Improving house security using an IoT infrastructure

by

Mirko Morandi

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master Degree in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

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.

Chapter five is an extensive analysis of the results obtained during the whole analysis of the scenario.

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

Literature

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

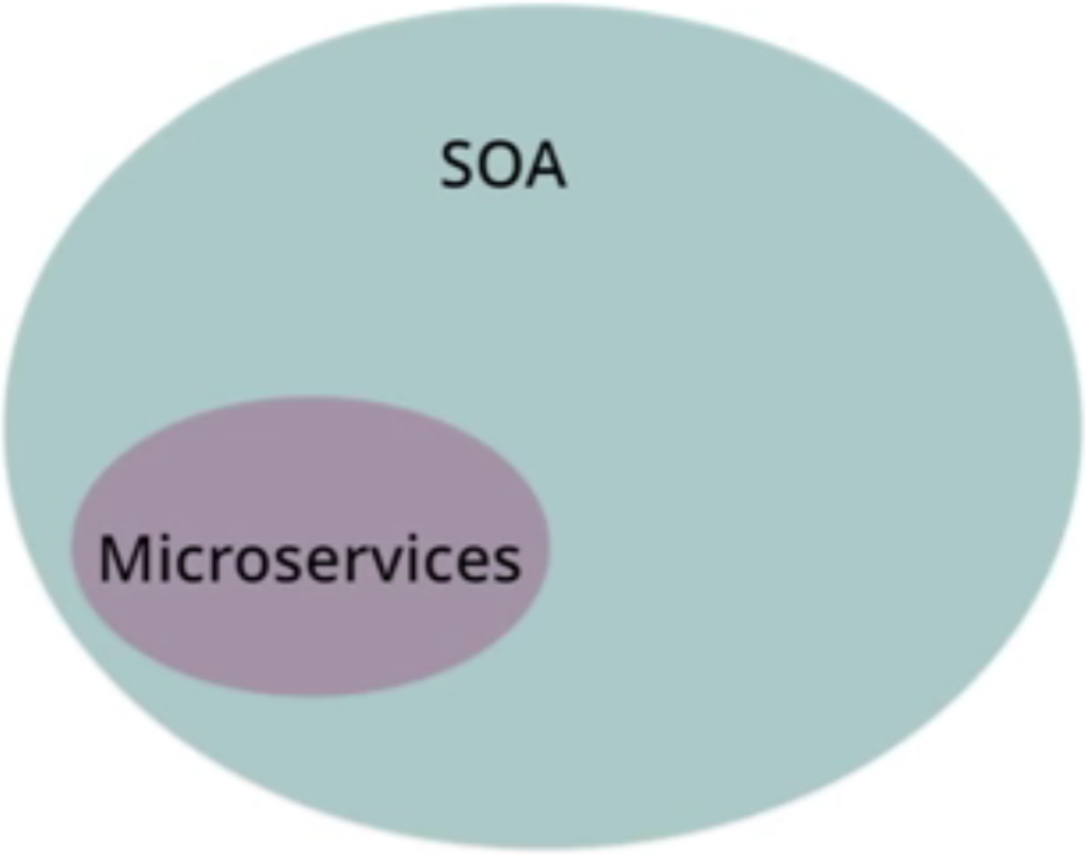
SOA vs Microservices

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]

Figure 2-1: Graphic illustration of microservices and SOA



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 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.

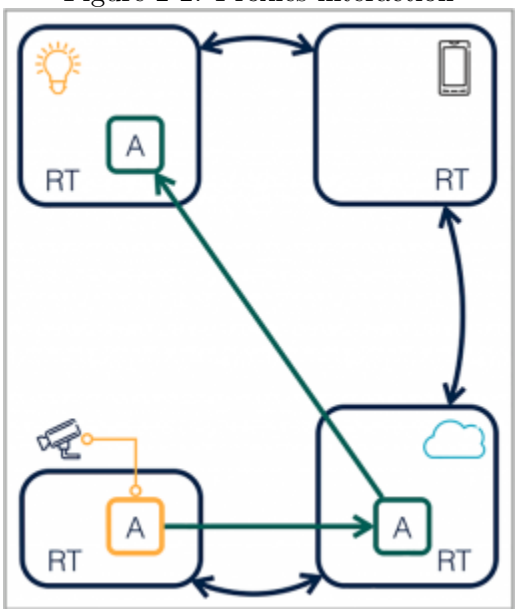
2.2.4 Microservice Oriented Internet of Things

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 shows the interaction of native Actors and Actors which acts as a proxy for

Figure 2-2: Proxies interaction



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.

2.3.2 Connect

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 **Calvin-Script**, to describe the various connections between actors and their input and output

Figure 2-3: Describe Phase

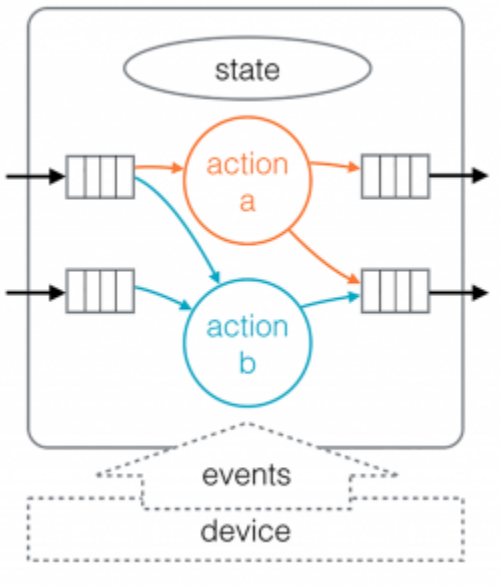


Figure 2-4: CalvinScript Example

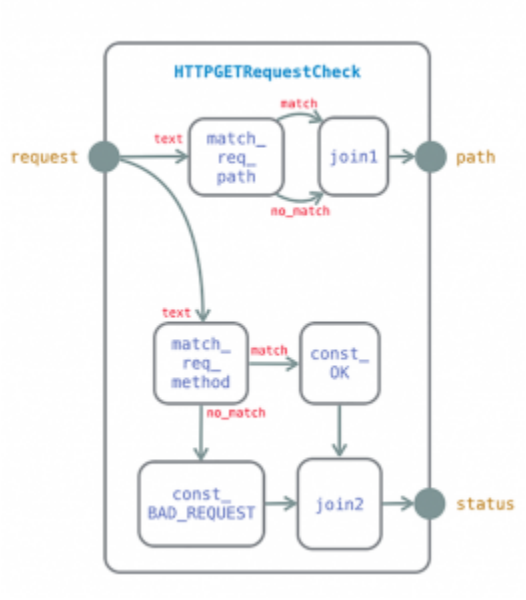
```
cam    : image.Camera(loc="door", fps=1)
detect : image.FaceDetect()
alert  : io.Alert()

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

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.

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

Figure 2-5: Actors interaction



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.

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.