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

I confirm that this thesis type (bachelor's thesis in informatics, master's thesis in robotics, . . .) 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 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 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, 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.

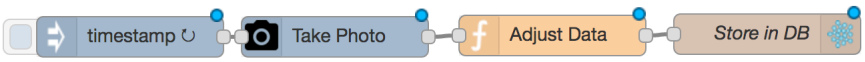

Figure 3.1: A node-red flow that stores an image in a database every time interval

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

$$f_1 : A \to B$$

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

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

Since composability should also be valid in a local or distributed environment. Thus, in the first case of local flow composability, 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.

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

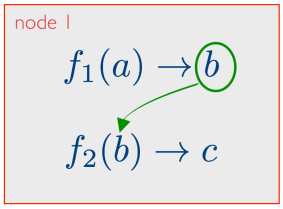

Figure 3.2: A node containing two composable flows

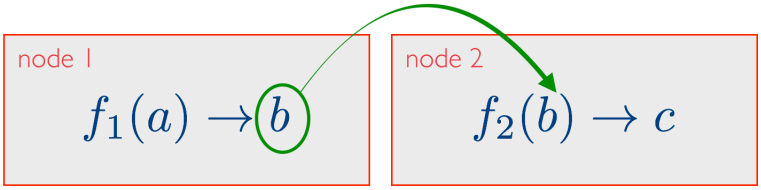

Figure 3.3: Two separate nodes having distributed composability

3.2 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 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, distributing them across nodes could have different semantics and approaches depending on the use case. Lets us first consider the two different semantics, a flow could be sent

on random basis to the nodes, and at the contrary, it could also be sent to specific nodes. This provides flexibility in applying the use case without wasting resources, in addition, it magnifies the effect of locational context in case we want to send a computation to a specific place. Moving on to approaches, a flow could be distributed across the nodes in a multiple ways explained as follows: (i) flows are pushed to all the available nodes even if they are disconnected (ii) flows are sent to n number of nodes whether they are selected or picked at random, (iii) choosing only one node to execute a specific flow. 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.

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.

3.2.2 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

3.2.3 Dependencies

Dependencies are one of the main requirements of computation execution, dropping one or more dependency would definitely stop the execution from proceeding. Thus, we need to deal with them and make sure they do not introduce any impediments during execution. 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 mandatory for the computation to run. 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.

Dynamic dependencies creates a bit of ambiguity, suppose that the dependency which is being sent by means of the communication mechanism is already on the receiving node. This introduces a versioning problem, also what makes it more complicated, is that the node does not know if it possesses an older version of the dependency or a newer one. Furthermore, imagine 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.

Nevertheless, there are multiple possible 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.

3.2.4 Resources

Resources are a different type of dependency 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 cannot be 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. 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.

3.2.4.1 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 hardware resource would be attached soon. A node should be aware of its available resources, otherwise, it would not be able to compare the requirements of incoming computations against its capabilities. Hence one possible solution, is for each node to have a specification file as a static dependency stating its available resources.

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.

3.2.4.2 Hardware Resources

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

Considering that each computation model has meta-data describing its resource consumption, then it is possible 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 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

3.3 Data Model

3.3.1 Data Types

3.3.2 Moving Data

3.3.3 IO Specification

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

3.4 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.

3.4.1 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 3.4

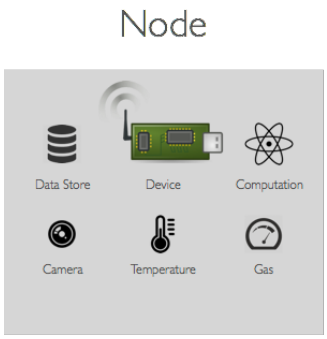

Figure 3.4: A typical node in the system

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*.

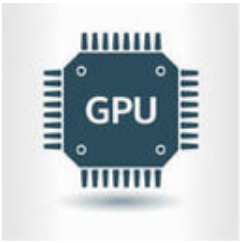

Figure 3.5: Figure denoting a Graphics Processing Unit GPU

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.

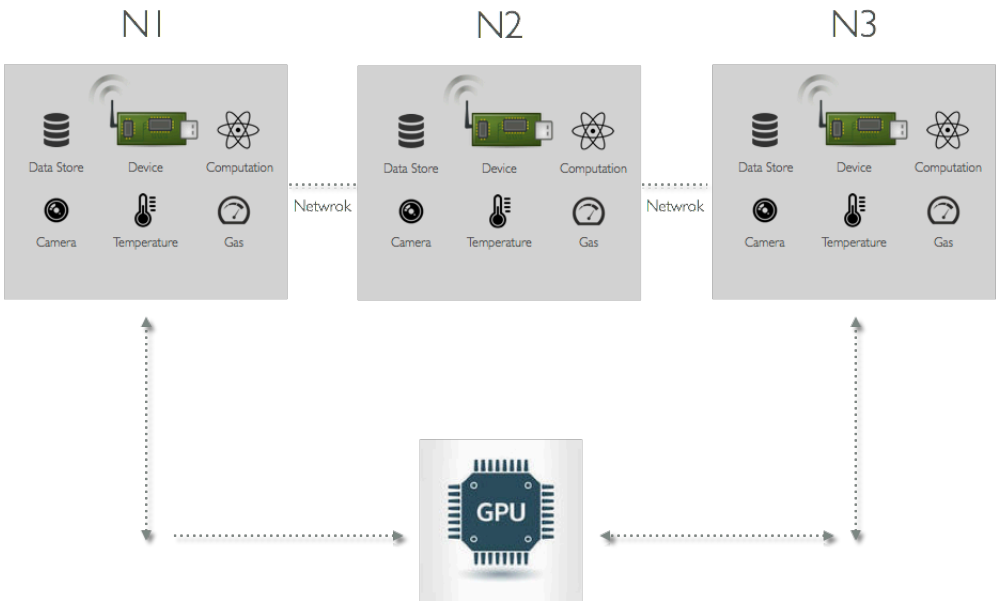

Figure 3.6: A network consisting of three connected nodes and a GPU

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

3.4.1.4 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 3.7: 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.

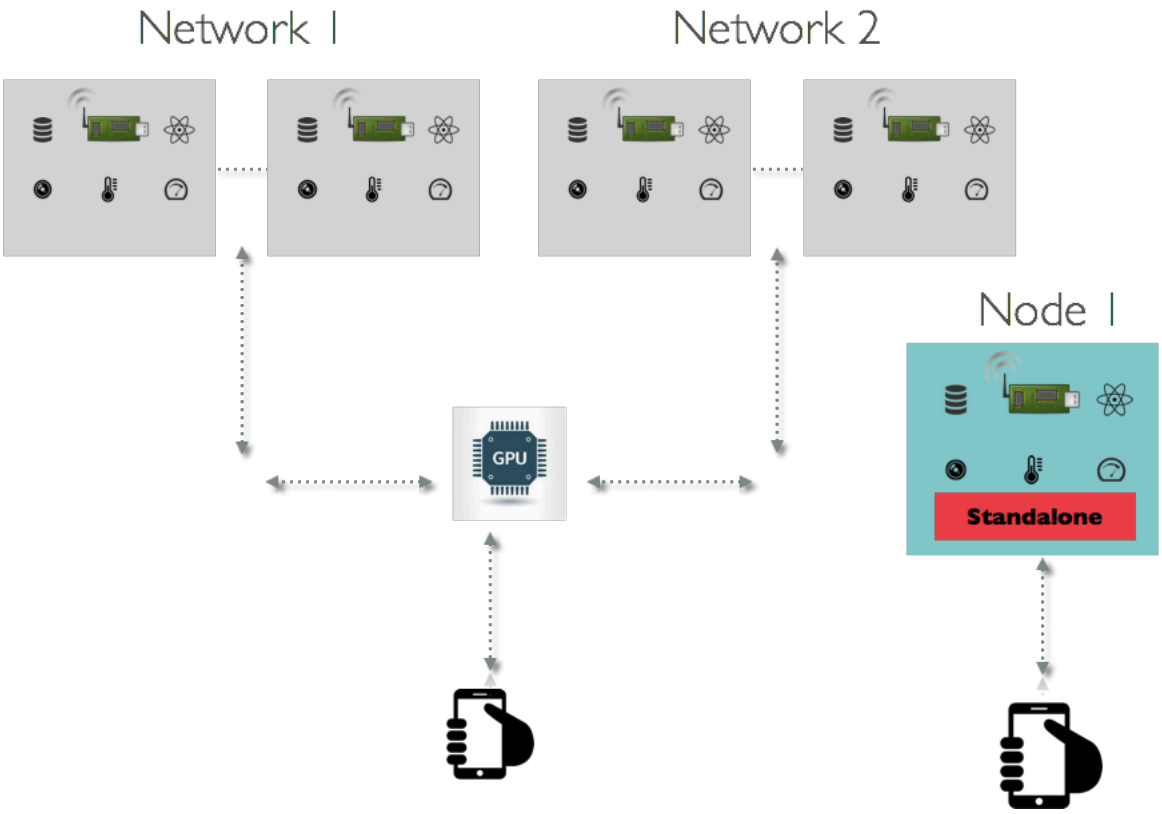

Figure 3.8: Two networks connected with a GPU and one standalone network

3.5 Summary

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

Turning now to consider the modularity of the input and output specification, as discussed above, we need the IO to have a specific structure that all nodes can understand regardless of the content. There are multiple ways in our design to specify how the IO data communicates which is explained below. But, the key idea however is that every type of IO communication can be described as a form of node-red flow in one way or another.

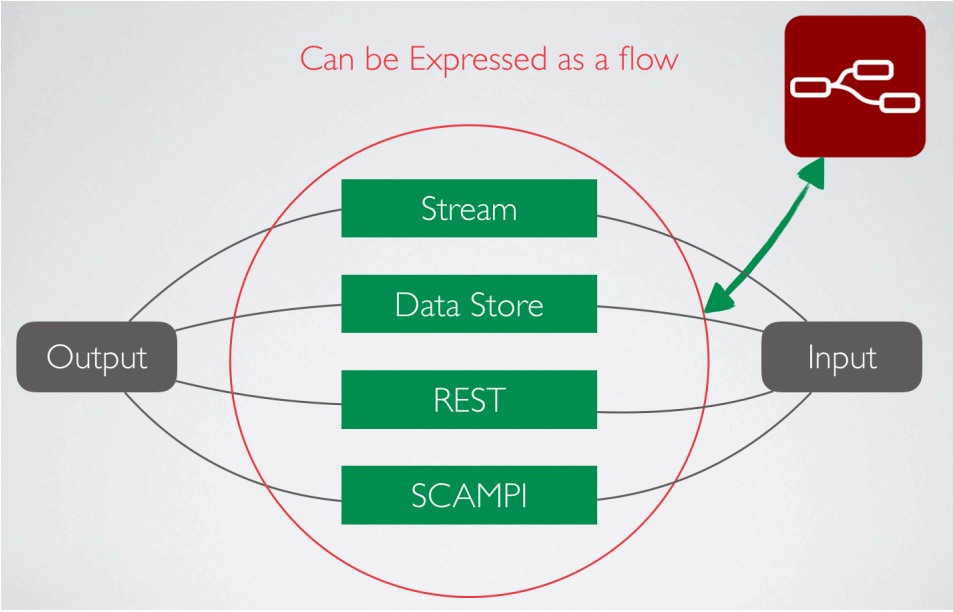

Figure 4.1: All IO types could be expressed as a flow

- 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 needs to use this is data is allowed to fetch it from this table. Since node-red allows database configuration from inside the computation flow there is no extra effort in writing an additional script to write data into a database. The example in 4.2 shows a flow that takes an image and then store it in the database, while the other pulls the data from the database upon receiving a request on a specific URL.

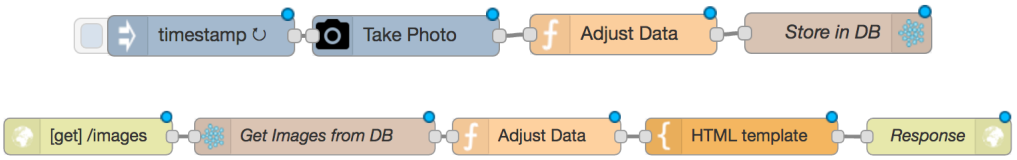

Figure 4.2: Two separate computational flows describing the IO through a database

- Another way is to use SCAMPI publish-subscribe messaging pattern to communicate, this way computations does not have to be on the same node, in fact, they do not have to be connected at all. The reason for that, is because SCAMPI can connect to mobile devices passing by, hence allowing them to make the data transfer to another 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. The figure 4.3 shows two flows as an example of this method, the first flow generates data and publishes it to SCAMPI. Then, on any node, the data could be received via subscribing to the same topic and any computation could be carried out with the data on the flow.

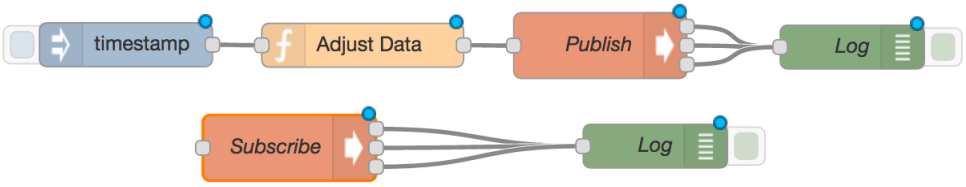

Figure 4.3

- Streaming

4.4 Data Model

4.4.1 Data Types

4.4.2 Moving Data

4.4.3 IO Specification

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

4.5 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.