and any other standard or custom libraries that are used by most computations. In addition to, the messaging system which implements the communication model allowing interaction between nodes.

Second, are the dynamic dependencies which are specific to each computation such as additional scripts, configuration files or libraries. In this case, they cannot be installed at node initialization since we can not know what are the custom dependencies any computation would need beforehand. Therefore, the computational model design should allow a way to configure additional dependencies. Moreover, the communication mechanism should support this configuration and grant a way to carry the configured dependencies forward to other nodes.

Static dependencies create ambiguity. Suppose that we want to upgrade the versions of current libraries installed on the nodes. This introduces a versioning problem, imagine that there is a computation on the node that uses an older version of the same library while the maintainer is upgrading to a newer version of the same library that is not backward compatible.

Nevertheless, there are multiple possible solutions to remove the ambiguity and make version upgrading more concrete; one solution would be to give the dependencies different names according to their versions before shipping them, hence, any different version would not replace the existing ones. Another solution would be to design a system that links each running computation on the node to its dependencies and once a collision appears, the new computation renames its dependency and uses the renamed one.

3.2.3 Resources " Physical Dependencies"

Resources are physical dependencies which are also necessary for computations to run. However, they might differ or not exist at all on each node. If one of the needed resources is missing then the computation could be either dismissed or queued depending on the type of resource. Moreover, the maintainers cannot make any assumptions about the resources, meaning, an assumption stating that each node has a camera is not necessarily true. Since the resources are not standardized across all nodes, each computation must provide meta data expressing the resources it is going to require, also, the node must realize its available resources. Then a node can check against its capabilities and decide whether it could carry out the execution or not. Further, the meta data can be exposed to the routing layer, thus helping the router take an informed decision whether a specific route contains nodes with the required resources or not. This could also provide an insight for developing better routing algorithms.

Considering that each computation model has meta data describing its resource consumption, then it is possible to know if it is going to be deployed on a specific node or not. Additionally, if it is not going to be deployed then it should be decided whether the computation is going to be queued or dismissed according to the possibility of acquiring the resource. The idea of queuing computations however develops a scheduling problem. Since we have a queue of computations inside each node, we will have a race condition on which computation should be deployed first according to available resources. Furthermore, since some computations might be dismissed, a rather bigger scheduling problem will come up when we try to fit the all computations across nodes in the whole system framework.

There are two types of resources; sensors and actuators which are used throughout a computation, and the hardware resources which influences the performance requirements of a specific computation.

3.2.3.1 Sensors and Actuators

Sensors and actuators are resources attached to a node such as cameras, temperature and gas sensors. Executing a computation missing this type of resource on a node should have a lower possibility of being queued, since its highly unlikely that this resource would be attached soon. However sensors and actuators are dynamic, they can be added or removed on demand, therefore, having them in a specification file as a static dependency which is only set at initialization time will be troublesome. Of course, we can always edit the specification file once we change the state of any of these resources, but this solution is not very efficient nor scalable, as it increases the manual work. It would be much easier if the node could run resource discovery to find its attached resources each time it receives a computation.

Moving on to consider computations acquiring the same resource at the same time, for instance, two computations that want to take a photo at the same time. This is problematic because whichever computation acquires a lock on the camera first will succeed while the other will fail. Therefore as a resolution, we could use resource decoupling; instead of having the computation ask a specific resource directly for information, the data will be pushed into a database. Afterwards, the different computations could query the data from a database.

3.2.3.2 Hardware Resources

The second type of resources is related to the node performance, its power and memory capabilities, it is heavily biased by the node processor and its random access

memory type and size. Computations vary in terms of resource consumption and hence a heavy computation should not be deployed to a node which is already loaded.

Queuing this type of dependencies should have a higher probability because it is highly possible that one of the computations will finish soon, thus decreasing the cpu usage and freeing more memory.

3.2.4 Pub-Sub Messaging Queues

The communication model is an essential part of this framework, it solves some of the biggest challenges, which are in a nutshell, service discovery, carrying dependencies, sending and receiving of data or computations whether nodes have end-to-end paths or not. Moreover, given our distributed approach and the need for service discovery, the communications model cannot be end-point centric since in the general case we are unable to target the actual nodes with their host names as endpoints. The reason for that is, we do not know their respective addresses or either they are connected or not. Rather our communication model is data-centric or information-centric meaning it assumes that there are some parties interested in sending data and others willing to receive the same data given a certain context and regardless their network location.

A possible solution to the framework demands and challenges is to use publish-subscribe message queues. The pub-sub pattern is data centric messaging architecture in which senders also known as *publishers* do not send messages directly to receivers, rather send to specific topics. Then, *subscribers* receive messages which are relevant to them by subscribing to these topics.

Commonly the pub-sub message queues contains a centralized bus *broker* which handles publishing, subscribing, notifying and persistence. When the bus receives a message published on a specific topic, the data is stored on a local storage for persistence and failure recovery. Parties can subscribe to the data topics on the bus hence notified when a new message is published. Figure 3.5 shows publishers publishing data to topic *t1* on the bus, subscribers subscribing to *t1* and notified when a new message is on the bus.

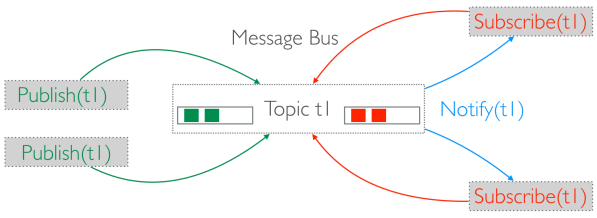

Figure 3.5: Common message queues

However having a pub-sub message queue with a centralized bus can not be used in our case. Since we do not know machines addresses or host names thus knowing where is the centralized message queue located is not possible. Therefore, each party or node would have its own message bus and local storage which are then synchronized together whenever there is a connection.

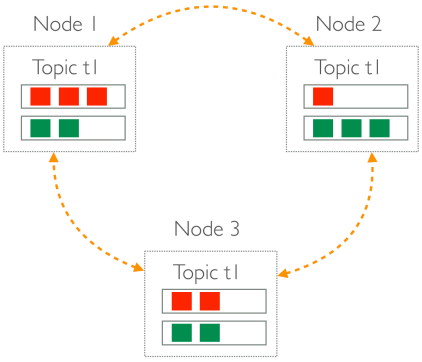

Figure 3.6: De-centralized message queues

Now addressing the mentioned challenges:

- First off, nodes service discovery, messaging queues are able to discover and synchronize messages across all the nodes connected to the messaging system through any kind of network. They are also dynamic in the sense that they are sensitive to the addition or removal of new nodes to or from the system.
- Having solved the problem of service discovery, nodes can now send and receive messages. But since we also care to send computations as well, we must differentiate between data and computations messages. Thus, a possible solution is to reserve a unique topic only for exchanging computations between nodes.
- For sending data or computation messages to a random set of n nodes, a routing algorithm can be used to ensure that no more than n nodes will receive the message. Further, if we exposed the meta data of the computation, the routing algorithm can make sure that receiving nodes will have a higher probability of being able to execute the flow.
- Sending data or computation messages to specific node or set of nodes can be done by reserving a unique topic for each. Therefore, to target any node, the message should be published once to each unique topic. For example, if we use the general identifier as a unique topic and want to publish a message to node 1, then, we can create unique topic *"N #1 "* and send the messages over this topic.

- Pub-sub messaging queues allow carrying arbitrary kinds of data inside the message body. Therefore, computation dependencies can be added to the message body of their respective computations. Thus solving the obstacle of carrying dependencies mentioned earlier and creating self-contained computations that are ready to execute anywhere.
- Last but not least, in order to send data or computation messages to disconnected nodes, either because there exists no end-to-end path or the nodes are experiencing network connectivity issues. The pub-sub messaging system should be delay tolerant and implements the store-carry-forward technique, this will allow the messaging system to store messages until connectivity is back or keep the message hopping from one device to another till it finds its end destination.

Having found a solution for our communication model challenges assists us to focus on the framework design and implementation. Knowing that the underlying network model will not fail us to connect the nodes even without and end-to-end path. In addition, we can directly send and receive message to the nodes on data topics without the framework being aware of their host names or network addresses.

3.3 Data Model

In this section we describe the data model which includes the structure of the data sent between nodes through the messaging system, how the data travels from one node to another and the input/output specification used to combine and compose different flows.

3.3.1 Data Types

A computational flow can generate different types of data depending on the use case. This data could be intermediate processing data or a computational result. Also, we should not attempt restrict the data types in order to make sure the framework is as dynamic as possible. The challenge is to be able to represent these data types in a composable way. Therefore, if a developer wants to create a composable flow he/she should define an IO specification explained latter. However, the good thing is that, the developer dictates how the input or output data are structured while developing the computational flow. Hence, he/she is in complete control and can structure the data in any way as long as it can be used afterwards. Different data types include: i) structured data that could be stored either in a relational or non-relational data base, ii) unstructured data, iii) data streams.

3.3.2 Moving Data

Moving data is the idea to send/receive raw or processed data to any flow. We should be to able use data from different remote or local sources in any computation. Some use cases for moving data are:

- Composing flows is one of the main use cases, we would like to get input data for a computation from the output of another.
- We can send data to be processed by a computational flow on any node and then obtain the outcome. For instance, we can send an image to a node containing computational flow with image recognition algorithm, then the image gets processed on that node and we get back the computation results.
- A Node can act as a monitoring node in which it is interested in all outputs of a certain computation running on several nodes.

As mentioned in 3.2.4, our approach and communication model is data-centric. Therefore, flows could subscribe/publish to a certain data topic in the distributed pub-sub messaging queue. Thus, data should reach any node which contains a flow subscribed to a certain topic. This allows us to move data freely and at will, we just need to express how a flow receives or publishes data.

3.3.3 IO Specification

Turning now to consider the input and output specification, the IO spec. explains how the output of a flow in one node can be linked to the input of another flow either on the same node or on a remote one. There are multiple ways to specify how the IO data communicates which are explained below:

- The first way allows data communication between computations of the same node through a database. One computation writes interesting data into a specific table with locally unique name in a database. Then, any other local computation which wants to use this data is allowed to fetch it from this table. Unique names are suggested to decrease the possibility of database inconsistencies if someone is using a table with the same name.

  Flows can be used to describe the database configuration from inside the computation flow, thus the maintainer should make sure when developing locally composable flows that the database configuration and table names match. An example in figure 3.7 shows that the first flow takes an image and then store it in the database with a unique table name. While the second flow, pulls the data from the database upon receiving a request on a specific URL.

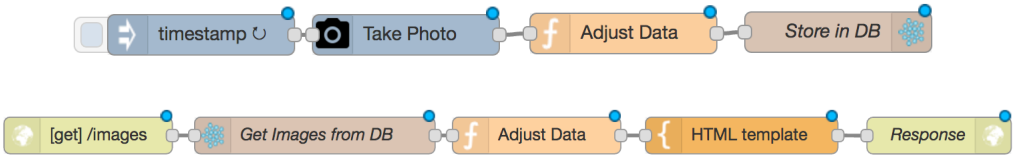

Figure 3.7: Two separate computational flows describing the IO through a database.

- Another way is to use publish-subscribe messaging pattern to communicate through different nodes. The node which generates the data publishes its resulting data to a generally unique topic, therefore any node interested in the data could simply subscribe to that topic and process the data accordingly.

  Figure 3.8 shows two flows as an example of this method, the first flow generates data and publishes it to the messaging system. Then, on any node, the data could be received via subscribing to the same topic.

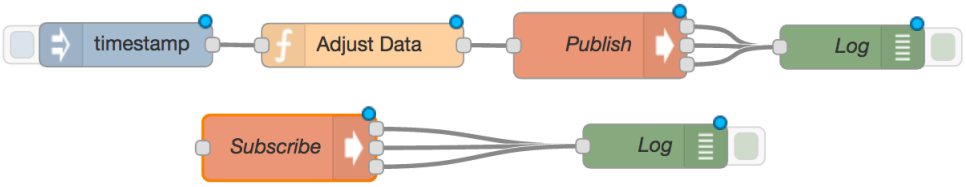

Figure 3.8: Two composable flows exchanging data via the messaging system.

- Streaming data is also possible, one node can have a computation serving as a streaming server while other nodes have computations which act as clients. However, the global reference of the streaming server must be known to clients or to make it more dynamic, a service discovery mechanism could be implemented in order to help clients find streaming servers.

3.4 Summary

In this chapter we explained the key concepts and foundations behind this framework, what is a flow and how it can be composed. We also described the computational model as an abstraction the framework design, and illustrated the challenges that faces this model which included dealing with dependencies, resources and having a distributed system model in addition to their possible solutions . We also elaborated the communication model and how it solves some of the challenges in our design. Moreover, we have shown how data inside the computation can be structured, and how input and output data of different nodes could be connected together.

4 Approach

In this chapter we explain the use cases and requirements to design a context-aware pervasive software framework. Then, we illustrate the proposed architecture and design following from the concepts introduced in the background chapter and taking into account the challenges and possible solutions shown in the foundation chapter. To recap, our aim is to design a context-aware pervasive software framework to manage and distribute computations while considering resources, dependencies and networking even with no end-to-end path.

4.1 Use Cases

4.2 Requirements

4.3 Framework Architecture

In this section we explain the software framework architecture designed for pervasive environments and challenged networks to distribute and manage computations with their respective resources and dependencies. The main idea behind this design is to harness the features of node-RED to create, deploy and share computations of any kind. In addition to having SCAMPI as an information-centric, publish-subscribe and delay tolerant messaging system that gives the framework the skill to deliver messages even without an end-to-end path. However, node-RED and SCAMPI are two different environments that cannot manage computations, resources and dependencies on their own. They need a middleware to orchestrate the communication between them.

In general the architecture stack on each node should look like figure 4.1. With SCAMPI at the stack bottom relying only on the JVM. Then on top of SCAMPI is its Java API to communicate with the TCP API of SCAMPI. Afterwards, comes the middleware which acts mediator in between node-RED and SCAMPI. Finally, at the very top exits node-RED to run computations and interact with the user if needed. However, if SCAMPI is used on an android device, there might be no need to run neither the middelware nor node-RED since the device will be used to transport data

from one node to another having no end-to-end path. Below we explain how each component of the system work.

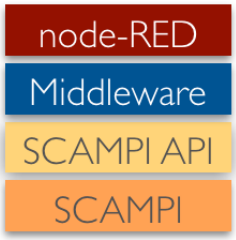

Figure 4.1: The framework architecture stack.

4.3.1 SCAMPI

As mentioned before SCAMPI is an information-centric, publish-subscribe and delay-tolerant messaging system. In this framework we use SCAMPI to send or receive messages that include computations and data. SCAMPI is also broker-less meaning we do not have to set up a server as broker which is one of the main reasons we chose SCAMPI, in order to be dynamic as possible. The other main reason is to reach nodes which do not have direct connectivity to the publishing node or do not have end-to-end path.

Being a delay-tolerant networking architecture, SCAMPI can use its store-carry-forward routing to deliver messages to challenged environments. In figure 4.2, we show how SCAMPI uses mobile devices to connect nodes that do not have a route or direct connection and want to exchange messages. In the figure, there are three Raspberry Pi devices running our proposed stack. The first two have network connection therefore it is rather easy to exchange messages between themselves. However, the third one is isolated, nevertheless it can be connected to a Wi-Fi network or run as an access point. In this case, an android device passing between network *N1* and *N2* can carry the message bundles from one network and forward it to the other by connecting to both networks alternatingly. Thus, reaching out to challenged environments that can not be reached using wired or wireless connections.

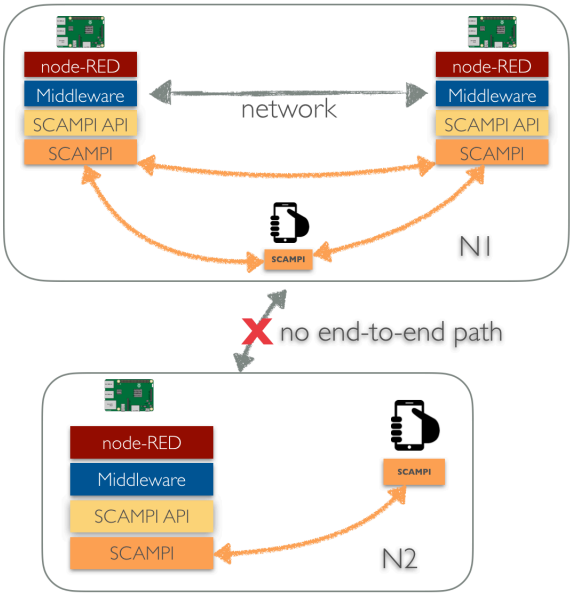

Figure 4.2: SCAMPI synchronization even without an end-to-end path.

Being information-centric, data-centric with publish-subscribe pattern and having peer discovery helps us in achieving our dynamic framework without knowing any host names. It also supports adding and removing nodes at will without any additional configuration. We can also use the general identifiers of the nodes as topics in order to target each of them independently. As stated, SCAMPI does not have any dependencies other than the JVM so we just run it on each node and we are good to go.

The SCAMPI Java API acts as a client to the server allowing us to publish or subscribe for any topic from the Java environment. Therefore, we are able to extend this API and create a running Java application that uses the TCP API for SCAMPI underneath. The API also allows the client to override the functions for SCAMPI status changes for instance if it is disconnected, stopped or most importantly when a new message is received. Furthermore, there is a model called *SCAMPIMessage* used to create messages. The model assists in assigning attributes to the message whether strings, integers or even binaries. Also, you can assign meta-data and lifetime to the message.

4.3.2 Node-RED

Node-RED is a tool used for wiring IoT applications, its flows describe the intended computations. It has exporting and importing endpoints for flows via REST which makes it easy to deploy without human interaction. Flows can be also configured to access certain tables or collections on a running database instance and this configura-

tion can be serialized along with the flow. In this framework whenever a flow wants to send a message to another flow on the same node-RED instance or on other instances, it uses the REST API that the middleware provides to publish and subscribe to data topics. This allows node-RED to send or receive data and allow composability both locally and globally. Further, node-RED is rich with predefined elements that can be used to run flows on time intervals, connect to emails, twitter accounts or even access a gpio pin on Raspberry Pi. Node-RED usage is intuitive since it is based on flow-programming, it does not need a developer to create a flow.

4.3.3 Middleware

The middleware is this framework's mediator and the main contribution in this work, it is deployed along with node-RED and SCAMPI on each instance in this architecture. It runs a jar file containing a web application server. It has several duties in orchestrating SCAMPI messages to node-RED instances.

- It reads the machine specifications to initialize the machine's resources, sensors and actuators .
- It includes the SCAMPI Java API and has a REST API over it which allows any other script from any other language ("including node-RED") to use the publish subscribe feature of SCAMPI.
- It analyzes flows by checking the meta-data attached to the message thus if the middleware finds out that a flow does not have the necessary hardware requirements, sensors or actuators, it will not deploy the flow.
- It is responsible for attaching dependencies of node-RED flows when one is published, also for putting them into the correct directory when receiving them.
- It provides a message caching mechanism in order to make sure messages are not handled twice.
- it provides a mapping between the topics and node-RED flows meaning if one or more flows are interested in the same topic, all of them should get the data.

4.3.4 Architecture Usage

The following figure 4.3 explains how the framework works and shows how data flows between the stack components. In this section, we explain the basic usage of the architecture.

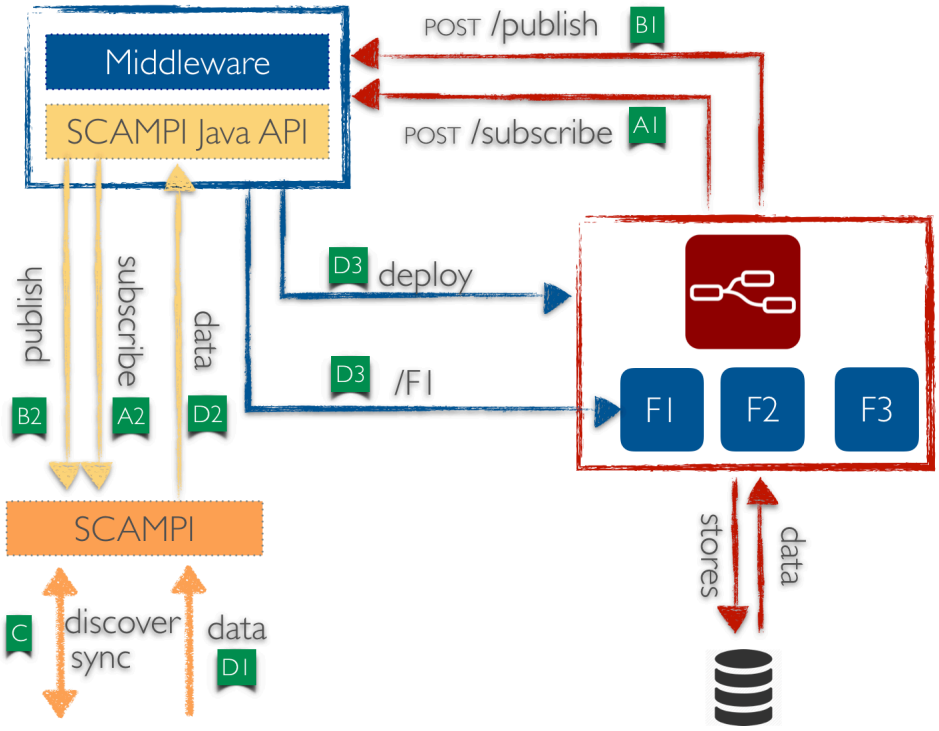

Figure 4.3: Software framework architecture summary.

(A) Flows are developed using node-RED UI, they can include publishing and subscribing REST calls to the middleware. If a flow subscribes to a certain topic, the middelware creates a topic-endpoint mapping between the topic and an endpoint for this flow specifically. Then send a subscribe request to SCAMPI. If another flow on the same instance wants to subscribe to the same topic, the middleware extends the mapping to include it, hence, once a message is received it gets forwarded to all subscribed flow endpoints.

(B) When the middleware receives a publish request from node-RED, it attaches the dependencies and an indicator that states if the response should be received by the sending node only. Then the message is forwarded to SCAMPI server.

(C) SCAMPI keeps synchronizing messages and discovering new peers continuously as long as its running. Also, storing some message for the store-carry-forward routing functionality.

(D) When a SCAMPI instance receives data it is forwarded to the middleware, which then checks the meta-data, resources, dependencies and then send the data to the subscribing flows from the topic-endpoint mapping. However, if the message is sent on the computation reserved topic, the middleware will deploy the flow to node-RED.

4.4 Summary

5 Implementation

Building on the framework architecture explained in Chapter 4, we describe the middleware proof-of-concept (POC) implementation in details to show that the framework architecture is sound. Moreover we describe the flows implementation used to create the use cases in order tp provide the evaluation for the framework.

5.1 Middleware

The middleware is written in Java, it uses *Maven* as a software project dependency management which handles the middleware build. *Maven* projects contain a Project Object Model (POM) file which describes the dependencies and libraries used by the project. In addition, it has build configuration management that is used to build the project and create a runnable jar file. The middleware can be compiled and packaged to a jar using the following command `mvn clean package`. The project contains dependencies for *Lombok* which is a library that generates getters, setters and constructors during compile time without the need to write them in the Java classes, *Gson* dependency in order to be able to read and write in JSON format, *SCAMPI API* which allows us to publish, subscribe and override functions from SCAMPI, dependencies to create a web application via *Spring Boot*.

The project is divided into several packages, we describe them in alphabetical order.

- First, the package `com.middleware.api` which contains two classes, `SCAMPApi.java` and `MiddlewareApi.java`. The class `SCAMPIApi.java` has a field of type `APP_LIB` from SCAMPI Java API which contains the core methods of SCAMPI server to connect, add status listeners, publish and subscribe to messages. As shown in 5.1, `SCAMPIApi` implements two classes via the `APP_LIB` for the SCAMPI server which includes functions like `OnConnected()`, `OnDisconnected()`, `OnConnectFailed()` and `OnStopped()`. It also implements `messageReceived()` which handles the messages once they are received from SCAMPI. Further, It contains a cache for messages and a field indicating the machine specification.

  The class `MiddlewareApi.java` contains REST API for the middleware web application that runs on port `8080`. It has two requests `POST /publish` and `POST /subscribe`

that connect to `APPL_LIB` in order to submit publish and subscribe requests to SCAMPI server. The class `MiddlewareAPI.java` has a field for the `topicMapping` which is the topic-endpoint mapping we explained in 4.3.4. Last but no least, it has a main method which in Java is used to run a class. In this case, it is used to run the built jar file from *Maven* which means once the jar file runs via the command `java -jar interface-1.0-SNAPSHOT.jar`, the main method is only thing that runs and therein it starts a web application via *Spring Boot*.

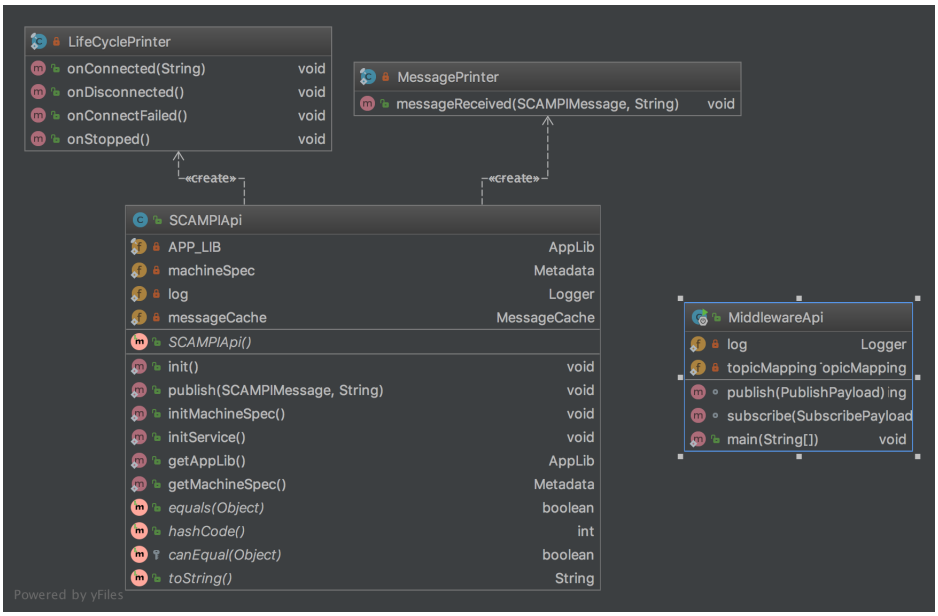

Figure 5.1: Class diagram for the api package.

- The second package is called `com.middleware.constants` and has one class `Constants.java` which contains all the constant fields used in the middelware across all other classes. This includes string keys for SCAMPI messages, some Linux commands, URL for the local node-RED instance, path for user home directory on the hosting machine and path for the JSON file that includes the machine specification. Furthermore, it contains the topic reserved for computations named *Main*

- The third package is the most important one `com.middleware.domain` which contains most of the services that is handled by the middleware. There are two singleton classes namely `ToppicMapping.java` and `MessageCache.java` that means there exist only one instance across the whole web application no matter how long it runs and no other class can create a new instance of these types. Thats because both are types of caches which should be consistent and global to any class which would need these caches.

The classes `CommandRunner.java` and `RESTHandler.java` are helpers. The first is used to run commands on the machine and has two methods `run(String)` and `getFreeRam()` which is equivalent to calling `run("free -m" )`. The second class is used as a REST client for sending requests to node-RED which are used to deploy computations and send data to endpoints.

Next are the classes `MessageHandler.java` and `Publisher.java` which contain the services for handling incoming messages from SCAMPI and publishing new message respectively. The `MessageHandler.java` is called from `SCAMPIApi.java` method `messageRecieved()` and differentiates between two types of the messages; computation ones which are received on the topic *Main* and handles them with the method `handleMainTopic(SCAMPIMessage)` which takes care of the resources, machine specifications, puts dependencies on node-RED local directory then the `RESTHandler.java` class is used to deploy the computation to node-RED, and data messages which are received on other topics and handles them with the method `handleSpecialTopic(ScampiMessage)`, it sends the data to any subscribed endpoint in the `toppicMapping` cache using also the `RESTHandler.java`.

The `Pubisher` is responsible for submitting publish requests to the `APP_LIB`. It is called from the `MiddlewareApi.java`. It creates a unique identifier for each new message, and adds the publisher global identifier to the *SCAMPIMessage* as well. It has the same topic differentiation as `MessageHandler.java`. On the one hand, if the publish topic is *Main*, it collects the dependencies and attach it to the *SCAMPIMessage* before it sends the message to SCMAPI server. On the other hand, if it is a data topic it checks the attachments and adjusts the response endpoint to whether it should be sent back to this node only, if this is the case, it creates a mapping between the global identifier of the node and the endpoint that sent the publish request then send it to SCAMPI server.

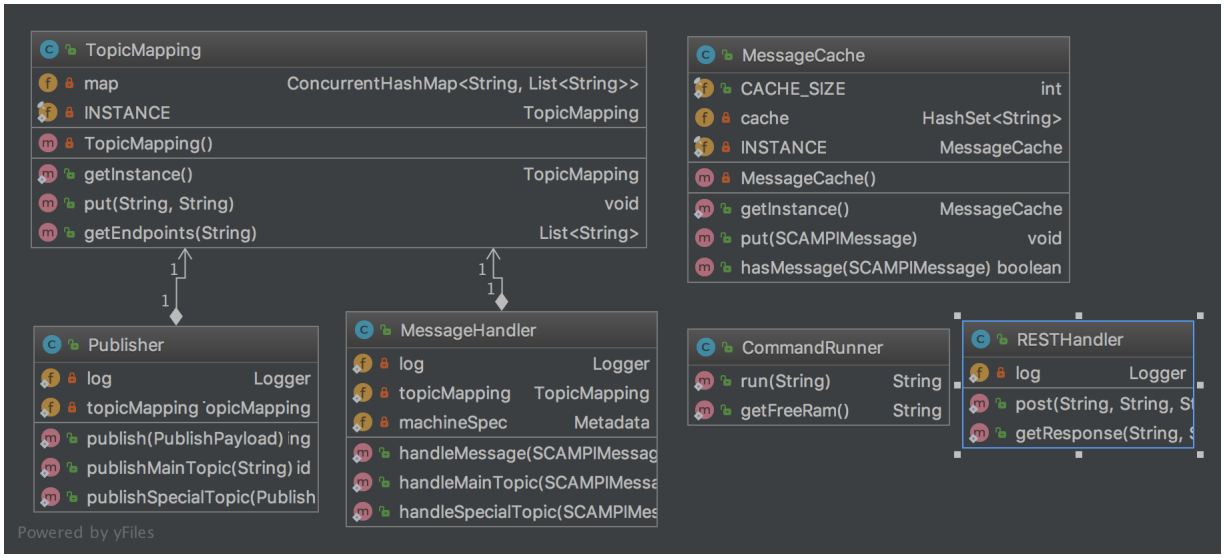

Figure 5.2: Class diagram for the domain package.

- The fourth package `com.middleware.exception` which contains the project's custom exceptions. Currently it only has a `RESTFailedException.java` class which handles node-RED deployments and data requests failures.
- The fifth and last package is `com.middleware.model`, it holds the models which are classes used to hold data and encapsulate them. Note that the Java classes do not contain any getters and setters or constructors. However, using *Lombok* library they are generated in compile time and they can be picked up by the Integrated Development Environment (IDE) as shown in 5.3.

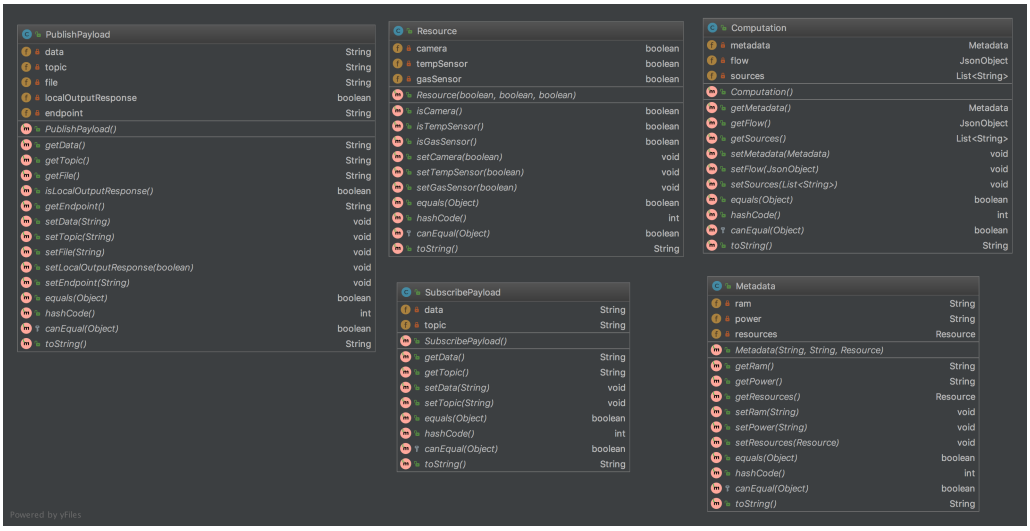

Figure 5.3: Class diagram for the model package.

5.2 Flows

This section explains all the flows developed in this thesis. It shows how the flows can be implemented to achieve different use cases. It also shows how we use node-RED flows in order to send computations.

5.2.1 Send Computations

In order to deliver flows which expresses computations for all nodes or selected ones according to available resources, sensors, actuators and also attach dependencies, we developed a node-RED flow shown in figure 5.4. The flow responds to the endpoint `GET localhost:1880/publish` and returns the HTML page in figure 5.5. Afterwards, the user can adjust the computation power needed by the flow, necessary free Random Access Memory(RAM), sensors and actuators. Then he/she must also write the flow identifier and attach dependencies, and eventually click on the button *push*, which calls another endpoint on the same flow `localhost:1880/push`, it receives the data, fetches the flow details and publish it to the middleware which handles the rest.

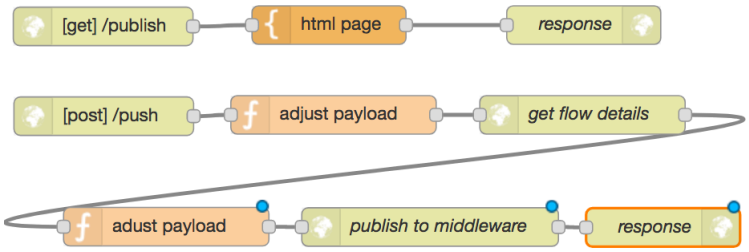

Figure 5.4: A flow that publishes computations to the middleware dn thus to SCAMPI.

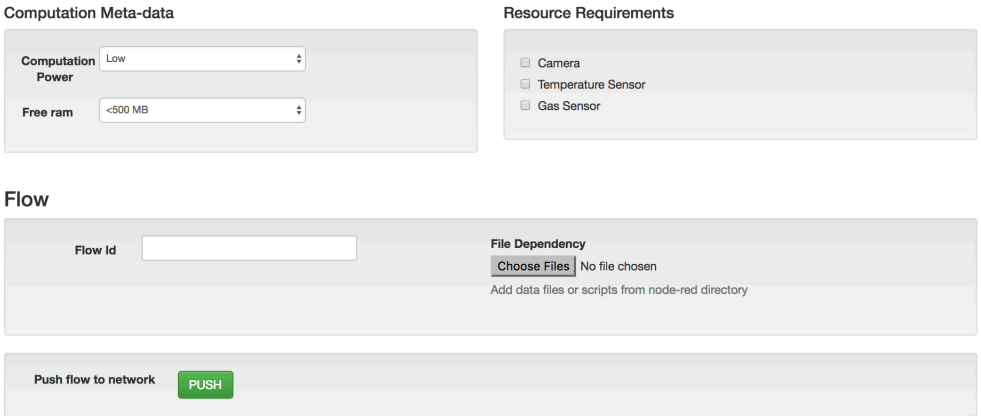

Figure 5.5: The HTML page used to publish computations conveyed in *html template* element.

5.2.2 Temperature Sensor Alert

This flow was developed to make sure that the software framework works against the basic IoT usage of sensors, actuators and time-series databases. As presented in figure 5.6, the flow runs once it was deployed to any node. Note that it would not have been deployed by the middleware without checking that it has the required resources and dependencies. Once the flow is deployed, it starts a script for sensing temperature which is sent as a dependency while sending the flow to other nodes. It then checks if the temperature is a valid number, then it gets stored in a database. Further, if the temperature is above 30 degree Celsius it runs a script, which is also sent as a dependency, to light a red led lamp. On top of that, the flow responds to the endpoint `localhost:1880/temp` and returns all the collected temperature data in the last two minutes.

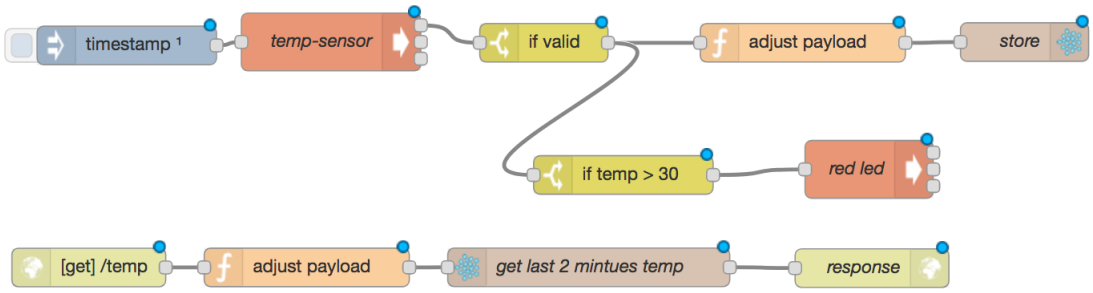

Figure 5.6: A flow that reads temperature and stores it, also start a red lamp if temperature is above 30 degree Celsius.

5.2.3 Detect Movement and Store Image Responses

As part of the implementation evaluation, we developed this flow to take part in a bigger use case. The flow is designed to detect movement through the infrared sensor and then take a picture, which is then published on the topic *NUC* for image recognition. However, in the push payload along with the picture, there is a field stating that the response should come back to this exact node, therefore, the middleware creates a mapping between the device global identifier and the endpoint before publishing. Also, as demonstrated in 5.7, the flow awaits messages on its endpoint and executes a script which starts a red led, it also store the recognized image in a database along with its accompanying data.

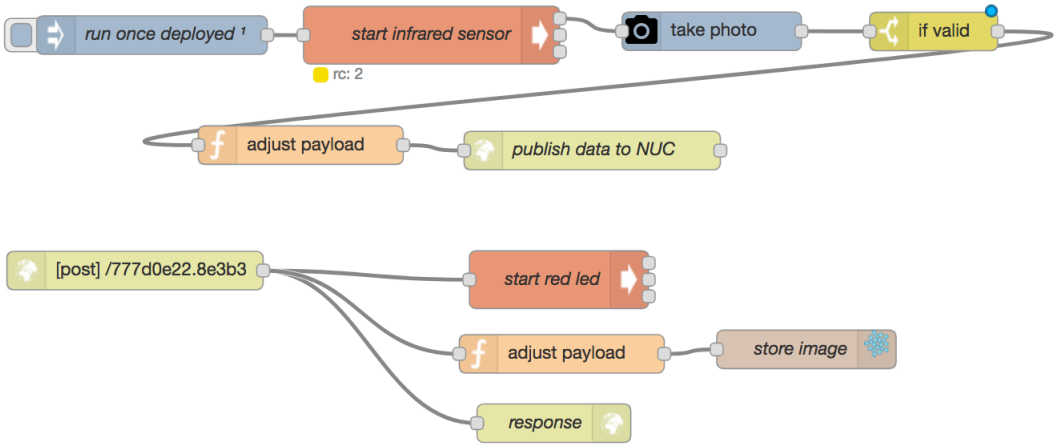

Figure 5.7: A flow that detects motion, take an image, publish message and store response.

5.2.4 Show Recognized images

Since compatibility is an important part to validate in our framework. We created a simple flow to show local composable flows. It simply queries the database for recognized images from the previous flow 5.2.3 when requested on the endpoint `GET localhost:1880/images`. It has an HTML page response showing all the images along with their image recognition confidence percentage data and time-stamp.

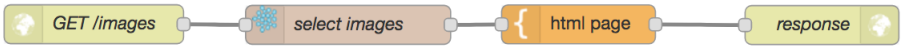

Figure 5.8: A flow that creates an endpoint for stored database images.

5.2.5 TensorFlow Water Bottle Recognition

To prove that we can send flows to machines with heavy computation power and lots of free RAM. We developed an image recognition flow using *TensorFlow* which is an open-source software library for machine intelligence. *TensorFlow* needs a 54MB image recognition model that recognizes 1000 object classes, one of them is a water bottle. It also needs the code to run the recognition algorithm which is a Java jar file of 29MB size. Therefore, the flow needs to carry all the mentioned dependencies. As show figure 5.9, the flow subscribes to the topic *NUC* once it is deployed. Then, when the middleware that runs on the same machine receives a message on the topic *NUC*, the middleware sends it to the corresponding endpoint from the topic-endpoint mapping, it also puts the dependencies in node-RED directory. Thereafter, the flow uses the message payload which should be an image, and runs the jar. If it results in

a water bottle as a best match, the flow responds to the sender with the result and a confidence percentage.

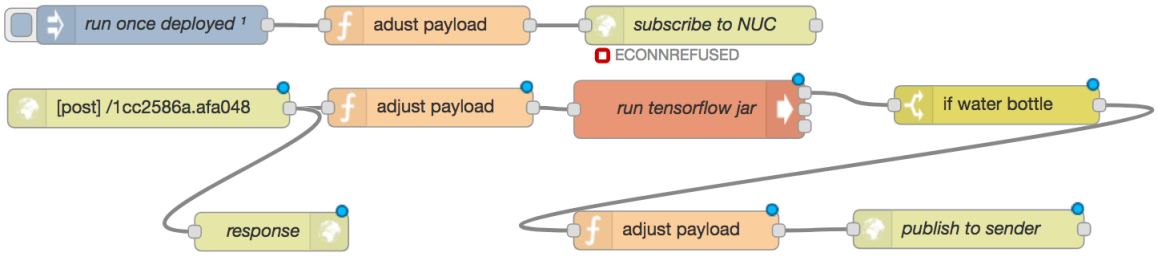

Figure 5.9: A flow that uses tensorflow to recognize a water bottle.

5.3 Starting the framework

The framework stack is run in the following order; SCAMPI, middleware and then node-red. The script used for starting the stack is as follows:

```
#!/bin/bash
java -jar SCAMPI.jar default_settings.txt > scampi-log.txt 2>&1 & disown
sleep 10
java -jar interface-1.0-SNAPSHOT.jar > interface-log.txt 2>&1 & disown
sleep 5
node-red > node-red-log.txt 2>&1 & disown
echo "All Set Up"
```

There is a sleeping period between each command in order to make sure that the previous one has started. The commands have to be executed in this order, however, if the middleware starts before SCAMPI it will not break and keep waiting for the server to start. But, node-RED must wait for both of them to start, as there are flows that only executes at the time of deployment, therefore the whole stack has to be running at this point. Note also, that there are logs for each process that can be found in the node-RED directory.

5.4 Summary

In this chapter we have described the software framework implementation following from the specifications, requirements and architecture design explained 4. Specifically, the middleware implementation and extending the Java API of SCAMPI. The other applications used in the software framework are only used but not implemented.

Moreover, we described the implementation of each flow used as part of implementing the use cases for the framework evaluation. At the end, a brief is given on how to start the framework and the order in which it should be run.

6 Evaluation

The goal of this chapter is to validate the framework architecture design for context-aware pervasive computing in challenged environments which was described in Chapter 4. It also evaluates the middleware implementation which was explained in Chapter 5. Note that, the delays and performances are to be taken with a grain of salt, as the system is designed as a proof-of-concept and not to run in production systems.

To satisfy all the requirements and specification for the software framework, we divided the evaluation into several sections. Each section validates certain specification via implementing a use case scenario. We describe which requirements where targeted at the beginning of each section. The devices used for the use cases have different hardware capabilities. They might also have sensors and actuators according to each use case. All the devices must be running our stack framework as explained in 5.3 except for android phones which has *Liberouter*, an implementation for SCAMPI on android phones. The devices we used in this evaluation are:

Name	Count	Stack	Performance
Intel NUC	1	Our framework	CPU:Intel Core i5-6260U Processor (4M Cache, up to 2.90 GHz) RAM: 16GB
Raspberry Pi 3 model B	2	Our framework	CPU: 1.2GHz RAM: 1GB
HTC One M9	1	Liberouter	CPU: Octa-core 4 x 2.0GHz + 4 x 1.5GHz RAM: 3GB

Table 6.1: Devices used for the implementation evaluation.

6.1 Basic IoT Usage

First, we wanted to evaluate that the software framework works with the basic IoT use cases using high rate data sensors and storing them in time-series database. The flow explained in 5.2.2 reads temperature on a regular time basis and stores it into a database, it also has an endpoint that can query for data between specific time intervals and if no time interval is specified, it will return temperature readings in the last two minutes. The flow also alerts for high temperatures by igniting a red led lamp.

The use case is an example of pervasive computing that checks if the temperature is above certain degree and then act by lighting the red led. It also serves as an abstraction for other use cases with real life purposes. For example, we might have a goal to start or close an air conditioning system in a building according to the temperature., Thus, the flow can be adjusted to include an API for the air conditioning system instead of lighting a red led. The use case can also be extended to include a monitoring system, that can show temperatures collected from different devices. Further, it can include location information along with the temperature data thus knowing what are the temperatures in different locations by syncing database instance on each device to the cloud.

In this experiment we used a two Raspberry Pis running our proposed stack that are connected along with a switch and router. The publishing PC is connected to the network through the router via Wi-Fi as shown in 6.1 . First, a PC sends the flow for temperature reading, data storage and creating an endpoint to query the data. Then, once the flow is received and deployed, the temperature sensors starts gathering temperatures and stores it into a local database.

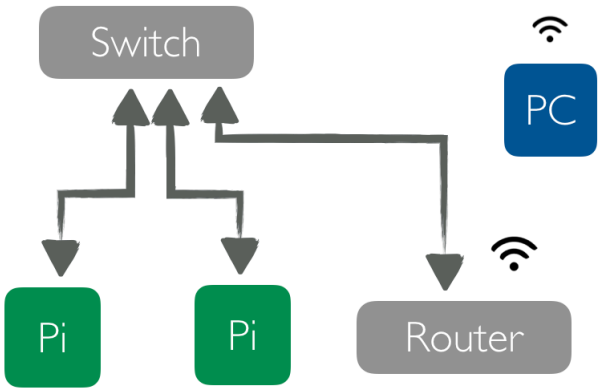

Figure 6.1: Testbed setup for temperature sensing flow.

The experiment was run 8 times and each time we measured the delay between pub-

lishing the flow from the PC t_{PC} till it is received and deployed by node-RED instances on the Raspberry Pis t_{PI}, we also measured the delay between flow deployment and the first database temperature insert query on the device's instance t_{db}.

	$t_{Pi1} - t_{PC}$	$t_{db} - t_{Pi1}$	$t_{Pi2} - t_{PC}$	$t_{db2} - t_{Pi2}$
meant	2.870	1.723	2.437	1.717
standard deviation	0.769	0.0376	1.003	0.013

Table 6.2: Results, mean and standard deviation of temperature flow delays.

Given these delays, the average time a temperature flow takes to reach the Raspberry Pi and gets deployed is 2.653 seconds , the average time till the first temperature insert in the database is 1.720 seconds. Which means that in our proof-of-concept implementation, it takes on average 4.373 seconds from the moment a flow which can sense the environment and stores data into a database till the first metric insertion is done.

6.2 Recognizing Water Bottles

Moving on to more complex scenarios, This use case is as follows, movements are detected around low computation devices portrayed as the Raspberry Pis. Once they detect movements around them, they take an image and send it to the topic *NUC*. A high performance machine "an Intel NUC" should be waiting for input on the same topic. As soon as the NUC receives an image, it runs an image recognition algorithm and responds back to the Raspberry Pi which sent the original message only if the recognizer recognizes a water bottle. When a Raspberry Pi receives the recognition result on its endpoint, this means that the image was a watter bottle with a certain confidence, therefore, the Pi signals a red led to light up and stores the result in a database.

The flows implementation used to create this use case can be found in 5.2.5 and 5.2.3. As part of this experiment when a flow requiring high amount of performance such as the image recognition flow is received by low performing devices they will not get deployed and will log that requirements were not satisfied. The same case applies when a flow requiring low amount of computing power is received by a high performing device, of course, this could be optimized because clearly high performance devices can execute flows which are not needy. The computational requirements needed by each flow are decided before sending the flow over to other devices using the HTML page implemented in the publishing flow explained in 5.2.1.

This scenario helps us validate several requirements for the software framework evaluation:

1. Sending flows to all nodes connected to a network running our framework stack.
2. Controlling which devices can deploy the flows by sending meta-data for the resources and computation power along with flow. Therefore, being able to only send to some sets of nodes.
3. Checking requirements of flows and rejecting the deployment if they were not satisfied at the receiving devices.
4. Sending data or flows to a specific node using its global identifier.
5. Carrying flow dependencies in order to guarantee a successful run at the receiving node.

As stated every device must have the framework stack running before we start our use case. So after making sure its running we start publishing the flows. In this use case, the testbed setup is shown in figure 6.2, it consists of two Raspberry Pis, an Intel NUC and a router connected via switch. The PC which will publish the computations is connected through the router via WI-FI.

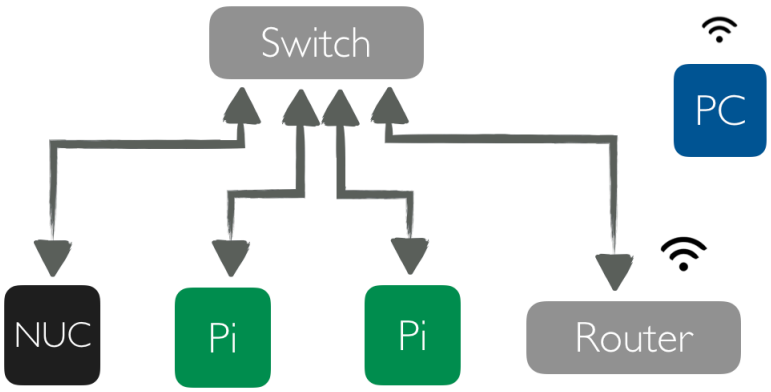

Figure 6.2: Testbed setup for recognizing water bottles.

We started by publishing the flow 5.2.5 for image recognition with all its dependencies in total 83MB, then we published the motion detection flow 5.2.3 with the sensor scripts. We measured the delay between publishing flows from the PC till it was received and deployed by node-RED on each instance. The use case was run 8 times, we also measured the delay when a motion was detected by Raspberry Pi and an image was sent to the NUC and the recognizer reply, this was also run 8 times for each Pi,

a total of 16 runs. The sequence diagram shown in 6.3, illustrates the procedure in addition to the time initials for each part of the process.

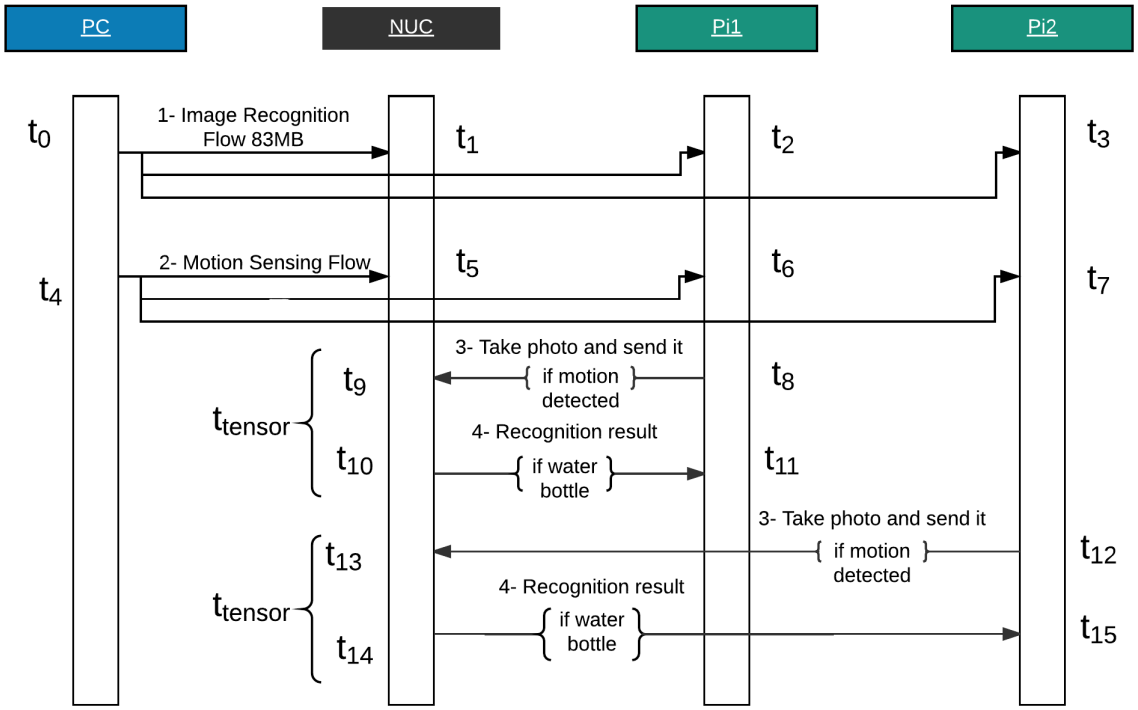

Figure 6.3: Sequence diagram for recognizing water bottles.

At time t_0 the image recognition flow was published, at times t_1, t_2 and t_3 it was received by the NUC, Pi1 and Pi2 respectively. Then at time t_4 the motion sensing flow was published and at times t_5, t_6 and t_7 it was received by the other devices as well. A times t_8 and t_{12} the Pis detected motion, took an image and then sent a message to the NUC. At times $t9$ and $t13$, NUC received messages from the Pis and started processing, detected a watter bottle and then sent the results back to the senders at t_{10} and t_{14}. The Pis received recognition responses with the confidence percentage at t_{11} and t_{12}. The tables 6.3, 6.4 and 6.5 show the mean and standard deviation of the delays.

	$t_1 - t_0$	$t_2 - t_0$	$t_3 - t_0$
1	23.245	28.226	20.826
2	20.699	49.666	19.051
3	22.330	21.756	29.425
4	21.387	22.265	32.680
5	22.549	27.662	22.627
6	24.095	27.222	15.922
7	19.597	25.561	43.311
8	22.586	26.898	28.627
mean	23.245 s	28.226 s	20.826 s
standard deviation	1.440 s	8.830 s	8.851 s

Table 6.3: Results, mean and standard deviation of the image recognition flow delays.

	$t_5 - t_4$	$t_6 - t_4$	$t_7 - t_4$
1	0.104	2.386	2.233
2	0.106	2.323	2.348
3	0.090	2.321	1.380
4	0.080	2.340	1.275
5	0.076	2.210	1.441
6	0.081	2.299	2.247
7	0.088	2.329	2.211
8	0.074	2.368	2.450
mean t	0.087 s	2.322 s	1.948 s
standard deviation	0.012 s	0.053 s	0.491 s

Table 6.4: Results, mean and standard deviation of the motion detection flow delays.

	$t_9 - t_8$	$t_{11} - t_{10}$	$t_{13} - t_{12}$	$t_{15} - t_{14}$	t_{tensor}
1	0.266	0.659	0.276	0.686	5.582
2	0.296	0.674	0.323	0.79	5.640
3	0.186	0.728	0.313	0.676	5.573
4	0.342	0.807	0.222	0.661	4.940
5	0.227	0.752	0.219	0.722	5.603
6	0.262	0.713	0.322	0.742	5.599
7	0.306	0.684	0.212	0.663	5.575
8	0.374	0.582	0.293	0.648	5.581
mean	0.282 s	0.700 s	0.273 s	0.6985 s	5.512 s
standard deviation	0.061 s	0.067 s	0.048 s	0.0488 s	0.217 s

Table 6.5: Results, mean and standard deviation for sending and receiving data delays.

It is clear that delays of the flow carrying image recognition dependencies took long times compared to the motion flow and the data messages between the NUC and Pis. That is mostly because the size of data in which the message is carrying. The data messages included images with size from 50K to 90K and motion sensing flow had dependencies with size less than 2K. It is also clear that when the receiving side are the Raspberry Pis, delays are usually larger than when the receiving side is the NUC. This is evident in the motion sensing delays 6.4 and also in 6.5 despite the message sent from the Raspberry Pis to the NUC are heavier than their replies in terms of size, but the delays are much less.

6.3 Local Composability

In order to prove that we can compose flows locally using our framework, we created a use case that builds on the previous experiment. Our goal is to use the same database configuration in both flows, one to write into a database while the other reads. Therefore, being able to compose two flows in order to achieve a bigger use case.

Continuing on the same test bed setup as 6.2 and same scenario. After the Raspberry Pis stored the recognized images of water bottles into their respective databases, we sent a flow that retrieves these images from the database into a web endpoint along with their confidence percentages explained in 5.2.4. By doing this, we make sure that flows can be locally composed using the same database configuration. We also measured the delay between sending the images flow from the PC until it was deployed by node-RED on other devices. The flow required low computational effort, therefore,

ir will not get deployed on the NUC device. The experiment was also run 8 times.

	$t_{17} - t_{16}$	$t_{18} - t_{16}$	$t_{19} - t_{16}$
1	0.102	0.761	0.838
2	0.163	0.799	0.799
3	0.125	0.927	0.923
4	0.091	0.752	0.868
5	0.070	0.814	0.778
6	0.095	0.824	0.861
7	0.094	0.736	0.835
8	0.098	0.798	0.769
mean	0.105	0.801	0.834
standard deviation	0.028	0.060	0.051

Table 6.6: Results, mean and standard deviation for retrieving recognized images flow delays.

Delays in this use case is quite low because this flow does not have any dependencies at all. Comparing the times in which the Raspberry Pi's have received and deployed flows, this one has quite the least delay. However, the NUC in this flow seemed to have a bit higher delay average than the motion sensing flow with not so much dependencies as well.

6.4 Challenged Networks

In this section, our aim is to evaluate that the framework works in challenged networks with no end-to-end path between sender and receiver. We used the same setup as 6.2 but with two major changes. The first change is that we disconnected a Raspberry PI from the network switch, therefore, it is no longer connected to the other devices or the publishing PC, we also set-upped the disconnected node as an access point in which other devices can connect to using Wi-Fi. The second change is that we introduced an android phone that can connect to both the Raspberry Pi's access point and the router's Wi-Fi connected to the switch. Additionally, the device switches it's Wi-Fi between the access point and router Wi-Fi each 80 seconds. The system setup is shown in figure 6.4.

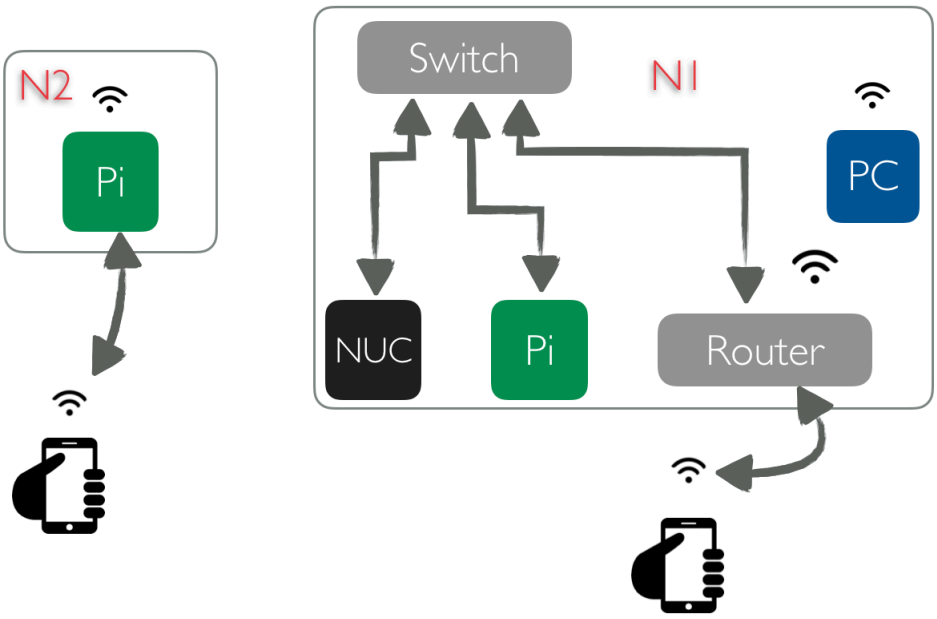

Figure 6.4: Testbed setup for challenged and delay tolerant networks.

The experiment was run only one time for an extended amount of 50 minutes. First, we used the PC to publish the image recognition flow and the motion sensing flow. Second, we waited some time until we made sure that the flows have reached the disconnected Pi. Afterwards, we passed hands over the infrared sensor of the disconnected Pi and put a water bottle in front of the camera. We repeated this action multiple times.

This use case is evaluated differently, since in this experiment, we do not have a way to map image recognition requests sent from the disconnected Pi to the NUC with their respective responses. Also, there are some recognition requests that was not detected as a water bottle therefore there was no response from the NUC. Thats why we evaluate this use case on 3 different levels. First, the delay between publishing flows from the PC and receiving them on all devices. Second, the delay between sending the images from the disconnected Pi to the NUC. Third, the delay between sending the result from NUC till it is deployed to the disconnected Pi. Note that the number of messages sent from the Pi to the NUC and vice versa may not be equal due to the fact that some images were not recognized as water bottles. In addition, we have no means to map a message which was sent as a image recognition request to another one sent as a response which could be future work.

The following table shows the first evaluation phase, t_{NUC}, t_{Pi} and $t_{disconnected-pi}$ are the delays between publishing the flows from the PC till it reach each device.

	t_{NUC}	t_{Pi1}	$t_{disconnected-pi}$
image recognition flow	39.414	53.719	312.072s
motion sensing flow	3.163	5.401	215.657

Table 6.7: The delays for sending flows to the network devices including the disconnected PI.

From the above table we can definitely see the delay added to the disconnected Pi. This main reason for that is, beside not having a direct connection, the switching window of the android device. It can affect the transfer in different ways and has its trade-offs. If the switching time is too big, the delay will most probably increase because after a flow is uploaded to the phone, it will have to wait for some time until the window closes before it switches to the other network. But also, having it too small, flows with large size of dependencies will not succeed to upload data to the android phone in time.

Next we show the delays of some messages that were sent from the disconnected Pi to the NUC carrying images with their average and standard deviation.

	t
1	284.957
2	216.409
3	180.415
4	228.811
5	228.329
6	220.11
7	214.213
8	111.267
9	104.166
10	101.188
11	179.825
12	178.946
13	166.999
mean	185.818
standard deviation	54.966

Table 6.8: Messages delays sent from the disconnected Raspberry Pi to the NUC.

Finally, we evaluated the response message returning back from the NUC to the disconnected Raspberry Pi having successfully recognized a water bottle.

	t
1	59.531
2	51.51
3	51.826
4	213.212
5	67.856
6	67.673
7	68.188
8	68.103
9	64.292
10	192.479
mean	90.467
standard deviation	59.770

Table 6.9: Messages delays sent from the NUC to disconnected Raspberry Pi having successfully recognized water bottles.

Clearly, delays are much bigger, but the good thing is that several devices were able to communicate despite not having any direct connection between each other. We were also able to deploy computations to a device which was not in our publishing network. On another note, these delays can be drastically optimized if we decreased the switching window because unlike flows which can carry huge dependencies, these data message do not.

6.5 Summary

In this section we have distributed our framework and middleware implementation evaluation on several parts each describing a different use case with various requirements. We started by showing the basic IoT usage of sensor networks in which we deployed a computation that monitors temperature on several devices. Then, we went on with a more complex scenario where we ran image recognition algorithm for images on high performing machines coming from devices with low computation capabilities. Afterwards, we showed how flows can be composed locally using the same database configuration. Finally, we created a delay-tolerant network with a disconnected device and showed that we can deploy computations and communicate with it.

7 Conclusion

7.1 Summary

7.2 Future Work

7.2.1 Streaming API

Bibliography

[Ahl+12] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. "A survey of information-centric networking." In: *IEEE Communications Magazine* 50.7 (July 2012), pp. 26–36. ISSN: 0163-6804. DOI: 10.1109/MCOM.2012.6231276.

[AMQ] AMQP. *Advanced Message Queuing Protocol*. https://www.amqp.org, Accessed: 2017-05-23.

[CDK01] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems (3rd Ed.): Concepts and Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0-201-61918-0.

[CFJ03] H. Chen, T. Finin, and A. Joshi. "An Ontology for Context-aware Pervasive Computing Environments." In: *Knowl. Eng. Rev.* 18.3 (Sept. 2003), pp. 197–207. ISSN: 0269-8889. DOI: DOI:10.1017/S0269888904000025.

[Cha+08] M. Chan, D. Estève, C. Escriba, and E. Campo. "A Review of Smart homes-Present State and Future Challenges." In: *Comput. Methods Prog. Biomed.* 91.1 (July 2008), pp. 55–81. ISSN: 0169-2607. DOI: 10.1016/j.cmpb.2008.02.001.

[Cra] CrateDB. https://crate.io/, Accessed: 2017-05-09.

[CZ16] M. Chiang and T. Zhang. "Fog and IoT: An Overview of Research Opportunities." In: *IEEE Internet of Things Journal* 3.6 (Dec. 2016), pp. 854–864. ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2584538.

[Dan+13] C. Dannewitz, D. Kutscher, B. Ohlman, S. Farrell, B. Ahlgren, and H. Karl. "Network of Information (NetInf) - An Information-centric Networking Architecture." In: *Comput. Commun.* 36.7 (Apr. 2013), pp. 721–735. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2013.01.009.

[Doe+08] M. Doering, S. Lahde, J. Morgenroth, and L. Wolf. "IBR-DTN: An Efficient Implementation for Embedded Systems." In: *Proceedings of the Third ACM Workshop on Challenged Networks*. CHANTS '08. San Francisco, California, USA: ACM, 2008, pp. 117–120. ISBN: 978-1-60558-186-6. DOI: 10.1145/1409985.1410008.

[EFA] EFA. *Raspberry PI model 3 design*. Accessed: 2017-05-09, https://commons.wikimedia.org/wiki/File:RaspberryPi_3B.svg Efa at English Wikipedia [GFDL(http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons.

[Ela] Elasticsearch. Elasticsearch as a time-series database https://www.elastic.co/blog/elasticsearch-as-a-time-series-data-store, Accessed: 2017-05-09.

[Fal+07] K. Fall, K. L. Scott, S. C. Burleigh, L. Torgerson, A. J. Hooke, H. S. Weiss, R. C. Durst, and V. Cerf. "Delay-tolerant networking architecture." In: (2007).

[Fal03] K. Fall. "A Delay-tolerant Network Architecture for Challenged Internets." In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '03. Karlsruhe, Germany: ACM, 2003, pp. 27–34. ISBN: 1-58113-735-4. DOI: 10.1145/863955.863960.

[Gil16] A. Gilchrist. *Industry 4.0: The Industrial Internet of Things*. 1st. Berkely, CA, USA: Apress, 2016. ISBN: 1484220463, 9781484220467.

[Inf] InfluxDB. https://docs.influxdata.com/influxdb/v1.2, Accessed: 2017-05-09.

[Jac+09] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. "Networking Named Content." In: *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*. CoNEXT '09. Rome, Italy: ACM, 2009, pp. 1–12. ISBN: 978-1-60558-636-6. DOI: 10.1145/1658939.1658941.

[Kär+12] T. Kärkkäinen, M. Pitkänen, P. Houghton, and J. Ott. "SCAMPI Application Platform." In: *Proceedings of the Seventh ACM International Workshop on Challenged Networks*. CHANTS '12. Istanbul, Turkey: ACM, 2012, pp. 83–86. ISBN: 978-1-4503-1284-4. DOI: 10.1145/2348616.2348636.

[Lei+15] B. Leighton, S. J. D. Cox, N. J. Car, M. P. Stenson, J. Vleeshouwer, and J. Hodge. "A Best of Both Worlds Approach to Complex, Efficient, Time Series Data Delivery." In: *Environmental Software Systems. Infrastructures, Services and Applications: 11th IFIP WG 5.11 International Symposium, ISESS 2015, Melbourne, VIC, Australia, March 25-27, 2015. Proceedings*. Ed. by R. Denzer, R. M. Argent, G. Schimak, and J. Hebíek. Cham: Springer International Publishing, 2015, pp. 371–379. ISBN: 978-3-319-15994-2. DOI: 10.1007/978-3-319-15994-2_37.

[Mio+12] D. Miorandi, S. Sicari, F. D. Pellegrini, and I. Chlamtac. "Internet of things: Vision, applications and research challenges." In: *Ad Hoc Networks* 10.7 (2012), pp. 1497–1516. ISSN: 1570-8705. DOI: http://doi.org/10.1016/j.adhoc.2012.02.016.

[Mon] MongoDB. MongoDB as a time-series database https://www.mongodb.com/blog/post/schema-design-for-time-series-data-in-mongodb, Accessed: 2017-05-09.

[MQT] MQTT. *Message Queue Telemetry Transport*. http://mqtt.org/, Accessed: 2017-05-23.

[Nod] Node-red. https://nodered.org, Accessed: 2017-05-09.

[Ope] OpenTSDB. http://opentsdb.net/, Accessed: 2017-05-09.

[Ras] Raspberry-Pi-Foundation. https://www.raspberrypi.org, Accessed: 2017-05-09.

[SB07] K. L. Scott and S. Burleigh. "Bundle protocol specification." In: (2007).

[SHB10] G. Schiele, M. Handte, and C. Becker. "Pervasive Computing Middleware." In: *Handbook of Ambient Intelligence and Smart Environments*. Ed. by H. Nakashima, H. Aghajan, and J. C. Augusto. Boston, MA: Springer US, 2010, pp. 201–227. ISBN: 978-0-387-93808-0. DOI: 10.1007/978-0-387-93808-0_8.

[SM03] D. Saha and A. Mukherjee. "Pervasive Computing: A Paradigm for the 21st Century." In: *Computer* 36.3 (Mar. 2003), pp. 25–31. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1185214.

[TSS10] D. Trossen, M. Sarela, and K. Sollins. "Arguments for an Information-centric Internetworking Architecture." In: *SIGCOMM Comput. Commun. Rev.* 40.2 (Apr. 2010), pp. 26–33. ISSN: 0146-4833. DOI: 10.1145/1764873.1764878.

[Wei91] M. Weiser. "The Computer for the 21st Century." In: *Scientific American* 265.3 (Jan. 1991), pp. 66–75.

[Xia+12] F. Xia, L. T. Yang, L. Wang, and A. Vinel. "Internet of Things." In: *International Journal of Communication Systems* 25.9 (2012), pp. 1101–1102. ISSN: 1099-1131. DOI: 10.1002/dac.2417.

[Xyl+14] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos. "A Survey of Information-Centric Networking Research." In: *IEEE Communications Surveys Tutorials* 16.2 (Second 2014), pp. 1024–1049. ISSN: 1553-877X. DOI: 10.1109/SURV.2013.070813.00063.

[YLL15] S. Yi, C. Li, and Q. Li. "A Survey of Fog Computing: Concepts, Applications and Issues." In: *Proceedings of the 2015 Workshop on Mobile Big Data*. Mobidata '15. Hangzhou, China: ACM, 2015, pp. 37–42. ISBN: 978-1-4503-3524-9. DOI: 10.1145/2757384.2757397.

[Z S14] C. B. Z. Shelby K. Hartke. *The Constrained Application Protocol (CoAP)*. https://tools.ietf.org/html/rfc7252, Accessed: 2017-05-23. 2014.

[Zan+14] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. "Internet of Things for Smart Cities." In: *IEEE Internet of Things Journal* 1.1 (Feb. 2014), pp. 22–32. ISSN: 2327-4662. DOI: 10.1109/JIOT.2014.2306328.