FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

A Pervasive Computing Framework for Distributed Computational Flow Composition and Execution

Aly Saleh

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

A Pervasive Computing Framework for Distributed Computational Flow Composition and Execution

Pervasive Computing Framework für Verteilte Rechenbetonte Ablauf-Zusammensetzung und Ausführung

Author:	Aly Saleh
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	M.Sc. Teemu Kärkkäinen
Submission Date:	June 11, 2017

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

Munich, June 11, 2017 Aly Saleh

Acknowledgments

Abstract

In recent years, research and development in Internet of Things, pervasive computing and machine-to-machine communication have been trending in the computer science field alongside the evolution of wireless sensor networks and single board computers. This has motivated researches and developers to harness the power of combining intelligent agents capable of acting on their own with small computer devices, in order to enhance the life quality of human beings, preserve the environment, improve the manufacturing process and provide better health care. As a result, the usage of networking concepts such as Information-Centric, Delay-Tolerant architectures and publish-subscribe messaging systems has nurtured in order to provide efficient machine communication mechanisms to smart devices. Despite the increase of pervasive use cases, there is still a lack of frameworks that facilitates the distribution and harmonization of pervasive use cases. A framework that would assist in designing and deploying different use cases to edge devices in a network and allows communication and data sharing between them. As a result, we propose a software framework for distributing flows which are representations of computations describing use cases to smart devices in a network. The framework also takes care of the flows dependencies, resources, data communication, service discovery, delivering computations and data to challenged networks. It relies on the concepts of pervasive computing, information-centric and delay-tolerant networking. The framework also allows having inter-relationships between flows either locally on the same device or globally across different devices. The framework can help researches, industries and individuals to design and distribute computations to smart devices using flow-based programming with minimal amount of code.

Contents

List of Figures

List of Tables

1. Introduction

The concept of Pervasive or Ubiquitous computing, in which devices are integrated with intelligent agents and expected to support human needs anytime and anywhere without their interference, existed long ago since 1991 in Mark Weiser's paper "The Computer for the 21st Century" [Wei91]. Even though pervasive computing was there for a long time, research and development in this area have recently flourished again following the prosperity of wireless sensor networks, single board computers and embedded micro-processors in our daily objects. Today, pervasive devices are expected to act on their own, be context-aware and include intelligent agents to support human beings and increase their life quality while being indistinguishable from everyday objects. In 1999 the term Internet of Things (IoT) appeared and since then it has been used to refer to networks of smart things and ideas around establishing smart cities, homes and factories. Smart devices used in these contexts are heterogeneous and expected to communicate and share knowledge with either each other or the cloud.

Despite the various applications for pervasive agents and IoT networks, there is still a lack of frameworks that could harmonize and orchestrate the deployment of different use cases to these networks. There are many reasons for this absence, first is the heterogeneity of devices, they differ in their computing capabilities. Second is the varying availability of gadgets including sensors and actuators attached to the smart devices in the network. Third, the different nature of use cases and their dependencies develops an obstacle towards using a unified framework for most use cases. Fourth, providing a communication mechanism to the framework that works seamlessly and discovers other peer smart devices remains a challenge. Last but not least, the existence of challenged networks makes it rather hard to reach devices without an end-to-end path. Advances in research specifically pervasive computing, IoT, Delay-Tolerant and Information-Centric networking in addition to current available platforms, have given us insights in order to pursue creating a framework for pervasive computing.

The main goal of our work is to propose a framework architecture that can distribute pervasive use case computations with their dependencies to all smart devices in a network with respect to their resources. Also, the framework allows peer discovery and communication for smart devices without host names which makes it even more dynamic. The framework also allows computations to be self-contained and have

inter-relationships between each other in addition to providing the proper execution environment. Further, we extended the framework architecture to deliver the computations to smart devices even in challenged networks.

Our framework architecture suggests a stack that is installed on each smart device and composed of a delay-tolerant, information-centric and publish subscribe messaging system. It handles the communication between all smart devices and implements service discovery at the stack bottom. Above the messaging system, a middleware exists that harmonizes and orchestrates computation dependencies, resources and deployment. This is done by ensuring that each computation has its dependencies attached, and verifies that the smart device has the required resources from hardware requirement, sensors and actuators. Then the middleware decides whether to deploy the computation or not. Finally on top of the stack, an execution environment that acts as a host for all computations deployed from the middleware and as a user interface to design and compose different use case computations.

1.1. Scope and Goals

The scope of this thesis is to present the design and architecture of a pervasive framework for distributing, composing and executing computations which implement various use cases. The design is based on the concepts of delay-tolerant, information-centric networking, pervasive computing and executing computations on the network edge. Requirements for the framework design are extracted from real life use cases. We base our design on these requirements and also present a proof-of-concept implementation to show that the design is feasible and can be realized. Then, the implementation and design are evaluated by running several experiments each targeting a specific set of requirements.

The framework will enable smart devices to communicate together through a publish-subscribe scheme even without an end-to-end path, thus sending and receiving data and computations. It will also provide the user an interface to design flows which are required to implement use cases as well as to publish flows either to all the smart devices in a network, a set or a specific device. In addition, it will allow the user to adjust the resources and attach dependencies to flows in order to compensate for the heterogeneous devices in the network and make sure flows will be executed successfully.

More specifically, the goal of this thesis is to provide a framework that enables the user to: i) develop a flow that portrays a pervasive use case, ii) choose the flow hardware requirements such as the computation power and the random access memory

that fits the use case, iii) choose the required sensors and actuators needed by the flow, iv) attach necessary flow dependencies in order to ensure the computation will run successfully, v) send the flow to all devices in a network, vi) ensures the flow is received by all targeted devices, vii) handle the deployment and execution of the flow if it satisfies the needed requirements and resources, viii) allow data communication between the framework devices using information-centric architecture, ix) compose flows by matching their the input and output, x) reach isolated devices in separate networks using delay-tolerant architecture.

1.2. Thesis Structure

This thesis is constructed in the following way; in the first chapter we present the problem along with a brief introduction to the proposed solution. The second chapter consists of the background information that represent the underlying concepts of this thesis in addition to the platforms used to implement the proposed framework. In the third chapter we define the theoretical terms, key concepts and possible solutions to the framework's main challenges. Afterwards in Chapter 4, we explain some of the real life use cases, requirements elicitation and the approach we took to design our framework architecture. Chapter 5 describes the implementation details of the main contribution of this work besides the use cases implementation used in the evaluation chapter. Then, we evaluate the framework implementation and architecture in Chapter 6 by running several experimental use cases each satisfying a different set of requirements. Finally we conclude our work, summarize the results and also explain the room for improvement and future work in Chapter 7.

2. Background

This chapter describes the concepts and background information that this thesis uses and relies on. It gives a brief introduction to the 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 proposed 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 these 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 of small-scale, high-performance, low-cost single board computers such as Raspberry Pi [Ras] and Banana Pi [Ban], 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 [Gar], 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 these 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 [LXZ15].

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*.

2.2. 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 and 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

the 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.

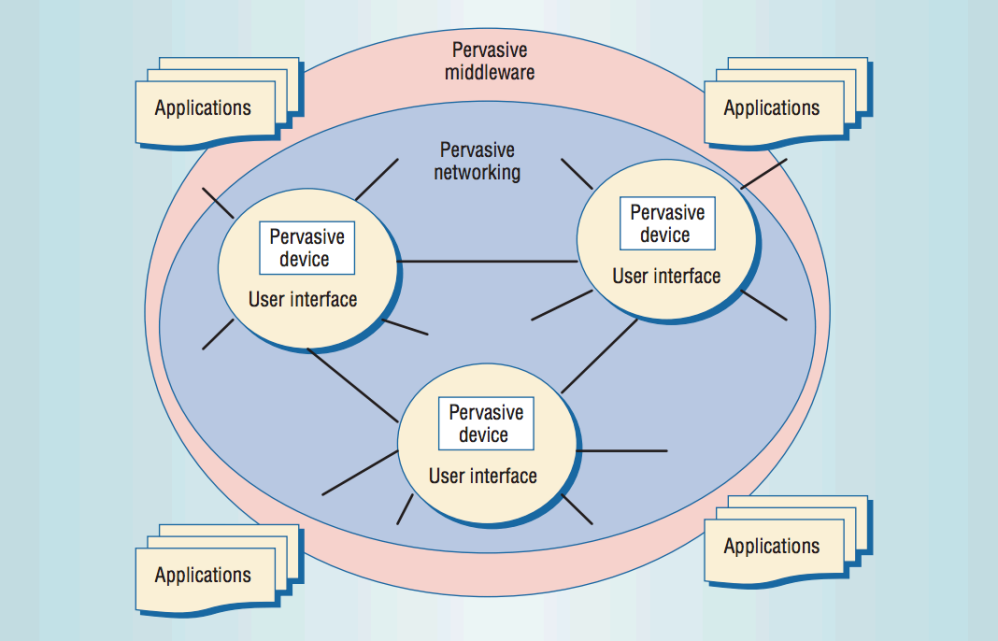

Figure 2.1.: Pervasive Computing environment architecture. *Adopted from [SM03].*

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.

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 on smart devices, readings uncertainty of sensors 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.

2.2.1. Fog Computing

The fog is an extension of cloud computing to 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 is yet to determine which applications should go where. The fog is characterized by having lower latency than the cloud, thus it is better suited for 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].

2.3. 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 enable 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 centralized intermediary. 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.

The broker-less architecture means that machines communicate directly with 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, the architecture needs to deal with unavailable machines that may not have started yet. In addition, it has to handle message delivery to machines with no end-to-end path or direct connectivity. ZeroMQ [Zer] is one implementation of broker-less architectures.

There is yet another approach which is endpoint centric such as RESTful services, web sockets and protocols like Constrained Application Protocol CoAP [Z S14] that uses the REST architectural style and is built over UDP. It also supports service discovery, multicast and asynchronous messaging exchange.

2.4. Networking

The following is a brief introduction of the main networking grounds that are used in this thesis. Since the proposed software framework focuses on exchanging data and computation between devices even without an end-to-end path, it is imperative that we shed light on Information-Centric Networking which proposes replacing current host-centric Internet architecture with a content-centric one. Additionally, having no end-to-end paths between some of our devices, 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.

2.4.1. 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 described by Dirk Trossen et al. as a networking architecture that aims 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.

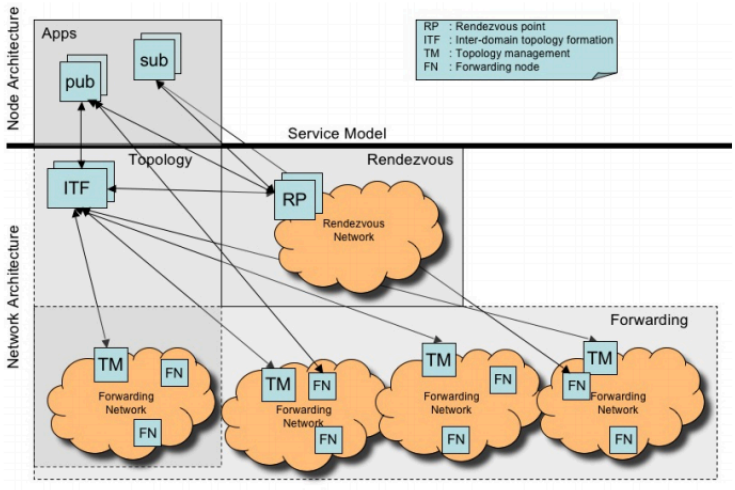

Figure 2.2.: ICN architecture by Dirk Trossen et al. *Adopted from* [TSS10].

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. It comprises 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 (NDOs). The NDO is only concerned by its name and data. As long as the 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 on 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].

2.4.1.1. 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 responds 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 supports on-path caching with name-based routing, however, off-path can also be supported [Xyl+14].

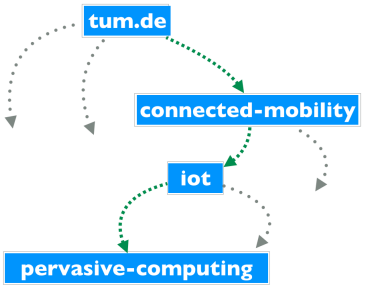

Figure 2.3.: Hierarchal namespace example for CCN.

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

- *Pending Interest Table (PIT)*: which stores the subscriptions and interests of NDOs. The subscription does not have to originate from the node itself, rather can be 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.

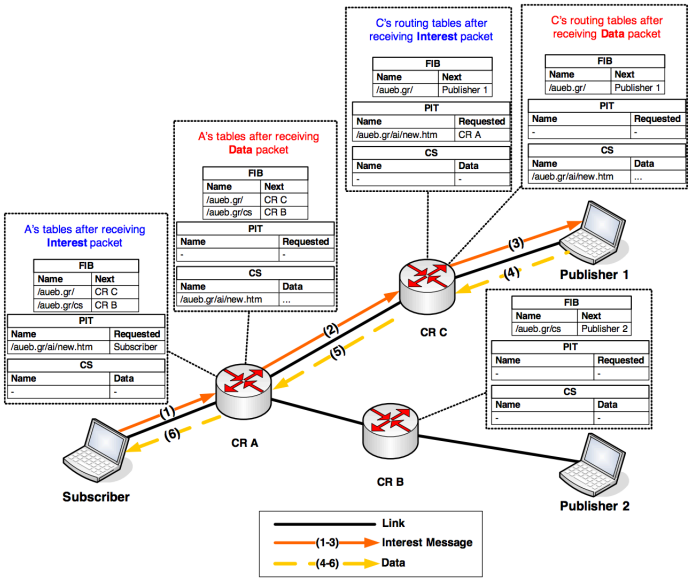

Figure 2.4.: Content centric networking architecture and flow. *Adopted from [Xyl+14].*

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.

2.4.1.2. 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 but 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].

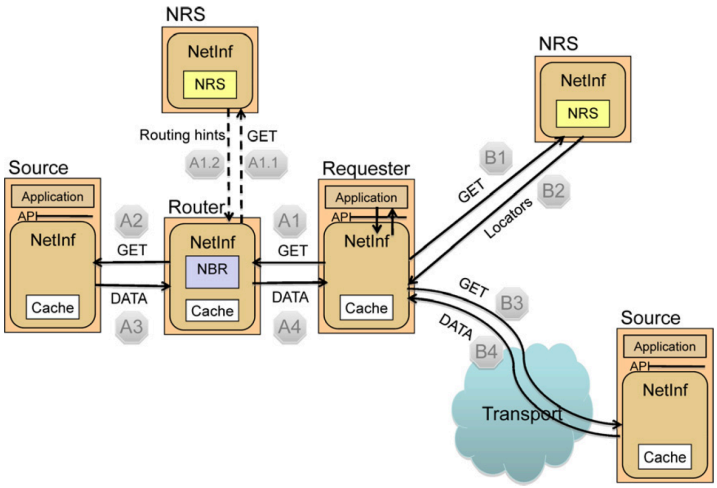

Figure 2.5.: NetInf routing flow example. *Adopted from [Dan+13].*

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**.

2.4.2. Delay Tolerant Networking

Delay-Tolerant Networking (DTN) is an overlay architecture proposed to overcome unreliable connections between devices through asynchronous message forwarding. In some challenged networks, there might not exist an end-to-end path between devices 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 resides above the existing protocol stacks thus 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 the *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 routing 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 [Dem05] is a reference implementation by the DTNRG to demonstrate the basic functionalities thus the performance is not optimized. There are also ION [Bur06] and DTNLite [PN03], however, they do not allow the use of common programming languages. IBR-DTN [Doe+08] is yet another implementation designed for embedded systems, it also has discovery via broadcasting. Also, SCAMPI application platform [Kär+12] which will be explained in details.

2.5. Used Platforms

Turning now to explain the platforms that we used in order to implement 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.

2.5.1. 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 a persistent background process SCAMPI router, allowing Android phones to route data through the native TCP API. Having SCAMPI running on the Android phone allows it to carry messages from one endpoint to another without even having neither wired connection or a wireless one between the sender and receiver, simply the phone is used to carry the data from one network 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 provides reliability for 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, it is handy that SCAMPI allows the publish-subscribe messaging scheme on top of the DTN architecture.

2.5.2. Raspberry Pi

Raspberry Pi [Ras] 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 today's computers and mobile devices. Nevertheless, its graphical processing units equals that of today's latest mobile devices, it can stream high definition videos and run 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 designed especially to cope with the edge devices and the Raspberry Pi such as *Ubuntu MATE*, *Windows 10 IoT Core* and *Raspbian* which is officially supported by the foundation.

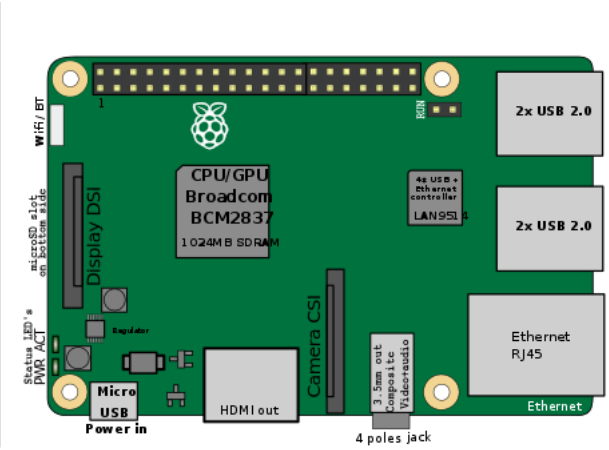

Figure 2.6.: Raspberry Pi 3 model B design. *Adopted from [EFA].*

In this thesis we use the Raspberry PI 3 model B 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 and hosting the operating system.
- Audio jack, HDMI port and 4 USB ports.
- Camera, display interfaces and 40 GPIO pins.

There are other models of the Raspberry Pi such as Pi 2 Model B, Pi Zero, and 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.

2.5.3. 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 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 creates new flows and extends current ones for their own use cases and make them available to the public, they also 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 empty 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 the configured time-stamp element. The Info tab on the right side, explains each node once it is selected, shows its documentation and how to use it.

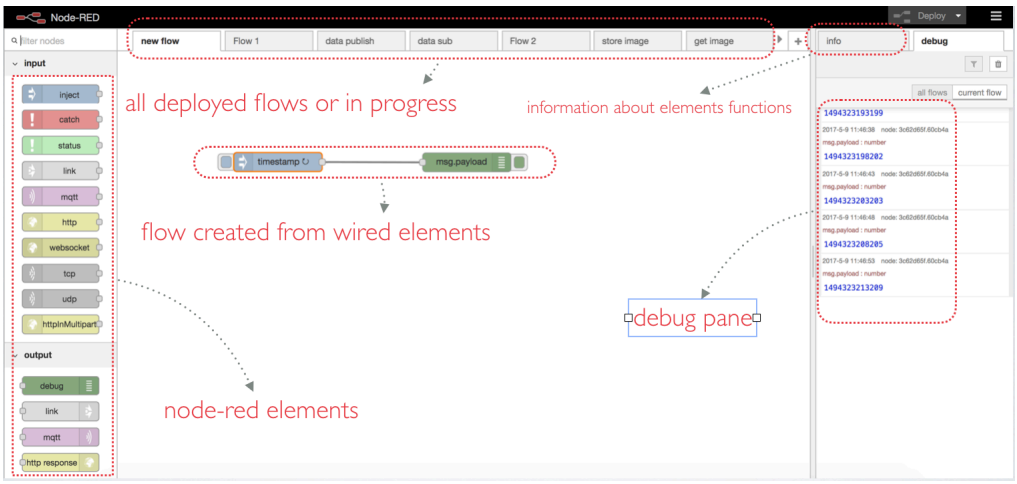

Figure 2.7.: Node-Red user interface to create and deploy flows for IoT applications.

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 along with their dependencies, meta-data to other nodes running node-RED instances and deploy the computation on them.

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

2.6. Summary

In this chapter we explained the fundamental concepts that we rely on in this work. We started by explaining Internet of Things and Pervasive computing, we also gave a brief about Fog computing which is yet another model of executing computation on

the network edge. Then, we elaborated networking concepts such as Information-Centric architectures which are content-centric in contrast to current network approaches which are host-centric and gave two example architectures based on this models. Later, we explained Delay-Tolerant networking architectures which are used to overcome unreliable connections and challenged networks. Eventually, we explained all the platforms and applications used to implement our framework including SCAMPI, node-RED and time-series databases in addition to explaining Raspberry Pi the low cost single board computer.

3. Theoretical Foundation

In this chapter we explain the framework model in theory, the key concepts behind it, challenges facing the design and their possible solutions. We first discuss the core element of this framework, then we go through the computational and data model.

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.

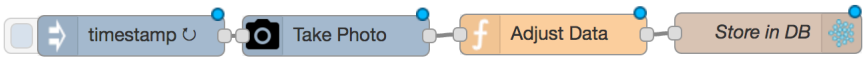

Figure 3.1.: A node-red flow is an example of the computational unit.

After having defined flows, the next step is to 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 device. As previously mentioned, since flows may interact, they need a way to communicate. Moreover, since devices might be disconnected, the communication mechanism must not require end-to-end paths and should handle sending the computations themselves from one device 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, is the dependencies and resources needed to carry out the execution. By dependencies we mean the custom libraries and perhaps scripts that are needed by the flow to ensure a successful run. Further, the resources can be either hardware in the form of computational capabilities by the host

devices or sensors and actuators needed by the flow for a specific use case. Dependencies and resources vary from one use case to another, thus need to be orchestrated across the heterogeneous devices by ensuring the delivery of relevant dependencies along with their respective computations and deploying the computation only to devices which fulfill the resource requirements.

Now assuming that we can send flows to the devices, 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 for example 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 device 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

$$f_1 : A \rightarrow B$$

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

$$f_2 : B \rightarrow C$$

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

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

Composability ensures that regardless of 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 Figure 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 Figure 3.3.

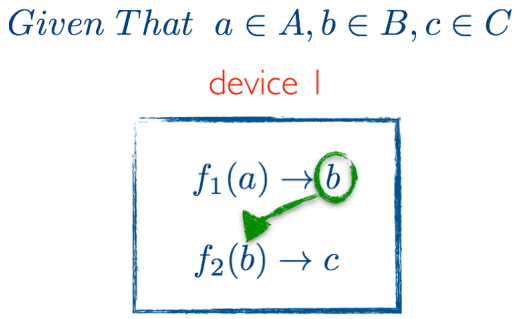

Figure 3.2.: A device containing two composable flows.

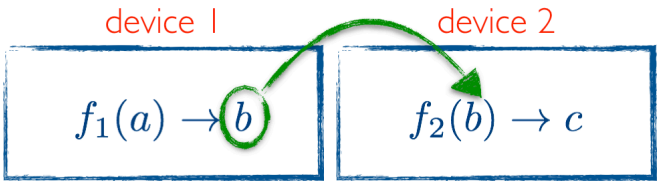

Figure 3.3.: Two separate devices having distributed composability.

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 hence allowing re-use and extensibility.

3.2. Computational Model

This section explains 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 Devices & 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 edges "devices" 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 device 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, devices should be able to communicate seamlessly even though they might be disconnected.

Turning to flows, distributing them across devices is a challenge because the distribution 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 devices in the system and call it $S(t)$ which is a function of time since devices can be removed or added to the system dynamically at any instant of time. Then comes the candidate set $S_C(t)$, which consists of devices that satisfy the required dependencies and resources of a certain computation. Take into account that devices inside any of these sets might not have an end-to-end path. Given $S_C(t)$, the flows could be either sent to a random set of devices or to a specific set. 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 devices residing in this location, we can send a flow to this exact set of devices with our desired computation. Continuing to explain flows approaches with figure 3.4, a flow could be distributed across devices of the candidate set $S_C(t)$ in multiple ways explained as follows: (i) flows are sent to all devices in $S_C(t)$, (ii) flows are sent to a set of n devices where $n > 1$, whether they are selected as case *C #2* in the figure or picked at random, (iii) choosing only one device to execute a specific flow as *C #1*. Consequently, 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 devices.

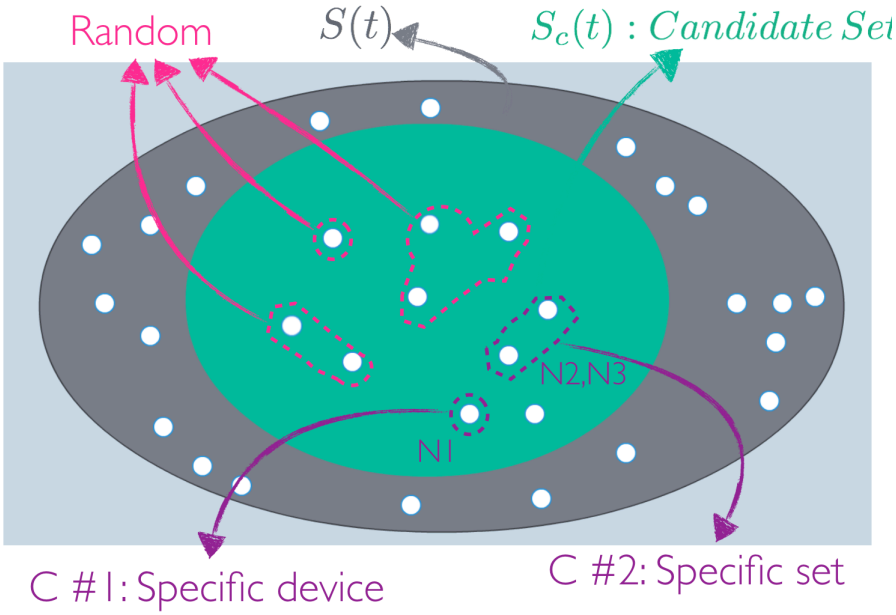

Figure 3.4.: Distributing flows approaches.

Another main challenge is to actually find the connected devices. Distributed and pervasive environments are dynamic, their components are not known to be alive or dead at compile-time. Thus the framework should be able to run service discovery at run-time in order to find the connected devices or it should be able to broadcast its message to all the other devices 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 device, however, this requires having static IP's for all the devices. Another solution is to use Dynamic Domain Name System (DDNS) [BR97] or multicast Domain Name System (mDNS) [CK13] to update the domain name whenever an IP address changes dynamically.

3.2.2. Software Dependencies

Dependencies are one of the main requirements of computation execution, missing 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 device, and the dynamic dependencies that are specific to each computation.

First, the static dependencies which are mainly the common libraries and software that most of the computations would require. That is why these dependencies are installed to each device 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 thus allowing interaction between devices.

Second, the dynamic dependencies that are specific to each computation such as additional scripts, configuration files or libraries. In this case, they cannot be installed at device 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 devices.

Static dependencies create ambiguity. Suppose that we want to upgrade the versions of current libraries installed on the devices. This introduces a versioning problem, imagine that there is a computation on the device 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 clarify the ambiguity and make version upgrading more controlled; 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 device to its dependencies and once a collision appears, the new computation renames its dependency and uses the renamed one.

3.2.3. Resources

Resources are physical dependencies such as sensors, actuators and devices capabilities that are necessary for computations to run. However, they might differ or not exist at all on each device. 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 device has a camera is not necessarily true. Since the resources are not standardized across all devices, each computation must provide meta-data expressing the resources it is going to require, also the device must realize its available resources. Then a device 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 devices 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 device 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, which can be high if it is free random access memory or CPU usage. The idea of queuing computations however develops a scheduling problem. Since we have a queue of computations inside each device, the queued flows will compete together to be deployed 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 devices 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 device such as cameras, temperature and gas sensors. Executing a computation missing this type of resource on a device should have a lower possibility of being queued, since its highly unlikely that this resource would be attached soon. However sensors and actuators 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 these resources, but this solution is not very efficient nor scalable, as it increases the manual work. It would be much easier if the device 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 snap a photo at the same time. This is problematic and can be looked at as a scheduling problem because only one flow can access the same resource at one time and whichever computation acquires a lock on the camera first will succeed while the other will wait or 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 device performance, its power and memory capabilities, it is heavily biased by the device 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 device which is already loaded.

Since we do not know what is the type or function of the computation being deployed, we must be careful of deploying computations which might result in abnormal CPU usage and memory leaks. Consequently, the framework should keep an eye on the running computations, monitor the CPU usage and memory for strange behavior specially that the framework knows exactly the computation requirements and metadata. It could also have CPU and memory usage quotas to control the hosting device resources.

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 devices 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 devices 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 a data centric messaging architecture in which senders also known as *publishers* do not send messages directly to receivers, but 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 han-

dles 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.

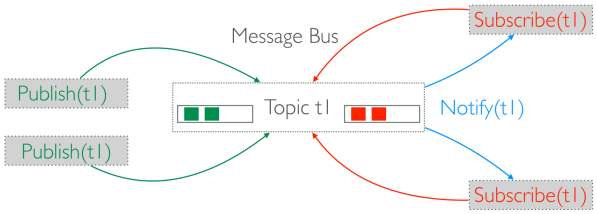

Figure 3.5.: Common message queues.

However having a pub-sub message queue with a centralized bus can not be used in our case. Since we do not assume that we have a connection to any centralized entity. In addition to not knowing machines addresses or host names at compile time thus connecting to a centralized message queue is not possible. Therefore, each party or device would have its own message bus and local storage that are then synchronized together whenever there is a connection.

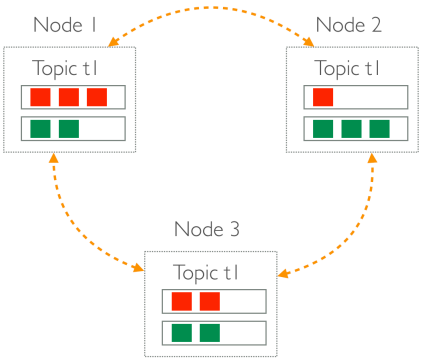

Figure 3.6.: De-centralized message queues.

Now addressing the mentioned challenges:

- First off, devices service discovery, messaging queues are able to discover and synchronize messages across all the devices 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 devices to or from the system.

- Having solved the problem of service discovery, devices 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 devices.
- For sending data or computation messages to a random set of n devices, a routing algorithm can be used to ensure that no more than n devices will receive the message. Further, if we exposed the meta-data of the computation, the routing algorithm can make sure that receiving devices will have a higher probability of being able to execute the flow.
- Sending data or computation messages to specific device or set of devices can be done by reserving a unique topic for each. Therefore, to target any device, 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 device 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 devices, either because there exists no end-to-end path or the devices are experiencing network connectivity issues. The pub-sub messaging system should be delay-tolerant and implements the store-carry-forward routing technique, this will allow the messaging system to store messages until connectivity is back or keep the message hopping from one device to another until 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 devices even without and end-to-end path. In addition, we can directly send and receive messages to the devices 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 devices through the messaging system, how the data travels from one

device 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, the framework should not attempt to restrict the data types that are communicated between the devices in order to make sure the framework is as dynamic as possible. In other words, the flow developer can have a weakly typed input and output data, which will result in more possibilities and flexibility when designing the use case. Yet, he/she might prefer to enforce strongly typed input and output data to make the flow behavior more explicit. 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 later. 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 able to 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 have inter-relationships between devices and request input data for a computation from the output of another.
- Sending data to be processed by a computational flow on any device and then obtain the outcome. For instance, sending an image to a device containing computational flow with image recognition algorithm, then the image gets processed and the results are sent back the original sender.
- A device can be used for monitoring in which it subscribes to all outputs of a certain computation running on several devices.

As mentioned in Section 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 device 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 device can be linked to the input of another flow either on the same device 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 device 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.

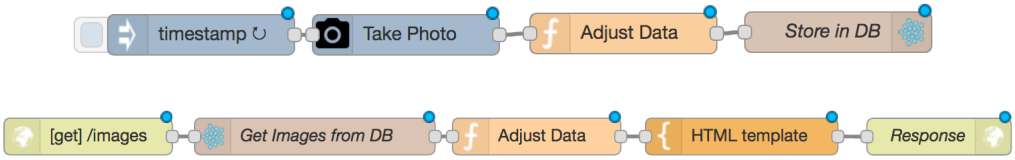

Figure 3.7.: Two separate computational flows describing the local IO composability through a database.

- Another way is to use publish-subscribe messaging pattern to communicate through different devices. The device which generates the data publishes its resulting data to a generally unique topic, therefore any device 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 device, the data could be received via subscribing to the same topic.

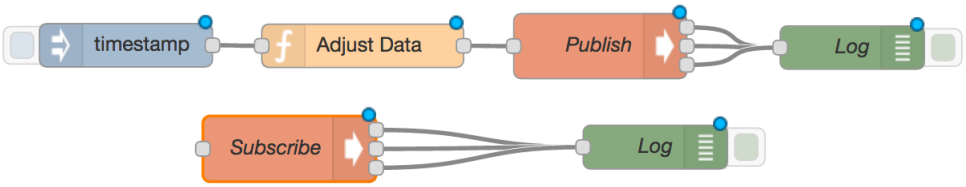

Figure 3.8.: Two composable flows exchanging data via the messaging system.

- Streaming data is also possible, one device can have a computation serving as a streaming server while other devices 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

To conclude this chapter, we explained the key concepts and foundations behind this framework including the definition of a flow and how they can be composed. We also described the computational model as an abstraction the framework design, and illustrated the challenges that face 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 IO data of different devices could be connected together.

4. System Design

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 an end-to-end path.

4.1. Use Cases

This section provides real life scenarios targeted by our proposed framework. Requirements are then elicited from the use cases and then the framework implementation and design are evaluated against these requirements. Keep in mind that the framework idea is not just to implement these use cases, but to provide the ability to distribute, compose and execute various use cases for context-aware pervasive computing. The use cases are mainly targeted to help human beings, increasing their life quality and preserve the environment.

4.1.1. Smart Cities

One of the most researched areas in the field of IoT is making our cities more advanced, connected and helpful to human beings and the environment. Researchers and professionals have had many ideas to make use of the context-aware sensor networks that can communicate and act independently whenever the situation needs intervention. We take some examples of the smart cities applications and show how they can be implemented using our software framework.

4.1.1.1. Smart lighting

Smart lighting is a use case for automatic control of outdoor lamp posts hence optimizing costs for the governments and enterprises in addition to helping the environment by saving energy [Zan+14]. The outdoor lighting can be automatically started or closed according to certain circumstances. For example, by lighting up when motion

is detected around it and turning off when there is no further motion, or by detecting natural light and weather circumstances. Moreover, the lamp posts can be monitored in order to see when a lamp needs replacement. Further, lamp posts can detect what is the best lighting percentage in a certain situation according to a machine learning algorithm based on the history of decisions taken and feedback provided by users.

In order to implement this use case in our proposed software framework, we must be able to distribute a computational flow to all lamp posts using a messaging system. Also, the flow should be able to access sensors and be capable of adjusting the light according to the input gathered from several sources including motion, light and humidity sensors. The decisions taken should be stored into a database so that the flow can access previous decisions in order to asses the current situation. In addition, a flow with an endpoint that allows users to send feedback on the lighting system can be locally composed to enhance the learning algorithm. Having sent the computational flow to all the lamp posts, the framework should ensure that it does not get deployed on lamp posts without the required resources, sensors and actuators.

4.1.1.2. Smart Parking

Automatically detecting empty parking spots in crowded streets is the main idea behind smart parking [LYG08]. Since the number of vehicles on the streets are in tremendous increase [For], the urge for finding a parking spot in city centers and crowded places has also increased. Having a smart parking mechanism helps in wasting less time, decreasing energy consumption and maintaining a clean environment by reducing emissions. Different information sources can be used to detect empty parking spots, for instance, image and video footage of street cameras, crowd sourcing of information and APIs for parking services. Implementing this use case has almost similar requirements to smart lighting, however, it adds having a camera integrated with smart devices which might produce the need for streaming the footage in real-time to other devices.

4.1.2. Mining Applications

In underground mines it is necessary to always monitor gas and temperature levels in order to prevent miners suffocation [Osu13]. At the same time, it is very hard to maintain a connection between sensors in the inside and monitoring systems outside [Gin+10]. Therefore, the devices incorporated with sensors inside the fields are expected to be pervasive and warn miners from unexpected increase of gas levels on their own. In addition to delivering the data to the outside world, the system must be able to gather data about the situation inside. This requires miners to act as middlemen who store delay-tolerant data into their mobile devices and carry it forward to

the outside world. Of course the miners cannot do this process manually, therefore, there should be a messaging system on their mobile devices that carry the data and handle the synchronization with other device. Also, carrying the data in and out is not the only issue, importing and visualizing data in the monitoring system should be also done automatically.

The mining use case is a delay-tolerant application because there is a big chance there are no wiring going in and out of the mine where new tunnels are constructed thus a delay-tolerant messaging system should be used. But also, it is an information-centric application because data should be sent from devices in the mine to be imported in the monitoring system without knowing their host names. Having smart devices installed in the mine means it is very hard to target them as endpoints without a valid connection, even if there was one, knowing the host names of these devices might still be an issue. Same applies to the monitoring system outside, data cannot be sent to a specific host name, rather sent to whoever is interested in these data.

4.1.3. Privacy and Security

The surveillance systems used at the moment uses closed-circuit television cameras to send tapes, images and footage of people using public transportation to storage systems [Ash17] . The police could acquire these tapes in case of incidents to observe, monitor and apply facial recognition algorithms in order to detect faces of wanted criminals to prevent crimes and anti-social behavior [Lon]. Despite this being of significant importance to the national security, this puts everyone's privacy in jeopardy. Therefore, we thought of replacing this model by a pervasive one, where the facial detection algorithm and faces of wanted criminals are pushed to the smart devices in public transportation means and whenever a match is found the national security is notified. This could also be triggered only whenever there is an incident using computer vision, which helps greatly in increasing privacy, decreasing latency between the footage and detection in addition to minimizing network bandwidth since the streaming footage will not be uploaded to the cloud unless there is a match. Similar approaches were introduced in [FS08] and [WR10] where the footage is analyzed on spot to protect privacy. Nevertheless, this is not trivial, facial recognition algorithms are complex and depends on other libraries and detection models. Therefore, the computation which includes the recognition algorithm must be able to carry dependencies otherwise the computation will not run. Moreover, the national security should be able to update the list of criminal faces thus messages sent to the smart devices should allow also attachments and dependencies as well.

4.2. Requirements

What follows is a listing of the requirements extracted from the real life use cases mentioned in the previous section. We then evaluate the design and implementation of the framework according to the derived requirements which are classified in the following points:

1. *Service discovery*: since we do not know the host names of the smart devices nor their addresses, the framework should have service discovery mechanism encapsulated in the messaging system as explained in Section 3.2.4 to discover peers and allow any interaction between smart devices.

2. *Send and deploy computations*: whatever the use case may be, one of the main requirements of this work, is to be able to send computations to smart devices. This includes sending to one, a set or all smart devices connected together in a network. However, to deploy a computation, the receiving side must have the required hardware resources, sensors and actuators to be qualified for deploying the computation.

3. *Computation dependencies*: complex computations which cannot be executed using the basic operating system and common libraries installed on the smart devices, should carry their own custom dependencies in order to guarantee a successful run on the receiving smart devices.

4. *Disconnected devices*: since the framework is also designed to be used in challenged networks, isolated devices in separate networks should also send and receive data or computations using the delay-tolerant architecture.

5. *Communication*: smart devices should be able to communicate, send and receive data in an information-centric and a publish-subscribe manner.

6. *Global identifier*: each smart device must have its own unique global identifier which can be used to send data and computations to this device only.

7. *Composability*: the framework should allow composition of multiple computations either locally through a database or globally by allowing data exchange via the publish-subscribe messaging system.

8. *Pervasiveness*: each computation should be able to act on its own, trigger actuators, persist data and access resources such as cameras and sensors if required by the use case.

4.3. Framework Stack

This section explains the software framework stack 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 ability 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 software stack on each device looks 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 as a mediator between node-RED and SCAMPI to handle the deployment logic and harmonize the communication between them. 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 middleware nor node-RED since the phone will be merely used to transport data from one device to another having no end-to-end path. Below we explain how each layer of the stack works.

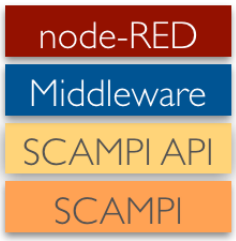

Figure 4.1.: The framework architecture stack.

4.3.1. SCAMPI

As mentioned in Section 2.5.1 SCAMPI is an information-centric, publish-subscribe and delay-tolerant messaging system. In this framework we use SCAMPI to send and 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 so that we do not have to connect to a centralized broker which allows SCAMPI to run as a standalone service without any dependencies to other hosts. Another main reason is to reach devices which do not have direct connectivity

to the publishing device 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 in challenged environments. In figure 4.2, we show how SCAMPI uses mobile phones to connect devices 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 and therefore they can 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 cannot be reached using wired or wireless connections.

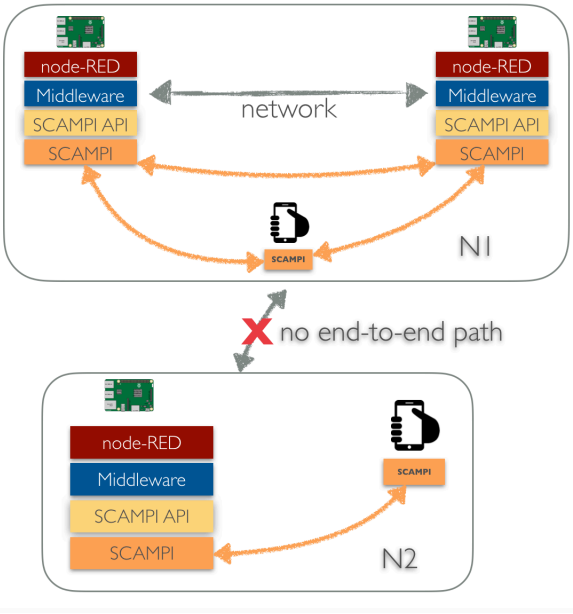

Figure 4.2.: SCAMPI synchronization even without an end-to-end path.

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 devices at will without any additional configuration. We can also use the general identifiers of the devices 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 device and we are good to go.

The SCAMPI Java API allows the use of services provided by a local SCAMPI instance granting us to publish or subscribe for any topic from the Java environment. The API

also allows the client to get information about 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, it assists in assigning attributes to the message whether strings, integers or even binaries. Also, one 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 possible to deploy flows without human interaction. Flows can be also configured to access certain tables or collections in a local database instance and this configuration can be serialized 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 either uses the REST API that the middleware provides to publish and subscribe to data topics or the same database configuration in both flows to be able to communicate data through the database. This allows node-RED to send or receive data and allow composability both locally and globally.

Node-RED is rich with predefined nodes that can be used to run flows on time intervals, connect to emails, twitter accounts or even access a GPIO pins 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. Maestro

Maestro is this framework's middleware and the main contribution of 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 that allows other entities to fetch and post data to Maestro via an API. 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 provides a REST API which allows any other script from any other language (including node-RED) to use the publish-subscribe feature of SCAMPI.
- Maestro subscribes by default to the computation topic in order to send and receive computations.

- Maestro subscribes to the unique global identifier of SCAMPI at the framework initialization to act as an identifier for the whole stack.
- It analyzes flows by checking the meta-data attached to the message thus if Maestro finds out that a device 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 more than once.
- 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 exchanged on this topic.

4.4. Framework Architecture

What follows is an explanation of the framework design and architecture and how it can be used to achieve the goals of this thesis. We describe a step by step guide to design and distribute a flow that portrays a pervasive use case.

4.4.1. Physical Components

Before we get into the software components and how we can use the framework stack to implement a pervasive use case, we first describe the physical components that are used by our framework. There are three main devices used in this architecture:

- *Raspberry Pi*: we use the Pi as a low-cost single board computer that contains GPIO pins which can be connected to arbitrary sensors and actuators. However, the Pi can be replaced by any single board computer that runs an operating system supporting the JVM and can host node-RED as an execution environment. An example Raspberry Pi in this framework has our stack installed in addition to a camera, an RGB LED, a temperature and a motion sensor as shown in Figure 4.3, we stick with this example because it will be used in the framework evaluation. All the Raspberry Pis in our design have the sensors connected to the same GPIO pins, for instance, all the temperature sensors are connected to GPIO pin 11 on each Raspberry Pi. This is important because when we send the computation script for sensing temperatures to all the Raspberry Pis, the script will be accessing the same pin to get a hold of the sensor data. A more advanced solution can be used by creating a name resolution service in Maestro

that resolves a sensor name to a specific pin, but this is outside the scope of this thesis.

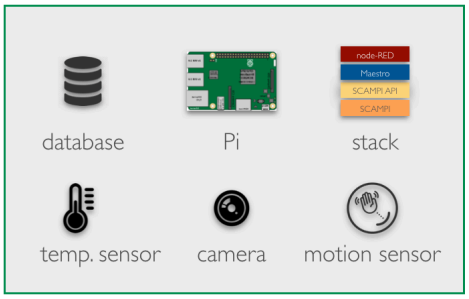

Figure 4.3.: A Raspberry Pi in our example connected to sensors and running our stack.

Now recalling the issue described in Section 3.2.3.1, which explained that multiple flows might attempt to access a resource at the same time (temperature sensor for example). If we sent another computation which also tries to access the same resource, it will fail with a high probability. Since a resource can only be accessed by one computation at a time. Therefore, we have only flow that access the resource and stores data into a database and then other computations can query the data from the database.

- *Intel NUC*: we use the NUC as a computer with high performance capabilities compared to the Raspberry Pi. NUC devices in the architecture also run our stack and are used to prove that we can delegate the computations requiring high performance from the Raspberry Pis to high performing devices. It is not connected to any sensors or actuators. The NUC can be replaced as well with any high computing device with an operating system supporting the JVM and capable of hosting node-RED.

- *Android Phone:* Smartphones in general are typical devices for accessing services and can serve as data carriers. Thus to confirm that we can deliver computations and data to challenged networks, we used an Android phone running SCAMPI so that we can carry data from one network to another. The phone runs only SCAMPI and not whole stack, however, node-RED supports Android devices while Maestro also only relies on the JVM therefore we can deploy the whole stack if we have a use case for executing computations on Android phones.

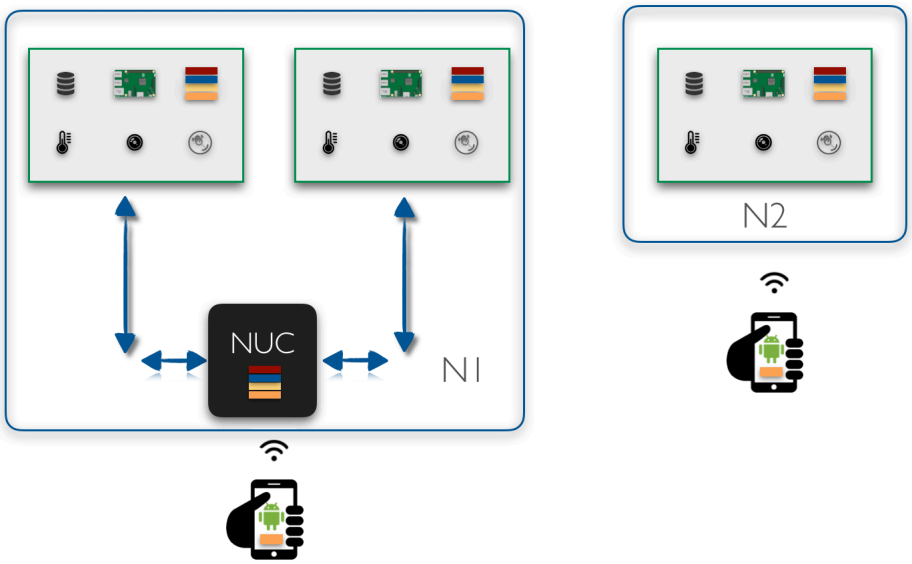

Figure 4.4.: Physical components of the architecture.

Figure 4.4 shows one possibility to connect the devices in our architecture where we have two Raspberry Pi's connected together with an Intel NUC in a network, and another Raspberry Pi in a separate one that communicates with the other devices using the Android device switches between both networks.

4.4.2. Designing a Flow

The first step to distribute a pervasive use case and send the computational flow to all devices in a network is to design the computation which implements the use case. Since we use node-RED to design and execute flows, therefore the first step is to use node-RED to create a flow by wiring and connecting nodes. Each existing node in node-RED has a different functionality, also additional nodes developed by other contributers can be installed into the local instance. If a specific functionality can not be found in the existing nodes or the ones available from the contributors, the functionality can be implemented using any programming language and then the `exec` node available in node-RED can be used to access the scripts in node-RED directory and run it. Moreover, the existing `function` node can be used to implement the functionality using JavaScript.

Triggering flows is very important as explained in Section 3.1, there are multiple options to trigger flows. The first one is using the `inject` node which allows triggering the flows in multiple ways such as every time interval, once a flow is deployed, between time intervals or at specific time. The second way to trigger computations is via receiving a request using a `http request` node on a specific endpoint that is wired to

the flow just as Maestro does to deliver data to node-RED flows. There are also other ways that could be devised according to the desired use case.

Accessing sensors and actuators can be done in several ways using node-RED. The first way is to use the existing GPIO nodes which can access the Raspberry Pi pins to read and give digital signals to the sensors and actuators. However, we found out that the existing GPIO nodes are not stable, therefore, we used the second option to access sensors and actuators using python scripts that are executed using the `exec` nodes. The python scripts are then transferred with the computation as dependencies.

There are also different database nodes that can be installed on the local node-RED instance, for example, *InfluxDB* the time-series database and *MongoDB*. Each supports writing and querying data from the local database instance. When a flow is configured to have a database configuration, the user can specify whether it should live across all node-RED flows or can only be used by a specific flow. Yet when we chose to have the database configuration across all node-RED flows, the configuration was not serialized along with each flow. Thus, it is advised to configure the flows in a way that each has its own configuration so that it can be serialized along with the flow and can reach other devices in the network.

4.4.3. Sending the Flow to Heterogeneous Devices

Having designed a flow representing a use case, we can send the flow along with its dependencies to devices in the network while accounting for their available sensors, actuators and hardware resources. To achieve this, we developed a node-RED flow with a user interface that can be used by the framework to send other flows to the devices running our stack. Once we have designed a flow and have the flow identifier, we can adjust the computation power and the required free random access memory for each specific flow through the HTML page demonstrated in Figure 4.5. We can also select the required sensors and actuators to cater for the heterogeneous devices and their sensors availability. Further, the scripts used by the `exec` nodes inside the flow can be selected in the HTML page as dependencies. Once we set everything regarding the flow meta-data and we click on the `push` button, a request is sent to Maestro in order to handle sending the computations and meta-data to SCAMPI and other devices.

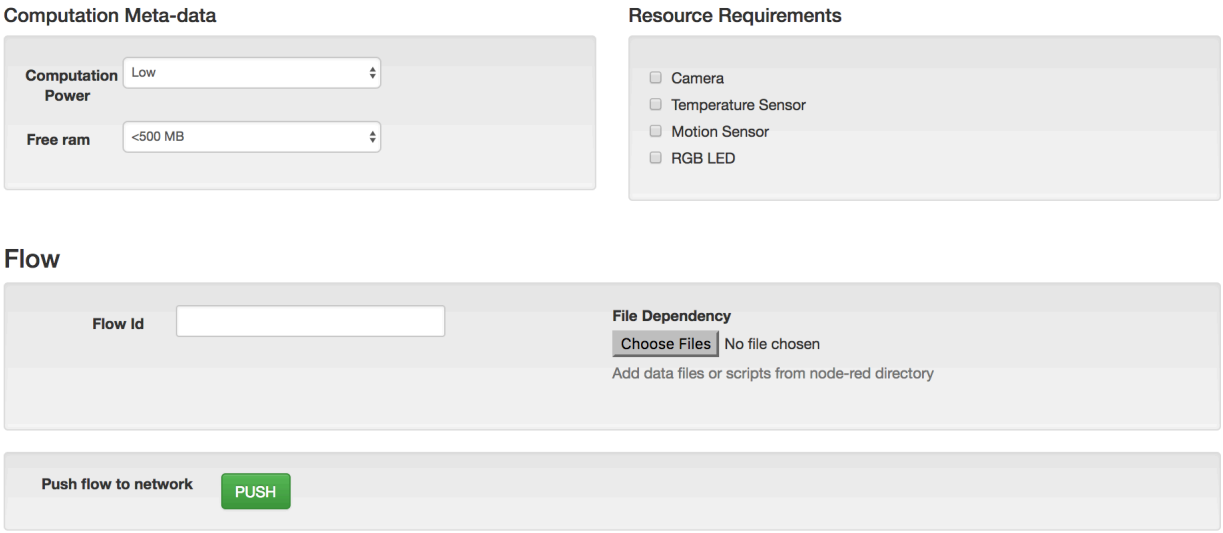

Figure 4.5.: The HTML page used to publish computations.

When Maestro receives the computational flow together with the meta-data and dependencies names from node-RED. It creates a new SCAMPIMessage with a global unique identifier, then it attaches the flow's dependencies as binary data in the message. Finally, it publishes the message to a specific topic to SCAMPI and returns a success response to node-RED. After SCAMPI receives the message from Maestro, it publishes the message to all discovered peers. It also keeps a copy in the cache in case it discovers new devices.

4.4.4. Receiving the Flow

Since Maestro subscribes by default to the topic of exchanging computational flows, whenever SCAMPI on the receiving devices gets a new message it is forwarded to Maestro. Afterwards, Maestro checks the hardware resources, sensors and actuators of the incoming computation and compares it with the machine specification. Then Maestro puts the dependencies carried by the SCAMPI message to node-RED directory before deploying the computation to node-RED.

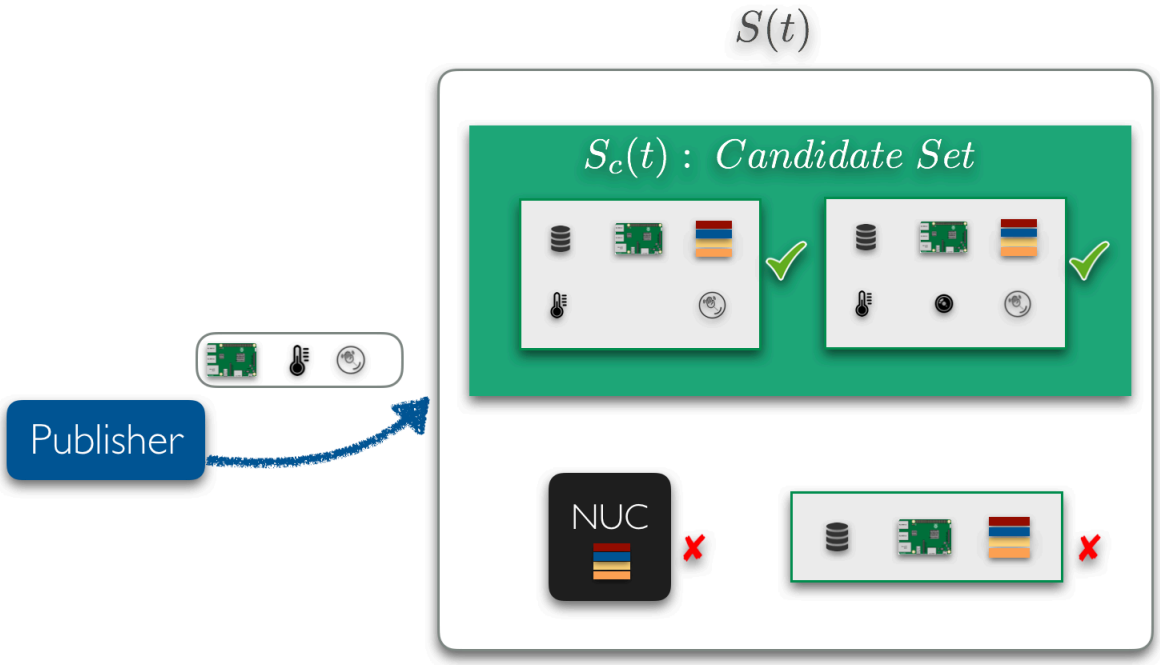

Figure 4.6.: Deploying computations to heterogeneous devices.

As described in Section 3.2.1, $S(t)$ is a set that contains all devices in a system. From this set, we can have a subset $S_c(t)$ which is the candidate set that satisfies the requirements for a certain computation. Applying this to our design and as shown in Figure 4.6, if we publish a computation that needs low computing power, motion and temperature sensors to a network of devices. The flow should only be deployed by the nodes that fulfill the requirements only. In the figure, neither the Intel NUC nor the Raspberry Pi without the required sensors were able to deploy the computation. Further, in the current Maestro implementation, if it receives a computation and the requirements are not met, then the computations are discarded completely. There is a possibility to queue the messages which failed the hardware requirements, such as low random access memory or high CPU usage of the current device, in order to be deployed again when the resources are freed. As explained in Section 3.2.3, this introduces a scheduling problem, after the resources are freed, queued computations will compete for the free resources.

4.5. Sending and Receiving Data

We explain how the stack can send and receive data using our framework architecture in the following points:

- As discussed in Section 3.3.3, the flows should send and receive data using

the messaging system described in Section 3.2.4 in order to achieve global composability. By using our framework, we can communicate data on a any topic except the computation topic. This is done by using a `http request` node that uses Maestro REST API to publish and subscribe to data on a specific topic.

- Local composability, discussed in 3.3.3, can also be achieved by sending two flows with the same database configuration which get serialized along with the computational flows to other devices. Each one of them can read or write to the same database instance. Moreover, since collections in time-series databases do not need a schema we can insert data of any form immediately.

- On the sender side, when Maestro receives a new publish request, it creates a new SCAMPIMessage with a unique global identifier and publishes it to SCAMPI which handles delivering the message to other devices. Alternatively, for each flow that subscribes to a specific data topic to Maestro, it adds an entry in the topic-endpoint mapping that maps between a topic and an endpoint identifying the flow.

  On the receiving side, when Maestro gets a message from SCAMPI because it had subscribed for a specific topic, Maestro forwards the data to all endpoints on node-RED that had previously subscribed to this topic, this enables the flexibility of having multiple flows subscribing to the same data.

- When publishing a data message, the publisher can attach a parameter that indicates whether the response of this message should be sent to the publisher only, or sent to specific topic that might have multiple subscribers. This helps publishers in sending data to be processed on other devices and then control who should receive the processing results.

- Since Maestro subscribes to the global identifier of SCAMPI and thus the framework, it is possible to publish flows or data to specific nodes or set of nodes as discussed in 3.2.1 by publishing the message once for each device. However if the set is too big, this might flood the network with the same message published to different topics for various devices. More optimized solutions could be achieved by exposing the information to the routing algorithm of SCAMPI thus being able to publish the message only once as explained in 3.2.3.

4.5.1. Architecture Summary

As a recap, the following Figure 4.7 summarizes how the framework works and shows how flows are distributed and data is exchanged between the stack components.

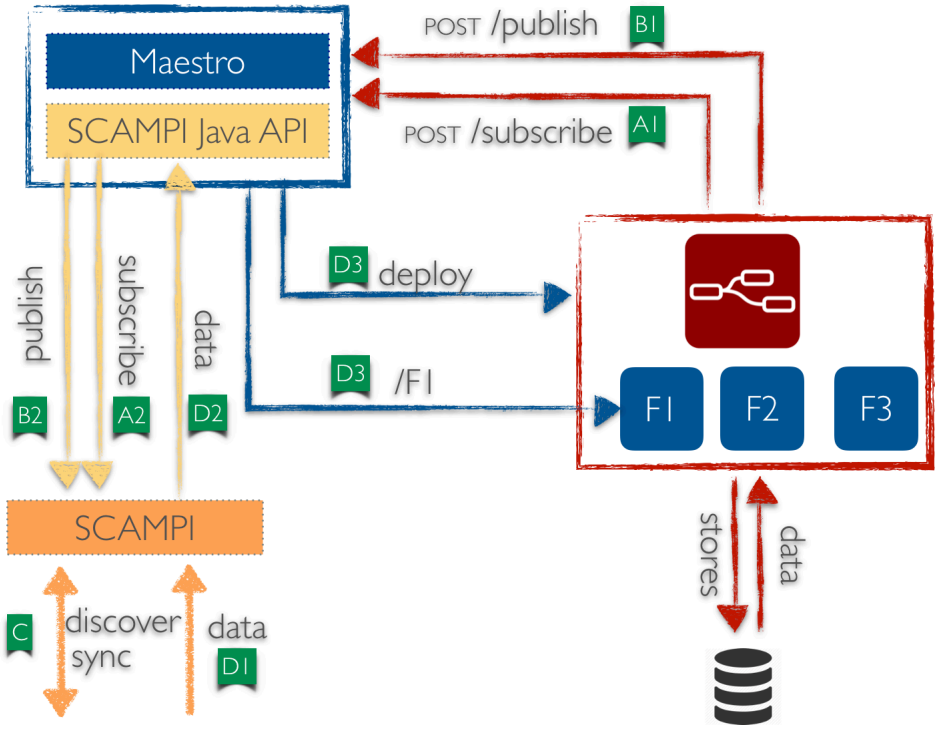

Figure 4.7.: Software framework architecture design.

(A) Flows are developed using node-RED UI, they can include publishing and subscribing REST calls to Maestro. If a flow subscribes to a certain topic, Maestro 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, Maestro extends the mapping to include it, hence, once a message is received it gets forwarded to all subscribed flow endpoints.

(B) When Maestro 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 device 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 a message it is forwarded to Maestro, which then verifies the topic. If it was a computation message then Maestro checks meta-data, resources, dependencies and then either deploy the computation to node-RED or discard it. Otherwise, if the message was not a computation, Maestro forwards it to the subscribing flows from the topic-endpoint mapping.

4.6. Summary

In this chapter we introduced real life use cases for our software framework that was utilized to elicit the requirements used to evaluate the implementation and design of this framework. Afterwards, we described the stack to implement this framework starting by SCAMPI the delay-tolerant information-centric messaging system, going through node-RED the platform used to implement, export and execute custom computations serving different use cases, finally we explained Maestro which is the middleware that orchestrates the communication between SCAMPI and node-RED. Then we explained how the whole framework is expected to operate fulfilling the thesis goals.

5. Implementation

Building on the framework architecture explained in Chapter 4, we describe Maestro the 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 experimental use cases to validate elicited requirements in the evaluation chapter.

5.1. Maestro

Maestro is implemented as a Java project and web application that act as a middleman between node-RED and SCMAPI. It accepts publish and subscribe requests from node-RED flows through REST API and uses SCAMPI Java API to propagate these requests to SCAMPI server. Further, whenever it receives a message that it had subscribed to from SCAMPI server, it forwards the message to node-RED flows who issued the subscription requests. It 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 Figure 5.1, `SCAMPIApi` implements two classes via the `APP_LIB` for the SCAMPI server which includes functions like `OnDisconnected()`, `OnConnectFailed()`, `OnStopped()` and `OnConnected()`. 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 which is read from a file on the device. Also, `SCAMPIApi.java` subscribes to the global identifier of SCAMPI on service initialization which is used as a global identifier for the whole framework.

  The class `MiddlewareApi.java` contains REST API for Maestro's web application that runs on port `8080`. It has two requests `POST /publish` and `POST /subscribe` that reference the `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 4.5. 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*.

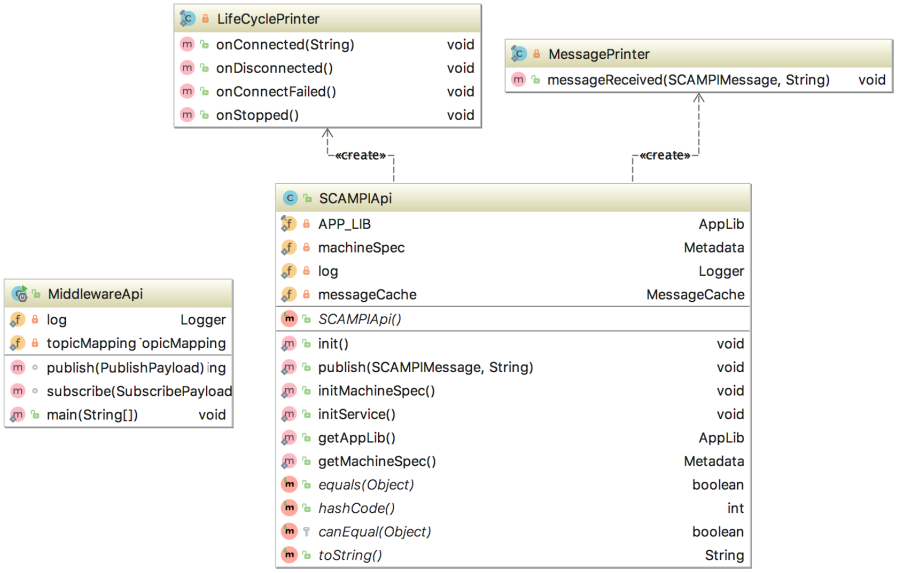

Figure 5.1.: Class diagram for the api package.

- The second package is called `com.middleware.constants` and has one class `Constants.java` which contains all the constant fields used in Maestro 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 Maestro. There are two singleton classes namely `ToppicMapping.java` and `MessageCache.java`, the first class is a cache that maps between the subscribed topics and the node-RED endpoints who issued a subscription request to them. The second is a cache that makes sure a received message is not handled more than once. A singleton class 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 or re-initialize the existing ones. That's 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 messages respectively. The `MessageHandler.java` is invoked 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)` that 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 invoked 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 device only, if this is the case, it creates a mapping between the global identifier of the device and the endpoint that sent the publish request then send it to SCAMPI server.

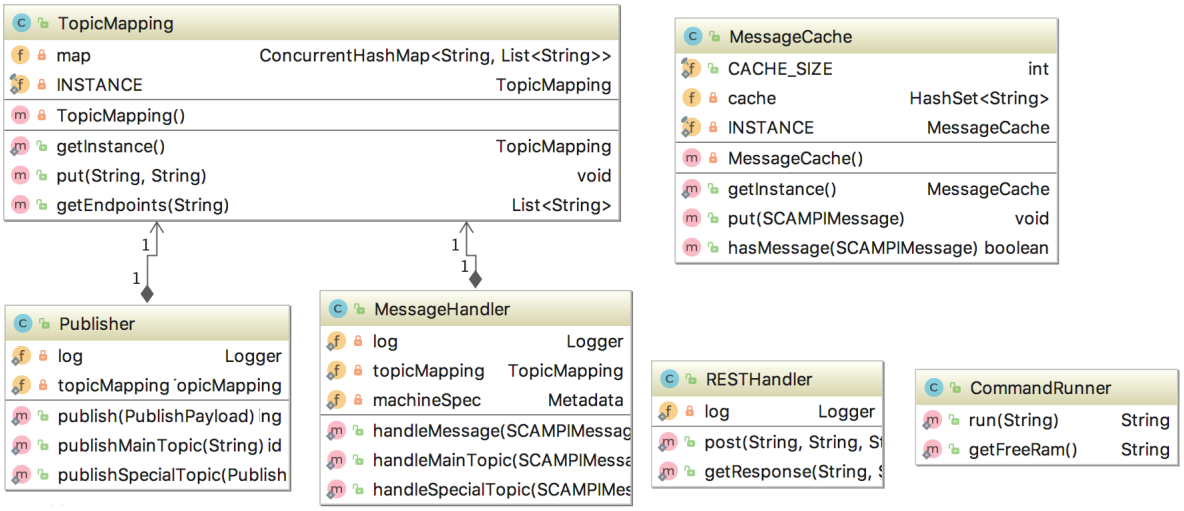

Figure 5.2.: Class diagram for the domain package.

- 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. Remark up on 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 displayed in Figure 5.3.

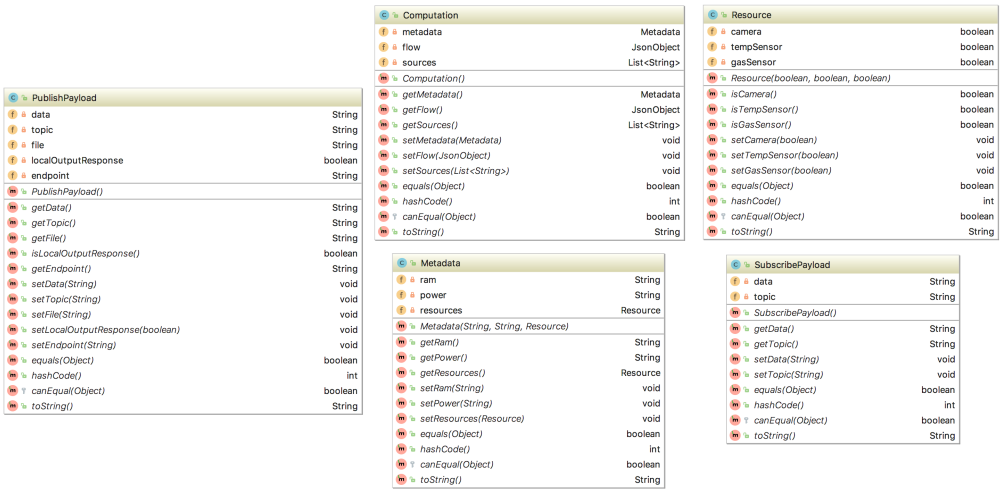

Figure 5.3.: Class diagram for the model package.

Maestro uses *Maven* as a software project dependency management which handles Maestro's 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. Maestro can be compiled and packaged to a jar using the 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 and dependencies to create a web application via *Spring Boot*.

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

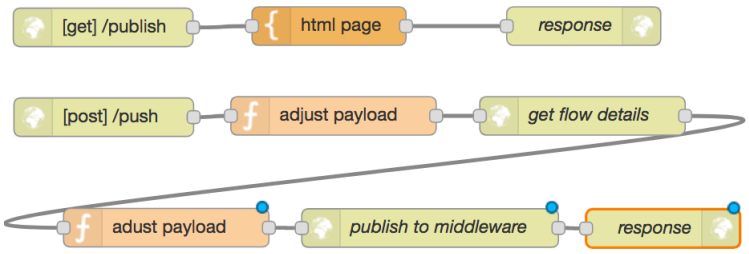

Figure 5.4.: A flow that publishes computations to Maestro and thus to SCAMPI.

In order to deliver flows which expresses computations for all devices 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 4.5. 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, attach dependencies and eventually click on the button *push* which calls another endpoint on the same flow `localhost:1880/push` with the form data. The

endpoint receives the data, fetches the flow details and publish it to Maestro which handles the rest.

5.2.2. Temperature Sensor Alert

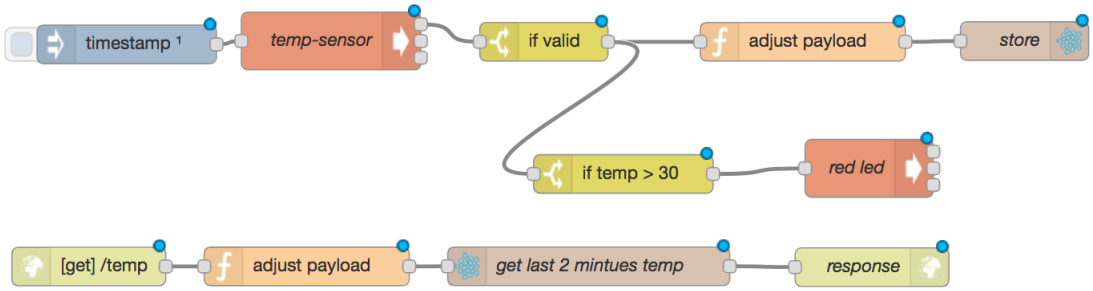

Figure 5.5.: A flow that reads temperature and stores it, also start a red LED if temperature is above 30 degree Celsius.

This flow was developed to make sure that the framework works against the typical IoT usage of sensors, actuators and time-series databases. As presented in Figure 5.5, the flow runs once it is deployed to any device. Take into account that, it would not have been deployed by Maestro 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 devices. 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 up 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.

5.2.3. Detect Movement and Store Image Responses

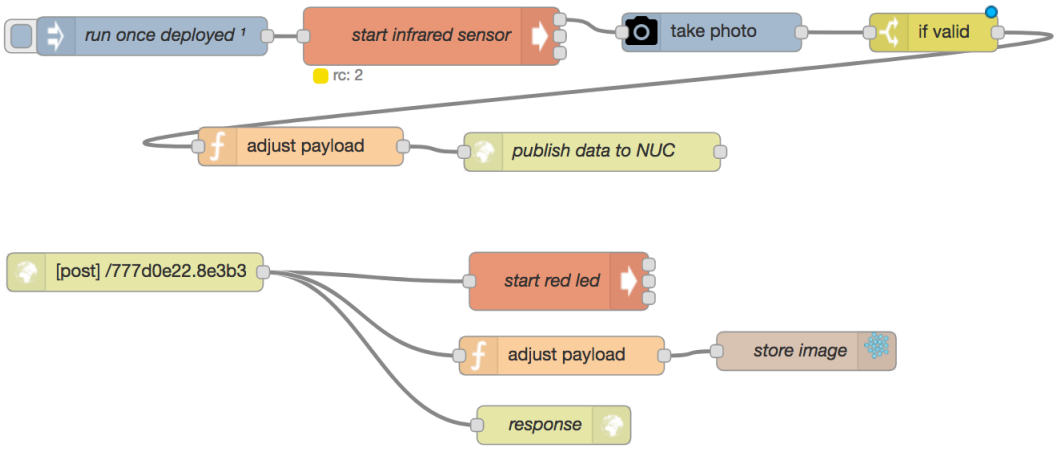

Figure 5.6.: A flow that detects motion, take an image, publish message and store response.

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 device, therefore, Maestro creates a mapping between the device global identifier and the endpoint before publishing. Also, as demonstrated in Figure 5.6, the flow awaits messages on its endpoint and executes a script which starts a red LED, it also stores the recognized image in a database along with its accompanying data.

5.2.4. Show Recognized images

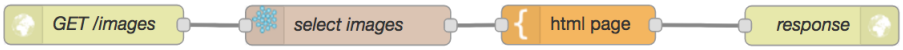

Figure 5.7.: A flow that creates an endpoint for stored database images.

Since composability 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. The flow is invoked once it is deployed by the receiving device.

5.2.5. TensorFlow Water Bottle Recognition

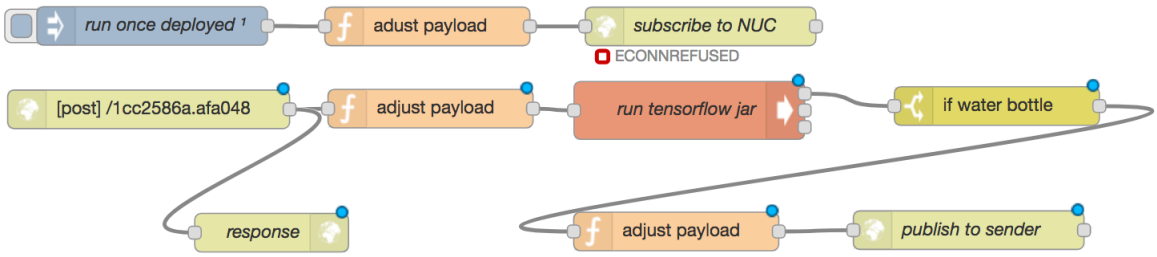

Figure 5.8.: A flow that uses tensorflow to recognize a water bottle.

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 shown in Figure 5.8, the flow subscribes to the topic *NUC* once it is deployed. Then, when Maestro instance that runs on the same machine receives a message on the topic *NUC* carrying an image, Maestro sends it to the corresponding endpoint from the topic-endpoint mapping, it also puts the dependencies (an image in this case) in node-RED directory. Thereafter, the flow runs tensorflow jar that reads the image from node-RED directory. If it results in a water bottle as a best match, the flow responds to the sender with the result and a confidence percentage.

5.3. Starting the framework

The framework stack runs in the following order; SCAMPI, Maestro 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 Maestro 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 in Chapter 4. Specifically, Maestro's implementation and extending the Java API of SCAMPI. 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 in the right order.

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 Section 4.4. It also evaluates Maestro's implementation which was explained in Chapter 5 according to the requirements mentioned in Section 4.2. Note that, the delays in the experiments are not meant for performance evaluation, but rather to prove and validate that the intended features of the framework work as expected.

To satisfy all the requirements and specifications for the software framework, we divided the evaluation into several sections. Each section validates certain specifications via implementing a minimal 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 have *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. Typical IoT Usage

First, we wanted to evaluate that the software framework works with the typical IoT use cases using high rate data sensors and storing them in time-series database. The flow explained in Section 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. Each Pi had a red LED and a temperature sensor attached. 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 the data is stored into a local database.

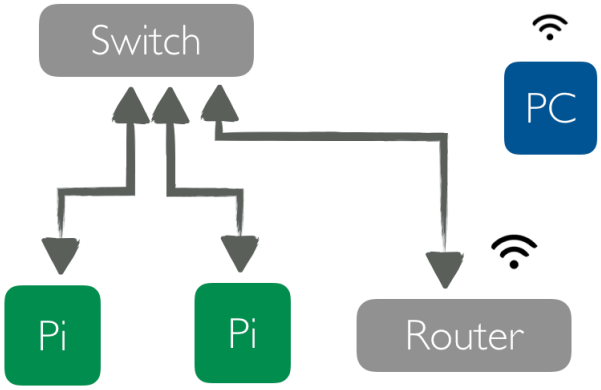

Figure 6.1.: Testbed setup for temperature sensing flow.

The experiment was tested 8 times and each time we measured the delay between publishing the flow from the PC t_{PC} until it was 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 each device's instance t_{db}.

	$t_{Pi1} - t_{PC}$	$t_{db} - t_{Pi1}$	$t_{Pi2} - t_{PC}$	$t_{db2} - t_{Pi2}$
$\bar{t}$	2.870	1.723	2.437	1.717
σt	0.769	0.0376	1.003	0.013

Table 6.2.: Mean and standard deviation of temperature flow delays.

Given these delays, the average time the temperature flow takes to reach the Raspberry Pi and gets deployed is 2.653 seconds which the mean of the two Raspberry Pi delays. The average time from the flow deployment until the first temperature measurement is inserted 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 the temperature flow is deployed until the first database insertion is done. Given the testbed setup and that the flow does not have large size dependencies, the time should have been less although we think that the Raspberry Pi low performance capabilities increases the delays and this can be seen in the next experiments. In this experiment we have satisfied some of the requirements mentioned in Section 4.2: i) service discovery, ii) send and deploy computations, iii) computation dependencies, iv) pervasiveness.

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 which is a flow itself 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 water bottle with a certain confidence, therefore, the Pi signals a red LED to light up and stores the result in a database.

The flow implementation used to create this use case can be found in Sections 5.2.5 and 5.2.3. As part of this experiment when a flow requiring high amount of perfor-

mance 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 framework evaluation:

1. Service discovery, through discovering all devices running SCAMPI.
2. Send and deploy flows, by sending them to all devices connected to a network running our framework stack and 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. Global Identifier, by sending data to a specific device using its global identifier.
4. Computation Dependencies, through carrying tensorflow image recognition and the motion flow dependencies in order to guarantee a successful run at the receiving device.
5. Communication, by having publish-subscribe messages between the NUC and the Raspberry Pis.
6. Pervasiveness, through lighting a red LED once an image bottle is successfully recognized and a message is sent back to the Raspberry Pi.

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.

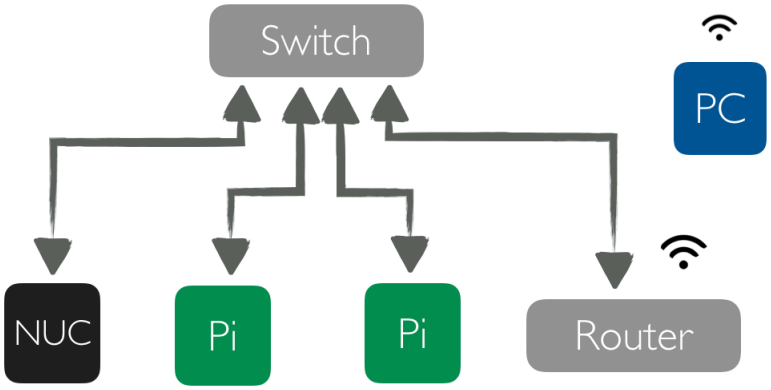

Figure 6.2.: Testbed setup for recognizing water bottles.

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 tested 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 tested 8 times for each Pi, a total of 16 tests. The sequence diagram displayed in 6.3, illustrates the procedure in addition to the time initials for each part of the process.

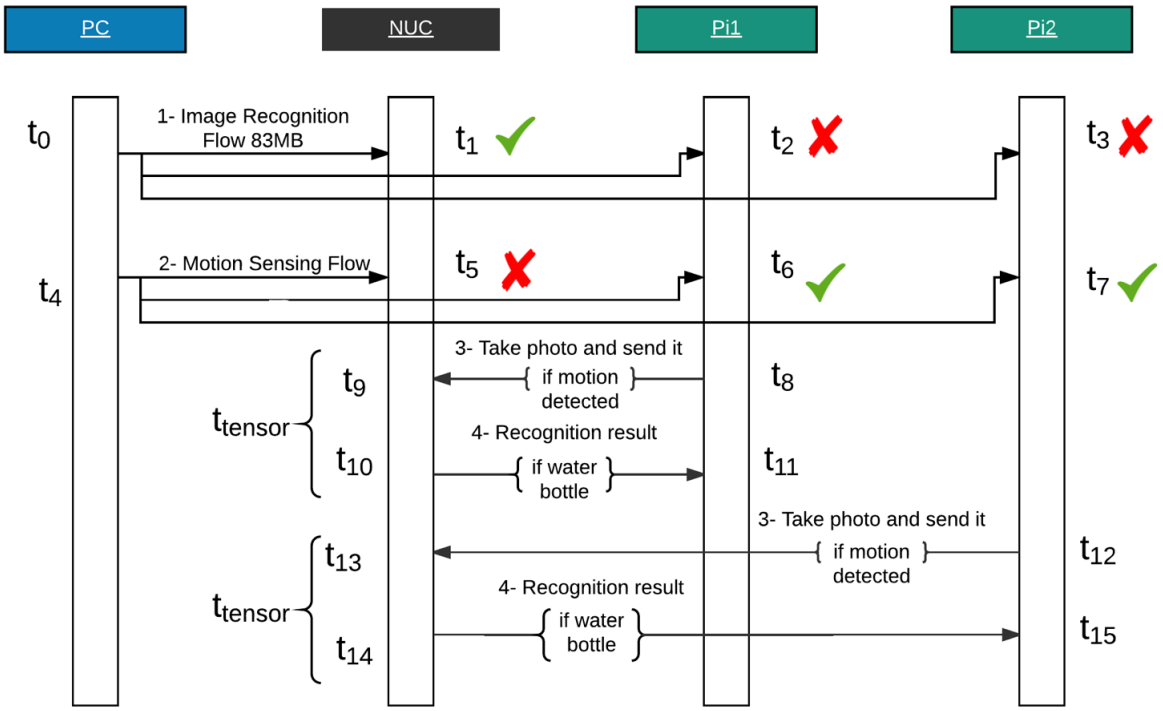

Figure 6.3.: Sequence diagram for recognizing water bottles.

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. The check mark shows that the flow was deployed and the cross mark means deployment was refused because of insufficient resources. 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 t_9 and t_{13}, NUC received messages from the Pis and started processing, detected a water 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_{15}. The tables 6.3, 6.4 and 6.5 show the mean and standard deviation of the delays.

	$t_1 - t_0$	$t_2 - t_0$	$t_3 - t_0$
$\bar{t}$	23.245 s	28.226 s	20.826 s
σt	1.440 s	8.830 s	8.851 s

Table 6.3.: Mean and standard deviation of the image recognition flow delays.

	$t_5 - t_4$	$t_6 - t_4$	$t_7 - t_4$
$\bar{t}$	0.087 s	2.322 s	1.948 s
σt	0.012 s	0.053 s	0.491 s

Table 6.4.: 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}
$\bar{t}$	0.282 s	0.700 s	0.273 s	0.6985 s	5.512 s
σt	0.061 s	0.067 s	0.048 s	0.0488 s	0.217 s

Table 6.5.: Mean and standard deviation for sending and receiving data delays.

	$t_{11} - t_8$	$t_{15} - t_{12}$
$\bar{t}$	6.566 s	6.592 s
σt	0.097 s	0.093 s

Table 6.6.: Mean and standard deviation for the whole data exchange period delays.

It is apparent 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 that range between 50K to 90K and the motion sensing flow had dependencies with sizes 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 in Tables 6.4 and also in 6.5 despite that the message sent from the Raspberry Pis to the NUC are heavier in terms of size than their replies, but the delays are much less. We can also conclude that it takes almost 1 second to have a successful data communication carrying dependencies between the Raspberry Pi and the NUC in our framework, by deducting the time taken in the image recognition process which is on average 5.5 seconds shown in Table 6.5 from the total time taken to exchange data between the Pi and the NUC displayed in Table 6.6.

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.

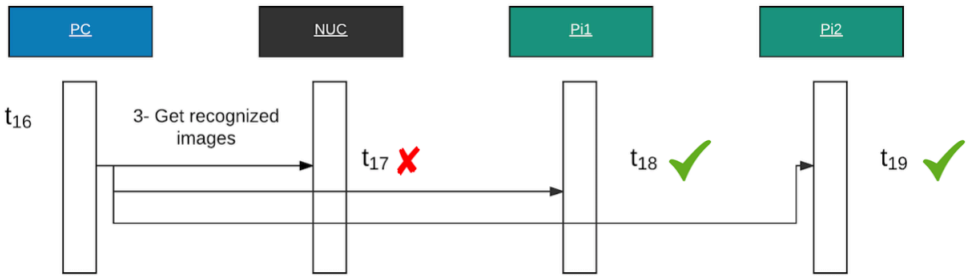

Figure 6.4.: Sequence diagram extension for getting recognized water bottles.

Continuing on the same testbed setup as Section 6.2 and the 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 as explained in 5.2.4. We retrieved some of the images displayed in Figure 6.5 from the Raspberry Pi endpoint to prove that the local composability flow works as expected. 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 flow from the PC until it was deployed by node-RED on other devices. The flow required low computational effort, therefore, it was not deployed on the NUC device. The experiment was tested 8 times.

	$t_{17} - t_{16}$	$t_{18} - t_{16}$	$t_{19} - t_{16}$
$\bar{t}$	0.105	0.801	0.834
σt	0.028	0.060	0.051

Table 6.7.: Mean and standard deviation for retrieving recognized images flow delays.

Figure 6.5.: Images of the water bottles received from the NUC.

Closer inspection shows that delays in this use case are quite low because this flow does not have any dependencies at all. Comparing the times in which the Raspberry Pis 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

The section below shows our aim 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 up the disconnected Pi 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 exchanges its Wi-Fi connection between the access point and router Wi-Fi each 80 seconds. The system setup is demonstrated in Figure 6.6.

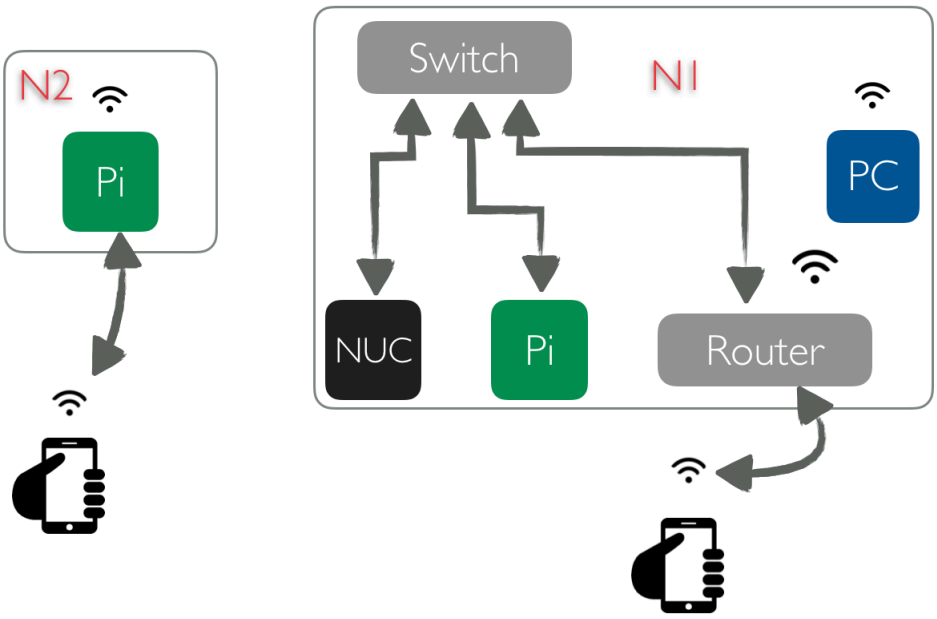

Figure 6.6.: Testbed setup for challenged and delay tolerant networks.

The experiment was tested 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 the current Maestro implementation, we do not have a way to map requests sent with their respective responses. In the experiment explained in Section 6.2, each request was only sent once hence we received only one response thus mapping them together was trivial, however this is not the case here. Since we sent more than one request at once and not all of them were recognized as water bottles, therefore there was no response from the NUC for some of these requests. That's 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 results from NUC till they are deployed to the disconnected Pi. Take into account 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 an image recognition request to another one sent as a response which could be an enhancement to Maestro.

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 reaches 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 6.8.: 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. 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 back to the other network. But also, having it too small, flows with large size of dependencies will not succeed to upload their 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
$\bar{t}$	185.818
σt	54.966

Table 6.9.: Delay mean and standard deviation of 13 messages 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
$\bar{t}$	90.467
σt	59.770

Table 6.10.: Delay mean and standard deviation of 10 messages sent from the NUC to disconnected Raspberry Pi having successfully recognized water bottles.

What stands out in the previous tables is that 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.

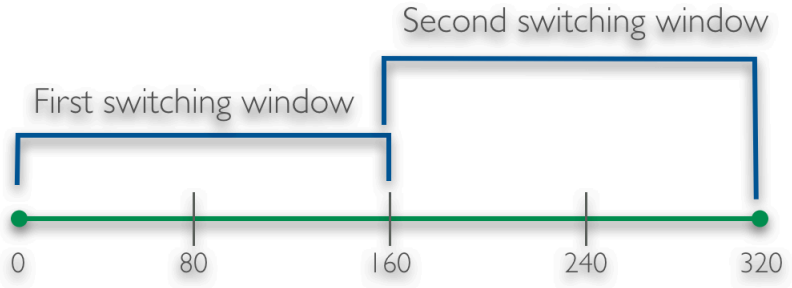

Figure 6.7.: Timeline for the Android device switching process.

When the messages are sent or the flows are published, we do not know whether the Android device is connected to the sender at that point or not, that's why we evaluate the delay time according to the whole transfer window which is 160 seconds as demonstrated in 6.7. We think that the average delay in Table 6.10 for delivering responses which is 90.467 seconds is acceptable. Since it means that the transfer was done in the first switching window. Further, The average times of the flows and messages sent from the Pi to the NUC in Tables 6.8 and 6.9 are also acceptable. Being from 160 to 320 seconds means that it had to wait for the second switching window which makes sense because these messages were carrying images, scripts and libraries as dependencies.

6.5. Summary

During this chapter, 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 typical 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 on high performing machines which received messages 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 the disconnected devices.

7. Conclusion

This chapter concludes the work of this thesis, we first give a summary about the proposed software framework, thesis flow and main contribution of this work. Then, we demonstrate the results and outcome of this thesis. Finally, we discuss the future work and enhancements for this work.

7.1. Summary

To sum up, we designed and presented a delay-tolerant and information-centric framework architecture for pervasive computing that distributes, composes and executes flows which are representations of computations describing pervasive use cases. The framework uses service discovery, allows sending and receiving of flows while accounting for their dependencies, required hardware resources, sensors and actuators even without an end-to-end path between senders and receivers. Moreover, it provides the execution environment for the flows and a user interface for publishing and designing flows. Further, the framework allows devices to communicate with each other and exchange data in a publish-subscribe manner which enables them to compose and have inter-relationships in addition to being able to locally exchange data through databases hosted on the same device.

In this thesis, we first gave an introduction to the problem and our intent to design a pervasive framework for computation distribution, composition and execution even in challenged networks. Therefore in the background chapter, we researched the current accomplishments in the fields of pervasive computing, delay-tolerant and information-centric networking. After harnessing these concepts and architectures, we came up with ideas and foundations which led us to design our framework architecture. However, we needed concrete real life use cases for requirements elicitation so that we can evaluate our framework against these requirements which were discussed in the system design chapter. We also explained the middleware implementation and used the extracted requirements to devise experimental use cases to evaluate our framework.

The main contribution of this work is both the framework design and the implementation of Maestro which acts as a middleman between the messaging system and the execution environment. It handles the flows necessities by attaching the depen-

dencies, checks the flow meta-data in order to make sure the device meets desired requirements and has the required resources. Once the flow passes all checks, it deploys the flow to the execution environment.

The evaluation results of our proof-of-concept implementation and system design showed the feasibility of our architecture. The use case experiments that we ran satisfied the requirements which were devised from real life use cases. We have proved the viability of distributing and executing flows with dependencies to smart devices with different resources and gadgets. Further, we showed that devices can exchange messages and compose different flows locally and globally. Beyond that, we confirmed that messages are delivered to challenged networks that does not have an end-to-end path. Finally, we provided the mean and standard deviation of the delays taken between sending and deploying messages to the execution environment

7.2. Future Work

The framework can be extended and enhanced on several aspects:

- *Streaming*: A possible extension to the framework is to allow live feed or footage to be streamed from one device to another. Of course this could be done by composing flows in which one device captures a video and sends the frames to other devices. However, this might experience some delay; it would be more efficient to have a streaming API in the messaging system which can be exposed to both the middleware and execution environment.

- *Request-Response Mapping*: In the middleware implementation there is no way at the moment to map a published message sent as a request with another sent as a response. More specifically, in the challenged networks experiment, we could not be sure which requests for image recognition sent from the Raspberry Pi lead to the successful replies sent from the NUC back to the Raspberry Pi. This is not the case with recognizing water bottles experiment because we sent one request each run and waited for the response. Unlike the challenged networks experiment, we sent a lot of requests and received less replies. This could possibly be done by forwarding the unique message identifier from the request to the reply thus mapping between them.

- *Security*: This thesis does not focus on securing the communication between devices and ensuring that requests and deployments to the execution environment are authenticated. This could be enhanced by providing a layer of security in the messaging and deployment process.

- *Resources Discovery*: The current middleware implementation reads the resources from a specification file on the smart devices. It would be more dynamic and flexible if the middleware could discover the attached resources dynamically and be sensitive to the addition or removal of gadgets.

Appendices

A. Use case Evaluation Results

A.1. Typical IoT Usage Flow

	$t_{Pi1} - t_{PC}$	$t_{db} - t_{Pi1}$	$t_{Pi2} - t_{PC}$	$t_{db2} - t_{Pi2}$
1	4.542	1.713	4.693	1.726
2	2.438	1.706	2.533	1.716
3	2.885	1.701	2.065	1.717
4	2.451	1.815	2.462	1.723
5	2.333	1.713	1.536	1.685
6	2.423	1.724	1.444	1.727
7	3.437	1.705	2.414	1.722
8	2.453	1.711	2.35	1.72

Table A.1.: Delays for the temperature flow.

A.2. Recognizing Water Bottles Experiment Delays

A.2.1. Image Recognition Flow

$	$t_1 - t_0$	$t_2 - t_0$	$t_3 - t_0$
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

Table A.2.: Delays for sending image recognition flow for all devices.

A.2.2. Detect Movement and Store Image Responses Flow

$	$t_5 - t_4$	$t_6 - t_4$	$t_7 - t_4$
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

Table A.3.: Delays for sending the motion detection flow for all devices.

A.2.3. Images and Recognition Results

$	$t_9 - t_8$	$t_{11} - t_{10}$	$t_{13} - t_{12}$	$t_{15} - t_{14}$	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

Table A.4.: Delays for data sent between the Raspberry Pis and the NUC.

$	$t_{11} - t_8$	$t_{15} - t_{12}$
1	6.484	6.591
2	6.64	6.768
3	6.507	6.57
4	6.737	6.476
5	6.584	6.563
6	6.582	6.684
7	6.423	6.572
8	6.57	6.513

Table A.5.: Delays for the whole data exchange period between the Raspberry Pis and the NUC.

A.3. Get Recognized Images Experiment Delays

$	$t_{17} - t_{16}$	$t_{18} - t_{16}$	$t_{19} - t_{16}$
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

Table A.6.: Delays for the flow which retrieves the recognized images for all devices.

A.4. Challenged Networks

$	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

Table A.7.: Delays for messages sent from the disconnected Raspberry Pi to the NUC.

$	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

Table A.8.: Delays for messages sent from the NUC to disconnected Raspberry Pi having successfully recognized water bottles.

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.

[Ash17] M. P. J. Ashby. "The Value of CCTV Surveillance Cameras as an Investigative Tool: An Empirical Analysis." In: *European Journal on Criminal Policy and Research* (2017), pp. 1–19. ISSN: 1572-9869. DOI: 10.1007/s10610-017-9341-6.

[Ban] Banana-pi. *Banana-Pi single board computers*. http://www.banana-pi.org/, Accessed: 2017-06-08.

[BR97] J. Bound and Y. Rekhter. "Dynamic updates in the domain name system (DNS UPDATE)." In: (1997).

[Bur06] S. Burleigh. "Interplanetary Overlay Network." In: (2006).

[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.

[CK13] S. Cheshire and M. Krochmal. "Multicast dns." In: (2013).

[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.

[Dem05] M. Demmer. "DTN2 Reference Implementation." In: *presentation at IETF DTNRG Meeting-March*. Vol. 9. 2005.

[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.

[For] W. E. Forum. *Number of cars by 2040*. https://www.weforum.org/agenda/2016/04/the-number-of-cars-worldwide-is-set-to-double-by-2040" , Accessed: 2017-06-06.

[FS08] S. Fleck and W. StraSSer. "Smart Camera Based Monitoring System and Its Application to Assisted Living." In: *Proceedings of the IEEE* 96.10 (Oct. 2008), pp. 1698–1714. ISSN: 0018-9219. DOI: 10.1109/JPROC.2008.928765.

[Gar] Gartner. *Estimated number of connected devices in 2017*. http://www.gartner.com/newsroom/id/3598917, Accessed: 2017-06-08.

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

[Gin+10] P. Ginzboorg, T. Kärkkäinen, A. Ruotsalainen, M. Andersson, and J. Ott. "DTN communication in a mine." In: 2010.

[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.

[Lon] T. for London. *CCTV in London underground*. https://tfl.gov.uk/corporate/privacy-and-cookies/cctv, Accessed: 2017-06-08.

[LXZ15] S. Li, L. D. Xu, and S. Zhao. "The Internet of Things: A Survey." In: *Information Systems Frontiers* 17.2 (Apr. 2015), pp. 243–259. ISSN: 1387-3326. DOI: 10.1007/s10796-014-9492-7.

[LYG08] S. Lee, D. Yoon, and A. Ghosh. "Intelligent parking lot application using wireless sensor networks." In: *2008 International Symposium on Collaborative Technologies and Systems*. May 2008, pp. 48–57. DOI: 10.1109/CTS.2008.4543911.

[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.

[Osu13] I. O. Osunmakinde. "Towards Safety from Toxic Gases in Underground Mines Using Wireless Sensor Networks and Ambient Intelligence." In: *International Journal of Distributed Sensor Networks* 9.2 (2013), p. 159273. DOI: 10.1155/2013/159273. eprint: http://dx.doi.org/10.1155/2013/159273.

[PN03] R. Patra and S. Nedevschi. "Dtnlite: A reliable data transfer architecture for sensor networks." In: *CS294-1: Deeply Embedded Networks (Fall 2003)* (2003).

[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.

[WR10] T. Winkler and B. Rinner. "Trustcam: Security and privacy-protection for an embedded smart camera based on trusted computing." In: *Advanced Video and Signal Based Surveillance (AVSS), 2010 Seventh IEEE International Conference on*. IEEE. 2010, pp. 593–600.

[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.

[Zer] ZeroMQ. *ZeroMQ Distributed Messaging*. http://zeromq.org/, Accessed: 2017-05-23.