DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Thesis type (Bachelor's Thesis in Informatics, Master's Thesis in Robotics, . . .)

Thesis title

Aly Saleh

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Thesis type (Bachelor's Thesis in Informatics, Master's Thesis in Robotics, . . .)

Thesis title

Titel der Abschlussarbeit

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

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

2.1 Introduce Edge, Fog and Pervasive computing, how they are used in this context

2.2 Explain how sensor data data is modeled and distributed in the current published approaches

2.3 Illustrate what are the ideas and possible network mechanisms and protocols that could be used data transfer

2.3.1 Server To Server

2.3.2 Server To Device

2.3.3 Device To Device

2.4 Explain Opportunistic networks and SCAMPI architecture

2.5 Show other approaches in the literature

3 Approach

3.1 Requirements

3.2 Use Cases

3.3 Modeling of Input Sensor Data

3.3.1 Show how the different sensors have data been modeled to fit our requirements for further use in computations

3.4 Framework

3.4.1 Foundation

The fundamental core element of this framework is the computational unit, which is responsible for describing the use case. One possible abstraction of the computational unit is the *flow*, which is a purposeful unit of computation that contains sequentially meaningful instructions. Also, it can be either standalone self-contained computation or interactive in which they collaborate with other flows for data gathering, sharing and processing. Moreover, the flow is composed of elements which could have a significant meaning such as snapping a photo or could act as an intermediary to harmonize data as shown in figure 3.1.

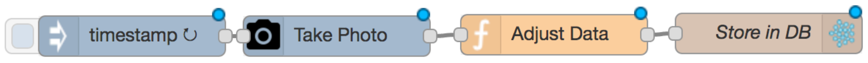

Figure 3.1: A node-red flow that stores an image in a database every time interval

After having defined flows, it is important to give an idea on how they end up being executed. To begin with, we must address the challenge that flows are distributed in the sense that each flow could reside on different node. In addition, as previously mentioned, flows may be interactive, therefore, they need a way to communicate through distributed systems. Moreover, since nodes might be disconnected, the communication mechanism must guarantee that there exit a way in which data could be transfered between these disconnected nodes. Furthermore, it should handle sending the computations themselves from one node to another, at the end we would like to be pervasive and manage sending computations everywhere.

Another challenge that faces the flows execution, is the dependencies and resources which could be demanded in order to carry on the execution. They vary from one use case to another, thus needs to be orchestrated across nodes through messaging system.

Now assuming that we can send flows to the nodes, make them communicate and care for their dependencies and resources, one aspect remains, which is triggering the execution of flows. There are multiple ways to start carrying on an execution, one simple example is a time interval trigger as shown in 3.1. Other ways include, starting computation execution when new incoming data has been received or other events have been triggered.

A flow should be modular having a specific functionality with defined interfaces that helps in reducing the complexity, allowing re-use and re-assembly. Moreover, since flows need to interact and exchange data, they should be composable. 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

$$f_1 : A \rightarrow B$$

then we have 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 taking f_1's input and giving f_2's output, resulting in final flow f_3 which is a composite of both.

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

As mentioned before, composability should also be valid in a local or distributed environment. In the case of locally composing flows within the same node, there should be a way to connect the output of a flow to the input of another as shown in 3.2. On the other hand if it is distributed composability, the messaging system should connect the dots and serve as a broker to deliver the data.

$$a \in A, b \in B, c \in C$$

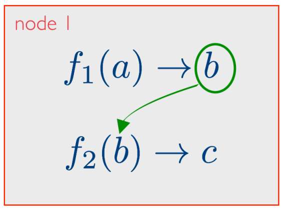

Figure 3.2: A node containing two composable flows

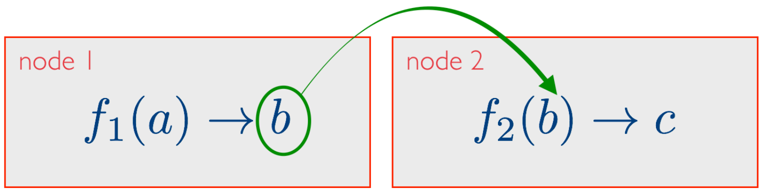

Figure 3.3: Two separate nodes having distributed composability

3.4.2 Computational Model

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

3.4.2.1 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 edges "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 exactly the case. 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.

Turning to flows, in essence every challenge related to making the nodes distributed also apply to flows because nodes host flows. However, there are more to flows, applying the concept of distributed system to flows could have different meanings and approaches. It could be distributed in multiple ways depending on the use cases explained as follows: (i) the general and most intuitive approach in regards to pervasive computing, in which flows are pushed to all the available nodes either the connected or disconnected ones, (ii) pushing flows to n number of nodes whether they are selected or picked at random, (iii) choosing only one node to execute a specific flow. Therefore, the communications model is the one of the most crucial parts to grantee a distributed system, it should have the flexibility to provide these approaches and overcome the hurdle of disconnected nodes.

Another main challenge would be 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.

3.4.2.2 Pub-Sub Messaging Queues

As previously stated, a smart communication system is essential to our computational model, it solves some of the biggest challenges in our approach which are mainly nodes service discovery, sending and receiving messages of distributed, connected and disconnected nodes. Also, considering that service discovery is dynamic, the communications model is not end-point centric since we cannot 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 meaning it knows that there are some parties interested in sending data and others willing to revive the same data given the same context regardless their network location. For all the stated facts, publish-subscribe messaging queues would best fit our needs. It implements service discovery, hence, it can discover all the other nodes who has the

3.4.2.3 Dependencies

Before proceeding to examine the computation itself and explain how it is designed, we must first introduce the dependencies that the computation would need to execute. There are different types of dependencies; first are the software frameworks that the whole design relies on and must exist on each node. These are the common libraries and systems that most of the computations would need to execute. That's why, these dependencies are shipped to each node in our design, examples of these dependencies include the operating system, data store, node-red and any other standard or custom libraries that is needed to guarantee a successful execution. In addition to, a messaging system which must be included to allow communication between nodes. These dependencies need only to be shipped once while initializing the node.

Second, are the dependencies that are specific to each computation such as additional scripts, data files or libraries. In this case, they cannot be shipped at node initialization since we cannot know what are the custom dependencies any computation would need beforehand. Therefore, the design of the computation model allows a way to configure additional dependencies, which are sent accordingly to any node that is going to execute this computation. This creates a bit of ambiguity because what if the dependency that is being shipped already is on the receiving node, also what makes it more complicated, is that the node does not know if it is an older version of the dependency or a newer one. Furthermore, what if there is a computation on the node that uses an older version of the same library while the maintainer is sending a new computation with a newer version of the same library that is not backward compatible. However, there are multiple proposed solutions to remove the ambiguity and make the custom dependency shipping 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.

3.4.2.4 Resources

Resources are a different type of 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 to carry out the computation is missing then it could be either dismissed or queued and that depends greatly on the type of resource. Moreover, the maintainers cannot make any assumptions about them, meaning, an assumption stating that each node has a camera is not necessarily true. Since the resources cannot be standardized, each computation must specify the resources it is going to need, then a node can check against its capabilities and decide whether it could carry out this computation or not. This kind of information is also known as computation meta-data. Resources may be classified into two main types explained below.

3.4.2.4.1 Hardware Resources

Hardware resources are attached to a node such as cameras, temperature and gas sensors. Also, executing a computation on a node missing this type of resource should have a lower possibility of being queued, since its highly unlikely that this hardware resource would be attached soon. The computation model suggests several ways to describe a node resource capabilities; first by using a specification file that expresses the resources in a certain node. Of course this approach has its drawbacks since if we attach a new hardware resource we must also edit the specification file correspondingly and that increases the manual work. Secondly, by using operating system commands to discover the attached resources each time a computation needs to be deployed on a system.

Moving on to consider the risk of computations acquiring the same hardware 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 the lock on the camera first will succeed while the other will fail. Therefore as a resolution, the design proposes resource decoupling; instead of having the computation to ask a specific resource directly for information, the resource will always push its data to a database. Afterwards, the different computations could query the data from the database.

3.4.2.4.2 Computational Resources

The second type of resources is related to the node performance, its power and memory capabilities. Computations vary in terms of resource consumption and hence a heavy computation should not be deployed to a node which is already loaded. Considering that each computation model has meta-data describing its resource consumption, then it rather easy to decide 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 computations is going to be queued or dismissed according to the possibility of acquiring the resource.

3.5 Data Model

3.5.1 Data Types

3.5.2 Moving Data

3.5.3 IO Specification

- I/O spec design for databases for two composable flows

3.6 System Design

A System Design can be broadly described as an architecture of the system, which includes an explanation of each and every hardware component of the system, the connection between these components if there is any, and the data flowing between these components. Moreover, it provides a wide glimpse of the whole system but not its exact functionality, hence, giving a simple understanding of the architecture without jumping into much detail.
Initially, the components of the System Design is introduced, then, the connection between these components is shown, and eventually, the flow of the data is pointed out.

3.6.1 Components

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

3.6.1.1 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 3.4

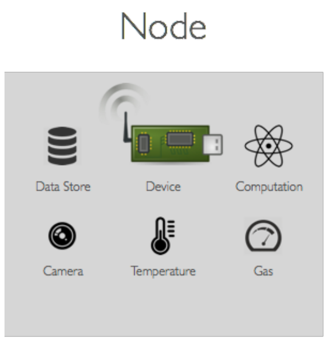

Figure 3.4: A typical node in the system

3.6.1.2 High Performance Units

CPUs in the proposed system nodes in 3.6.1.1. An example of a high processing unit is a Graphics Processing unit *GPU*.

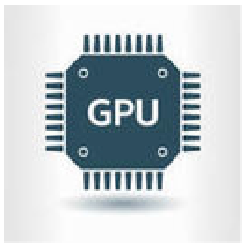

Figure 3.5: Figure denoting a Graphics Processing Unit GPU

3.6.1.3 Network

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

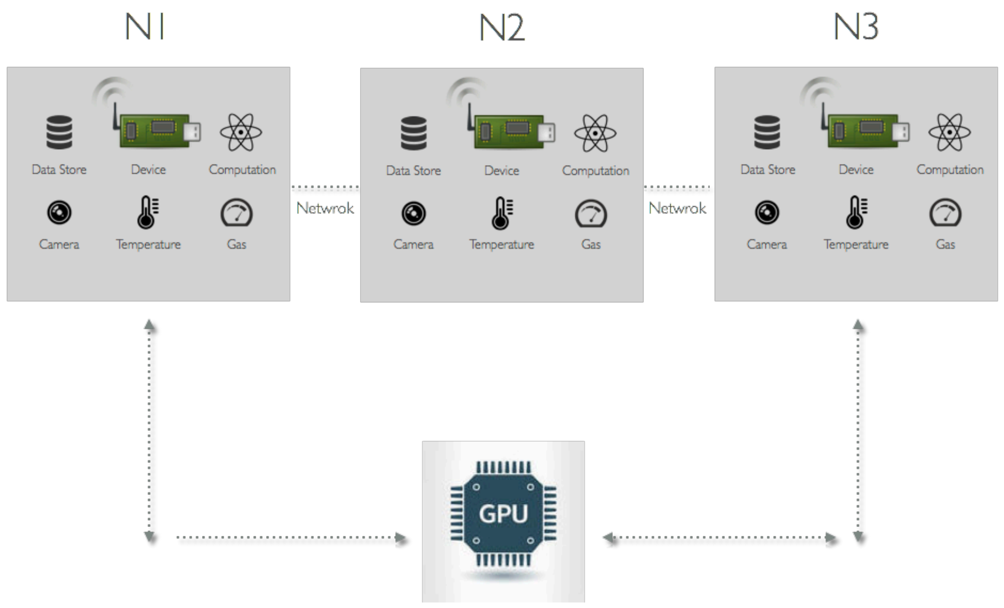

Figure 3.6: A network consisting of three connected nodes and a GPU

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

3.6.1.4 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 3.7: Figure denoting a Mobile Device

3.6.2 Connectivity and Data Flow

A Network described in 3.6.1.3, 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.

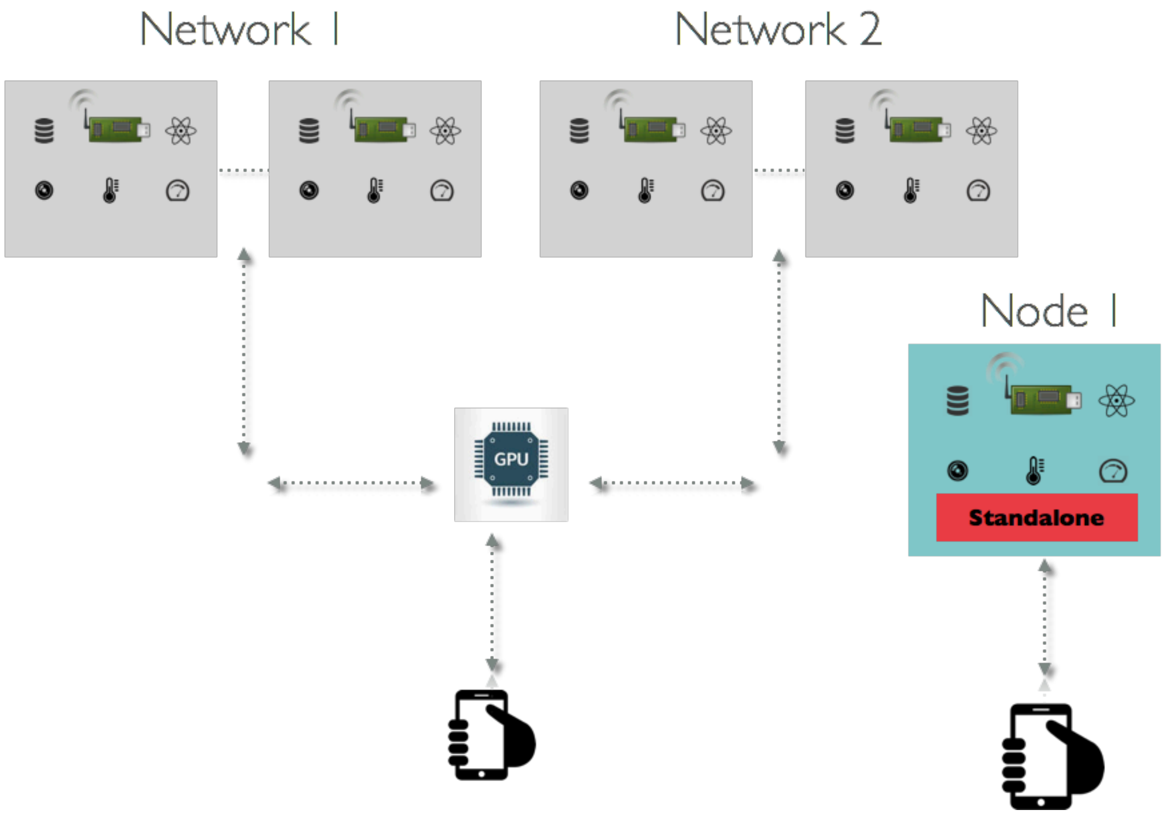

Figure 3.8: Two networks connected with a GPU and one standalone network

3.7 Summary

4 Evaluation

4.1 Implementation

4.2 Use Cases

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

4.3 Performance Tests

4.4 Limitations

5 Conclusion

5.1 Summary

5.2 Future Work

5.2.1 Streaming API

Bibliography

[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.

[GFT10] D. Guinard, M. Fischer, and V. Trifa. "Sharing using social networks in a composable Web of Things." In: *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*. Mar. 2010, pp. 702–707. DOI: 10.1109/PERCOMW.2010.5470524.