UPPSALA
UNIVERSITET

UPTEC IT 14 021
Examensarbete 30 hp
December 2014

Distribution of Tasks to Processing Resources in the Internet of Things

Andreas Moregård Haubenwaller

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Distribution of Tasks to Processing Resources in the Internet of Things

Andreas Moregård Haubenwaller

In the Internet of Things (IoT), many applications focus on gathering data which can then be processed and visualized. However, such computations are usually distributed depending on parameters such as CPU and/or network load. This may mean that a significant amount of data needs to be transported over the network (either directly, or transparently using a network file system) in order for the data to be available to the node that is responsible for processing. In this thesis a method is proposed for deploying computations that can take factors such as data proximity into consideration. Thus, processing can be moved from central high-powered processing nodes to smaller devices on the edge of the network. By doing this, costs for gathering, processing and actuation can be minimized. In order to capture data dependencies among computations, and also to deploy and handle individual processing tasks in an easy way, the actor-model programming paradigm is used. To minimize the overall cost and to handle extra factors that influence the distribution of tasks, a constraint programming approach is used. The combination of these two techniques results in an efficient distribution of tasks to processing resources in IoT. Taking into consideration the NP-hard nature of this problem, we present empirical results that illustrate how this technique performs in relation to the amount of devices/actors.

Handledare: Konstantinos Vandikas
Ämnesgranskare: Justin Pearson
Examinator: Roland Bol
ISSN: 1401-5749, UPTEC IT14 021
Tryckt av: Reprocentralen ITC

Contents

1 Introduction 3

2 Background 5

3 Design 7
- 3.1 The Task Graph 9
  - 3.1.1 Actor Based Programming 10
- 3.2 Framework for Deployment 13
  - 3.2.1 Fault Tolerance 14
- 3.3 Deployment Logic 14
  - 3.3.1 Establishing Cost 15
  - 3.3.2 Other Deployment Factors 15
  - 3.3.3 Deployment Algorithm 15

4 Implementation 17
- 4.1 Actors 17
- 4.2 Framework 18
- 4.3 Deployment Algorithm 20
  - 4.3.1 Input and Output 22
  - 4.3.2 Example 22
- 4.4 Code Example - Actors and Deployment Request 24

5 Evaluation and Testing 27
- 5.1 Test 1 - Centralized vs Migrated 27
  - 5.1.1 The Scenario 27
  - 5.1.2 Specifications 28
  - 5.1.3 Results 29
- 5.2 Test 2 - Algorithm Scalability 29
  - 5.2.1 Specification 31
  - 5.2.2 Results 31

6 Related Work 33

7 Conclusion and Future Work 34
- 7.1 Future Work 34
  - 7.1.1 Algorithm 34
  - 7.1.2 Framework 34
  - 7.1.3 Tasks and Actors 35
- 7.2 Conclusion 36

References 37

1. Introduction

New devices, or things, are being connected to the Internet every day. This massive influx of devices has coined the term Internet of Things or IoT. The devices in question are often quite different from general-purpose computers and their capabilities vary. Big technology companies such as Cisco and Ericsson foresee a rapid growth in the number of connected devices in the near future, both companies think that by the year 2020, 50 billion devices will be connected to the Internet[1, 2]. These billions of devices are producing vast amounts of data and all this data needs to be processed somewhere in order for it to be useful.

A common practice for processing IoT data would be to send it to a central server or server cluster. However, this approach could lead to network congestion as the central location struggles to handle millions of requests simultaneously and might also lead to long response times if the server is far away from the source of the data.

The things in the Internet of Things are often only seen as producers of data, however many of them also have processing capabilities. If these processing capabilities could be utilized in an efficient way, then the incoming raw data could be reduced and pre-processed before it travels over the network to a central server for storage or and further processing. The time it would take to make a decision based on the data could also be drastically reduced if nearby devices could process the data and make the decision.

This paper proposes an approach where the data will be processed by IoT devices. The processing tasks that are to be carried out will be deployed based on a minimization of cost. Depending on what cost metric is used, different costs can be minimized, for example one might want to reduce the amount of data flowing through the network or one might strive to reduce latency. This will be accomplished by using a constraint programming model to minimize the cost and find a deployment for a specific task graph.

The focus of this thesis is on finding an efficient distribution of tasks in a heterogeneous cloud network comprised of an arbitrary number of connected

devices. Security and stability issues are not considered in this paper, thus the distribution will not take into account things such as sensitivity of data or the event of device failure.

2. Background

By the year 2020, major technology companies expect that the number of devices connected to the internet will number in the range of 25-50 billion. Cisco and Ericsson believe that the number of connected devices will have reached 50 billion by the year 2020[1, 2], Gartner on the other hand expect that number to be around 26 billion[3]. Despite the large disparity in numbers, they all see a huge increase in the number of connected devices. As more and more devices are being connected, new uses and markets are springing up. Some examples would be smart homes and home automation, environmental monitoring, and smart cities. All of these connected devices make up the Internet of Things, or IoT. The devices in IoT are usually smaller and less capable than generic home computers or servers, for example single-board computers such as a Raspberry Pi[4] or Arduinos[5]. Internet of things is currently getting a lot of attention, for example it is on the peak of Gartner's Hype Cycle for 2014[6]. Along with this hype and attention, several platforms specifically aimed for IoT were introduced, for example, SicsthSense[7], Xively[8], SensorCloud[9] and IoT-Framework[10, 11].

Many of these IoT platforms share the same kind of approach when it comes to data; everything should be done on the platform itself. This means that vast amounts of data will be sent to the same location. The data is usually gathered by either "push" or "pull". Pushing data means that the devices are sending data on their own accord whereas pulling is that they need to be asked for the data, see Figure 2.1. After the data has been gathered, it can be processed, visualized and sometimes also acted upon.

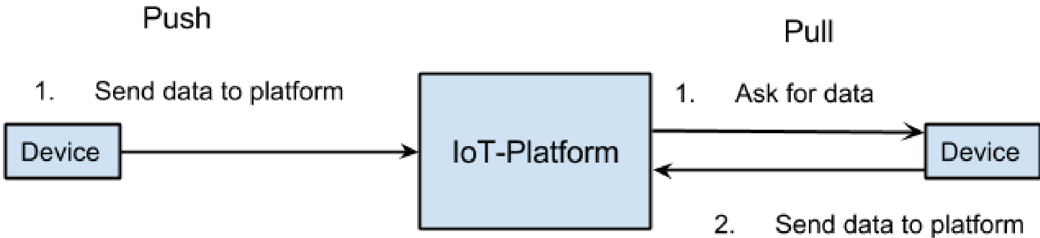

Figure 2.1. A picture showing the push and pull data gathering strategies.

With millions or maybe even billions of devices producing data that is sent to the same place for processing, network congestion and slow response times

can be a big issue. In a system with devices that are both sensing and actuating, time between sensing and actuation could be critical. If the devices are connected to a centralized IoT platform as mentioned above, the response time for an actuation based on sensory data could be long and that would lead to reduced productivity or loss of resources. Another example would be where we have a local network of sensors and actuators and in order to reach the IoT platform we need to send the data over an expensive WAN link. In both these cases it would be a lot more efficient if the data could be initially processed and/or reduced before it is sent to the IoT platform. This would reduce response times for actuation and/or reduce the cost of sending large amounts of data over expensive links. These platforms mostly share a common view when it comes to IoT devices, and that is that they are mainly seen as producers of data; the fact that many of these devices can also do processing of data is overlooked. By using the processing capabilities of the IoT devices, the data could be processed or reduced by the devices themselves or by other nearby devices. This could reduce response times since the data would be processed closer to the source and it would also lessen the strain on the IoT platform.

The next chapter will detail an approach that aims to make the handling of IoT data more efficient by utilizing the processing capabilities of IoT devices.

3. Design

The idea behind the thesis is to find a way to efficiently use the processing resources available in the many devices connected to the Internet. Platforms in which you can register devices or data streams and then gather and visualize the data, already exist. The question is how to evolve such a platform in order to gain access to the processing capabilities of the many devices. Some of the platforms mentioned in the earlier chapter might already support things such as triggers on data and actuation, however, the processing is done centrally. By moving the processing to devices that are closer to the source of the data, the response times for actuation can be reduced. If the platform is using a pull type strategy when it comes to gathering data, then that functionality could be moved to nearby devices so that data can be packed together and compressed before being sent which would reduce overall network traffic. If the platform is using a push type strategy, then the destination of the data can be changed from the platform to a nearby device so that data can be packed and compressed.

One important question is: How do we know which parts of the data handling can be moved from the central platform to the smaller, less capable, devices? Things such as permanent storage and large scale analysis will probably not work well on the smaller devices. However, gathering, packing and compressing, simple processing, and actuation will work. Some might argue that large scale analysis and harder computations can also be done in a widely distributed fashion, such as SETI@home[12] or Folding@home[13], that is however, not the focus of this thesis and will not be discussed further.

All the steps, except sending data and doing the actuation, of the IoT data flow are usually handled by the IoT-Platform. By decomposing the computation, we can find smaller pieces of work, called tasks. This is a common technique when trying to find concurrent tasks in a parallel program[14, p. 86]. In Figure 3.1 we can see that the analysis and processing has been split up and that the data flow will essentially be split up into two different paths. One path handles the storage in the repository which enables visualization and analysis of the data, and the other path relates to simple processing and triggers or decisions based on individual or aggregated data which might lead to actuation. Different parts of the data flow(gathering, processing, storage, etc.) can be

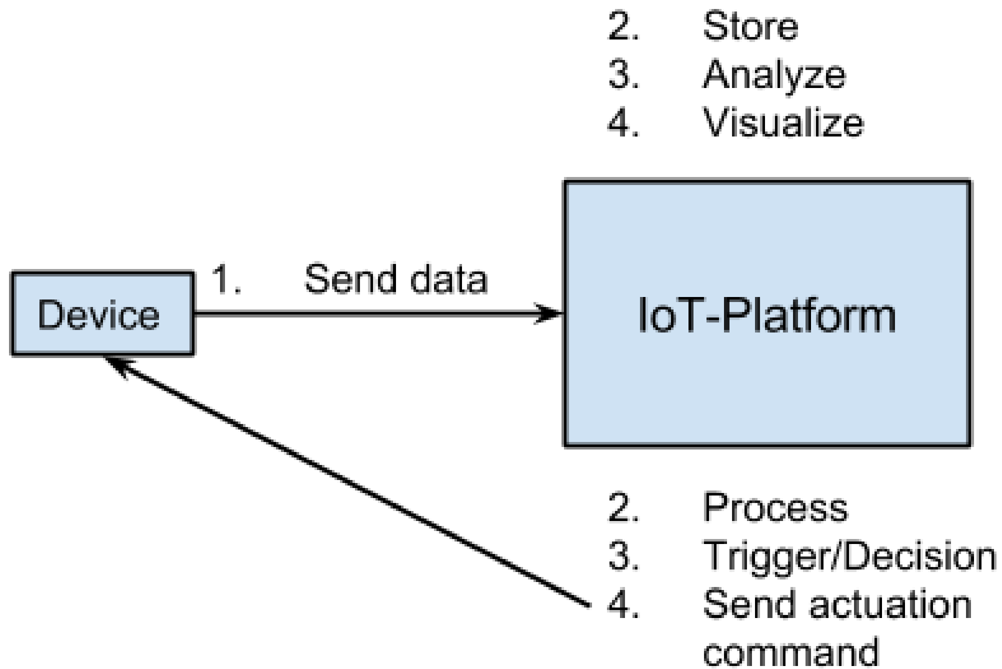

Figure 3.1. A common dataflow in IoT platforms and frameworks

seen as separate tasks. Some tasks require access to the platform or the central repository whereas some tasks require access to the device(s). By connecting the tasks and adding these prerequisites, we get a task graph in which we can see that some tasks are without prerequisites and thus they can be moved from the central platform to other devices , see Figure3.2.

With this is mind, we can break down the problem into three main areas, namely:

- The task graph, which tasks that can be moved from the platform and how they are defined (see Section 3.1)

- The deployment framework, how tasks are deployed on the individual devices (see Section 3.2)

- The deployment logic, how to decide which task goes where (see Section 3.3)

The following sections will discuss each of these areas as well as give a proposition on how these problem areas can be handled.

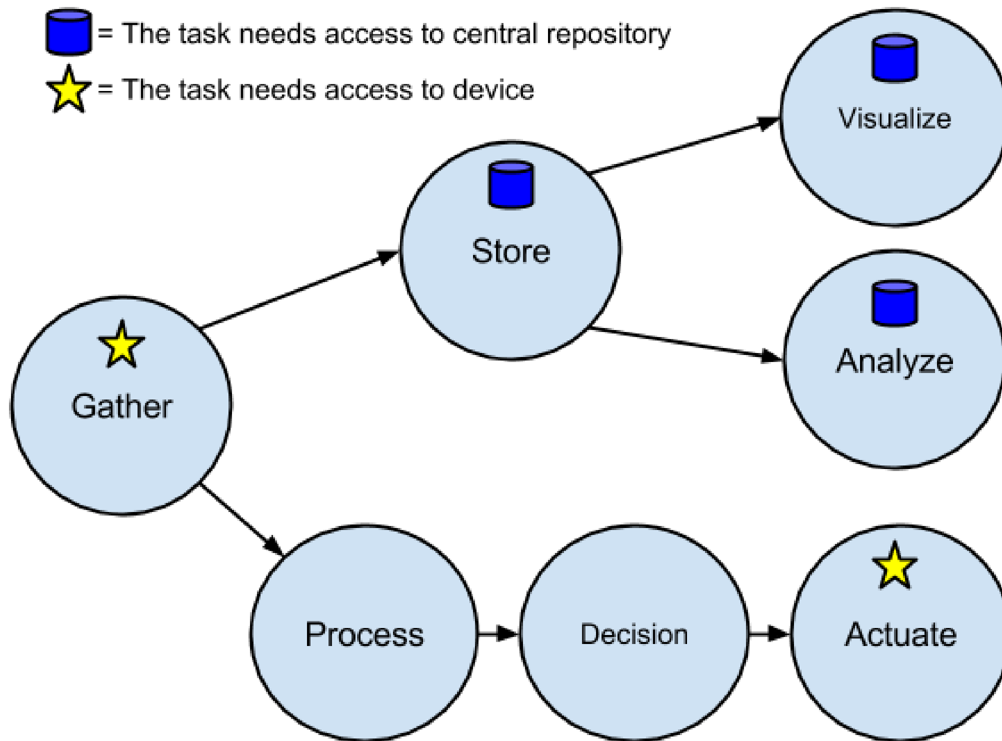

Figure 3.2. A task graph containing the tasks that are normally executed by an IoT platform or framework.

3.1 The Task Graph

As mentioned above, the flow of data can be seen as a task graph where the data originates from different devices, the data can then be aggregated and possibly reduced; After this the data is sent along for further processing, storage and also possibly acted upon. When the data is acted upon, it would most likely cause actuation in a nearby device. If the central repository is far away, then the data has to travel a long way, first it travels from the device with the sensor all the way to the repository where it is processed, and then it is returned the same way for the actuation. In order to get rid of this behavior, the data flow or task graph need to be split up. Splitting the task graph up or rearranging it, is quite easy. However, splitting up complex software programs can be a daunting task. Chances are that these tasks are not represented as individual tasks at all in a large code base such as an IoT platform, this in turn would require quite a lot of code to be rewritten. Thus, if the program was already split up into smaller distinguishable pieces or tasks then it would be a lot easier to break out individual tasks and place them on different locations(devices). The proposed idea is to use actor based programming so that each task could be represented by an actor, the message flow between actors could stay the same, but the actual location of the actor can be changed.

3.1.1 Actor Based Programming

Actors has been proposed as an easy way of creating distributed systems. Carl Hewit et.al states that "The architecture [The ACTOR architecture] will efficiently run ... languages ... requiring a high degree of parallelism."[15] As a high degree of parallelism is a major priority for running efficient and highly concurrent distributed systems, this architecture seems to be a good fit. The actor model has the philosophy that "everything is an actor". Every actor is an encapsulated computational entity that reacts to messages it receives. Because the actors only do things when messages are received, they are asynchronous by nature. This also means that the actors are inherently concurrent. The tasks in the task graph can be displayed as independent entities that are connected by arcs or edges. These tasks send messages over the arcs to other tasks which will then perform something based on the message that was received. These tasks can easily be translated into actors which makes it a good fit for this proposed IoT solution. There are several programming languages that implement the actor model in one way or another. A brief explanation about three different actor languages is found below as well as a small code example for each actor language. The code example will illustrate how two actors communicate between each other. One actor will send a ping message to the other actor which in turn will reply with a pong message.

CAL and Caltoopia

CAL(Cal Actor Language)[16] is an actor based language that was created in 2001 as a part of the Ptolemy II project. CAL uses a very high level abstraction when defining an actor. An actor in CAL has specified inputs and outputs. The inputs and outputs acts as message buffers, by reading from the buffer you consume a token. By writing to an output buffer you produce a token. You can also specify patterns on how to receive, handle, and produce tokens. The CAL language has a run-time called Calvin that enables fast and easy distribution of actors onto distributed nodes. However, Calvin lacks the ability to redeploy actors. CAL also lacks overall documentation and learning resources.

In the code snippets below you can see how actors are defined and also how they are linked using a network language. The actor definition combined with the network structure will be compiled into C files which can then be run by a run-time such as Calvin.

```
1 namespace pingpong:
2
3   actor Ping () pong_in  ==> ping_out:
4     ping := 1;
5     initialize ==> ping_out: [ping] end
6     action pong_in: [x] ==> end
```

```
7     end
8
9     actor Pong () ping_in ==> pong_out:
10      action ping_in: [x] ==> pong_out: [x] end
11    end
12
13    network pingnetwork () ==> :
14    entities
15
16      p1 = Ping();
17      p2 = Pong();
18    structure
19      p1.ping_out --> p2.ping_in
20      p2.pong_out --> p1.pong_in
21    end
22  end
```

Here, two actors are specified, `Ping` and `Pong`. The `Ping` actor has one input, `pong_in`, and one output, `ping_out`. The `Pong` actor has one input, `ping_in`, and one output, `pong_out`. These inputs and outputs are connected in the structure section. `Ping` has an initialize action which will be performed when the actor is started. The initialize action will produce a token on `pong_out`. Since `pong_out` is connected to `Pong`'s `pong_in`, `Pong` will receive a token and return the same token on the `pong_out` output which will be received by `Ping` on the `pong_in` input.

Scala and Akka

Scala is an acronym for "Scalable Language". The integration between functional programming and object-oriented concepts is the root to the scalability of Scala. Scala is used in systems made by large companies such as Twitter, LinkedIn and Intel. Scala has an actor library, however it will be deprecated in Scala version 2.11.0 and already in version 2.10.0 the default library has changed to Akka.

"Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM." - Akka.io

Akka is basically an actor toolkit for Scala. Akka is relatively new(released in 2009) and it has an active community. Akka is quite flexible in that it can use pretty much any Scala code in the actors. In Akka, actors are not as strict as they are in CAL since you do not have specified inputs and outputs in the same way as they do in CAL. Sending and receiving messages is more dynamic since you can sent different types of messages to different actors that can be changed during runtime by just sending it to another actor reference or by doing a lookup of a certain actor type. This cannot be done in CAL as the inputs and outputs are all defined during compilation. Actors are run in so

called actor systems and to send a message you need to specify the reference of another actor or to do a "lookup" of sorts for a certain actor type.

In the code snippets below you can see how an actor is defined and how it can be started in an actor system.

```
case object Ping
case object Pong

class PingActor(ref: ActorRef) extends Actor {
  ref ! Ping
  def receive = {
    case Pong =>
      println("Received Pong!")
  }
}

class PongActor() extends Actor {
  def receive = {
    case Ping =>
      println("Received Ping!")
      sender ! Pong
  }
}

object PingApp{
  def main(args: Array[String]): Unit = {
    val system = ActorSystem()
    val pongactor = system.actorOf(Props[PongActor])
    val pingactor = system.actorOf(Props(classOf[
        PingActor],pongactor))
  }
}
```

First, the objects (`Ping` and `Pong`) that we send between actors are specified, then we specify two actors, `PingActor` and `PongActor`, last we specify the actorsystem on which the actors will run and then we start them. `PongActor` is started first since it has no arguments, and then the `PingActor` is started with the reference to `PongActor` as the argument. `PingActor` will then send a `Ping` object to `PongActor`, and `PongActor` will reply with a `Pong` by sending a message to the sender reference.

Erlang

Erlang is a functional programming language that was developed with large, distributed, fault-tolerant systems in mind[17]. Erlang was developed by Ericsson and started off as a proprietary language but was later open sourced. Its use of actors is not as straight-forward as that of CAL or Akka, rather it is

the concurrency model of Erlang that follows the actor model. In Erlang they have so called "processes", which are modeled after actors.

In the code snippet below you will see some processes in Erlang.

```
1  -module(pingpong).
2  -export([pinger/1, ponger/0, start/0]).
3
4  pinger(Pid) ->
5    Pid ! {ping, self()},
6    io:format("PING!\n"),
7    receive
8      pong ->
9        io:format("Pong received!\n")
10   end.
11
12 ponger() ->
13   receive
14     {ping, Pid} ->
15       io:format("Ping received!\n"),
16       Pid ! pong,
17       io:format("PONG!\n")
18   end.
19
20 start() ->
21   Pid = spawn(pingpong, ponger, []),
22   spawn(pingpong, pinger, [Pid]).
```

In this example we have two processes that are spawned by the `start()` function. Each process is basically just running a function, one is running `pinger` and the other `ponger`. `ponger` has no arguments but `pinger` is started with the pid, which is a unique process identifier, from the `ponger` process as the argument. The pid is needed in order to send a message to another process. So `pinger` sends a tuple to `pong` that contains the atom `ping` and the pid to itself (obtained by calling `self()`), with this pid `ponger` can respond with a `pong`.

3.2 Framework for Deployment

The framework is the actual system that will deploy actors on other devices. It will not contain features that most IoT platforms already have such as registration, analysis and visualization. The main focus is on the ability to use the processing capabilities in the devices and how such a solution could possibly be combined with an existing IoT platform. The idea is that the framework will be given some actors as input and then deploy them based on some deployment algorithm. The use case would be that a user would want a specific task or set of tasks done, the user would then specify these tasks as a set of

actors. This set of actors would then be sent to the framework. The job of the framework would be to deploy these actors on IoT devices.

3.2.1 Fault Tolerance

Fault tolerance can play a huge role in a framework like this. If a device goes down that has an actor on it, then it could possibly stop the entire chain of data. Some mechanism for redeploying actors on nearby devices should exist; However, it was decided that fault tolerance is not in scope for this thesis, therefore, possible device failure will not be taken into account.

3.3 Deployment Logic

In order for the deployment and usage of actors to be efficient, the actors need to be deployed in a smart way. There needs to be some underlying logic that decides whether or not an actor should be deployed to another IoT device rather than on the IoT platform. The IoT devices will most likely not be as powerful as the server(s) running the IoT platform, so the actual processing might be slower if the actors are deployed on IoT devices. This is where data locality comes into play, the time that is lost by using a slower device for processing can be gained by not having to send the data over long, and possibly congested, network links. By deploying the actors on nearby devices we can get a more efficient message flow and at the same time reduce the work load of the IoT platform. To be able to deploy actors on nearby devices, there needs to be a way to establish the distance between devices, this can be done by setting a certain cost between devices. A cost should also exist between actors, for example if we have three actors, A, B and C. A and B will always be sending messages to each other and once in a while B will also send a message to C. The traffic between A and B is much higher than it is between B and C, so deploying A and B closer to each other is of a higher priority than deploying B and C close to each other.

The proposed idea is to assign a certain cost to all of the connections between actors as well as a cost to all of the connections between devices. Rather than just having a measurement of distance, the cost can be composed of several factors that could affect the deployment. If an actor is deployed on a device, then the actor connection cost will be multiplied by the device connection. By calculating an overall cost for the whole deployed actor system we will get a total cost. The goal will then be to minimize this cost which would result in an optimal actor to device mapping.

3.3.1 Establishing Cost

The costs between actors does not need to be specified as a certain property, rather it can be decided by the owner of the actor system, it all comes down to what part you wish to optimize. For example, the cost could be the amount of data between actors, the cost could be the priority between actors, it could also be a measure of how sensitive the link is to delay. In the same way, the cost between devices can be specified as different properties, examples of that cost could be: latency, throughput, or actual monetary cost of data being sent over that specific link. Which cost is most important is something that differs between deployment scenarios.

3.3.2 Other Deployment Factors

There could of course be other factors than just the cost that needs to be taken into consideration. By just taking the cost into consideration we are ignoring other important factors such as current CPU load and the memory usage of the devices. As all of this is in the realm of the Internet of Things, another big factor that needs to be considered while deploying the task graph is the ability to sense the outside world and gather data. In a task graph related to the Internet of Things, there will, most likely, exist a number of tasks that are using sensors or actuators in one way or another. This means that those specific tasks can only be deployed on a specific device or a subset of devices. The same is true for storing data, not all of the devices will have the access to a data store. In summary, this means that the deployment cannot be based purely on the cost, rather is must take a number of other factors into consideration.

3.3.3 Deployment Algorithm

In order for the framework to deploy actors, it needs to have an algorithm that will find the optimal deployment. This optimal deployment will be based on the minimal cost of the total deployment that does not break any other deployment factors. The problem can be specified as such: Assign n actors on m devices. Several actors can be placed on the same device. Each actor has a flow to all other actors and each device has the cost to communicate with all other devices, the flows are then multiplied by the communication costs in order to get the total cost. This problem quite similar to the Quadratic Assignment Problem which is a well known operations research problem. The flow and the communication cost can be seen as two matrices, the actor matrix $F = n \times n$ and the device matrix $D = m \times m$. x is a general function from F to D, or the assignment of actors to devices. The total cost of the deployed task

graph can be expressed as Equation (1).

$$\sum_{i \in n} \sum_{j \in m} F(i,j) * D(x(i), x(j)) \qquad (1)$$

Most actors will only have one or two connections to other actors so the F matrix will most likely be quite sparse. Therefore we can model the cost by the sum of all of the connections or arcs between nodes. An arc $a(i,j)$ exists iff $F(i,j) \neq 0$. Let A denote the set of all arcs. The cost can now be expressed as seen in Equation (2).

$$\sum_{a(i,j) \in A} F(i,j) * D(x(i), x(j)) \qquad (2)$$

This total cost can now be minimized by placing actors on different devices until the optimal(minimal) solution is found. This is what the deployment algorithm will have to do.

4. Implementation

In the beginning of the project, CAL was used as the programming language for the framework. However, the lack of documentation and learning resources led to a very steep learning curve. CAL and Calvin also lacked the ability to dynamically relocate actors between devices, this led to a change of programming language. The choice was then between Erlang and Scala. Scala was seen as a good choice since it runs on the JVM as many devices are able to run the JVM. It also sets a limit on the scope of devices that can be used and seen as a "processing resource" in this thesis. Thus, any device that can run a JVM is seen as a processing resource. Akka is also quite new (Initial release was 2009), and it has a very active community and extensive documentation. Erlang was also seen as a good choice, but Scala was chosen over Erlang. The basic framework for handling the deployment of actors was therefore implemented in Scala and Akka. The algorithm for finding the optimal deployment was also implemented in Scala. These implementations will be explained more in depth in this following chapter.

4.1 Actors

The actors that are to be deployed in the system are not restricted. That is, anything that is possible in Akka should also be possible in this system. However, in order for it to be possible to move actors and possibly also replicate actors, there is some additional functionality that must be provided. In Akka, the references to other actors are based on where they are deployed, so if an actor is redeployed somewhere else, then the reference needs to be updated. Unlike CAL, Akka does not have static inputs and outputs, the message passing is a lot more flexible. This can prove to be a problem when we need to separate messages of the same type but are from different senders, especially if these senders can change their actor references. Thus, the framework needs a way to separate different inputs independent of the actor reference. To be able to do this, we need to be able to register inputs and outputs in each actor and we also need to be able to specify an input number and an output number for each message sent. To enable this, a specific trait ,`IOTrait`, was made. A trait in Scala is similar to an interface in Java, they are used to defined object types by specifying method signatures. A requirement for the seperation of

inputs to work is that every actor needs to extend this `IOTrait` and they need to use it for every message sent, the actors also need to have two `receive` cases, `RegisterInput`, and `RegisterOutput`.

In the code below, you can see that the actor extends the `IOTrait` and that it specifies that the actor will have two inputs and one output by calling `addInputs (2)` and `addOutputs(1)`. The actor also has the `receive` cases `RegisterOutput( output_nr, input_nr, actor)` and `RegisterInput(input_nr, actor)`. These `receive` cases are used by the system when the actors are deployed in order to connect the different inputs and outputs. The actor also has two other `receive` cases, namely `IntMessage(0, value)` and `StringMessage(1, message)`, these are case classes, each case class contains the input port and the message itself. So this actor specifies that on input 0 it will receive `IntMessage` and on input 1 it will receive `StringMessage`. Upon receiving such a message, it will send a `StringMessage` to output 0. The `StringMessage` will be sent to actor reference and input port which had been previously registered by the system with `RegisterOutput`. This is done calling `Outputs(0)_2` which will get the second element of the output tuple that exists in the Outputs array and it will provide the actor reference to that output. In the same way, the input port is retrieved by getting the first element of the output tuple by calling `Outputs(0)_1` , more on how inputs and outputs are registered in section 4.2

```
1  class TemplateActor extends Actor with IOTrait {
2    addInputs(2)
3    addOutputs(1)
4    def receive = {
5      case RegisterOutput(output_nr, input_nr, actor) =>
6        regOutput(output_nr, input_nr, actor)
7      case RegisterInput(input_nr, actor) =>
8        regInput(nr, actor)
9      case IntMessage(0,value) =>
10       Outputs(0)_2 ! StringMessage(Outputs(0)_1, "
           IntMessage from Input 0")
11     case StringMessage(1,message) =>
12       Outputs(0)_2 ! StringMessage(Outputs(0)_1, "
           StringMessage from Input 1")
13     }
14   }
15 }
```

4.2 Framework

As mentioned in section 3.2, the framework will essentially just deploy actors to devices. The framework consists of three actors, Deployer, Scheduler and

Landmark. A user will send a request for a set of actors to be deployed, this request will then be handled by the Deployer. The deployment request should contain five things:

- The number of actors
- An array of strings containing the class names of the actor classes(The framework assumes that the actor classes are available).
- An array of options containing the arguments of the actors.
- A list of tuples containing static actor placement. Each tuple contains an actor number and a device address.
- A list of connections between actors. Each connection contains information about the weight or flow of the connection as well as source and destination information.

In order for the deployment to work, the actors should meet the requirements discussed in section 4.1. The array of arguments does only allow for one argument for each actor class. However, the argument could be a list, which implicitly allows for more arguments. This array can be empty. The list containing the tuples of static actor placement can also be empty. The list of connections is very important, this is where the inputs and outputs are specified and it is also where the weight is specified for each link between actors. The weights are needed for the distribution algorithm and the inputs and outputs are needed in order for the system to register the connections when deploying actors. See Figure 4.1 and the explanation below on how a deployment request from a user is handled.

Explanation of deployment request:

1. User sends deployment request
2. Deployer asks Landmark for deviceinfo
3. Landmark replies with deviceinfo
4. Deviceinfo and actorinfo is sent to Scheduler
5. Scheduler replies with an actor to device mapping
6. Deployer deploys the actors on to devices as specified in the mapping

The Deployer is the main actor in the implementation. It receives the deployment requests and handles them all the way to deployment. Upon receiving a deployment request containing the actor information, the deployer will ask the Landmark for information about the devices. The Landmark has a matrix of costs between devices and a list of addresses to connected devices, this is the information that is sent back to the Deployer. The cost matrix in this scenario is made out of latencies between the devices, but, as previously mentioned, this cost can differ from deployment scenarios and implementations. It is assumed that if this framework is integrated with another IoT platform, then the

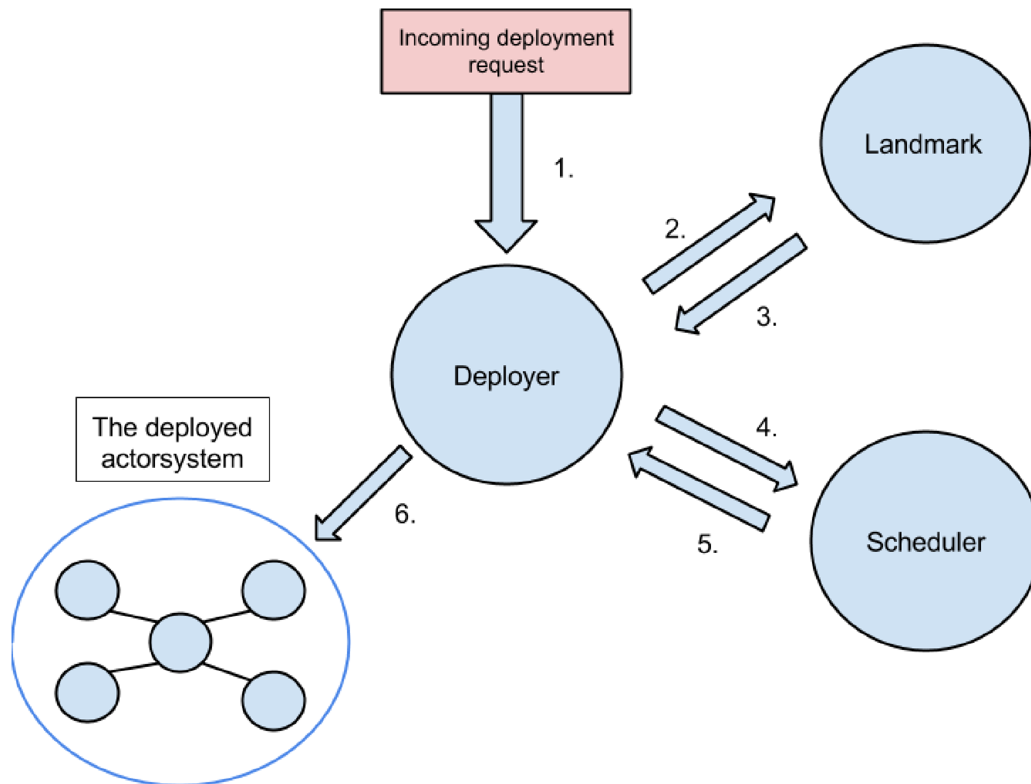

Figure 4.1. The order in which a deployment request is handled

role of the Landmark actor should be replaced by the underlying system that has the information about the devices and their capabilities.

After the Deployer has gotten the device information, then it will forward it along with the actor information to the Scheduler. The Scheduler actor is the actor that runs the deployment algorithm, more about this in the next section 4.3. The Scheduler will return a placement to the Deployer which will deploy the specified actors on the specified devices. The Deployer will also register all of the connections by sending a `RegisterInput` and a `RegisterOutput` for each connection contained in the list of connections.

4.3 Deployment Algorithm

The implementation of the algorithm is based on the ideas from section 3.3. The algorithm was implemented using constraint programming with the aid of the Scala toolkit OscaR [18]. Constraint programming is a programming paradigm for solving combinatorial problems. This is done by specifying constraints and decision variables. Decision variables have domains, and constraints can specify which values in these domains that are allowed, they can

be seen as a requirement on a decision variable. This requirement must hold in order for the solution to be valid. If we have a finite set of decision variables and constraints, we have a constraint satisfaction problem (CSP). We can then try to solve this CSP by assigning variables so that the constraints are met. This is done by searching the solution space. One naivé way is to simply take the first variable in the domain and assign it and then see if the solution is valid or not. By doing this for all decision variables and all values in the domains we will have exhausted the search space and if there was a solution, it is guaranteed that it would have been found[19].

Constraint programming was used as an approach because of the ease of adding and changing constraints that are not directly connected to the total cost of the distribution. The optimal deployment would be the one with the lowest cost. However, by adding other constraints, such as static actors, the solution could be different. Using constraint programming has the added bonus of making the algorithm very flexible so that it is easy to add, remove and change constraints between deployment scenarios. In order to get the optimal solution, the cost needs to be minimized. The built-in minimization in OscaR[20] for constraint programming was used, it uses the branch and bound branching strategy. There was not much time spent trying to find better heuristics or branching options so the default settings for the minimization were used.

The problem is still the same as specified in Section 3.3.3; Assign n actors on m devices so that the overall cost is minimized. The cost between actors i and j is $F(i,j)$ and the cost between devices a and b is $D(a,b)$. Initially, no constraints are specified. This means that we have to check the whole search space in order to find the optimal solution. An array x of size n contains the decision variables. The decision variables can be seen as the tasks and their domain is the set of possible devices on which they can be deployed. Now the problem looks as such:

$$min \sum_{a(i,j) \in A} F(i,j) * D(x(i), x(j))$$

Possible constraints might be that some tasks can only exist on certain devices and also that some devices can only handle a certain number of tasks. For example: $x(y) = z, y \in n, z \in m$ would mean that task y must be placed on device z. Another example would be where each device can only hold one task, then we can put x under the `alldifferent` constraint, this constraint says that all decision variables take distinct values, or in other words, every actor must be placed on different devices.

4.3.1 Input and Output

There are two different inputs, `actorinfo` and the `deviceinfo`. The `actorinfo` contains information relevant to the actors and actorsystem and the `deviceinfo` contains information relevant to the devices. The `actorinfo` is what we get from the user, so it is the info contained in the deployment request. The `deviceinfo` contains the cost matrix related to the devices and in this implementation this is provided by the Landmark actor. With these inputs, the algorithm will find an optimal placement and will return this as the output. So the output is an actor to device mapping that details which actor should be deployed on which device.

4.3.2 Example

In this example we have an actor system containing four actors and we have six devices. The actor system with the corresponding flows between actors can be seen in Figure 4.2 and the devices can be seen in Figure 4.3.

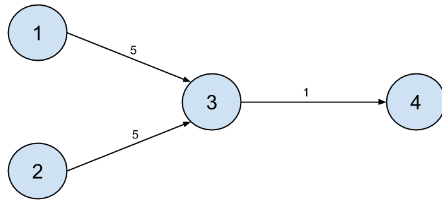

Figure 4.2. The actorsystem

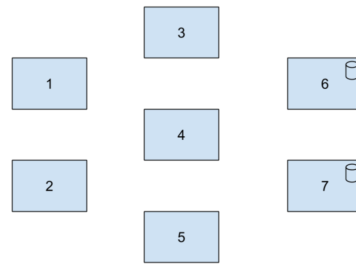

Figure 4.3. The available devices that are capable of processing

Actors one and two are producing data from sensors and they have to be placed on specific devices, namely devices one and two. Actor three will compress the data and send it to actor four that will store it. Actor four needs to be placed on a device that has access to the data store, in this case that is device six or seven. The total cost of the deployment depends on the three arcs: (1,3), (2,3), (3,4). We know that actor one and two are placed on devices one and two, we also know that actor four must be placed on six or seven. That means that the domains of the actors are: Actor1 = {1}, Actor2 = {2}, Actor3 = {1,2,3,4,5,6,7} and Actor4 = {6,7}. We can now start the search by taking the first value in the domains and calculating the cost. In order to find the minimized cost, we need to check all solutions. The cost can now be expressed as:

$$F(1,3) * D(x(1),x(3)) + F(2,3) * D(x(2),x(3)) + F(3,4) * D(x(3),x(4))$$

We know that x(1) = 1 and x(2) = 2. We also know the flow between actors, F(1,3) = 5, F(2,3) = 5 and F(3,4) = 1. Now the cost can be simplified as:

$$5 * D(1,x(3)) + 5 * D(2,x(3)) + D(x(3),x(4))$$

The latency matrix D can be seen in Table 4.1. The possible solutions and the cost for each solution can be seen in Table 4.2.

Table 4.1. *The latency matrix D*

Device1	Device2	Device3	Device4	Device5	Device6	Device7
0	20	20	20	40	50	70
20	0	40	20	20	70	50
20	40	0	20	50	30	50
20	20	20	0	20	30	30
40	20	40	20	0	50	30
50	70	30	30	50	0	20
0	0	50	30	30	20	0

Table 4.2. *These are the possible solutions for this deployment*

Actor1	Actor2	Actor3	Actor4	TotalCost
1	2	1	6	150
1	2	2	6	170
1	2	3	6	330
1	2	4	6	230
1	2	5	6	350
1	2	6	6	600
1	2	7	6	620
1	2	1	7	170
1	2	2	7	150
1	2	3	7	350
1	2	4	7	230
1	2	5	7	330
1	2	6	7	620
1	2	7	7	600

From the results, we can draw the conclusion that the optimal deployment would be either Actor1 = {1}, Actor2 = {2}, Actor3 = {1}, Actor4 = {6} (See Figure 4.4) or Actor1 = {1}, Actor2 = {2}, Actor3 = {2} and Actor4 = {7} (See Figure 4.5)

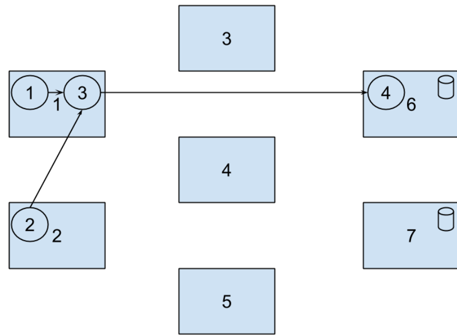

Figure 4.4. Deployment 1

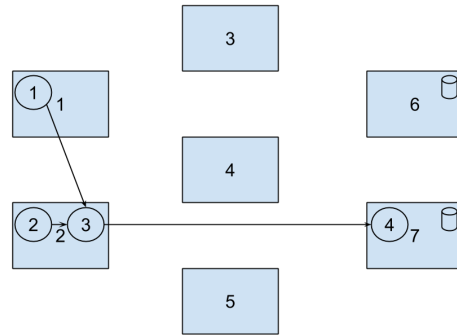

Figure 4.5. Deployment 2

4.4 Code Example - Actors and Deployment Request

In this example, we are going to deploy two actors using the framework and the actors should be specified according to the requirements listed earlier. The two actors are simple Ping and Pong actors. For the deployment code and actor code, see below:

```
1  package pingpong
2  import akka.actor._
3  import IOPackage._
4
5  class PingActor extends Actor with IOTrait {
6    addInputs(1)
7    addOutputs(1)
8    def receive = {
9      case RegisterOutput(output_nr, input_nr, actor) =>
10       regOutput(output_nr, input_nr, actor)
11       Outputs(0)_2 ! StringMessage(Outputs(0)_1, "PING")
12     case RegisterInput(nr, actor) =>
13       regInput(nr, actor)
14     case StringMessage(0,"PONG") =>
15       println("PING Succesful!")
16     case _ =>
17       // Unknown message
18     }
19   }
20 }
21
22 class PongActor extends Actor with IOTrait {
23   addInputs(1)
24   addOutputs(1)
25   def receive = {
26     case RegisterOutput(output_nr, input_nr, actor) =>
27       regOutput(output_nr, input_nr, actor)
28     case RegisterInput(nr, actor) =>
29       regInput(nr, actor)
30     case StringMessage(0,"PING") =>
```

```
31        Outputs(0)_2 ! StringMessage(Outputs(0)_1, "PONG")
32      case _ =>
33        // Unknown message
34      }
35    }
36  }
```

As soon as `PingActor` has an output registered, it will send a `StringMessage` to that actor containing the string "PING". `PongActor` will only respond if it gets a `StringMessage` on input 0 that has the message "PING". It will respond by sending a `StringMessage` containing "PONG" to its output. Now, to deploy these actors we need to create a deployment request and send it to the Deployer. The code for that can be found below:

```
1   import scheduler._
2   import akka.actor._
3   
4   object PingPongDeployment {
5           def main(args: Array[String]): Unit = {
6            sendDeployment()
7           }
8           def sendDeployment(): Unit = {
9   
10                  val classes: Array[String]   = Array("
                        pingpong.PingActor", "pingpong.
                        PongActor")
11                  val args: Array[Option[Any]] = Array(
                        None, None)
12                  val placement: List[(Int,Address)] =
                        List((0,AddressFromURIString("akka.
                        tcp://ClusterSystem@10.0.0.12:6000"))
                        )
13  
14                  val pingtopong = new Link(output_actor =
                         0, output_nr = 0, input_actor = 1,
                        input_nr = 0, flow = 1)
15                  val pongtoping = new Link(output_actor =
                         1, output_nr = 0, input_actor = 0,
                        input_nr = 0, flow = 1)
16                  var connections = List(pingtopong,
                        pongtoping)
17  
18                  val actorinfo = new ActorInfo(2, classes
                        , args, placement, connections)
19  
20          val system = ActorSystem("ClusterSystem", config
                )
21                  val deployerRef = system.actorSelection
                        ("akka.tcp://ClusterSystem@10
                        .0.0.1:6000/user/Deployer")
```

```
22                  deployerRef ! DeploymentRequest(
                        actorinfo)
23          }
24  }
```

Here the deployment request has been created, and it is sent to the Deployer actor that is running on the clustersystem "ClusterSystem" on the IP 10.0.0.1 and port 6000. In the request we specify that we have two actors, PingActor and PongActor from package pingpong, that they have no arguments, that PingActor must be placed on the clustersystem "ClusterSystem" on device with IP 10.0.0.12 and port 6000 and that we have two links, one from PingActor to PongActor, and another one from PongActor to PingActor.

5. Evaluation and Testing

For evaluation, two different tests were performed. The first test was to evaluate the conecpt of migrating processing to the edge rather than sending all data to a central repository for processing. This also tested all the functionality of the system. The second test was to see how much the algorithm can handle, how many tasks and devices it can find a solution for in a reasonable amount of time.

5.1 Test 1 - Centralized vs Migrated

In this test two different processing models will be compared, the centralized(Figure 5.1) and the migrated(Figure 5.2). The centralized version is a common scenario when dealing with IoT-data today, all data is sent to a centralized server or server cluster. The migrated version is a simplified version of what is proposed in this paper. As you can see in Figure 5.1, the data is sent directly from the source to the centralized server or cluster, in Figure 5.2 the data is first sent to an intermediate processing resource. This processing resource can pre-process the data and for example filter out unnecessary data or trigger a response.

5.1.1 The Scenario

Three sources are producing data and sending it for processing every second. Due to the large amount of data, not all of it can be stored. Every data point is processed to look for anomalies and then an average of all of the data during one minute is stored in the datastore. The three sources are all connected to the same local network, the datastore (a server) is located elsewhere and it needs to be communicated to over an expensive WAN link. There are more devices connected to the local network that are also capable of processing. In the centralized scenario, all of the data would be sent to the server. In the migrated scenario, the data would be pre-processed by a device in the local network and only the minute average would be sent to the server over the WAN link.

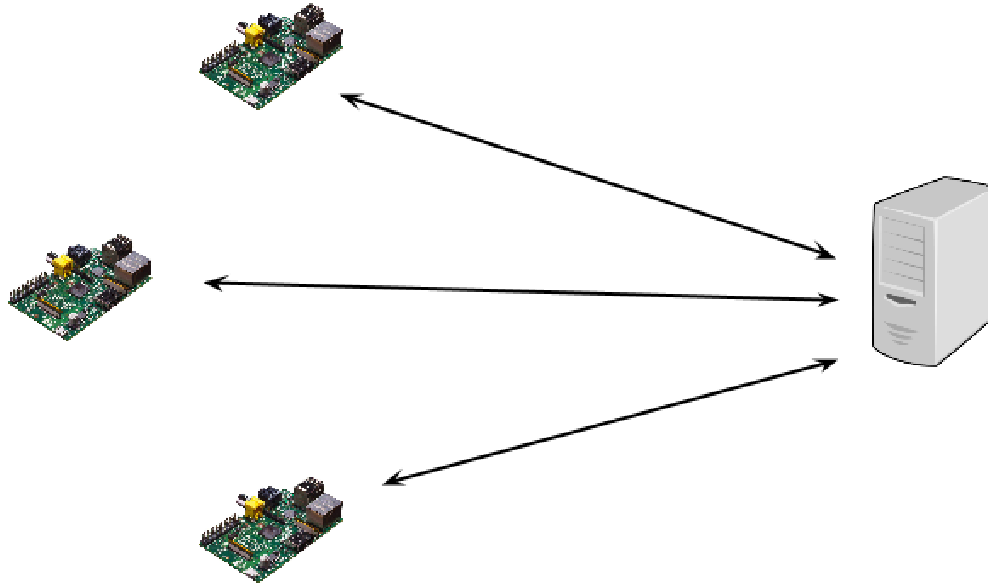

Figure 5.1. The communication path with a centralized server

5.1.2 Specifications

The testbed consisted of four Raspberry PIs and one desktop.

Desktop:
Intel Core2 Duo CPU E8400 3.00GHz x 2
8 GB RAM
Running Ubuntu 13.10

Raspberry Pi model B:
ARM1176JZF-S 700 MHz
512 MB RAM
Running RASPBIAN Debian Wheezy June 2014

The three data sources were RPI1, RPI2 and RPI3. The server was the Desktop and the other device in the local network was the RPI4. The netem software was used in order to emulate latency between devices. The latency was set up according to the following table:

	RPI1	RPI2	RPI3	RPI4	Desktop
RPI1	0ms	40ms	40ms	20ms	100ms
RPI2	40ms	0ms	40ms	20ms	100ms
RPI3	40ms	40ms	0ms	20ms	100ms
RPI4	20ms	20ms	20ms	0ms	80ms
Desktop	100ms	100ms	100ms	80ms	0ms

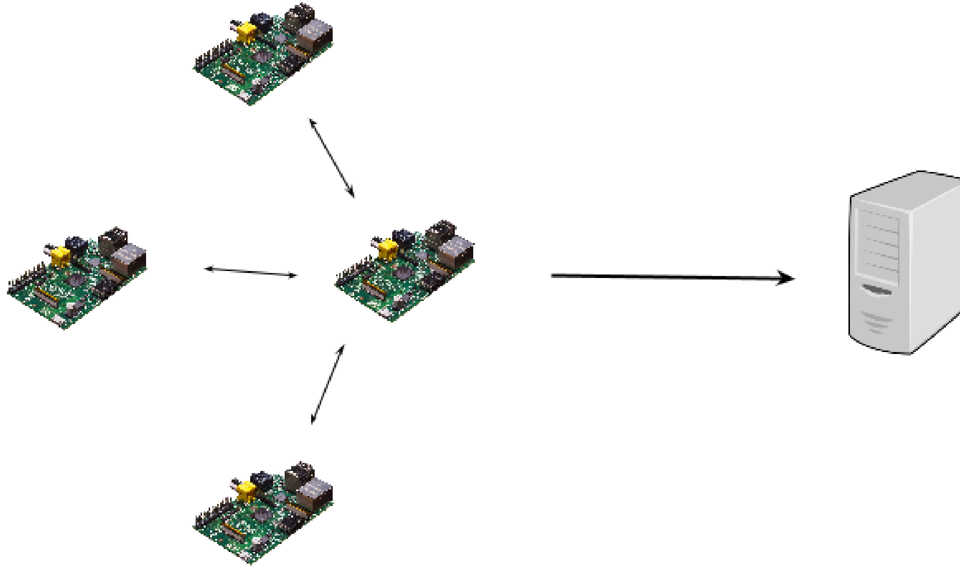

Figure 5.2. The communication path if the actor is migrated to an intermediate Raspberry Pi

5.1.3 Results

Response times	RPI1	RPI2	RPI3
Centralized	273ms	356ms	251ms
Migrated	55ms	57ms	55ms

Messages	Centralized	Migrated
Total	10897	11086
Over WAN	10897	177
Over LAN	0	10969
% Over WAN	100%	1,06%
% Over LAN	0%	98,94%

5.2 Test 2 - Algorithm Scalability

As the problem is NP-hard, it is interesting to see how many actors and how many devices the algorithm can actually handle. This test tries different numbers of actors and devices and measures the time it takes for algorithm to find the optimal solution. The timeout for this test is set at one(1) hour. The tests are also divided into two cases, the easy and the hard case.

The common criteria for the test is that the first and the last actors are static, the costs of the arcs between actors are random between 1-100 and the costs

of the links between devices are random between 1-100. The number of arcs between actors and how they are connected is what distinguishes the easy case from the hard case.

In the easy case, each actor has an arcs to the next one, except for the last actor(see Figure 5.3).

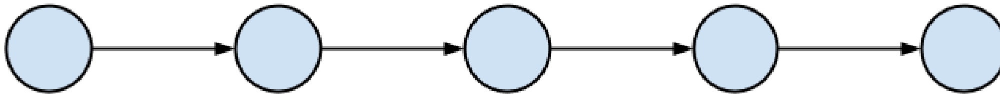

Figure 5.3. The actors and their arcs in the easy case

In the hard case, each actor has an arc to the next one, including the last one; This means that there is a circular flow. In addition to this, there are arcs from random actors to other random actors(see Figure 5.4). In total, the number of arcs in the hard case is the number of actors times two, whereas in the easy case the number of arcs is the number of actors minus one.

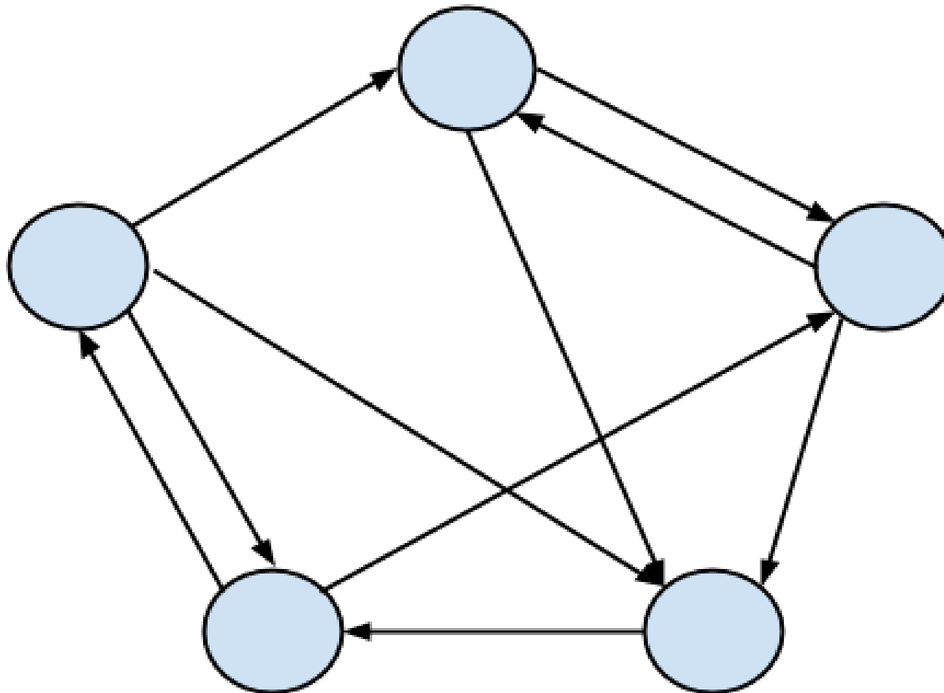

Figure 5.4. The actors and their arcs in a hard case scenario

5.2.1 Specification

The tests were performed on the following machine:
Intel Core i5-3570K @ 3.40GHz
8 GB RAM
Running Windows 8.1

5.2.2 Results

The results, as seen in Table 5.1 and Table 5.2 are the average of 4 test runs. All numbers under 50 ms were removed as they differed a lot between runs and the end result is still that they were below 50 ms and it can be seen as a negligible amount of time. The results show that for smaller numbers of actors, the deployment is found very fast, however, as the number of actors grow, the time it takes to find a solution increases significantly. That the algorithm might take 10 minutes to find a deployment is still a reasonable amount of time, since the actors might be deployed for days, weeks or even years. However, when the algorithm timed out, the time it would take to find a deployment is just too long. In such cases, one might want to change the algorithm and try some other heuristics so that an approximate solution is found instead, or simply take the best solution that has been found after a certain timeout. This would mean that the deployment is not guaranteed to be optimal. It should also be noted that most actor systems will most likely be closer to the easy case scenario than the hard case.

Table 5.1. *Easy Case:*

Devices Actors	5	10	15	20	25
5	< 50 ms	< 50 ms	< 50 ms	< 50 ms	131 ms
10	< 50 ms	< 50 ms	< 50 ms	< 50 ms	182 ms
15	< 50 ms	< 50 ms	68 ms	107 ms	2.638 s
20	< 50 ms	< 50 ms	99 ms	201 ms	10.069 s
30	< 50 ms	52 ms	123 ms	599 ms	79.755 s
50	< 50 ms	275 ms	800 ms	17.541 s	≈3 minutes
100	< 50 ms	329 ms	7.412 s	37.852 s	≈14 minutes

Table 5.2. *Hard Case:*

Devices Actors	5	10	15	20	25
5	< 50 ms	< 50 ms	< 50 ms	71 ms	5.634 s
10	< 50 ms	245 ms	57 ms	246 ms	≈3 minutes
15	< 50 ms	101 ms	1.942 s	≈9 minutes	≈47 minutes
20	< 50 ms	482 ms	2.940 s	≈13 minutes	≈56 minutes
30	< 50 ms	802 ms	37.870 s	≈29 minutes	>1 hour
50	72 ms	1.982 s	≈3 minutes	>1 hour	>1 hour
100	102 ms	31.622 s	≈10 minutes	>1 hour	>1 hour

6. Related Work

In this chapter, similar technologies and approaches will be discussed. As IoT is a relatively new area, there has not been much work done on the problem discussed in this thesis. There are, however, areas that are quite close. When talking about IoT, Big Data is usually a term that comes up, and when talking about Big Data, it is hard to overlook platforms and solutions such as Hadoop[21], Spark[22] and Storm[23]. However, Hadoop and Spark both align with the initial idea of IoT data, to gather it centrally and then process it. Storm focuses more on distribution of computation but it does not take such things as locality of data into mind.

Hadoop and Spark also mainly focus on batch processing, in that you run a batch, get a result and then it is done; Storm on the other hand focuses more on the fact that data will never stop coming, i.e.there is an endless stream of data and you process data as it arrives. This is more in line with the kind of approach to data processing that this thesis proposes. Although this thesis share the same kind of philosophy of processing data as Storm, the are still major differences. Storm is more aimed toward general computers/server and clusters whereas this thesis focuses on smaller devices which are characteristic for IoT. Overall there has not been much work done related to IoT devices and how to utilize their processing capabilities together with a larger IoT platform.

There is however, some related work to the cost-based deployment that was discussed in this paper. There is a constraint programming approach suited for wireless sensor networks[24]; This approach is quite similar to the cost-based approach discussed in this thesis and it also allows for other constraints in the model. The functionality of their program is also divided up into tasks, and where this thesis focused on the actor model and actor languages to represent the tasks, their article chose Abstract Task Graph macroprogramming(ATaG)[25]. The main difference is the scope of deployments. This thesis has a global IoT perspective and the distribution and deployment is seen as a part of a larger IoT platform whereas the WSN approach focuses more on single deployments. The authors of the article also had a bigger emphasis on the constraint programming model in comparison to this thesis.

7. Conclusion and Future Work

The actual implementation that has been discussed in the thesis could use some improvements and there are many avenues for future work, however the overall idea feels sound. Figuring out which cost metrics are most useful or limiting the different choices of cost metrics might be a necessity if a cost-based distribution is implemented in a dynamic IoT platform. To measure things such as latency or throughput would also put a strain on the network so it might not be the best way to measure cost between devices if it needs to be constantly updated. One approach to approximate distance or latency could be to cluster or layer different devices at time of registration.

7.1 Future Work

This section will discuss some possibilities of future work. The section is split up into the three main areas, Algorithm, Framework, and Tasks and Actors.

7.1.1 Algorithm

The results of the scalability test showed that in the easy case, it could handle 25 actors and 100 devices in a reasonable amount of time, however the hard case showed that it might take too long already at 20 actors. The algorithm can most likely be improved by adding and/or changing heuristics. Approximating solutions instead of going for the optimal solution might be a good choice if there are more than 25 actors. Doing additional scalability tests on the algorithm when testing out different heuristics could be a good way to find more efficient ways to find solutions. Adding additional constraints is also something that could change the time it takes in order to find the optimal solution and is also something that would prove interesting to look into.

7.1.2 Framework

The implementation in this thesis was just the bare minimum of what was needed in order to test the idea of cost-based distribution. In order for this

to be fully functional it should be integrated with a full-fledged IoT platform. In the current implementation, nothing is done after the actors have been deployed. Registering the deployed actor systems and providing ways of visualizing and administrating the deployed actor systems is another interesting venue for future work. There are also other things to consider, such as security and fault-tolerance as they were not in the scope of this thesis but remain very relevant to a live system.

Redeployment

An early idea was that if the user did not have cost information about the links between the actors, then the cost could initially be set to one in order to get an initial deployment. The actor system could then be deployed and statistics could be gathered, such as the amount of data sent between actors. With these statistics, the costs of the links could be set and a new and, hopefully, better deployment could be found. This means that the framework has to support dynamic redeployment. It would also be beneficial if the whole system would not need to be redeployed but rather only a single actor or a subset of actors. In order to satisfy this, the actors need to be dynamic enough that you can redirect their inputs and outputs. Another scenario would be where the network infrastructure has changed, for example new devices have been added. In such a scenario the optimal deployment might differ and it might be beneficial to redeploy the actors.

Replication

The idea behind replication is that if the original actor system has a certain actor that is under heavy load, then this actor could possibly be replicated so that the workload is split between them in order to reduce the risk of a bottleneck and to get a better distribution of workload. There are a number of difficulties with this, the main one is when an actor has a specific state. That state could be changed for every message it receives, so if the actor would be replicated, then the states would differ and the integrity of the program would be compromised. For a stateless actor however, replication can be quite simple. It could be done by just starting another actor and then redirecting some inputs from the old actor to the new one. However, replication was not in the scope for this thesis so no further work has been done in this area.

7.1.3 Tasks and Actors

In this implementation, Scala and Akka was used as the actor language, it could prove interesting to try other languages that support actor based programming and compare them to each other. This framework required of the

user to provide the actors, this might require the user to have prior knowledge in Scala and Akka in order for them to even be able to deploy some actors. To improve the usability of the framework there could exist generic or default actors that the user only had to connect and modify in order to have a fully functional actor system. To take this idea even further it would be interesting if it was possible to create a kind of drag-and-drop interface in which the user could create the actor system. This would allow for a wider range of users and applications while also maintaining a certain control over what kind of actors that could be deployed in the system.

7.2 Conclusion

With the enormous growth of the amount of connected devices that is foreseen by large technology companies, there will most likely be an increase in problems associated with network congestion and high server load in current IoT platforms. In order to solve, or at least reduce, these problems, the IoT platforms need to change the centralized way of gathering and processing data. Some functionality should be deployed closer to the source of the data, i.e. on the edge rather than in the center. Providing a way to utilize the untapped processing resources on the edge in the Internet of Things is thus the next big step in the evolution of IoT platforms.

The idea of using actor based programming seems to be a good fit for deployments in the Internet of Things. Actors and smaller IoT devices share the same kind of fundamental idea that they are standalone entities that communicate with other entities by sending messages. The overall concept of cost-based distribution also seems to be a good fit for IoT deployments. Deploying actors that are capable of initial processing closer to the source of the data will most likely reduce the amount of data sent, reduce the response times or reduce the network costs. Even though the reduction of costs for individual devices might be quite small, due to the sheer number of connected devices, the overall cost reduction could be significant.

The approach proposed in this thesis is a possible and plausible way of utilizing processing resources in IoT. Even though the work presented is not a complete solution, it provides a proof of concept for actors and cost-based deployments in IoT. It also provides avenues for further research and a possible direction for the next generation of IoT platforms.

References

[1] Ericsson, "More than 50 billion connected devices - taking connected devices to mass market and profitability," white paper, Ericsson, 02 2011.

[2] D. Evans, "The Internet of Things How the Next Evolution of the Internet Is Changing Everything," white paper, Cisco Internet Business Solutions Group (IBSG), 04 2011.

[3] T. G. Group, "Gartner says a thirty-fold increase in internet-connected physical devices by 2020 will significantly alter how the supply chain operates." Press Release, March 2014. http://www.gartner.com/newsroom/id/2688717.

[4] Raspberry Pi Foundation, "Raspberry Pi," 2014. Available from http://www.raspberrypi.org/.

[5] Arduino, "Arduino," 2014. Available from http://www.arduino.cc/.

[6] The Gartner Group, "Gartner's 2014 hype cycle for emerging technologies maps the journey to digital business." Press Release, August 2014. http://www.gartner.com/newsroom/id/2819918.

[7] SICS, "Sicsthsense," September 2014. http://sense.sics.se/.

[8] LogMeIn, "Xively," September 2014. https://xively.com/.

[9] SensorCloud, "Sensorcloud," September 2014. http://sensorcloud.com/.

[10] Ericsson, "IoT-Framework," November 2014. http://iot.cf.ericsson.net.

[11] U. U. Project CS group of 2013, "IoT-Framework," November 2014. https://github.com/projectcs13/sensor-cloud.

[12] University Of California, Berkeley, "SETI@home," 2014. Available from http://en.wikipedia.org/wiki/SETI@home.

[13] Pande laboratory, "Folding@home," 2014. Available from http://folding.stanford.edu/.

[14] A. Grama, A. Gupta, G. Karypis, and V. Kumar, "Introduction to parallel computing," *Introduction to Parallel Computing, 2nd edn, by A. Grama et al.*, vol. 1, 2003.

[15] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[16] J. Eker and J. W. Janneck, "CAL Language Report: Specification of the CAL actor language," Technical Memorandum No. UCB/ERL M03/48, University of California, Berkeley, CA, 94720, USA, 12 2003.

[17] B. Däcker, "Concurrent functional programming for telecommunications: A case study of technology introduction," *Licentiate Thesis, KTH Royal Institute of Technology*, 2000.

[18] OscaR Team, "OscaR: Scala in OR," 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[19] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[20] Pierre Schaus, "CP for the impatient," 2013. Available from `http://info.ucl.ac.be/~pschaus/cp4impatient/firststeps.html`.

[21] A. Hadoop, "Hadoop - reliable, scalable, distributed computing," June 2014. `http://hadoop.apache.org/`.

[22] A. Spark, "Spark - lightning-fast cluster computing," June 2014. `https://spark.apache.org/`.

[23] Storm, "Storm - distributed and fault-tolerant realtime computation," June 2014. `https://storm.incubator.apache.org/`.

[24] F. H. Bijarbooneh, A. Pathak, J. Pearson, V. Issarny, B. Jonsson, *et al.*, "A constraint programming approach for managing end-to-end requirements in sensor network macroprogramming," in *3rd International Conference on Sensor Networks: Sensornets 2014,*, 2014.

[25] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner, "The abstract task graph: a methodology for architecture-independent programming of networked sensor systems," in *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*, pp. 19–24, USENIX Association, 2005.