FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Thesis title

Aly Saleh

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Thesis title

Titel der Abschlussarbeit

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

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

Munich, Submission date Aly Saleh

Acknowledgments

Abstract

Contents

List of Figures

List of Tables

1 Introduction

1.1 IoT & Distributed Sensor Networks

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

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

1.2 Motivation

1.2.1 Show the need to explore Pervasive Computing

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

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

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

2 Background & Related Work

This chapter describes the concepts and background information that this thesis uses and relies on. It gives a brief introduction about Internet of things and other concepts that emerged from within such as pervasive and fog computing. Moreover, the chapter explains delay tolerant and information centric networking as they play an important role in this thesis. Further, we explain the software platforms and hardware used to implement the system framework.

2.1 Internet of Things

In general terms, IoT refers to a highly dynamic and scalable distributed network of connected devices equipped with context-aware gadgets that enables them to see, hear and think[Xia+12]. Then, transfer theses senses to a stream of information allowing them to digest the data and act intelligently through actuators if needed. They are also allowed to communicate and share knowledge, which make them smart, powerful and capable of acting independently. Smart devices in an IoT network are heterogeneous in terms of computation capabilities, also each device is energy optimized and able to communicate. Moreover, 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 identifier and are context-aware, they can be tracked and localized, which is very helpful when performing geospatial computations [Mio+12]. The huge demand on IoT has triggered the development small-scale, high-performance, low-cost computers, in addition, sensors and actuators are getting cheaper, smaller and more powerful which in turn increased the interest even more.

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

- Smart homes which tend to use sensors and actuators to monitor and optimize home resource consumption and control home devices in a way that increases

humans satisfaction. Further, expenses generated from resource usage such as gas, power, water and telecommunications can be sent directly to related authorities without any human intervention [Cha+08].

- Smart factories also known as "Industry 4.0" the fourth industrial revolution which are optimized machines that communicate together in order to improve the manufacturing process and gather data to analyze factories logistics, pipeline and product availability It also creates intelligent products that can be located and identified at all times in the process [Gil16].
- Smart cities is one of the most adopted applications in the IoT field, it comprises smart parking, traffic congestion monitoring and control, real time noise analysis, waste management and others. All this applications need enhanced communication and data infrastructure. It aims to increasing quality of living for individuals[Zan+14].
- There are also applications in health care, environmental monitoring, security and surveillance.

IoT is very diverse, one way of applying it is to gather data from the smart devices, then process data in the cloud via *Cloud Computing*. Afterwards, results could be sent back to smart devices in order to act somehow. However, exploiting the overwhelming capacity of IoT lead to more specialized and concrete terms that are more focused on pushing computations to the smart devices "Edges" . Consequently, more terms like *Edge Computing*, *Pervasive Computing* and *Fog Computing* emerged.

2.1.1 Wireless Sensor Networks

2.1.2 Fog Computing

The fog is an extension to cloud computing at the edge of the network. It provides computation, storage, networking and application services to end-users. Fog and cloud are independent, in fact, cloud can be used to manage the fog. They are also mutually beneficial, some use cases are better deployed in the fog and the other way around. Research yet to determine which applications should go where. The fog is characterized by having lower latency than the cloud, thus are more useful in time critical applications. Also, fog devices have location awareness with a better geographical distribution than the centralized cloud approach. It can distribute the computations and storage between the cloud, itself and idling devices on the network edge [CZ16].

2.1.3 Pervasive Computing

Pervasive computing also known as *Ubiquitous Computing* followed from the general IoT networks 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 they even hold a history of what has happened in the environment. They also communicate seamlessly in order to share knowledge and help each other take better decisions. Moreover, pervasive devices can be relocated from one place to another, thus changing the network and possibly environment. Therefore, devices can not be addressed with their respective networked addresses because they might eventually change.

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

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

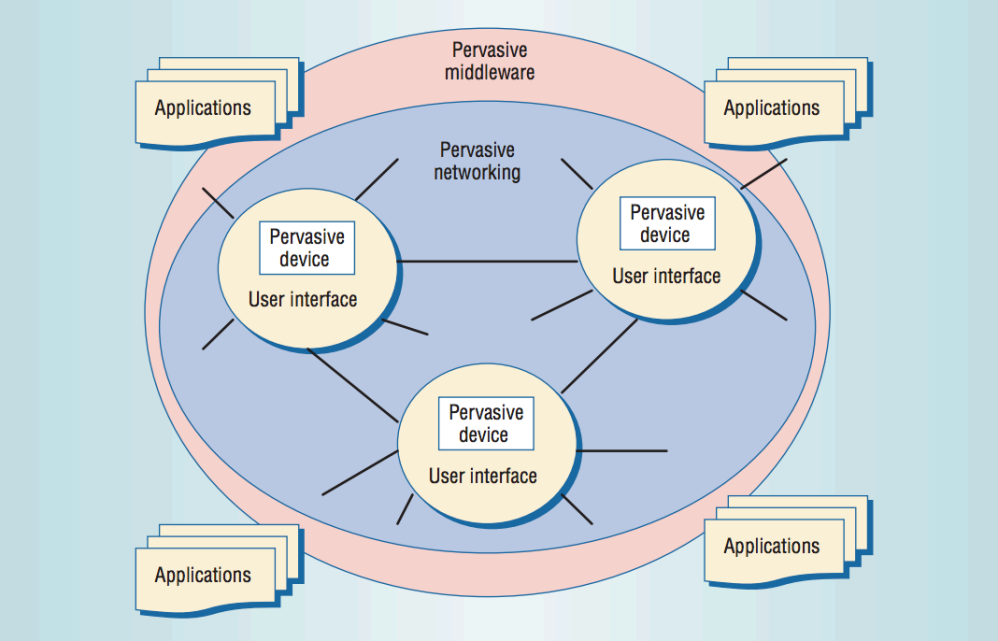

Figure 2.1: Pervasive Computing environment architecture. *Adapted from [SM03]*

The road to pervasiveness is not paved with gold, there are many challenges that faces the implementation and design of pervasive applications. Some of these challenges are [SHB10]:

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

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

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

4. Communication and cooperation between devices requires interoperability. There are three different says 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 Networking

2.2.1 Information Centric Networking

Information Centric Networking (ICN) is a concept that focuses on content sharing and distribution. In contrast to current network approaches which are host-centric where communication takes place between hosts such as servers, personal computers, etc. ICN was brought to light as a result of the increasing demand on content sharing in highly scalable, distributed and efficient fashion. ICN compromises network caching, replication across entities and resilience to failure.

The content types includes web-pages, videos, images, documents, streaming and others which are titled Named Data Objects (NDO). The NDO is only concerned by its name and data. As long as name identity is preserved, it does not matter where the NDO is going to be persisted, what is the storage method or which type of transport procedure is used. Therefore, copies of NDO are equivalent and can be supplied from any location or replica across an ICN network. However, since the name represents its identity, ICN requires unique naming for individual NDOs.

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

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

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

2.2.1.1 Content Centric Networking

Content centric networking (CCN) is an architecture based on the ICN concept. Namespace in CCN is hierarchal, for instance, `/tum.de/connected-mobility/iot` matches the figure 2.2. Names do not have to be meaningful or readable, they can include hashes, timestamps, ids, etc. A request matches an NDO if its name is a prefix of any named object, for example, a request with the name `/tum.de/connected-mobility/iot` matches an NDO with the name `/tum.de/connected-mobility/iot/pervasive-computing`. CCN natively support on-path caching, however, off-path can also be supported[Xyl+14].

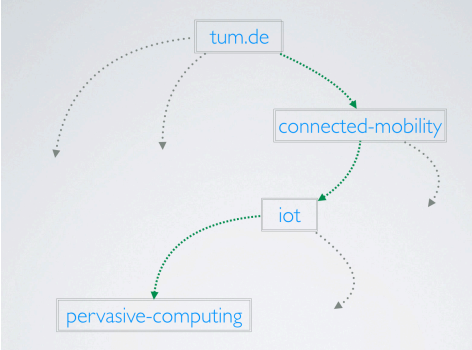

Figure 2.2: Hierarchal namespace example for CCN

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

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

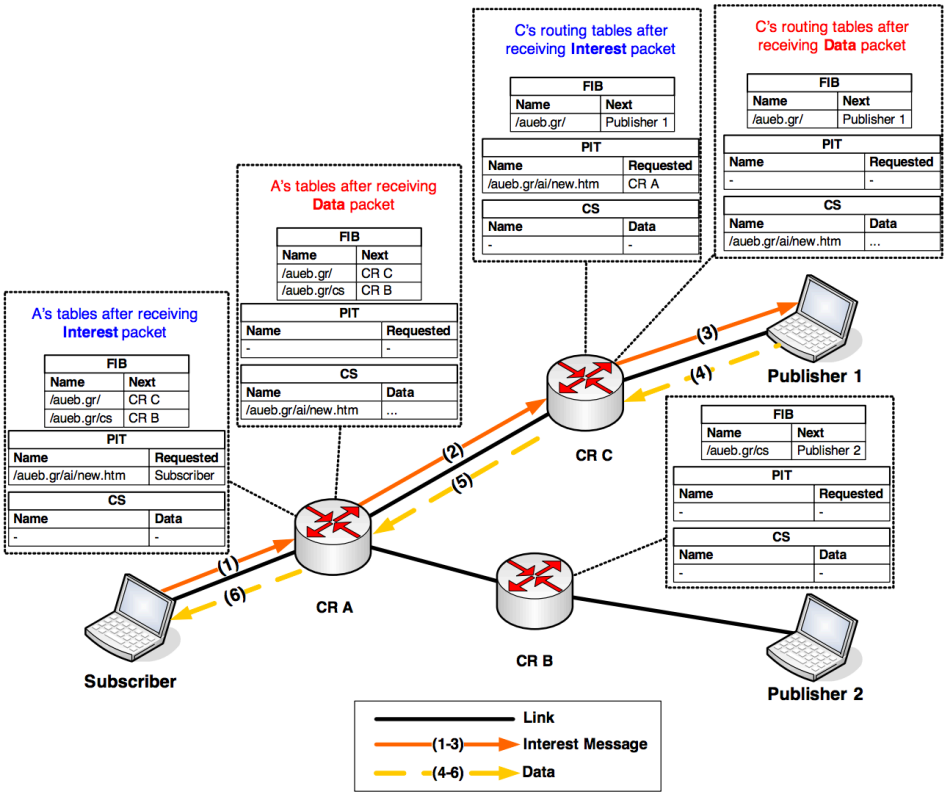

Figure 2.3: Content centric networking architecture and flow *Adapted from [Xyl+14]*

Figure 2.3 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.

2.2.2 Delay Tolerant Networking

2.3 Used Platforms

2.3.1 Node-Red

2.3.2 SCAMPI

2.3.3 Raspberry Pi

2.3.4 Time-series Databases

3 Framework in Theory

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

3.1 Foundation

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

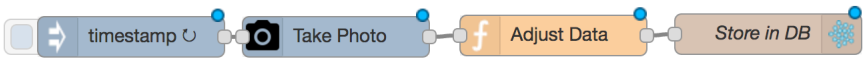

Figure 3.1: A node-red flow that stores an image in a database every time interval

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

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

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

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

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

$$f_1 : A \to B$$

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

$$f_2 : B \to C$$

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

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

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

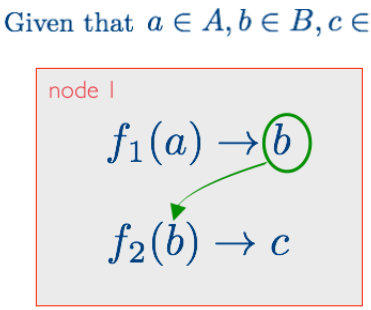

Figure 3.2: A node containing two composable flows

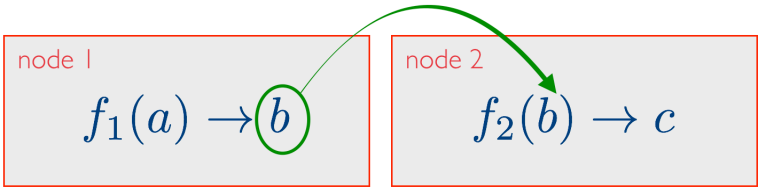

Figure 3.3: Two separate nodes having distributed composability

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

3.2 Computational Model

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

3.2.1 Distributed Nodes & Flows

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

Turning to flows, in essence every challenge related to making the nodes distributed also applies to flows because nodes host flows. However, there are more to flows, distributing them across nodes could have different approaches depending on the use case. Let us explain this with figure 3.4, to start with, lets take the set of all nodes in the system and call it $S(t)$ which is a function of time since nodes can be removed or added to the system dynamically at any instant of time. Moreover, nodes in Then comes the candidate set $S_C(t)$, which are nodes that satisfy the required dependencies and resources of a certain computation. Note that, nodes inside any of the these sets might not have end-to-end path.

Given $S_C(t)$, the flows could be either sent to a random set of nodes or to a specific set of nodes. This provides flexibility in applying the use case without wasting resources. In addition, it magnifies the effect of locational context, meaning that if we want to compute a certain computation or measurement in a specific location and we know the general identifiers of the nodes residing in this location, we can send a flow to this exact set of nodes with our desired computation.

Continuing to explain flows approaches with figure 3.4, a flow could be distributed across nodes of the candidate set $S_C(t)$ in multiple ways explained as follows: (i) flows are sent to all nodes in $S_C(t)$, (ii) flows are sent to a set of n nodes where $n > 1$, whether they are selected as case *C #2* in the figure or picked at random, (iii) choosing only one node to execute a specific flow *C #1*. As a result, the communication model is one of the most crucial parts to guarantee a distributed system, it should have the flexibility to provide these approaches and overcome the hurdle of disconnected nodes.

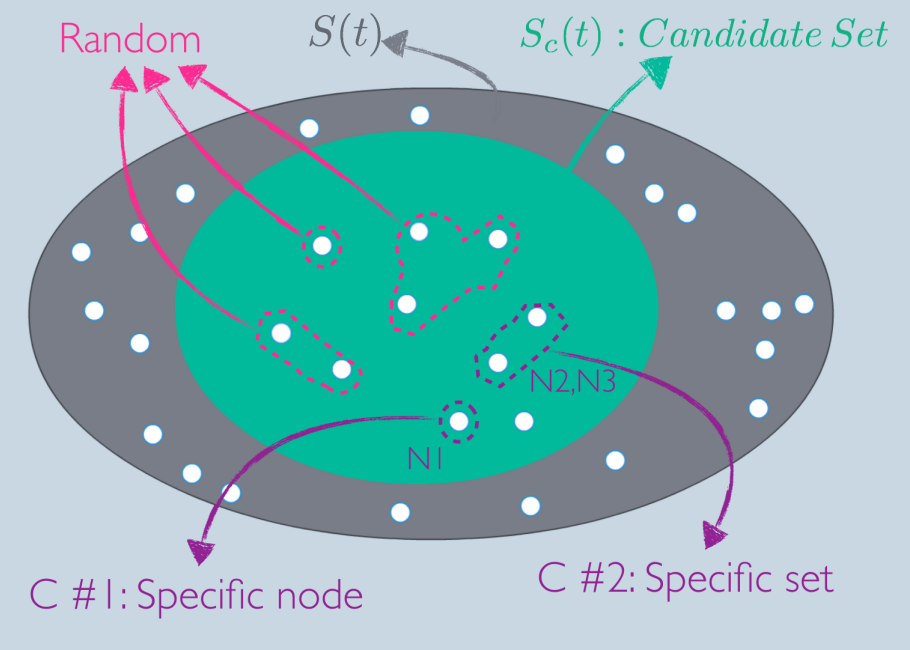

Figure 3.4: Distributing flows approaches

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

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

3.2.2 Software Dependencies

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

First, are the static dependencies are mainly the common libraries and software that most of the computations would require, they represent the base of framework. That is why these dependencies are installed to each node in our design, examples of

these dependencies include the operating system, data store and any other standard or custom libraries that are used by most computations. In addition to, the messaging system which implements the communication model allowing interaction between nodes.

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

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

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

3.2.3 Resources " Physical Dependencies"

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

Considering that each computation model has meta data describing its resource consumption, then it is possible to know if it is going to be deployed on a specific node

or not. Additionally, if it is not going to be deployed then it should be decided whether the computation is going to be queued or dismissed according to the possibility of acquiring the resource.

The idea of queuing computations however develops a scheduling problem. Since we have a queue of computations inside each node, we will have a race condition on which computation should be deployed first according to available resources. Furthermore, since some computations might be dismissed, a rather bigger scheduling problem will come up when we try to fit the all computations across nodes in the whole system framework.

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

3.2.3.1 Sensors and Actuators

Sensors and actuators are resources attached to a node such as cameras, temperature and gas sensors. Executing a computation missing this type of resource on a node should have a lower possibility of being queued, since its highly unlikely that this resource would be attached soon.

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

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

3.2.3.2 Hardware Resources

The second type of resources is related to the node performance, its power and memory capabilities, it is heavily biased by the node processor and its random access memory type and size. Computations vary in terms of resource consumption and hence a heavy computation should not be deployed to a node which is already loaded.

Queuing this type of dependencies should have a higher probability because it is

highly possible that one of the computations will finish soon, thus decreasing the cpu usage and freeing more memory.

3.2.4 Pub-Sub Messaging Queues

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

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

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

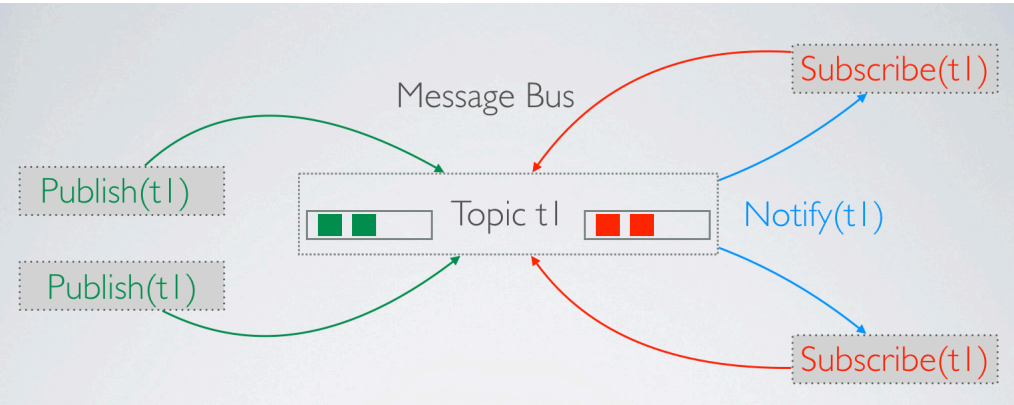

Figure 3.5: Common message queues

However having a pub-sub message queue with a centralized bus can not be used

in our case. Since we do not know machines addresses thus knowing where is the centralized message queue located is not possible. Therefore, each party or node would have its own message bus and local storage which are then synchronized together.

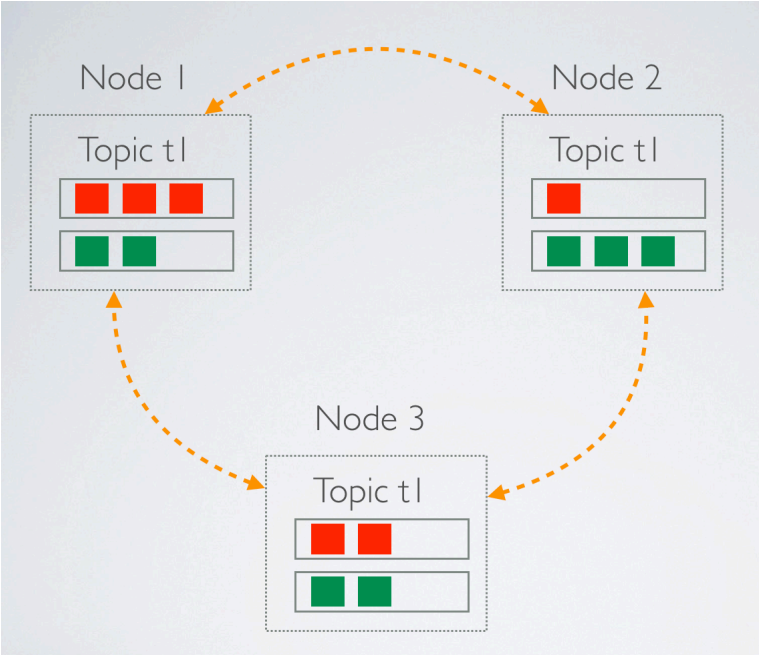

Figure 3.6: Common message queues

Now addressing the mentioned challenges:

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

message should be published once to each unique topic. For example, if we use the general identifier as a unique topic and want to publish a message to node 1, then, we can create unique topic *"N #1 "* and send the messages over this topic.

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

3.3 Data Model

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

3.3.1 Data Types

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

3.3.2 Moving Data

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

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

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

3.3.3 IO Specification

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

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

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

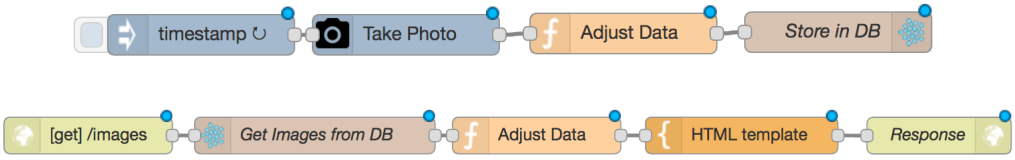

Figure 3.7: Two separate computational flows describing the IO through a database

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

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

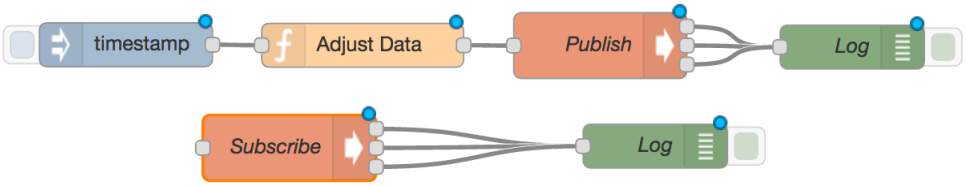

Figure 3.8

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

3.4 Summary

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

4 Approach

4.1 System Design

A system design can be broadly described as an architecture of the system, which includes an explanation of each hardware component of the system, the connection between these components if there is any, and the data flowing between these components. Moreover, it gives an idea of the whole system but not its exact functionality, hence, giving a simple understanding of the architecture without jumping into much detail.

4.1.1 Components

Below, each component of the proposed system design is explained.

4.1.1.1 Node

A Node is one of the core components of this design, it is a small computer device of low storage and computation capacity compared to nowadays portable computers, commonly a *Raspberry Pi* but could be any other device. It is connected to several sensors which typically detect certain changes in the environment and converts it into digital data, for instance, Gas sensor, Temperature sensor or a Camera. Then, the device either stores the data into a local database, performs a computation locally, does both or even asks other nodes to do computation instead, however, an assumption about which sensors or specifications does a specific node possess can not be made, meaning, each node might not have the exact number or types of sensors because each node may be deployed in a different timing or context. Thus, each node has a configuration file specifying its capabilities. A typical node is shown in figure 4.1

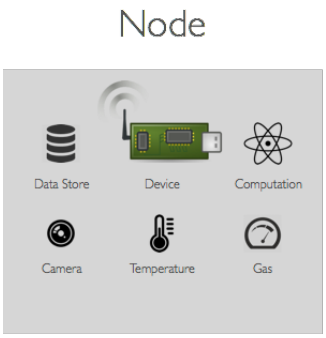

Figure 4.1: A typical node in the system

4.1.1.2 High Performance Units

CPUs in the proposed system nodes in 4.1.1.1. An example of a high processing unit is a Graphics Processing unit *GPU*.

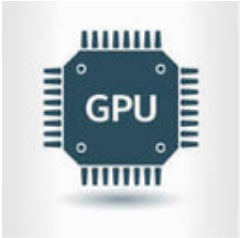

Figure 4.2: Figure denoting a Graphics Processing Unit GPU

4.1.1.3 Network

A Network in this design is a set of connected components which are capable of communicating and therefore allowing data sharing between them.

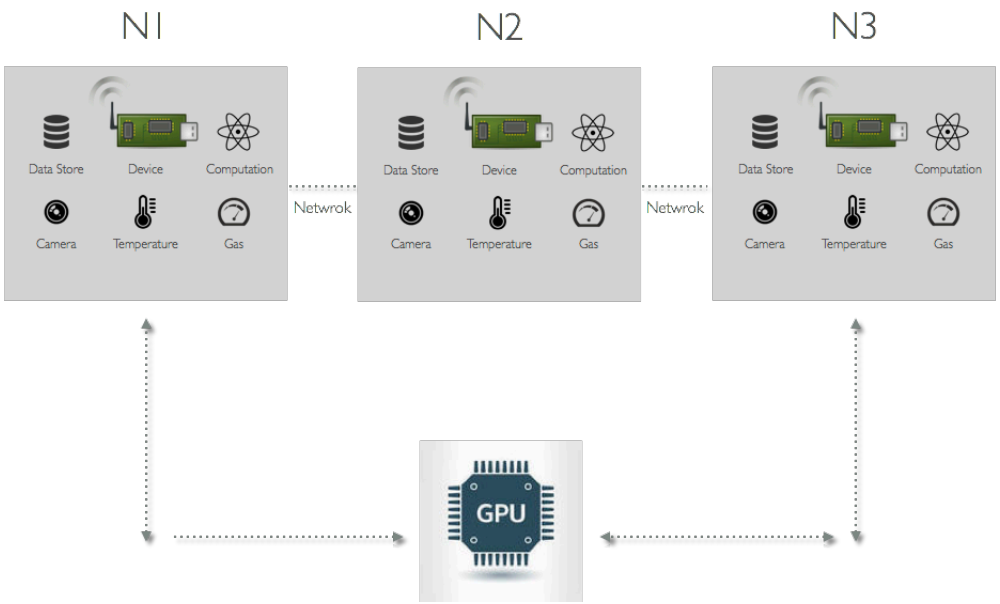

Figure 4.3: A network consisting of three connected nodes and a GPU

– TODO: Emphasis the difference between persistent and non persistent network links in system design.

4.1.1.4 Mobile Device

A Mobile Device in this context is any device that can connect to the network containing the nodes and is allowed to carry data from one network to another, hence, allowing a form of data sharing between networks or nodes which are not connected.

Figure 4.4: Figure denoting a Mobile Device

4.1.2 Connectivity and Data Flow

A Network described in 4.1.1.3, is a simple form of connectivity between components, however, components and specifically nodes are not necessarily connected, sometimes they are just a standalone component that cannot share any information via direct connectivity, also, networks could be disconnected as well, meaning, a network might not be connected to the whole system, thus, is a standalone network. In these cases, a mobile device could help in carrying information and data between these disconnected nodes or networks.

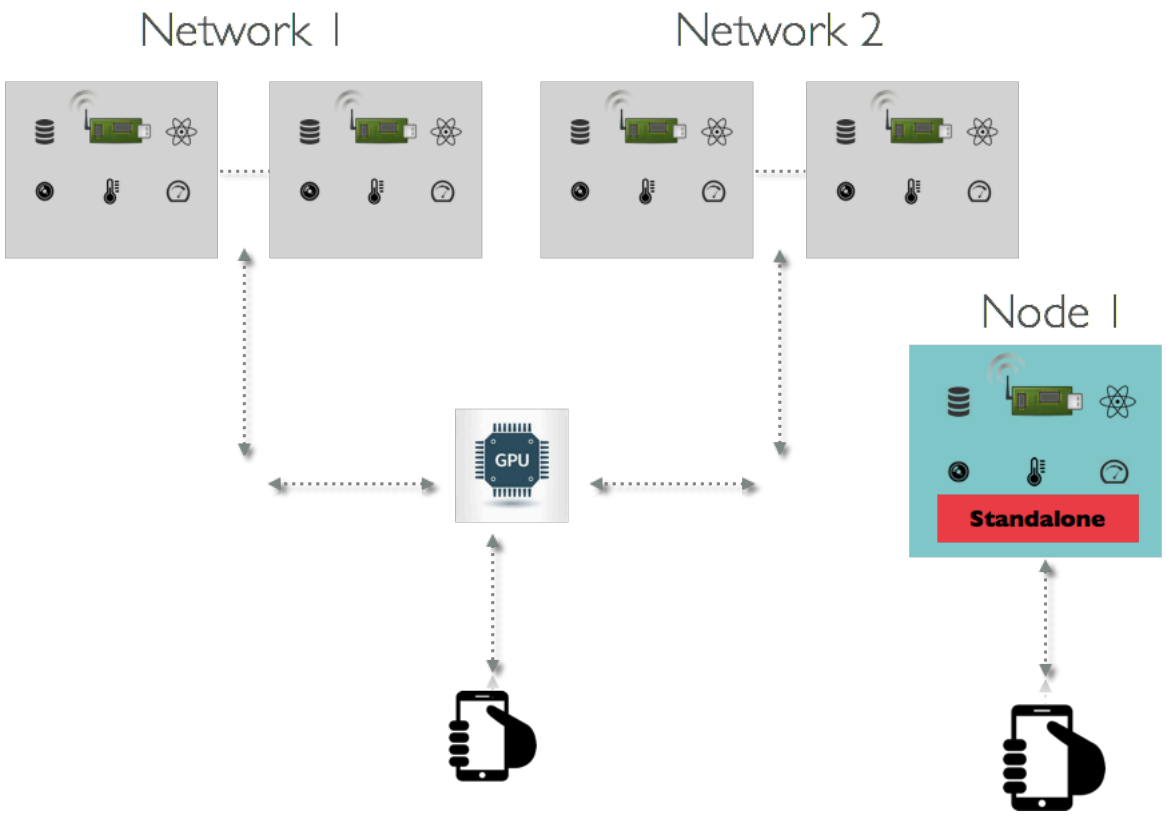

Figure 4.5: Two networks connected with a GPU and one standalone network

4.2 Use Cases

4.3 Requirements

4.4 Summary

5 Evaluation

5.1 Implementation

5.2 Use Cases

Design of two, three or more use cases, however, implement one or two.

5.3 Performance Tests

5.4 Limitations

6 Conclusion

6.1 Summary

6.2 Future Work

6.2.1 Streaming API

Bibliography

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

[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.

[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.

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

[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.

[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.

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

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

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

[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.