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

2.1 Internet of Things

Internet of things **IoT** is one of the most trending topics in the software industry. In general terms, IoT refers to a distributed network of connected devices equipped with context-aware gadgets that enables them to see, hear, think and act intelligently[Xia+12]. They are also allowed to communicate and share knowledge, which make them smart, powerful and capable of acting independently. Thus, a device is qualified to be a smart one if: it has physical existence, can communicate, has unique global identifier, has a name and address, can sense the environment and has basic computing capabilities[Mio+12]. Currently the number of connected devices are estimated in billions, they aim to automate everything around us and are mainly targeted to increase life quality of home and business applications.

Exploiting the powerfulness of these smart devices a lot of concepts emerged:

2.1.1 Distributed Wireless Sensor Networks

2.1.2 Pervasive Computing

2.1.3 Fog Computing

2.2 Networking

2.2.1 Delay Tolerant Networking

2.2.2 Information Centric 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.

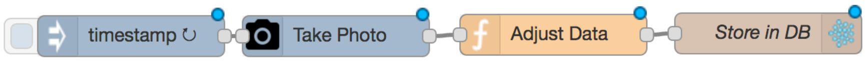

Figure 3.1: A node-red flow that stores an image in a database every time interval

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.

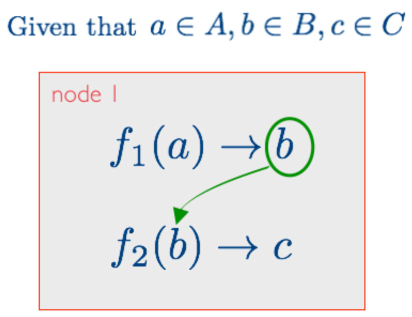

Figure 3.2: A node containing two composable flows

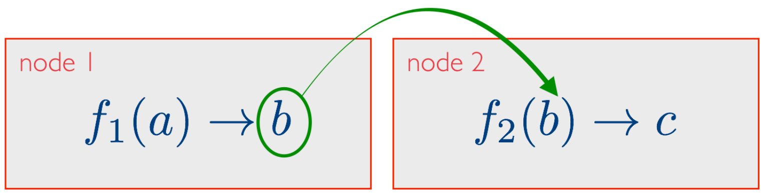

Figure 3.3: Two separate nodes having distributed composability

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.

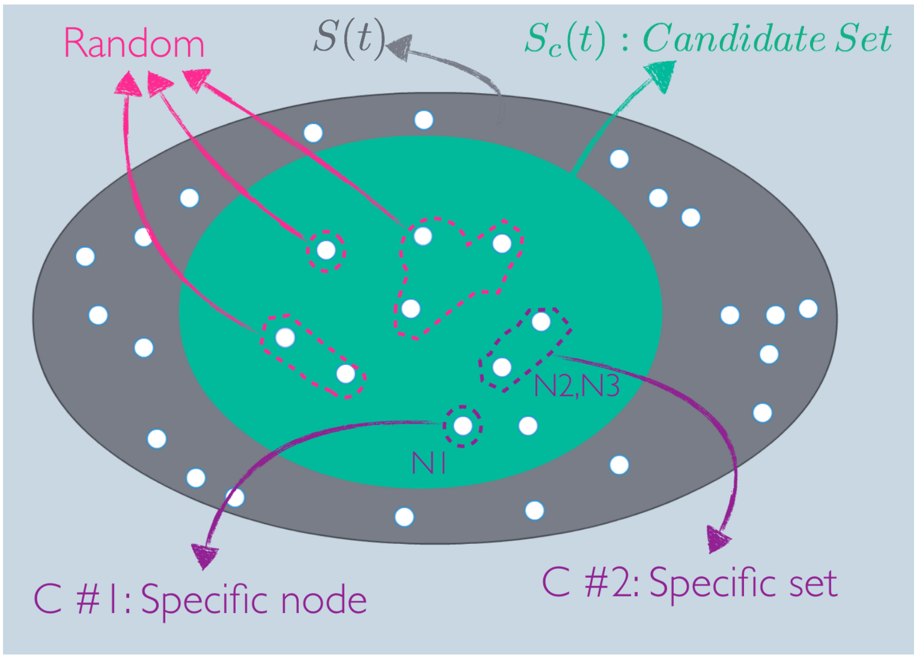

Figure 3.4: Distributing flows approaches

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 an 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 knows 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 a 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. 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 binary data inside the message body. Therefore, computation dependencies can be converted into binaries and added to the message body of their respective computations. Thus solving the obstacle of carrying dependencies mentioned earlier.
- 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 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 3.5 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.

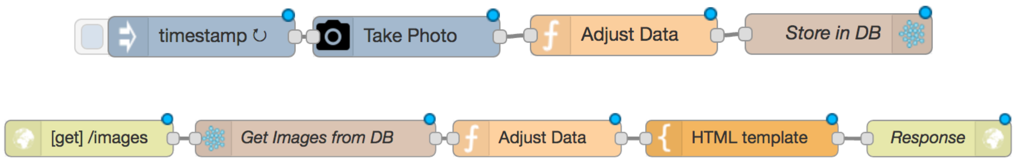

Figure 3.5: Two separate computational flows describing the IO through a database

- 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 3.6 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.

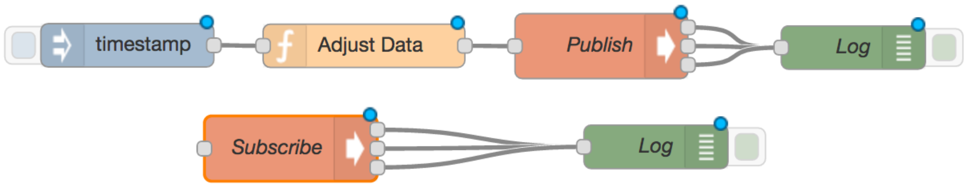

Figure 3.6

- 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 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.

3.4.1 Components

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

3.4.1.1 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 3.7

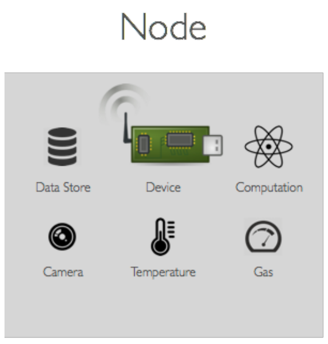

Figure 3.7: A typical node in the system

3.4.1.2 High Performance Units

CPUs in the proposed system nodes in 3.4.1.1. An example of a high processing unit is a Graphics Processing unit *GPU*.

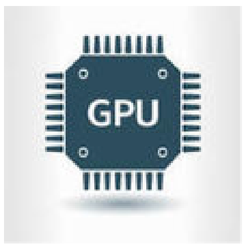

Figure 3.8: Figure denoting a Graphics Processing Unit GPU

3.4.1.3 Network

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

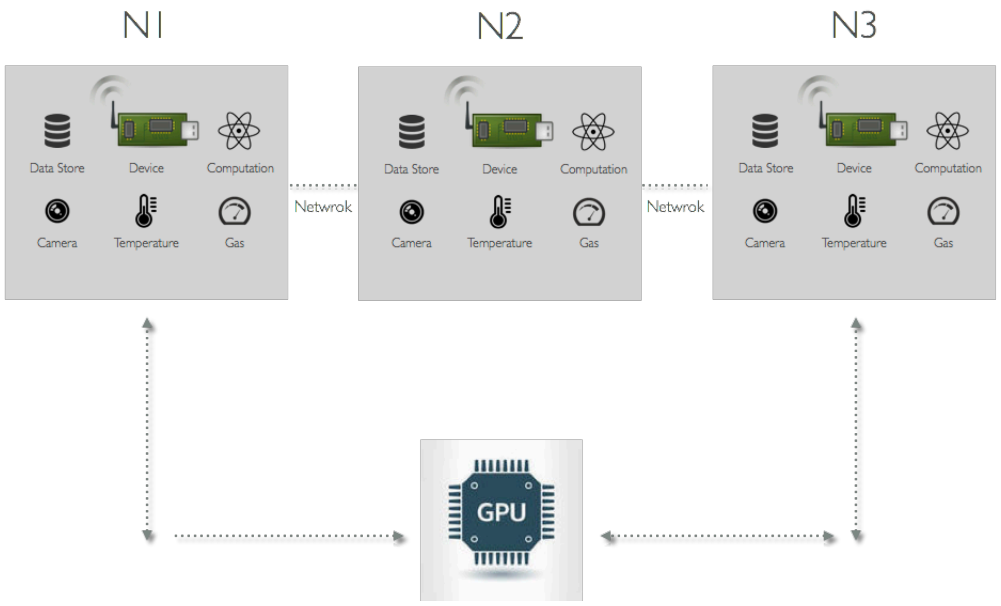

Figure 3.9: A network consisting of three connected nodes and a GPU

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

3.4.1.4 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 3.10: Figure denoting a Mobile Device

3.4.2 Connectivity and Data Flow

A Network described in 3.4.1.3, 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.

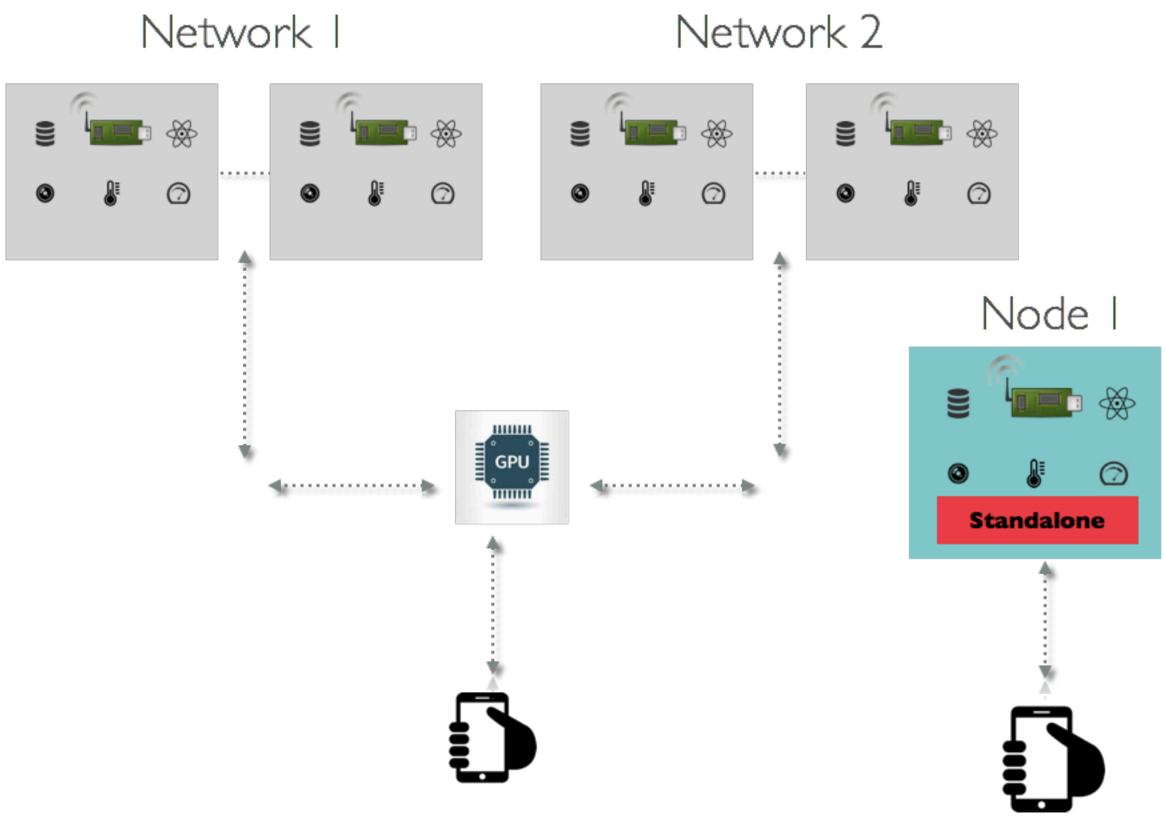

Figure 3.11: Two networks connected with a GPU and one standalone network

3.5 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. Last but not least, we introduced hardware architecture of system.

4 Approach

4.1 Requirements

4.2 Use Cases

4.3 Modeling of Input Sensor Data

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

4.3.2 Execution Model

- Which nodes should execute the data, is it all, some or a specific nodes. Also, how is the model specified in the computation meta data. - How do we know if a Computational Instance has been executed or not.

Since the goal is to push computations to the available nodes which have the capability to do so, there are three main aspects that must be addressed to achieve this goal.

- First, is to be able to express the computational requirements and resources that the computation is going to use while running on a specific node, which is also called **Computation Meta-data"**. Since we cannot make assumptions that a certain node has specific resources or specification as explained in 3.4.1.1, these requirements has to be pushed along with the computation. It includes, for instance, the amount of processing power and random access memory the computation is going to need, also, the resources required to perform such a computation. The meta-data is expressed as a *JSON* formatted object.

  include example

- Second, is the core of the meta-model, which is a model expressing the computation itself, and since node-red is the chosen platform to execute our computation on, then naturally, it is modeled into a *flow*, which is a model used by node-red to describe the capabilities and actions of a computation in JSON format. This simplifies the whole idea of making the computation meta-model travel through the network, because with node-red, the maintainer could develop a computation using the user interface of node-red, then export this computation as a flow, include the meta-data and publish it to the network. Afterwards, a node can read the meta-data and analyze if it could run this computation according to its resources, then import the flow into its node-red instance and deploy it for running. **include example**

- Last aspect in the meta-model is the output or results of the computation run. Since the node does not know what kind of data does the computation produce, therefore the maintainer must specify the way the output data is used or stored. In this case, nodes can understand whether the data needs to be sent back to the original node or just stored in a local database for further use.
  include example

4.3.3 Input Output Modeling & Flow Composability

It is believed that development of smart pervasive computing devices that uses sensors and actuators are mainly grouped into smaller disconnected architectures and thus hard to create a composite framework in which all devices can communicate and integrate[GFT10]. In order to tackle this problem and to ensure that our computational model design is dynamic and flexible enough, we must ensure that our model is composable. By Composability we mean that computational models should be able to communicate, send and receive input and output data.

Also, the models should be able to either communicate directly to each other via global referencing or through discovery in which they share the same interest. Moreover, input and output data structure should be modular to allow dynamic sending and receiving of data. Below we explain how the computational model design overcomes these challenges.

4.3.3.1 Controlled Vs Uncontrolled Communication

Controlled communication in our context basically means that we have one or more computation on more than one node, these nodes know each others global reference and wishes to exchange their input or output data. On the other hand, the uncontrolled communication suggests that there are several interested nodes in one or more computations and might not know each others global reference. To allow both types to communicate, our model proposes to use SCAMPI to transfer the message between them. In the case of controlled communication, then we add an attribute to the message with the receiving node global reference, hence, ensuring that only the node with the exact same global reference is allowed to pick up the message. In the other case of controlled communication, the global reference attribute is disregarded completely allowing all interested nodes to collect the message.

4.3.3.2 IO Specification

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

[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.

[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.

[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.