Improving house security using an IoT infrastructure

by

Mirko Morandi

June 2016

Author . . . .
Department of Computer Science
June 18, 2016

Certified by . . . .
Kitlei Ròbert
ELTE Supervisor
Thesis Supervisor

Certified by . . . .
Marchese Maurizio
Trento Supervisor
Thesis Supervisor

Certified by . . . .
László Szilágyi
Industrial Supervisor
Thesis Supervisor

Accepted by . . . .

Improving house security using an IoT infrastructure

by

Mirko Morandi

Submitted to the Department of Computer Science
on June 18, 2016, in partial fulfillment of the
requirements for the degree of
Master Degree in Computer Science

Abstract

The recent increase of power of computation combined to the decrease of power consumption of devices led to an explosion of smart devices for all the possible purposes, and it's called the ***Internet of Things***.
The IoT includes all devices that has an interface to the real world and are connected to the Internet, such as: *Smart Fridges, Smart TVs, Thermostats, Alarms, Cameras and etc*. These devices were born to simplify normal people's life, let's take the smart fridge as example: it will alert you when the food you have in the fridge is going to expire, the thermostat will try to save money applying smart strategies for heat consumption. One of the most important application of IoT to people's life is *House Security*: exploiting the IoT concept to improve the house security with smart sensors like cameras, leap motion sensors and microphones. House Security has to be of concern, because house invasion is an always trending crime. However these systems are not always perfect and the most common problem are false positives, for the which there is an active branch of research focused on.
The raise of devices connected introduced another problem: heterogeneity between IoT devices. As the popularity of smart devices increased more and more companies started producing their own different ecosystems. Communication between different ecosystems is another research topic which will analyzed in this document.

Thesis Supervisor: Kitlei Ròbert
Title: ELTE Supervisor

Thesis Supervisor: Marchese Maurizio
Title: Trento Supervisor

Thesis Supervisor: László Szilágyi
Title: Industrial Supervisor

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

IoT surveillance systems has been proved to be effective during the time, catching the burglar committing the crime and helping the authorities to catch him[1]. Nonetheless these systems are not yet perfect, and during this paper we will investigate on some optimizations with respect to the *integration* with other ecosystems and the *reduction* of false positives. Here follows the main structure of the document:

Chapter two introduces the technologies used in our scenario, with a brief introduction on the techniques adopted to face the various problems.

Chapter three illustrates the main architecture of the optimizations proposed during the document and the motivation behind them.

Chapter four shows the implementation of our use case with the optimizations we have described formerly.

1.1 Motivations

In this document we try to illustrate the common problems that arises when introducing a new IoT system in an existing environment, let's define it as an average person house, we'll define the scenario details later. The first motivating problem taken in consideration is the **integration** of a new system in a house where there's already a different coexisting ecosystem. The idea behind our research in this field is to improve the existing solutions with an innovative and different approach. Nowadays there are many standalone solutions for IoT surveillance, which acts separately from all the other components in the house. We address this isolation, trying to create a system capable of interconnecting with other existing components in the house. There are currently 4 billion of connected devices [2] as 2016, and the forecasts says they will be 13.5b in 2020. This means a growth in the heterogeneity of companies and products, which makes a must the interoperability between devices from different producers.
Most of the common Smart devices are built to work specifically with their own application,

without any external support which leads to a loose coupling between devices. This problem arises when introducing new *systems* in a house where there are is already a different ecosystem. The inability of a user to switch or adopt a new technology is also called *vendor lock-in*, a common commercial practice to retain users. However research is moving forward this direction investigating in new innovative and efficient methods to solve this problem, and there are already some approaches which will be discussed in the next chapter. **False positives** is another key issue in our investigation on the improvement of a house monitoring scenario. The reason behind this choice lies in the connection with the "idea" of a house with an alarm connection to a third party security agency. We need to have a real alarm when something is happening, reducing the probability of false alarms and not wasting time calling the police or alerting some vigilance agent.

1.2 The Scenario

In our document we will often refer to an imaginary scenario of a typical house of an average person. It is important to denote this detail since most of our solutions are designed for such a peculiar situation, and they may not be as efficient or innovative in a different scenario. The main idea is to have a surveillance system in a typical *American style* house, with different floors and rooms, each of them equipped with different and various smart devices. Having a clear idea of the scenario helps to focus more on the identification of issues while designing the systems rather than facing and solving problems at the last moment due to some unpredictable variables of a too varied scenario.

1.3 Microservices and IoT

The Microservice architecture is an innovative modeling pattern that aims to solve a well defined class of problems: scaling. The idea behind microservices is to split the architecture on different machines usually communicating through a RESTful interface. This pattern is similar to the *microkernel* architecture for Operative Systems, where the kernel contains the most vital functions and each functionality is a Plug-In (or Driver Module) that can be added externally. Each service holds a functionality isolated in it's context, which can be deployed at runtime without any interruption of the service. In microservices the equivalent of the kernel is the API interface that exposes all the functionalities to the external world. With the raising of the need of interoperability, *microservices* seems to hold the key for this problem. Vendors have published their protocols, and have exposed API's to their various hubs. A MicroService can serve as an adapter between various protocols. It can be lightweight and disposable, both desirable traits in a rapidly evolving environment.[3]

1.3.1 M2M: Machine to Machine

Most of common smart sensors, in order to be *smart* they need to provide a form of connectivity: let it be BLE (Blue-tooth Low Energy), Wi-Fi or RFID. *M2M* is trending with the raising of IoT, requiring a higher number of devices to be interconnected without any human interaction. Different devices means different protocols, which introduces difficulties in the communications between each other. The typical approach is to define a set of translators which are able to deal with both sides, also called *Gateways*. Gateways are high-level objects that knows the various protocols needed to communicate and provide these knowledge as a service to whom has the necessity to access the functionalities provided by the device. In our architecture each gateway is a device exposing a service using a **RESTful** architecture, for a higher simplicity of use.

1.3.2 A RESTful interface

The communication between different devices is a common and dated problem in *Computer Science*. Many approaches has been introduced during time, all of them with their relative pro and cons:

- **Remote Procedure Calls** used to execute procedures on a remote machine, with the advent of *CORBA* it was also possible to do that on machine with a different technology stack on it.
- The **Message Oriented Middleware** used queues to exchange messages to different platforms using different technologies. This subsequently led to the development of the common *ESB* technology.
- **SOAP** a high-level protocol for communication between web services over many popular application layer protocols such as HTTP or SMTP.

All of them are good candidates for the construction of a distributed system, but it's better to adopt a single protocol for communication instead of many to avoid over complicating the problem. We choose to use the **RESTful** architecture, which better suits our scenario. Our decision complies with the current trend in the implementation of microservices, where REST is the typical choice for exposing functionalities. Its simplicity made it very popular and widely adopted, making REST one of the "standards" for communication between services on the web, mainly due to its capability to work over *HTTP* without any additional infrastructure.

1.4 The power of Social Networks

Social networks plays a fundamental role in people's life nowadays, keeping in touch people from different countries or sharing their life moments with everyone. On average American's

spend 4.7 hours browsing on their social profile, usually around 17 times per day [4].
However social networks opened the door for some important issues for people in their daily/working life

Productivity The most common problem with social networks is of course the reduction of productivity/attention that affects people impacting negatively on their life.

Scams Scam are very common on social networks, people using fake profiles to perpetrate illegal actions such as blackmailing or phishing.

Spying This problem is the less recognized but most dangerous vulnerability introduced by social networks. Basically, if you share you're entire life with your friends, there may be someone else interested in your activity: thieves. Burglars checks people's social activity with fake accounts or looking at who has checked in in airports to select the houses to break into. This is the case we will work in the following pages of the document.

Although their vulnerabilities, social networks has the power to connect thousands or millions of people together, allowing users to spread news at unthinkable speed compared to traditional methods. Let's consider Belgium's police, who uses twitter to share the terrorists video hoping to reach someone who can recognize them.[5]
Furthermore this has been proved to be working, when a woman recorded a burglar who intruded in her house, after she shared her position, and tracked him posting his image on *Facebook*.[6] This episode is a clear example of how social networks can be exploited to use their power, and we will apply this concept in our scenario.

1.5 False positives

One of the most common problems that occurs when an alarm is armed are false positives. In most of the cases it can be just a pet triggering the alarm, some other times may be just yourself when you forget to deactivate the alarm.
Although this problem may look relatively unimportant, it assumes importance if the alarmed is linked to a surveillance company which is supposed to look after our house. That's why during this thesis we will take an effort to address this common issue with different approaches.
The false positive detection is calculated as a probability of an event happening, influenced by many factors. Each factor is one of the approaches adopted below, which will have a positive or negative influence on this factor, determining if with a certain precision a detection can be classified as a *false positive*.

1.5.1 Approaches

Here's a list of the proposed approaches taken in consideration during this document

Face detection Detecting a face when the alarm is triggered is a useful way of discriminating between a false positive and a real intrusion. However this approach has some limitations given by the position of the camera and the speed of the shutter which impacts the image quality. Its importance differs if used in combination with other different *false positives* reduction systems.

Face recognition Face recognition is just an improvement of the above technique, which would allow us to recognize a familiar face in the system and reduce significantly the probability of a false alarm. This approach however is affected by the same problems from the former one.

Speaker recognition This approach involves machine learning to learn and recognize familiar voices during an intrusion. Knowing with a certain precision that the voice belongs to someone familiar will drastically drop the probability of a possible false detection. Furthermore this approach does not suffer from the same problems as the former two approaches.

Dictation recognition People's biometric signature also includes the way they do talk. It is actually possible to identify someone by the words they say. This is an abstraction or a limitation of the approach described before. However it's a good idea to test it's performance on our use case.

Chapter 2

State of the Art

In this chapter there will be an extensive explanation to the technologies used throughout the document.

2.1 Internet of Things

What is the ***Internet of Things***?
Internet of Things, better known as IoT, is the world of interconnected devices connected to the real world and to the Internet. These devices has a very broad definition, from the smart sensors in a power plant to the smart fridge in a house. What connects these devices is their ability to be connected with the world, sharing, with the needed restrictions, their work. This opened the door to a new revolution in the IT sector, prompting new opportunities for developers, entrepreneurs and end users. Some of the most relevant endings of these technologies are *Smart Houses*, *Smart Cities*, Cars etc.

2.1.1 Ecosystems

What is an Ecosystem? An ecosystems is by definition *a system, or a group of interconnected elements, formed by the interaction of a community of organisms with their environment.*, in our case a set of smart devices connected between them. The IoT world is evolving from distinct single entities to more evolved clusters of devices which can communicate more easily between each other.These are usually are made by companies using specific protocols or standards to simplify the connection between their products, but at the same time closing it to the others. The list of products related to the *Internet of Things* is too broad for being even listed, due to the highly expanding sector related to smart devices. However we will consider the most common ecosystems which can be easily found in a common home, meaning services for: Smart Lights, Cameras, Thermostats etc. Luckily most of them are being bought and integrated in bigger environments run by famous companies such as *Google* or *Apple*. The reason for which we try to restrict the integration with these ecosystems is mainly practical, because they're the most common and they do offer simulators for testing, and because they suits perfectly our scenario.

Google - Nest

Nest is a home automation producer of smart devices which ranges from their famous Smart Thermostat to the locking system, from the washing machine to the light system, from the *Dropcamera* to the Hi-Fi Sound system,and these are just some examples. The number of components supported by this company is wide, which makes it a good choice to support in our scenario, also because of the high cost of these devices. The main added value given by these accessories is related to two main points: saving money with a smart consumption of electricity and the ability to remotely control the house with an easy to use application. Besides the commercial value of *Nest*, the platform offers many tools for developers to interact with their platform using their Cloud service. Third party developers are encouraged, with some limitations, to use their Cloud service which offers some *RESTful* APIs to gather access to the remote devices. The remote access through APIs unlinks developers from platform dependent libraries (see later HomeKit) that restricts the use or the integration with a specific technology. Furthermore it is possible to access directly *Nest* devices with their *Nest Wave* which allows direct communication with non-branded devices using two different communication protocols, mainly **802.11** standards.

Apple - HomeKit

HomeKit is a very similar ecosystem to the one described before, producing or supporting home automation devices. *HomeKit* relies on the large network of Apple devices, making it an environment to be considered even if it is relatively new on the market. The key point of *HomeKit* is the easy integration with Apple devices, fitting almost perfectly wherever there was an already existing Apple ecosystem. Developers are allowed to take control of the devices only through iOS applications, restricting the possibilities of integration with different ecosystems or technologies. Moreover it ties the developers to their technology making it really hard to be adopted in a different system. However as we'll see later it is possible to bypass the problem using the microservice architecture to make the system independent from specific technologies.

Samsung - SmartThings

SmartThings is another ecosystem from *Samsungs* with many similarities with the former producers.*SmartThings* focuses on four main Smart devices categories: *Security*,*Monitoring*, *Lighting and Energy* and *Convenience and Entertainment*. *SmartThings* as the previous ecosystem does not allow a direct interaction with the smart devices, but offers a developer-friendly interface to their Cloud Service. However, as *HomeKit* it is tied to a technology, making it harder for different technology stacks to access.

2.2 The Microservice Architectural Style

SOA vs Microservices

In short, the *microservice architectural style* is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.[7]
There is a close link between the *microservice architecture* and the *service oriented architecture*, thus due to their nature the community classified microservices as a subclass of the service oriented architecture. *Don Box* of Microsoft described the Service-Oriented paradigm with the following four principles [8]

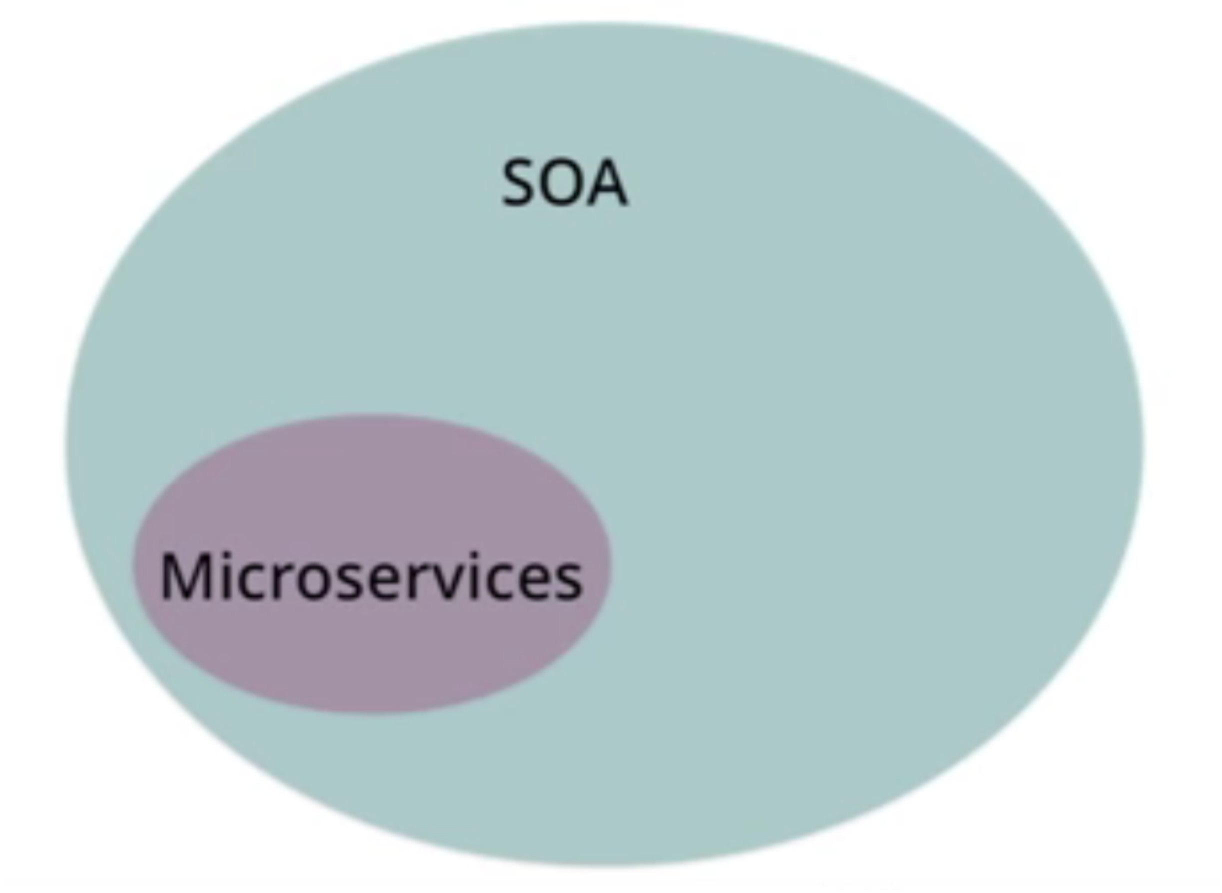
1. Boundaries are explicit
2. Services are autonomous
3. Services share schema and contracts, not class
4. Service compatibility is based on policy

Microservices fullfills the first two requirements, with a very strong focus on the second principle. However the functionalities are very frequently exposed using a *RESTful* interface, which doesn't expose any contract nor schema. Furthermore microservices holds another subtle difference related to their scope, where a microservice serves as a service inside its application meanwhile typical SOA services serves a broader scope and *can be* part of the same application. The difference between the two concepts is very subtle, and it wouldn't be impossible for them to be the same in certain situations. *Bob Ruhbart* from Oracle described shortly the difference: *Microservices are the kind of SOA we have been talking about for the last decade. Microservices must be independently deployable, whereas SOA services are often implemented in deployment monoliths. Classic SOA is more platform driven, so microservices offer more choices in all dimensions.*[10]

2.2.1 Internal integration

Typically when a new component has to be added to an existing project the approach consists in the development of a library which has the logic to deal with the new module to be supported. Subsequently the component will become part of the project itself, with its needs to be updated throughout time. However when the number of components to

Figure 2-1: Graphic illustration of microservices and SOA



integrate increases it will affect the size of the project and very likely its performances. In our case the component to be integrated will be the ecosystem driver, having a set of libraries which are capable of interacting with the remote APIs or with the direct wired connections. As we'll see in the next paragraph this is a typical monolith approach, meanwhile for this situation would be better to use a microservice approach.

2.2.2 Integration as a Service

Considering the *Microservice* architectural pattern we can decompose the above situation creating dedicated services capable of handling the required business logic to interact with an external system. This approach is also called **Componentization via Services**[7], where a component is defined as a *unit of software that is independently replaceable and upgradeable.* It is important to distinguish between *libraries* and *services*: the latter uses out-of-service components to communicate, mainly HTTP requests or remote procedure calls when libraries uses instead mechanisms like in-memory calls. The main advantage to build components as services instead of libraries is the possibility to deploy them individually without the need to redeploy the whole system. If a library is modified or removed the whole system will need to be redeployed, which in most of the cases it is converted in a loss of money and time. That's not the case if the system is composed by many independent systems, where only the changed service will need a new deployment. However this is not always true, there will be some circumstances where it will be necessary anyway to deploy again the whole system, but the aim of this approach is to reduce the number of these necessities.

2.2.3 Other benefits

Apart from the benefits already listed formerly, there are other benefits introduced by adopting the microservice architecture:

Heterogeneity between technologies Structuring the system as a set of services frees us from the limitations of a singular technology allowing us to adopt different frameworks for different tasks. This benefits on the possible optimization that can be achieved using the right technology for the right task. Furthermore this removes completely the problem of creating adapters for different technologies to integrate in the system if they do not exist. This is also called **Decentralized Governance**.

Evolutionary Design is a concept popularized by the *Extreme Programming*, where the system is continuously evolving during the phases of it's development. This key feature makes possible to evolve adopting a microservice architecture while keeping the old legacy monolith system, well tested and functional, without rewriting the whole system.

Designed for failure Building a system made of services instead of components leads the developers to take more effort in considering failures. Developers have to take in consideration the possible failure while reaching the service, and prevent the system to crash and handle the situation in the most gentle way. On the other side, this approach introduces more overhead to handle the possible situations.

2.2.4 Microservice Oriented Internet of Things

IoT with their advent brought up some, not yet resolved, challenges which fits very well with the Microservice architecture. *Interoperability* is one of these challenges aimed to be solved. Microservices *decentralized governance* feature, which allows the use of different technologies inside the same systems holds the key for solving the interoperability problem. Key IoT devices vendors realizes that if their products supports multiple interoperability standard they're likely to be adopted also by their competitors (e.g. Nest products are available on HomeKit).[3] Microservices here can be used as isolated adapters for the many different technologies, facing the spreading fragmentation of communication protocols between machines.

Microservices are by nature dynamic and reactive, which makes them a good choice for an environment where new technologies are frequently added or modified.

2.3 Calvin

Calvin is an Open-Source distributed framework designed mainly, but not only, for IoT systems. The key point of *Calvin* is the "Distributed cloud for IoT", meaning running the code where it best suits the performance needs, a crucial aspect since we are dealing with low-resource components. We will use *Calvin* for the whole course of the document, referring to it as the main "system", also used to integrate with the others.
Calvin has the ability to integrate new components without replacing them by writing proxies for the specific hardware to support. Proxies are actors written to handle communication with the legacy system. Such a proxy actor handles the task of converting data from the application into messages or requests the old system can handle, and converting the response back into tokens the Calvin application understands.[14]

Figure 2-2: Proxies interaction

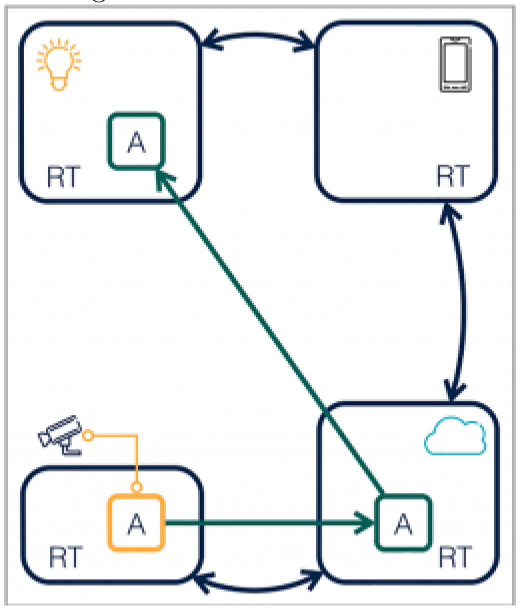


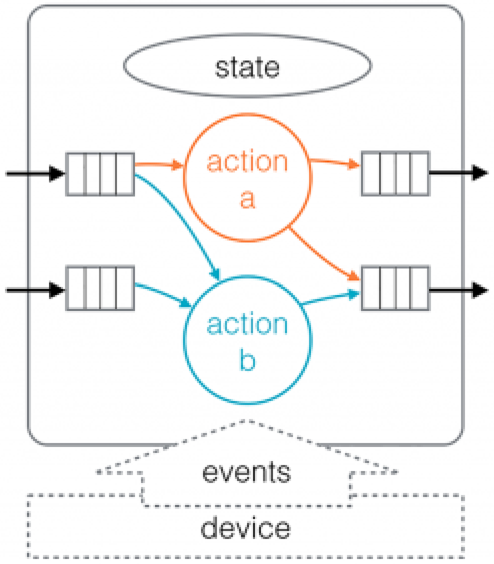
Figure 2-2 shows the interaction of native Actors and Actors which acts as a proxy for legacy components, in this example the camera.

Calvin is built upon the well-established actor model, using a methodology often referred to as dataflow programming[11], and its life cycle can be summarized in four, well-distinct, phases: *Describe*, *Connect*, *Manage* and *Deploy*.

2.3.1 Describe

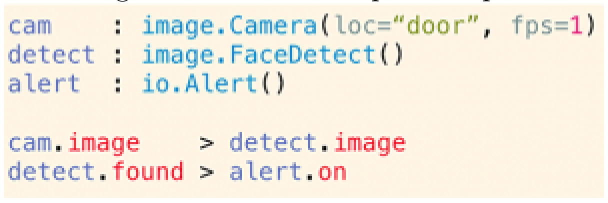
The smallest functional units in *Calvin* is an Actor. Actors do not share nor state or behavior, encapsulating all the logic. The key point of Actors in Calvin is their reactivity: they react to external events or when receiving inputs. Actors communicates only using data through ports, has to be defined before deploying the system. This way is possible to describe the possible interactions that the actor may have when connecting it to others. Having a non-shared internal state allow the Actor to be serialized and moved to another running machine or to be backed up if the machine crashes. However this is partly true because it may be tricky to serialize Actors with a very complex internal state.

Figure 2-3: Describe Phase



2.3.2 Connect

Figure 2-4: CalvinScript Example



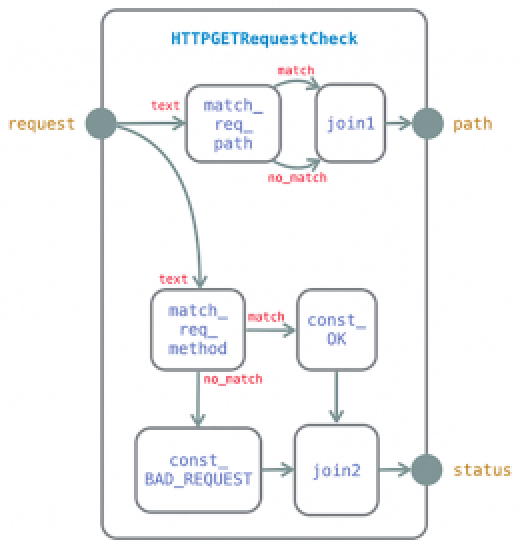
```
cam    : image.Camera(loc="door", fps=1)
detect : image.FaceDetect()
alert  : io.Alert()

cam.image    > detect.image
detect.found > alert.on
```

After describing the many functionalities provided by the various Actors in the system we need to tie them to build applications. *Calvin* offer a scripting language, named **CalvinScript**, to describe the various connections between actors and their input and output ports. In figure 2-4 there is an example of a script for detecting faces in an image. The

first part is relative to the actors declaration and initialization, giving a clear description of which actors will be playing in the current environment. The second part describes the links between each actors, structuring the flow of the process. In this case the actors in the system are 3: the *Camera*, the *Detector* and the *Alert*. The flow of the process is relatively simple in this case, and it is structured as follows: the *Camera* takes a picture, which will be passed to the *Detector*. Subsequently the *Detector* will send its result to the *Alert* actor, which will possibly fire an "alarm" if he detects any human face inside the picture.
Figure 2-5 shows a more complex interaction between actors.

Figure 2-5: Actors interaction



2.3.3 Deploy

Each Actor has different resource requirements needs to be satisfied in order to be deployed successfully. For example, referring to the previous scenario, an actor using a camera needs to be deployed on a system where there actually is a camera. These are also called *hard* or *unconditional requirements*, which determines the possibility to instantiate or migrate the actor on a machine.
On the other side there are also *non-functional requirements*, describing where an actor would suit best for its task. Always referring to the former example, when applying face detection it would be better to perform this task on a more performing machine compared to a low-resource machine, like a *Raspberry Pi*.
However at the moment *Calvin* supports only *static deployment*, needing the user to define where to deploy and execute the actors.

2.3.4 Manage

When the whole runtime is running it is more than needed to have a tool to keep track and monitor the activities in the system. The runtime can be queried to retrieve informations about the actors and locate them. It is also possible to track the actors firings, on which ports and step by step execution. These are mainly debugging tools, though it is also true that this is a recent framework and many functionalities are already in development.

2.3.5 The relationship between the Actor Model and Microservices

Actors are isolated, single-threaded components that encapsulates both state and behavior. Typically actors communicates using lightweight direct messaging systems, for example to receive or return inputs/parameters. Microservices are very similar to Actors in many of their key aspects, such as isolation, encapsulation and lightweight messaging, though usually microservices uses RESTful interfaces. There is some discussion in the community whether some argues that actors are actually microservices themselves [12] meanwhile others defines actors as a subset of microservices[13]. This however heavily depends on the actors framework adopted for the situation, which in our case an Actor can not be compared to a microservice due to an insight limitation: **Calvin Actors can not use different technologies**, at the moment.

2.3.6 Calvin's Three Tier architecture

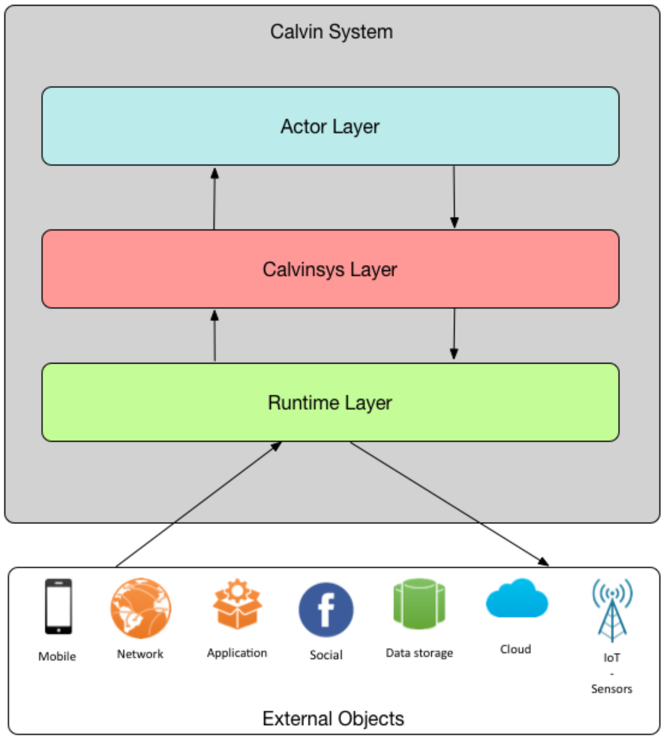
Calvin is structured with a three tier architecture: the actor layer, the system layer and the runtime layer. Each of them is isolated and can communicate with each other only passing values through the various layers. The architecture is summarized in picture 2-6

Actor Layer The actor layer is the upper layer which exposes all the actors and their functionalities. Actors can used only the functionalities offered by the lower level calvinsys with no direct access to the runtime system.

Calvinsys Layer Calvinsys is considered the middleware, where all the runtime logic is exposed through high level functions and objects. This layers wraps and uses the functionalities offered by the runtime layer, offering tools for the development of advanced actors.

Runtime Layer the runtime layer is considered the core, in this layer are stored all the low level implementation.In this layer can be found detailed implementations of how to communicate with different kinds of sensors, an asynchronous implementation of HTTP calls for Calvin and etc. Here's the core of all the functionalities adopted by the higher level objects such Actors or middleware libraries.

Figure 2-6: Calvin Architecture



2.3.7 Anatomy of an Actor

As previously introduced Actors are abstract reactive entities which exposes inbound and outbound connections. In contemplation of *reactivity* all the actions taken from an actor must be simple, fast and short. Nonetheless this optimal situation is hardly achievable in real life, it's usually more common to have blocking calls to services in the runtime which require on average more then 1-2 seconds. This limitation can be solved simply moving the logic in the runtime towards an asynchronous approach for long time tasks. Typical tasks requiring asynchronous interaction can be reading data from a sensor, sending an HTTP request or authenticating into any service, the list is longer but can summarize in most of interaction with out of the system objects.
An actor anatomy can summarized in the following parts:

- Declaration of the connections
- Initialization and setup
- Migration
- Actions, Guards and Conditions

Declaring the connections

All the connections of an actor has to be declared a priori, and hence be all used (or allocated) when the actor is used. The declaration is done simply adding under a comment the names of the many connections and their type: either input/output or both. Here is a simple example of an actor for HTTP GET requests:

```
Input:
  URL : URL to get
  params : Optional parameters to request as a JSON dictionary
  header: JSON dictionary with headers to include in request
Output:
  status: 200/404/...
  header: JSON dictionary of incoming headers
  data : body of request
```

In this example we can see the 3 input ports: URL, params and header, which will all be needed to be set in order to fire one of the actions, depending on the condition. These inputs can be used as a condition parameter using `action_input['var_name']` and `action_output['var_name']` for outputs.
However it is very likely that we won't need all these inputs or outputs, specially if they are optional like `params`. For this reason there exists two special actors which will connect to these ports without generating tokens or consuming them without doing anything (namely Void and Terminator).

Initial phase

The initial phase is denoted by the decorator `@manage['inputs']`, where all the variables are set and the required libraries are loaded. It is possible to pass inputs at instantiation time using manage and bypassing the input ports, though this approach is discouraged and advised only for particular situations. Furthermore in this phase all the components from the *calvinsys* will be loaded for a consequent use during the actor actions.

Migration

One of *Calvin's* key feature is the simplicity of migrating actors through the different nodes in a cloud system. Migration of the object is achieved using a special function `migrate(self):` to which can be passed the current actor object, and it will be serialized. Migrating implies serializing all the current variables which composes the actor's state to be sent, though problems may arise if these variable contains complex objects such as connections (e.g. DataBase connection, TCP connection etc). When the state will be sent to another machine, this will setup the actor passing from the former phase, hence creating a new (almost) identical actor.

Actions, Guards and Conditions

The core of an actor are the actions it can perform, but these needs some conditions to bet met before being fired (e.g. having a variable set to `true`). Most of actions are composed of three key elements:

Condition is a mandatory decorator which describes the inbound and outbound connections for the action. Although the name suggests it to be the condition statement to for firing an action, condition is related to the input and output condition of the action. An example of condition for a camera actor can be `@condition(action_input=['trigger'], action_output=['image'])`. In this case our actor when will receive a token on the `trigger` port will enable the action, but the next decorator will be the definitive check to pass to the action body.

Guard is an optional decorator, for checking some conditions before passing to the body of the action. Guard is more related to the internal state of the actor instead of the connected ports, thus it's the real condition check to fire the action. Guard can be used to see if a variable has been set or if its value is correct. For example if we want to check that an actor has completed an asynchronous task we can use a guard similar to this one: `@guard(lambda self:  self.camera.picture_taken)`. In this case *Calvin* will perform an evaluation of one the actor internal objects: it the value is set to `true`, it means that the picture has been taken and its stream has been saved on the disk (i.e. the file exists). This allow the system to prevent firing actions when the actor may not be ready for it even though the firing connections are set.

Action The action is a simple function which has to take as parameter the inbound values in the condition statement, and return an `ActionResult` object with the expected output bounded to the outbound values.

Moreover inside the actor the actions has to be listed by priority, to avoid ambiguity when choosing with the correct action to fire.

Chapter 3

Detecting Intruders

3.1 Preface

The major problem faced during this work is related to the detection of intruders while surveilling the house. Modern security systems applies different techniques to detect intrusions using many different sensors such as leap motion detection, face detection or sound analysis. They have been working fine during the last years, however thieves improves bypassing or partially avoiding these technique making the latter less effective.
Therefore most of the work will be done investigating and testing innovative and advanced techniques to deal with intrusions detection.

3.2 Fingerprinting a Human

During the time there has been many approaches to recognize a "Person" using some of its peculiarities: software side with passwords or access phrases, which should identify a single person; biometrics with finger prints, retina scans and voice recognition.
Usually such controls where available only with expensive solutions (and some still are, e.g. retina scanners), however with the introduction of IoT their price dropped allowing them to be sold for a reasonable price. These controls are also called **behavioral biometrics**, the set of behavioral traits which identifies an individual person. Usually these traits are used for authentication when accessing a restricted system, however we'll consider these to recognize unknown entities accessing our house.
These systems are mainly needed to prevent accidental triggering of the alarm, to be more precise when listening or watching, not detecting just something but a real intrusion.

3.2.1 Behavioral biometrics

Detecting if someone in the house is someone familiar is not an easy task, most of the approaches are precise in optimal conditions, much less when the situation is non optimal (low light, low resolution etc). This impacts negatively their efficiency when applied in a real life scenario, where usually the conditions are far from the optimal situation. For example facial recognition works in very limited cases, like the systems used on smartphones to unlock the device, has been reported to not be reliable enough to be the main unlocking system. The situation is also similar for voice recognition or speech to text systems, which are improving during the time and mainly **during the usage**. *Siri*, one of the most famous personal assistant had some minor troubles at the beginning being used to the different users voices, accents and languages. *Siri* uses different voice recognition algorithms to categorize your voice by learning your dictation and your accent, with good results over time.

3.3 Expanding the system

One of the biggest limitations of nowadays security system is their isolation when installed. Most of devices are stand alone solutions incapable of extending their functionalities to pre existing devices. If we want to expand *Calvin* to be adopted by others, including users and developers, we need to be able to extend its functionalities with the ability to communicate with different other systems.

3.4 The Architecture

In this section we will go deep in the details of the architecture of the solutions we propose.

3.4.1 Accessing different systems

Accessing different systems is a problem of technology interoperability between different technologies running on architecturally different hardwares. Most of the times *IoT* devices runs on low limited power devices such as an *Arduino* or a *Raspberry Pi*, both supporting different system languages (the first is limited to a subset of *C++* meanwhile the second one supports everything that can run under a normal linux distribution, but usually it's *Python*). However in our scenario the devices we consider are on a different layer of abstraction, we do target ecosystems and not single independent devices. **Ecosystems** has to provide an access to developers through a point of access, **REST api** for *Nest* and the **HomeKit.framework** for accessing *HomeKit* devices. The limitations are not always the same, when using *Nest* it's easier to communicate with it using a wrapper for their own api, which exists in many libraries, including *Python*. Nonetheless the situation with *HomeKit* is much more strict, having their ecosystem restricted only to their technology, *Objective-C* or *Swift*, which runs only on *Apple* branded device. Furthermore, the limitations are even more strict due to the limitation of the framework only for *iOS*, at the moment at least. Usually the latter is the common case, which implies the need of a different solution then "just wrapping" the apis in the code and plug them into the project. In our case *Calvin's* system is fully written in

Python, though it have many extension for a wide range of technologies, communicating with *HomeKit* is still hard to achieve. In the following sections we'll describe the approaches taken to solve the problem.

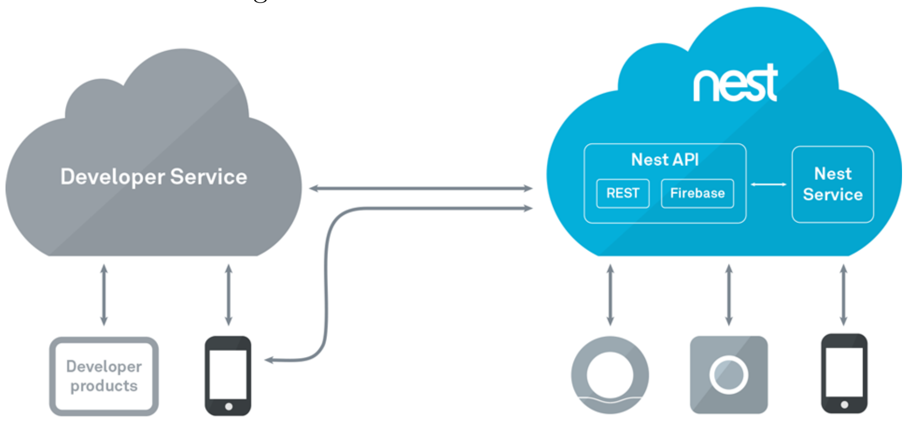
3.5 Accessing Nest

As said in the previous section *Nest* has some advantages due to their cloud apis, which makes it easier to reach the connected devices.

3.5.1 Nest API Model

Nest offers a cloud API near real time, based on a subscription, that allows users to build products accessing data on their devices, with the relative ability to read and write data. With the *Nest* cloud each element is identified as a resource and accessible through a unique address, called "data location". Furthermore, the cloud system also offers a *RESTful* service to access these resources, but only allowing GET and PUT operations with a call limit to prevent overuse. There is also a **Streaming** feature for REST, which allows our application to receive updates in real-time or to stream from a device, namely camera, since web-sockets are not supported. However it applies the same rule as before, with some limits in the data usage to prevent abuse.

Figure 3-1: Nest Cloud architecture



As can be seen in figure 3-1, the only way to access *Nest* devices at the moment is only through their cloud system. However this will add many layers of overhead and delay due to the various layers, including calling services external to our network. However, in the upcoming future it will be also possible to use *Nest Wave*, a direct device to device communication through protocols such as BLE or Wi-Fi, moving the integration to a higher lever of quality.

The cloud functions of *Nest* are limited to a restricted set of devices: Thermostat, Protect (alarm), Camera and the Home. Here are the functionalities offered by each of the products, with a brief description:

Nest Thermostat

Thermostat, as the name suggests, is a smart thermostat ,which can also be remotely controlled, with some energy saving settings to help the user avoiding unnecessary heating expenses.
Its remote functionalities are:

- Read the current temperature
- Read or set a target temperature
- Set the fan timer
- Read or set the temperature mode
- See humidity values
- View online status and last connection information
- Read structure name and device location in the home

Nest Protect

Protect is a location based alarm for dangerous gas leaks, such as Carbon monoxide CO, or smoke in case of fire.

- View CO or smoke status
- View battery health state
- View last manual test status and timestamp for last manual test
- View online status and last connection information
- Structure name and device location in the home

Nest Camera

Camera, previously known as Dropcamera, is one of they key components of the *Nest* suite. It provides screen capture, audio and video streaming and the other typical features offered by smart cameras.

- View camera online status or mic status
- View or change streaming status (turn video streaming on/off)

- View device name and where identifier
- View last online status change (last online/offline change)
- View subscription status (enrolled/not enrolled)
- Learn more about Nest Aware with Video History subscriptions >
- View deep links to the live camera feed in the Nest app (iOS, Android) or on the web at home.nest.com
- View content related to the last event that triggered a notification, including:
  Sound or motion event detected
  Event start/stop times
- Deep links to image and gif files
- Structure name and device location in the home

3.5.2 Nest with Calvin

One of the key aspects of *Calvin's* Actors is their abstraction to allow their reuse in different situations. However, due to the different nature of the ecosystem, specially *Nest* and *HomeKit* the actors can't be used for the same purpose. *Nest* structure can be decomposed in two big sections: Structures and Devices. Structure are objects to keep grouped together many devices, placing them in different part of the house, but still not the most important part in our scenario.
A device is an abstraction of all the smart products offered by Nest, but mainly the one we have listed in the latter paragraph.

Device Abstraction

We have abstracted a device in its most basic functions: setting and getting properties. This simplification allows us to follow the best practices for the development of *Calvin*, allowing a possible reuse of the actor. Here is the pseudocode for the high-level functions definition.

```
getproperty: in device_id, property_name; out property_value
setproperty: in device_id, property_name, value ; out True or False
```

However going deeper into the actor architecture we have to define some more constraints, like the events which trigger the execution and the tokens for which returning the results. First of all, all incoming port of an actor has to be linked, we can not have an unbounded port, both input or output. Here's a list of our actor's ports:

```
Inputs:
    device: the device identifier
    operation: string representing the operation to do (get or set)
    property_name: the property to be get/set
    value: the value to be used to set the property
Outputs:
    result: the property value
```

To simplify the work, we reduced the inbound port for the operation to only one, instead of having a token for a *get* or *set* operation. This task is completed using the `@condition` to declare all the elements which are needed to fire and will be used in a particular action. A simple condition for an action is the following example:

```
@condition(action_input=['operation'], action_output=[])
```

This condition requires the token operation to be bounded, and it will use it in its body function. Since no `action_output` is defined, the operation does not return any value.

Once all the inbound ports are connected, i.e. they have a token, the system will check the `@guard` condition, which acts as a conditional control: it will continue only if the condition is satisfied. An example guard condition can be written as:

```
@guard(lambda self: self.nest._in_progress is None and self.operation == "set")
```

This guard condition is used to enter the function only when the *nest* object is not working on an asynchronous task and the operation has to be the *set* operation.
One of the constraints that an actor has to follow is the **reactivity**, which implies non-blocking calls in the body of its functions. However, as in our case, the actor will have to make some calls to the *Nest* cloud, which is a blocking call. This is reached using a two step asynchronous system: first the value is acquired and the actor will delegate the request to a different thread; second when the delegate terminates it will modify a flag to mark its completion, at that point the actor will return the value obtained from the nest cloud. Asynchronous calls are needed since the scheduler loop can't be blocked during the execution otherwise it would break the reactivity of the other actors.

3.5.3 A Microservice Oriented approach

Despite of the correctness of the former solution, it does however contains some flaws in the architecture. The solution works with under the following circumstances:

1. Libraries for connecting to a different ecosystem are provided in the same language (e.g. *Python*).
2. We assume the actor does never crash, and if it does the system may crash subsequently.

3. The integration library won't require any change or modification in the future.

However these assumptions are too risky to be considered for an always running system, because the likeliness of one of the above problems to happen is very high.
To face these problems we adopted a **microservice** architecture style to implement the integration with other ecosystems. In the case of *Nest* is not very significative but it will be much more for *HomeKit*, as we'll see later.

Microservice Architecture

Figure 3-2: Microservice Integration architecture

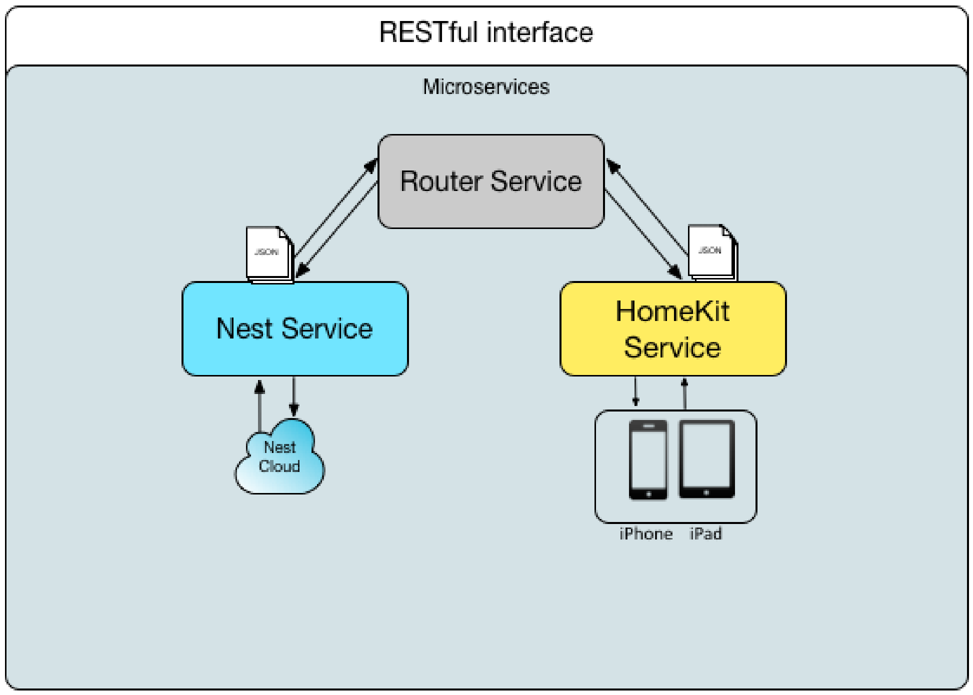


Figure 3-2 illustrates the architecture proposed in our solution. All the microservices can be reached through a router service, which doesn't expose the real location of each service. This approach moves the service finding logic outside of *Calvin*, simplifying the work of the actor who won't have to deal with possible re-deployments or re-locations of the service, it will just need to know where to reach the routing service. This task will be deferred to the *routing* service, which will act as a proxy to the other micro-services.
Each ecosystem has a microservice dedicated, this is due to the different internal structure of the different technologies. For example *HomeKit* uses a hierarchical structure divided in the following, from above to the bottom: **HMHome**, **HMRoom**, **HMAccessory**, **HMService** and **HMCharacteristic**. Which differs from the simpler one of *Nest*, where there are mostly Structures and Devices.

Moreover structuring the integration with ecosystems introduces the capability to link more accounts, more devices, to the *Calvin* runtime.

The microservice API

The API to describe the connection between *Nest* and the microservice resembles the former actor structure. Following the classic *RESTful* style, to access a device will look like

```
1.
Show details about device
GET /device/<deviceid>

Example Response
{"device":
    {
    "name": "DFF9", "temperature": 37, "humidity": 50, "fan": false,
    "mode": "heat", "online": false, "serial": "fake9396122EDFF9",
    "where": "basement","structure": "HomeTest", "target": 25
    }
}

2.
Show property value of a given device
GET /device/<deviceid>/<property>

{ "value": 37 }

3.
List all the devices
GET /devices

{
   "devices": [
        "4A32","12DE","DFF9",
        "18B430AAFAA447E5","18B430AA542D0893",
        "fakebb555fb772ce405b91729b3c4ac1962b"]}
4.
Set a value for a specified property of a device
PUT /device/<deviceid>/<property>/<value>

{ "success": true }
```

3.5.4 Accessing HomeKit

HomeKit, introduced in details in the previous chapter, is the other ecosystem for which we are extending *Calvin* integration. First of all, HomeKit doesn't expose any cloud API, it's functionalities can be accessed only through their **HomeKit.framework**, a library available only for *iOS*. As can be seen the limitations compared to Nest are much more strict, which implies no direct integration. The only possible solution for the moment is to have an *HTTP* web server exposing HomeKit functionalities to the outside of the world. As previously mentioned, their library is not available on both **Mac OS X** nor the Apple TV **tvOS**, which means the only possible approach is a mobile application. However it is predicted to be supported by the former OS in the future, on which would be a more elegant and nicer solution.

The Application

Before entering in the application logic of the solution, is better to review some concepts regarding mobile development. Here are some of the architectural issue we had to deal with:

1. First of all Swift or Objective-C is not meant for building web services, therefore there are no official libraries for starting and handling HTTP requests.

2. Second, the architecture for mobile applications doesn't fit well with the standard daemon running in the background.

3. Third, the *HomeKit* accessories are accessible only from a specific component, *UIView*,which means the server has to be started in one of these views.

However on the Internet there are many open source solutions, including some HTTP servers which also supports url matching. We choose to use **GCDWebServer**[15], a lightweight HTTP server embedded in both *iOS* and *Mac OS X*.

Grand Central Dispatch

In order to fully understand how to run a web server on a mobile application we have to briefly introduce the *Grand Central Dispatch* (GCD). The *Grand Central Dispatch* is a set of tools and libraries made available from Apple for developers to handle concurrency in their applications[16]. GCD allows developers to create and dispatch objects, which will be deferred to a differed thread and queued for concurrent execution.
GCD provides three different queues on which tasks will be submitted in a form of block object; all the submitted blocks are executed on a pool of threads fully managed by the system. The three queues are:

Main : task executed serially on the main application thread;

Concurrent : tasks dequeued following a FIFO policy but executed in parallel;

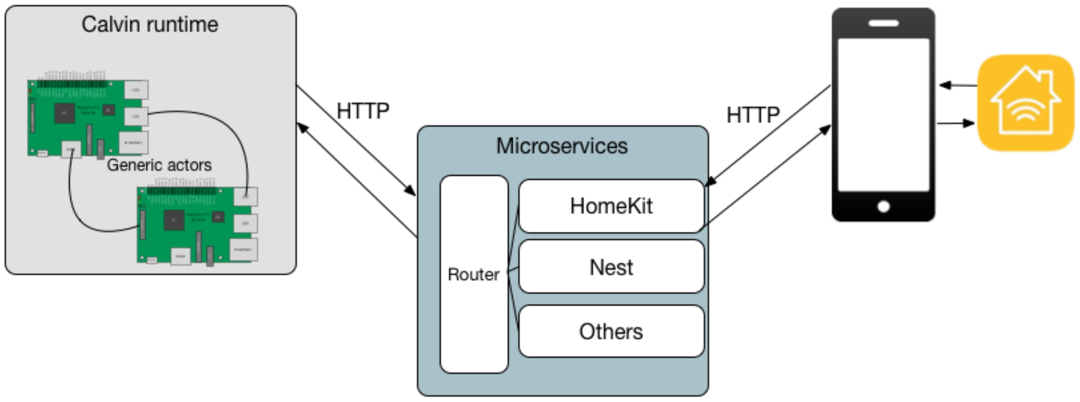
Serial : Same as before but executed in serial order;

Dispatching different parts of our system can be very helpful when we need to handle a request from a client without locking the main view of the application. In **GCDWebServer** to each handler for the different url paths, there is a block of code assigned to it, which will be handled by the GCD.

Calvin and HomeKit

The connection between a *Calvin* Actor and *HomeKit* is very similar to the former approach: the actor fires an async HTTP request, the request will travel through the whole microservice circuit and it will get the result.
The only difference lies in the number of hops to reach the device, in this case four(eight considering the returning path) including the device itself. However this implies more overhead and more latency, resulting in an overall reduction of performance, which we'll analyze more in details in the next chapter. Figure 3-3 illustrates the general architecture for *Calvin* to communicate with a *Homekit* device.

Figure 3-3: HomeKit communication architecture



Despite the overhead due to the different services, being on the same network, under the assumption of it being fast enough, the total delay should be more than acceptable. However the architecture present a weakness,due to the single point-of-failure in the iOS application, which is the only access to the HomeKit devices.

Advantages

3.5.5 lol

The answer is 42

Appendix A

Tables

Table A.1: Armadillos

Armadillos	are
our	friends

Appendix B

Figures

Figure B-1: Armadillo slaying lawyer.

Figure B-2: Armadillo eradicating national debt.