Computer Science Department

BACHELOR THESIS

Migration system for Zoe microservices

Author: Rafael Medina García

Supervisor: David Expósito Singh

Madrid, June 2016

Título: Migration system for Zoe microservices
Autor: Rafael Medina García
Tutor: David Expósito Singh

EL TRIBUNAL

Presidente: José Manuel Sánchez Pena

Secretario: Pedro Peris López

Vocal: María Paula de Toledo Heras

Realizado el acto de defensa y lectura del Trabajo Fin de Grado el día 7 de julio de 2016 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de:

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

Agradezco:

A mis padres y hermano, por vuestro apoyo, siempre; a Dorito-senpai y sus increíbles amigos: Jaime y Nacho; a Adrián, por esas interminables tardes trabajando en el despacho; al Grupo de Usuarios de Linux de la Universidad Carlos III de Madrid, verdaderamente habéis hecho que aproveche mi tiempo en la universidad; a David Expósito.

龍が我が敵を喰らう

"*Ryuu ga waga teki wo kurau*"

Hanzo Shimada

Abstract

The Zoe virtual assistant developed by the Linux User Group from Carlos III University is a project that aims to automate various tedious tasks of the association. Its architecture is based on independent *microservices* (agents) that communicate through a common server and can be implemented in almost any programming language thanks to the plain text messaging protocol.

The aim of this project is to extend the Zoe platform in order to add migration capabilities to the agents. That way, it would be possible to distribute the load among different machines, making efficient use of the available hardware resources by means of load balancing algorithms while maintaining the internal data across migrations.

The approach followed was to design a portable protocol to deal with file and data migrations and implement it in a distributed architecture using basic functionalities from the original project. Said architecture was devised as an addition to Zoe, at software level, that does not require any modification to the original components and uses SSH tunneling for inter-machine communication.

As a result, the system obtained is completely backward compatible and transparent, given that all the process is controlled by the independent *Scout* agent. Furthermore, the two load balancing algorithms implemented automatically migrate the agents either trying to maintain a regular load among all the machines or to use specific machines, which would allow to shut down unused machines. Moreover, the modular code greatly simplifies the addition of new algorithms.

As a conclusion, the proposed solution offers a new set of tools for developers of the Zoe project in the hopes of enriching the agent ecosystem. Finally, by releasing it as free software, anyone can expand the capabilities of this work or apply it to other fields, which could lead to new and interesting, projects.

Resumen

El projecto de asistente virtual Zoe desarrollado por el Grupo de Usuarios de Linux de la Universidad Carlos III de Madrid tiene como fin la automatización de diferentes tareas de la asociación. Su arquitectura se basa en *microservicios* independientes (agentes) que se comunican por medio de un servidor común y pueden implementarse en casi cualquier lenguaje de programación gracias al protocol de mensajería basado en texto plano.

El objetivo de este proyecto es extender Zoe para habilitar la migración de agentes. Así, sería posible distribuir la carga entre diferentes máquinas, haciendo un uso eficiente del hardware disponible mediante algoritmos de balanceo de carga, manteniendo además los datos internos durante la migración.

Para ello se diseñó un protocolo que se encargara de la migración de archivos y datos y se implementó en una arquitectura distribuida usando funcionalidades básicas del proyecto original. Dicha arquitectura se ideó como un añadido a Zoe a nivel de software que no requiere modificación de los componentes originales y utiliza túneles SSH para la comunicación entre máquinas.

El sistema obtenido como resultado es completamente compatible con el original, además de transparente, teniendo en cuenta que el proceso se controla por medio del agente independiente *Scout*. Además, los dos algoritmos de balanceo de carga implementados migran agentes automáticamente, ya sea para mantener una carga regular entre máquinas o utilizar máquinas específicas, lo que permitiría apagar aquellas en desuso. Aparte de esto, el código modular simplifica enormemente el añadir nuevos algoritmos.

Como conclusión, la solución propuesta ofrece un nuevo conjunto de herramientas para desarrolladores del proyecto Zoe con la esperanza de poder enriquecer el ecosistema de agentes. Finalmente, al liberarlo como software libre, cualquiera puede expandir este trabajo o aplicarlo a otros campos, lo cual podría llevar a proyectos interesantes.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Free software projects offer an excellent base to build on top of and extend the work of others with enhancements or additional features. Such is the case of the Zoe virtual assistant, developed by the Linux User Group (GUL[1]) from Carlos III University, which aims to automate tasks in several scopes such as administration, management or student memberships.

Given the distributed nature of this assistant, which will be explained in the next section of this document, and the fact that it is a real project currently in use, it provides a very good opportunity to apply the knowledge obtained throughout the years in the University.

This document includes an introduction to the state of the art of technologies related to the topic, a description of the proposed solution and its related architecture as well as additional details such as time/cost planning, regulations that apply in the case of this work, evaluation of the solution and, finally, conclusions on the work performed.

In addition to that, administrators may find the installation and user manuals in the appendices of this document.

1.1 Objectives

This section lists the objectives to achieve through the development of this work. These are summarized in four main points:

1. **Design of an architecture to support agent migration**. Such architecture

[1] http://gul.es

would be an extension to the Zoe project at application level in order to remain completely backward compatible

2. **Design of a portable and efficient communication protocol**. This protocol would need to be independent of the programming language used and serve as the foundation for migrating and communicating remote agents with the Zoe server

3. **Design of load balancing algorithms in order to achieve better management of the available hardware resources**. By performing automatic migrations based on decision algorithms it would be possible to, for instance, determine the best location for an agent by taking into account its load

4. **Implementation and evaluation in an heterogeneous architecture**. It would be necessary to ascertain the validity of the solution proposed and its behaviour when communicating machines with different configurations (both hardware and software)

Chapter 2

Background

This chapter introduces some key concepts required for understanding the problem to solve and the state of the art of the technology related to this project. In addition, an overview of the Zoe architecture is given, indicating how the system works and, finally, the alternatives considered before designing the proposed solution.

2.1 State of the art

This section offers an insight on technologies that are related to the problem to solve, directly or indirectly, and their impact on different aspects of society.

2.1.1 Virtual assistants

Apple launched the first PDA (*Personal Digital Assistant*) the Newton MessagePad in 1993. In the year 2000, nearly 12 million PDA units were sold worldwide [1, p. 1]. Since then, and especially due to the growth of the mobile device market and advances in technology, the original concept has evolved giving birth to the IPAs (*Intelligent Personal Assistant*), also known more commonly as *virtual assistants*.

While the PDA devices were designed with businesses or other fields such as medicine in mind, it is clear that current mobile devices supply most, if not all, of the features they offered. Recent versions of the Operating Systems used in these mobile devices (Android, iOS, Windows, etc.) usually bundle a virtual assistant software of their own which offers customized services to the user of the device.

Even though there are many virtual assistants in the market, the three most well-known are *Google Now* from Google, *Siri* from Apple and *Cortana* from Microsoft. Each of them works in a different way and/or platform (for instance, Siri is only available in iOS devices), although a general description for them would be that of an application which uses inputs such as the user's voice, vision (images), and contextual information to provide assistance by answering questions in natural language, making recommendations, and performing actions [2, p. 1].

Most of these actions and functionalities offered by the assistants are performed through Internet services, with examples such as natural language recognition, data fetching or interaction with other online services. This type of implementation is in place in order to reduce the load in the device as much as possible, as these mobile devices and *wearable devices* are constrained by their computational capabilities and, most importantly, the battery of the device.

2.1.2 Bot platforms

Recently, another field that has gained importance is the one of communication robots, or simply *bots*. These have existed for some time, for instance in customer service offered by phone companies, but are now in the spotlight, in part due to platforms such as Telegram[1], which apart from being an instant messaging system also offers free APIs (*Application Programming Interface*) to create bots that can communicate through the instant messaging service.

This, however, is not the only example, as bots can be created for virtually any communication platform, being notable examples IRC, Twitter, Facebook or any chat software available. The approach required for each platform varies greatly depending on its implementation details, although most share a common point: they are implemented as Internet services.

As opposed to virtual assistants, which share some of the computational load with remote servers, bots perform all the computations in the server they are installed in, meaning that user devices are only required to transmit the input (text, voice, images, etc.). Because the actions performed are usually not as load intensive as the ones performed by a regular virtual assistant, even though they can do things such as natural language recognition if the developer chooses to do so, it is an interesting solution for implementing small services.

There are, however, privacy implications when using this type of services, as the bots

[1] https://core.telegram.org/bots

are hosted in the private server of an individual rather than a company such as Google or Apple, therefore the user has to consider whether the information the bot will have access to is worth the services it offers.

2.1.3 Cloud computing

The previously mentioned Internet service trend is also considered part of *cloud/distributed* computing, which refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services, being also referred to as SaaS (*Software as a Service*) [3].

For users this simply implies the execution of remote procedures or actions by means of a communication interface such as the well-established RPC protocol, web applications or REST APIs, which are very common nowadays. However, service providers, developers or administrators must consider additional elements to this approach: any application needs a model of computation, a model of storage, and a model of communication [3, p. 2].

It is clear that software services require hardware to run on top of, and while a small application may be able to run seamlessly on a small machine, such as a Raspberry Pi, a popular and widely used service, like those offered by Amazon or Google, would require higher computation capabilities and resources that can only be achieved through powerful datacenters. In spite of that, there is a limit to what technology can achieve with the resources initially allocated for a service as it may end up requiring additional means to continue operation.

This issue is present in data storage as well. Hard disk drives have evolved greatly since the first implementations, recently reaching the first drive with a maximum capacity of 10 Terabytes [4], but these may fail over time and require a replacement. As it is not feasible to stop a datacenter for this reason, scenarios such as the previous one have to be considered when implementing the system: it should be possible to expand or replace resources as needed without affecting the deployed service(s).

There is an additional approach, that can also be applied in parallel, available for cases such as these: decentralized systems. This would consist in sharing load, data, etc. among several datacenters or simple servers in order to increase availability or performance. An example of this can be found in Cassandra, a distributed storage system for managing very large amounts of structured data spread out across many commodity servers [5, p. 1] originally created at Facebook.

Decentralized systems use the network to communicate as well as to provide services. The advantage of the approaches mentioned is that systems may be stopped or launched on demand to save resources.

2.1.4 Microservices

Directly related to cloud computing is the microservice architecture pattern, which defines a microservice as a small application which can be deployed independently, scaled independently, and tested independently and that has a single responsibility [6, p. 1]. These services may, or may not, communicate among themselves using a common channel which ranges from standard HTTP protocols to a messaging bus.

In addition to being independent, each service may be created using different technology stacks or programming languages. As more and more languages are focusing on web technologies, developers can greatly benefit from this fact and choose the best option to implement a microservice on a *by-case* basis.

All in all, this pattern is no panacea and whether to use it or not should be carefully considered beforehand. Whereas monolithic applications may be deployed to a small server or container, having several microservices implies a higher cost in terms of resources. The added complexity of a distributed system is apparent when dealing with asynchronous operations or tasks of each service, especially when a synchronous element is introduced (e.g., storing information in a database).

2.1.5 Secure network communications

Taking into account the aforementioned and given that the network plays a very important role, security of the communications is of primary importance. The way of securing the communication greatly depends on how the system works.

For example, regular web pages and applications (including REST APIs) employ standard security protocols such as SSL. This protocol was created by Netscape to ensure secure transactions between web servers and browsers, and uses a third party Certificate Authority (CA) to identify one end or both ends of the transactions [7, p. 2].

The usage of such certificates prevents Man in the Middle attacks and ensure secure communications with the server if correctly implemented. However, users could not always afford said certificates as their price range varies greatly depending on the features offered with the certificate (e.g., valid for all subdomains, authentication of the owner, etc.), resulting in personal webs and applications not implementing the secure HTTPS protocol.

Nowadays this is starting to change with the *Let's Encrypt* project started by the University of Michigan, Mozilla, the electronic Frontier Foundation and other partners and which will enable websites and other computerized systems to get free, secure, self-renewing SSL certificates that will be trusted by browsers as the project will have its own CA [8, p. 5].

Another scenario would be that of secure communication among private systems that, as opposed to web services or applications, do not actually offer access to external users. One of the most, if not the most, well-known protocol is the SSH (*Secure Shell*) protocol, which enables secure remote login and other secure network services over an insecure network [9, p. 2].

This protocol is based on the idea of establishing a secure communication channel on top of an insecure one where the host machine has its own private-public keys and performs authentication for a remote user. Apart from being an open standard protocol, it emphasizes points such as all encryption, integrity, and public key algorithms used being well-known, well-established algorithms [9, p. 6], making it easier to audit and more secure than proprietary protocols as anyone may participate in its development.

The way it is mostly used is through public key authentication, which creates the secure channel using public information from both the host and the user (or other host) and encryption algorithms with their respective private keys, supposing that neither have been compromised in any way.

While these are only a couple of examples of secure communications applied to system communication, closely related to the work presented in this document, there are multiple protocols that can be applied in other fields such as OpenPGP in email encryption.

2.2 Zoe architecture

The *Zoe virtual assistant* project is a free software (from here on, *free software*) project developed by the Linux User Group from UC3M with the objective of automating tasks. Overall, it implements a *microservice architecture pattern* where the services are named **agents** and can be implemented in virtually any programming language.

This allows for developers to easily develop new features in the language they feel most comfortable with.

These agents communicate using a common protocol, which is also independent from the programming language, through an internal server and work in an asynchronous manner performing very specific tasks.

The following sections offer a brief introduction to key concepts of this architecture.

2.2.1 Agents

Also called *actors*, agents are the services that perform all the operations. Usually, agents are not aware of the rest of the agents installed in the system and rely on the server for communication and handling of messages directed to them.

In general terms, an agent is a small server that awaits connections to the socket it is listening to and performs specific actions depending on the message received. Therefore the only method that is running continuously until the agent is stopped is the one that listens for such connections and receives the messages. An overview of such connections can be found in Figure 2.1, which shows four different agents connected to the server and not among themselves.

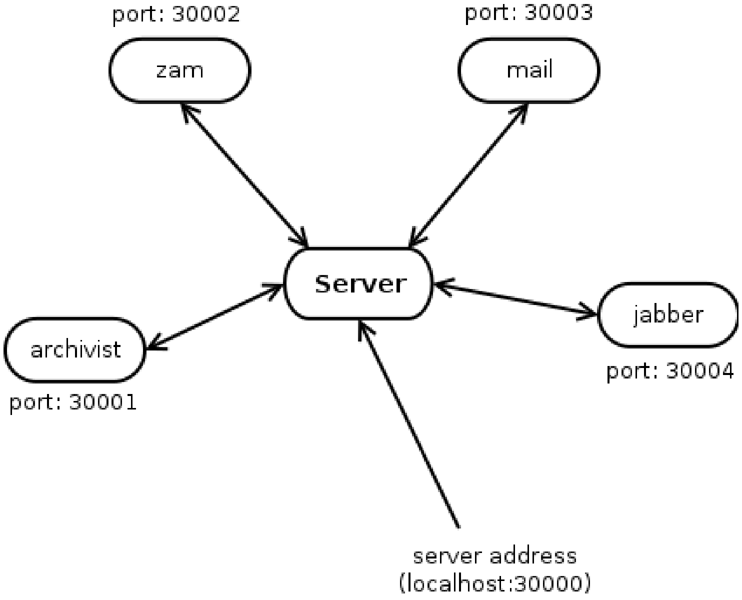


Figure 2.1: Example of agents connected to server

Once a message is received, its structure is parsed following the specification shown in the next section and an action/method is executed according to the parsed message.

While similar to how RPC works, this architecture does not necessarily reply once the requested method has been executed, being completely decoupled from the client that originally sent the request.

As this is an asynchronous system, the main thread in charge of receiving messages from the server should never perform the parsed actions itself. Instead, agents rely on either threads or additional processes to do so. The main advantage of this approach, apart from the fact that the main thread is not blocked by long-running operations, is that each child thread is isolated from the rest, even though the developer may choose to synchronize them at some point (e.g., shared memory access), hence being unaffected by errors in sibling threads.

Even though agents are intended to work in an asynchronous manner, it is also possible to run synchronous tasks. The principle is the same as before: when started, the main thread of the agent inspects the methods implemented and if a *timed method* is found, it creates a new child thread that will be in charge of running that method periodically, for instance checking mail inbox or machine status. However, and given that this is not a real time system, these synchronous tasks are executed on a *best-effort* basis and their scheduling time is not guaranteed.

At the time of writing this document, most of the agents are written in Python, although there are a couple written in Java, and there is a Zoe library available for both languages included in the *Zoe startup kit* distribution.

2.2.2 Communication

The communication protocol used by the Zoe server and its agents is intended to be simple to parse and implement. It is an indirect application of the Unix philosophy, which strongly encourages writing programs that read and write simple, textual, stream-oriented, device-independent formats [10, p. 15], and as such it is plain text with a specific structure.

On one hand, by using plain text agents (and server) can be written in any programming language taking into account that all of them have code in their respective standard libraries to deal with strings in some form or another. On the other hand, this imposes some limits as to what can be sent in said messages. Simple data can usually be converted to a string representation, but complex data structures are much more difficult to deal with, especially in inter-language communication.

Nonetheless, this communication protocol is very light and works incredibly well in this architecture. It is based around the idea of using *key-value* pairs to identify parts of the message, for instance (omit the *newlines*):

```
dst=zam
&tag=install
&name=archivist
&source=rmed/zoe-archivist
&sender=admin
&src=tg
```

Almost all agents recognize the following keys:

- `dst`: agent this message is intended for, in this case the Zoe Agent Manager (an agent named *zam*)
- `tag`: action the agent should execute, in this case install an agent from a git repository

The `dst` key is also used by the server to determine the address it should relay the message to, while the `tag` key is the standard way of identifying available methods in the agent. Tags are then checked by the agent, as specified before, when a message is received to determine the method to execute.

In addition, the previous message contains information which is only of interest to the agent that receives the message (*zam*):

- `name`: name of the agent to install
- `source`: GitHub repository from which to download the source
- `sender`: unique ID of the user that sent the message, usually from a communication channel such as a chat or an email
- `src`: name of the agent the message comes from, in this case Telegram, used to send feedback messages after operation

Although easy to read, these messages are tedious for humans to write. That is why Zoe implements communication channels that employ *natural language* recognition to extract the aforementioned information. Figure 2.2 shows the execution and communication flow (messages exchanged) when the previous message is received.

First, the user sends a natural language command (*Install "archivist" from "rmed/zoe-archivist"*) through a communication channel, for instance Telegram. This message is received by the agent in charge of that communication channel (agent *tgbot*), which extracts relevant information such as the message itself, the channel where the message came from and the sender of the message.

Given that communication channel agents do not know how to interpret natural language commands, they relay the extracted information to the *natural* agent, who parses the language using several scripts. These scripts extract information for the recipient agent (in this case: name and source repository) and relay the message.

Finally, the destination agent (*zam*) performs the action and may also return feedback to the user through the communication channel they first used (i.e., *"Agent archivist installed correctly"*).

As shown, the server is only in charge of dispatching messages to agents, being unaware of their contents.

There is another type of messages that deals with topics. Topics can be used to send the same message to several agents that work with a specific topic. Relation between agents and topics is stored in one of the configuration files in the Zoe distribution.

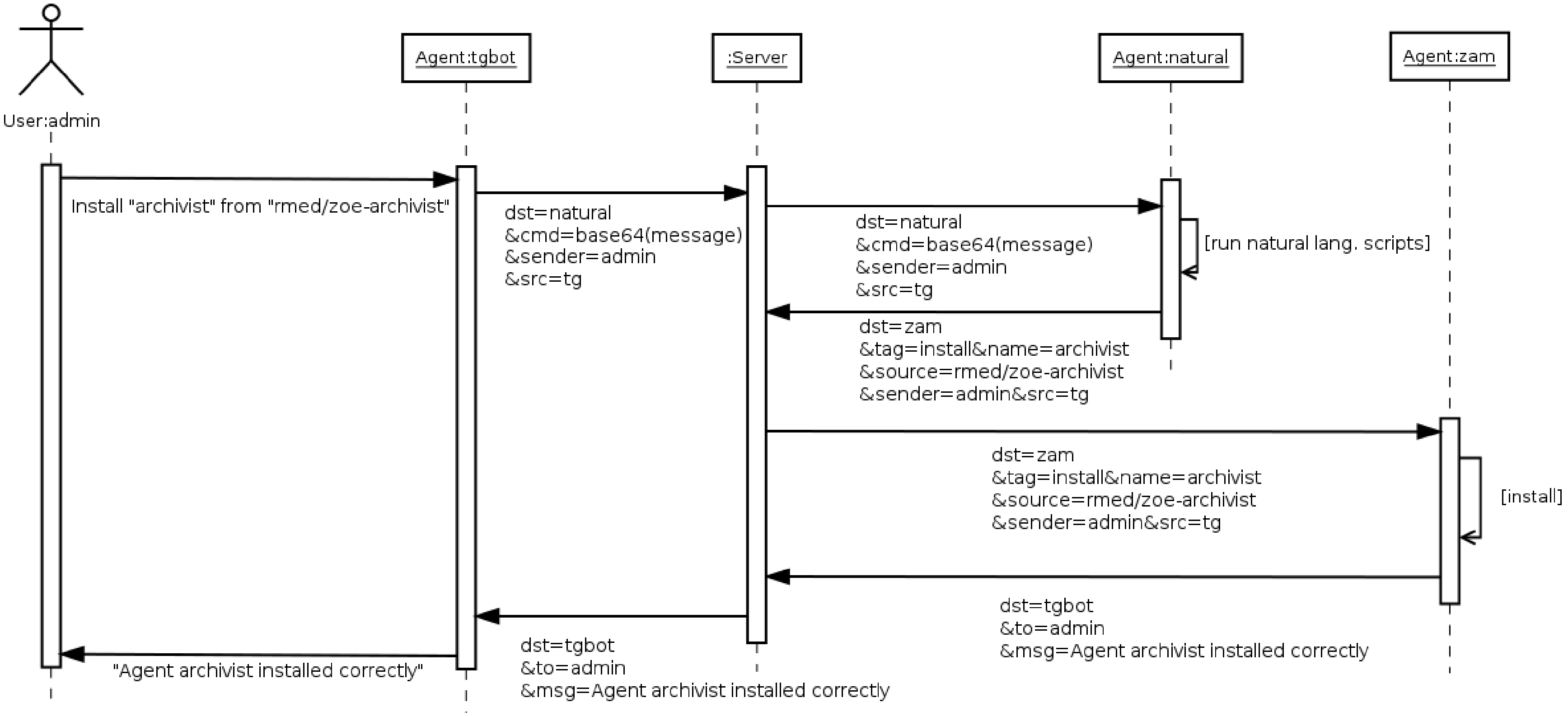

User:admin
Agent:tgbot
:Server
Agent:natural
Agent:zam
Install "archivist" from "rmed/zoe-archivist"
dst=natural
&cmd=base64(message)
&sender=admin
&src=tg
dst=natural
&cmd=base64(message)
&sender=admin
&src=tg
[run natural lang. scripts]
dst=zam
&tag=install&name=archivist
&source=rmed/zoe-archivist
&sender=admin&src=tg
dst=zam
&tag=install&name=archivist
&source=rmed/zoe-archivist
&sender=admin&src=tg
[install]
dst=tgbot
&to=admin
&msg=Agent archivist installed correctly
dst=tgbot
&to=admin
&msg=Agent archivist installed correctly
"Agent archivist installed correctly"

Figure 2.2: Command: installation of agent archivist

2.2.3 Server

The official Zoe server is written in the Scala programming language and requires the Java Virtual Machine to run. It has two main functionalities:

- **Proxy**: communicates with other Zoe instances (if configured) and parses messages sent to its socket (by default in port 30000)
- **Router**: routes the messages that each agent sends and delivers them accordingly

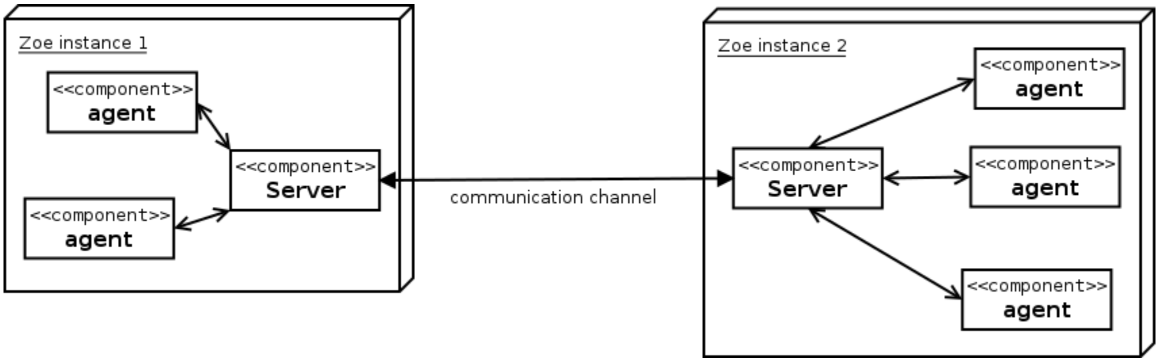


Figure 2.3: Distributed Zoe instances

This implies that Zoe can work in a distributed manner, both in terms of having agents in different machines and communicating with other instances (e.g., another instance has an agent that is not available in the local one), as represented in Figure 2.3. Both Zoe instances are configured to be aware of each other, usually indicating host and port of the other server, and are able to exchange messages when needed (e.g., one Zoe instance has an agent that the other needs).

However, as shown previously, the communication protocol does not implement any kind of encryption and sends the messages as plain text, so this type of distributed functionality must be secured through external means by an administrator.

In order to perform message routing, the server needs to know the ports (and addresses) in which each of the agents can be found. This information is stored in an internal map that is filled on startup with information from the agents configuration file. In addition, and given that agents may be added and removed dynamically from the server at any time, there is special message for the server that updates the internal map with the keys it contains. Its structure is as follows:

```
dst=server&tag=register&name=<AGENT_NAME>&host=<AGENT_HOST>&port=<AGENT_PORT>
```

2.3 Considered alternatives

This section exposes the different alternatives initially considered for this work, before reaching the final solution, with the main reasons for discarding them.

2.3.1 Virtualization

Virtualization consists on *running an operating system on top of another operating system* by means of an hypervisor. The hypervisor acts as a layer between the virtualized guest operating system and the real hardware [11], making the virtualized operating system (guest) see the real hardware as its own.

While there are several hypervisor solutions (Xen, VMWare, OpenVZ, etc.), the one that was mainly considered was KVM (Kernel-based Virtual Machine) due to the fact that the Linux kernel includes support for full KVM virtualization since version 2.6.20 (being 4.5.3 the latest stable version at the time of writing). With this, the Linux kernel is treated as the hypervisor so that the virtualized environment can benefit from all the ongoing work on the Linux kernel itself [11]. Figure 2.4 shows an overview of how KVM is implemented on top of the Linux kernel.

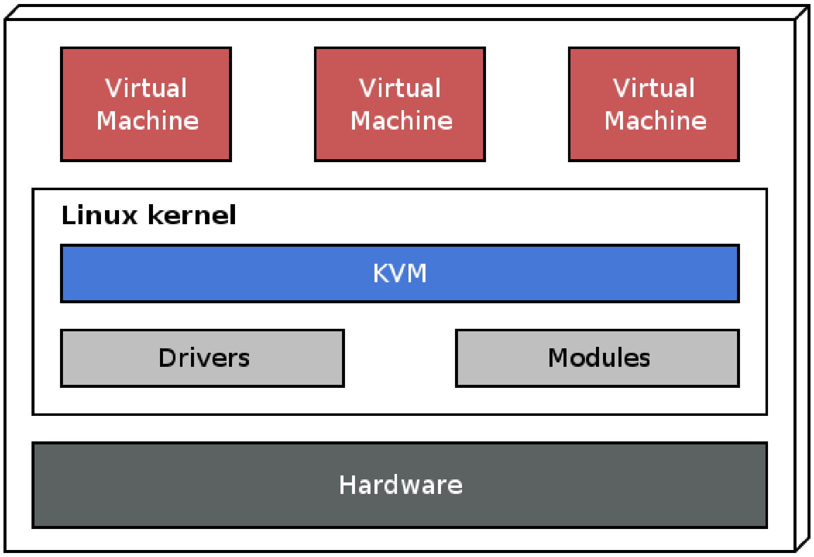


Figure 2.4: KVM Architecture. *Adapted from an original figure Copyright of Red Hat, Inc. [12]*

The main reason for using this type of virtualization would be live migration, which was one of the primary objectives of this work. This process works by copying guest memory to the target host in parallel with normal guest execution. If a guest page has been modified after it has been copied, it must be copied again [13, p. 229]. In the scope of the task at hand it would mean moving agents accross machines without interruption of the services they provide.

Although the aforementioned is a very important advantage, this solution showed several disadvantages that made it unfit for implementation. First and foremost, virtualization using KVM requires hardware support, reducing the number of platforms that the project could be executed on (Intel and AMD do support virtualization in most of their products).

Secondly, there is an important overhead to consider when dealing with virtual machines. It would be a practical implementation for conventional servers given that they often require a set of tools that run on top of an operating system but, in the case of a Zoe agent, the advantage of having easy live migrations is not worth the overhead and additional complexity of the system (e.g., each agent would run independently on an isolated virtual machine with its own operating system and requiring a complete modification of the Zoe architecture).

2.3.2 Containers

Containers, follow a similar approach to virtualization: execute multiple isolated applications in a single host machine. LXC (*Linux Containers*) is an operating system level virtualization technology that creates a completely sandboxed virtual environment in Linux without the overhead of a full-fledged virtual machine [14, p. 2].

Part of the recent popularity of such technology is due to the open source Docker framework, which provides a tool that greatly simplifies the interaction with LXC and other kernel functionalities, as well as the more efficient use of available resources. This means a computer running Docker can run many more simultaneous virtual instances than the same computer running typical virtual machines [15].

In addition, Docker focuses on application development and deployment, with features such as container versioning, allowing to rollback to a previous state, and easy deployment through a configuration file (*Dockerfile*). This *Dockerfile* contains the commands necessary to build a container and range from specifying a base image (such as Debian) to installing software and launching a web server.

Figure 2.5 shows an overview of the Docker architecture. In this architecture, there is **only one Operating System** (the host system) which is shared among the containers, as opposed to the case of virtualization, in which all virtual machines have

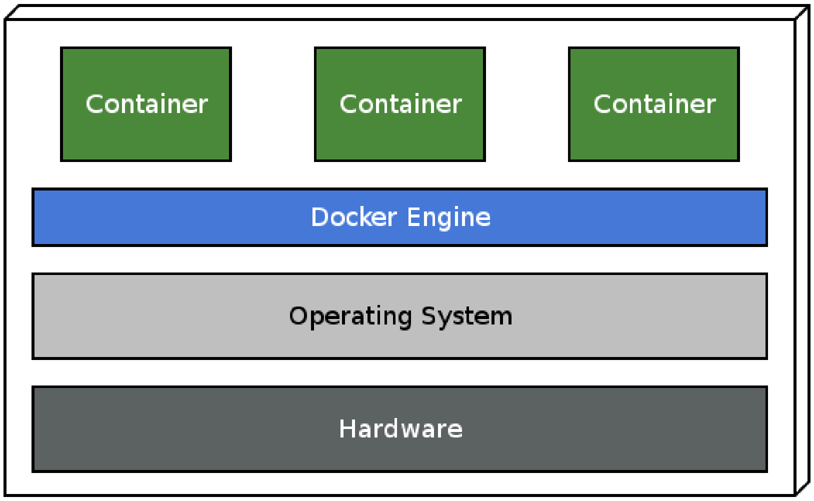


Figure 2.5: Docker Architecture. *Adapted from an original figure Copyright of Docker, Inc. [16]*

their own Operating System. This is one of the main reasons for the efficient use of resources discussed previously.

It is clear that containers offer many advantages over classic virtualization, although there are limitations as well: it is not possible to run containers with different processor architecture (i.e., ARM architecture cannot run applications packaged for x86).

The reasons for not using Docker containers for this work were similar to those from the previous virtualization section: even though it is also possible to perform live migration of containers and the overhead would not be as noticeable as the one from virtualization, only one processor architecture (the central server one) would be supported and the Zoe architecture would need modifications (albeit minimal when compared to virtualization).

2.3.3 Network Filesystem

The final alternative that was initially considered was using NFS (*Network Filesystem*) for sharing files across the central server and the outpost machines. NFS consists of three major pieces: the protocol, the server side and the client side [17, p. 119] and was originally developed by Sun Microsystems in 1984.

At its core, the remote filesystem is implemented using RPC (which was also developed by Sun Microsystems) in a stateless fashion in order to simplify the protocol and error recovery: when a server crashes, the client re-sends NFS requests until a response is received, and the server does no crash recovery at all [17, p. 120].

Figure 2.6 shows an overview of the architecture of NFS. As stated before, all operations on the server are performed through RPC protocol. The server routines rely on kernel system calls to affect the files of the filesystem, although such operations are transparent to the client.

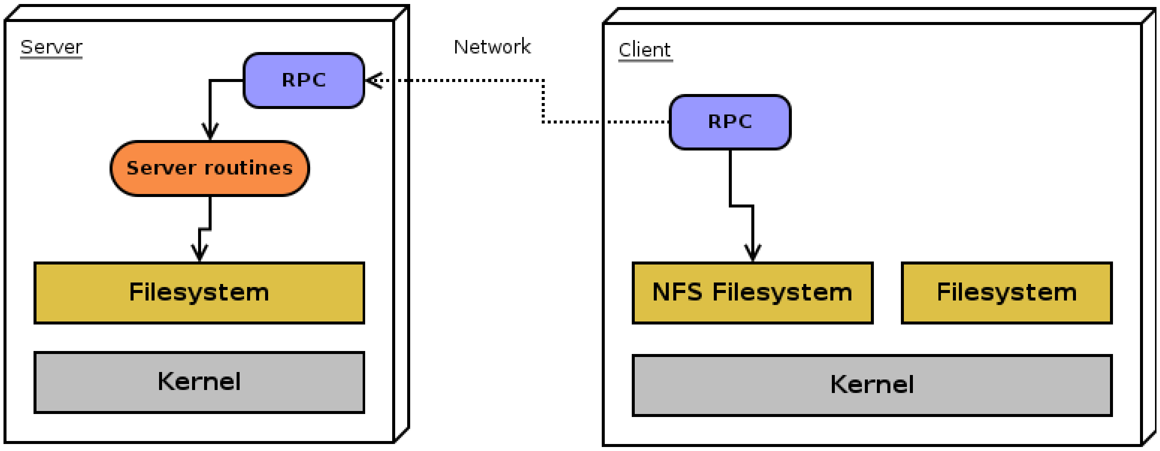


Figure 2.6: NFS architecture. *Adapted from [17, p. 122]*

Even though it was developed for Unix systems, nowadays it can be implemented in all major operating systems, including GNU/Linux based ones. In fact, it is very common to see such filesystem implemented in remote multi-user scenarios that enable user access from different machines, for instance in organizations or universities.

Given that it works in a transparent way for both users and applications, Zoe agent files would have been shared using a NFS (from the central server) and accessed directly by the outposts. By doing so, load balancing would consist on stopping the process of an agent in the machine it is running on and have another machine start the program, effectively using its own available resources for the computations.

Taking into account that most of the agents are written in interpreted languages (i.e., Python, Java), there would not have been an issue in architecture compatibility given that each machine would possess the necessary tools for the execution. However, connection to the Zoe server would have needed a separate implementation (ideally taking security into account).

Nonetheless, this option was also discarded for two main reasons. First, no state would have been saved during the downtime of the agent when migrated, losing any information the agent could have been working with up to that point. Second, the performance and complexity implications of using this protocol were a great concern: NFS, being based on RCP, relies on network stability for accessing the files and is much slower when compared to native access to the disk. In addition, configuring NFS could be a difficult task for users that do not have a high level of expertise in the field.

All in all, this NFS based approach proved to be most interesting and might be a compelling topic to study for its application in scenarios such as scientific computation that does not rely heavily on disk access.

Chapter 3

Proposed solution

The outpost system is a distributed system architecture that tries to solve the problem at hand employing free and open source technologies and software in its entirety. As a free software project itself, the system specification, which can be found in the next pages of this document, will also be published in the future in the hopes of providing a useful service to developers of the Zoe platform.

In addition to the architecture of the system, this section describes the development environment and relevant tools used for the design and implementation of the system, as well as the load balancing capabilities included within it.

3.1 Development environment and tools

The following tools and environment were used to develop the solution. Note that all of them are free software, which also helps reduce project costs.

3.1.1 Debian GNU/Linux

The operating system used to provide a base for the development environment was the Debian GNU/Linux distribution in its *testing* branch (at the time of writing, version 9.0 “stretch”). This operating system provides a solid foundation for both advanced and regular desktop users, with tools of many different fields available.

Apart from being *free/libre* software, it can be downloaded freely from the webpage of the project (https://www.debian.org).

3.1.2 Time planning

The gantt chart used for time planning (see *Planning*) was created using the GanttProject tool.

3.1.3 Code editor and file management

All the code was developed using the Vim editor, mainly due to past experience with the software, its extensibility and the fact that it integrates perfectly with the workflow employed throughout the development of the project (heavy terminal usage).

Every component of the project (*Scout*, outpost, library, etc.) was put under its own git repository for a more efficient version control management, allowing to safely change the code of individual components without affecting the rest and revert those changes if needed.

3.1.4 Programming language

As most of the Zoe agents are developed using the Python programming language, with a standard Zoe library available, and given the experience that has been gained through development of personal projects in the past years, it was chosen as the main development language.

Python is a dynamic programming language that can be executed in many different platforms and operating systems, offering portability of the code with no additional requirements as long as the target machine has a Python interpreter available. In particular, the code was developed and tested in versions **3.4** and **3.5**.

Furthermore, some additional modules and libraries, separate from the standard Python library, were used to write the code. These are:

- **Zoe libraries** for Python developed by David Muñoz and released under the MIT licence
- **Paramiko**: module that implements the SSHv2 protocol in Python. Developed/Maintained by Jeff Forcier (and contributors) and released under the LGPLv2 licence [18]
- **scp**: plugin module for Paramiko which implements file copying capabilities using the Paramiko connection. Developed/Maintained by James Bardin (and contributors) and released under the LGPLv2 licence [19]

- **peewee**: small ORM (*Object-Relational Mapping*) library used to manage the internal databases. Developed/Maintained by Charles Leifer (and contributors) and released under the MIT licence [20]
- **Blinker**: signaling module used in the dashboard implementation. Developed/Maintained by Jason Kirtland (and contributors) and released under the MIT licence [21]
- **PyGObject**: Python bindings for the GTK+3 toolkit used for the dashboard user interface. Developed/Maintained by the GNOME Foundation and released under the LGPLv2 licence [22]

3.1.5 HPLinpack

As each machine has different hardware, the total capacity of each one needs to be benchmarked beforehand. Although there are many benchmarks available, the one used during development and testing was LINPACK. In particular, the HPL suite, which is an implementation of the LINPACK benchmark, was compiled manually.

In essence, LINPACK is a collection of subroutines which analyze and solve various systems of simultaneous linear algebraic equations [23]. These operations place a lot of stress on the processor(s) and can be used to determine the capabilities of a machine in various scenarios that can be configured through a complex configuration file.

3.1.6 Documentation

The documentation for this work was written as another piece of code, with Vim and its own git repository, using the Markdown syntax (which is released under a BSD licence) [24] and some $\LaTeX$ for the source files. These source files are then *compiled* to PDF using Pandoc, which is a free software (released under the GPLv2 licence) that is able to convert documents to and from multiple formats [25].

This workflow has worked extremely well in all the reports delivered during the last two academic years and has simplified the creation of documents greatly.

3.1.7 Diagrams

Attached diagrams were created with the GNOME Dia, which is a cross-platform diagram creation program released under GPLv2 licence.

3.2 Architecture

The implementation, referred to as *outpost system* from here on, consists of three main blocks: a protocol, the *outpost* microserver that runs on a remote machine and an agent named *scout* which runs in the central Zoe server. The details for each of them are explained in this section.

Figure 3.1 shows a general overview of the architecture proposed, where the Zoe server works with both normal and outpost agents, allowing them to exchange messages in a transparent way, and communicates with other machines (outposts) by means of SSH tunnels. Outpost agents employ a different standard library that implements the protocol (see *Protocol*) necessary to enable agent migrations.

In addition, and as part of the *information gathering* process, both *Scout* agent and outpost servers use the *Linux perf* tool in order to access hardware counter information exposed by the Linux kernel.

Each of the points shown in the figure will be discussed and explained in the next sections.

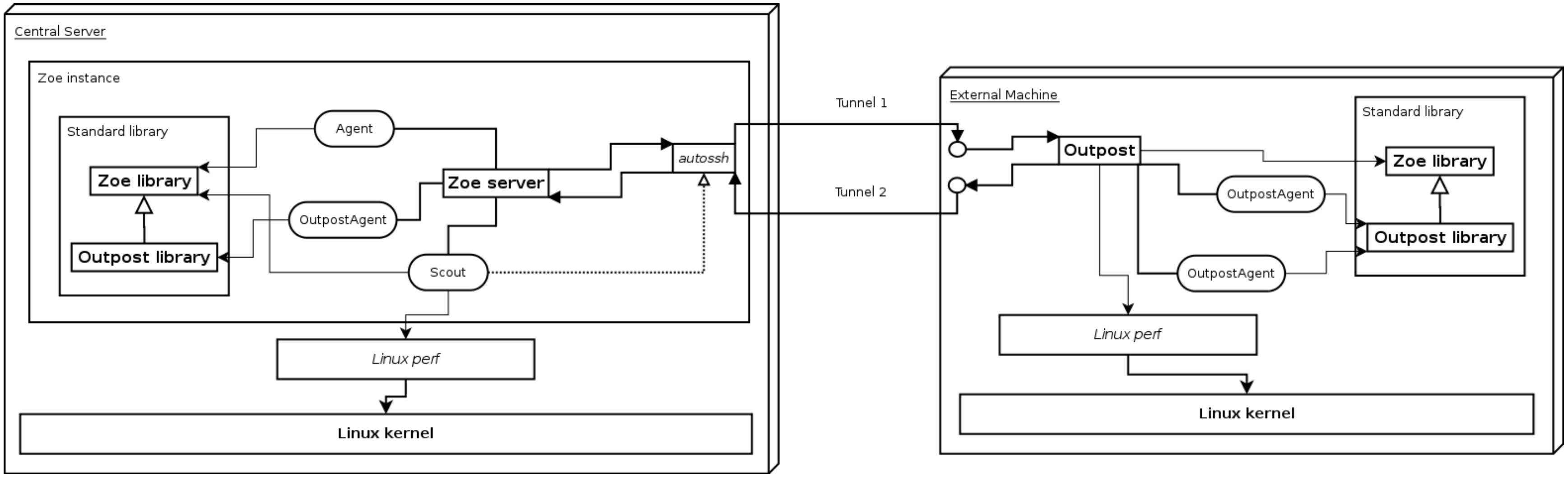


Figure 3.1: Architecture overview

3.2.1 Protocol

The outpost protocol was designed as a non-intrusive element that could be easily attached to the main Zoe library by using common core functionalities available in Zoe. The result is a fully compatible subprotocol that works entirely on the *agent-side* and does not require any internal change to the server code.

As shown previously (see *Agents* section), Zoe agents are processes which receive and send messages via socket connections. They can be easily executed in remote machines that have to be connected to the central server. However, moving agents between machines requires manual intervention from an user/administrator both for agent installation and proper configuration.

In order to automate these tasks, the outpost protocol introduces a series of *programming language independent*, and easy to implement, rules and functions that achieve the desired result by using concepts available in most, if not all, programming languages.

The migration process can be summarized in the following steps:

1. Agent is notified of migration
2. Data is serialized and stored
3. Agent is stopped
4. Agent files are copied to remote machine
5. Agent is launched in remote machine
6. Data is restored
7. Agent resumes operation

3.2.1.1 Data migration

The Java language offers RMI (*Remote Method Invocation*), which is similar to RPC in the sense of distributed computing. RMI uses a variant of the Java Object Serialization package to marshal and reconstruct objects [26], meaning that the object itself is being sent through the network, hence keeping all its information.

Approaches such as this one cannot be directly ported to other programming languages, but the concept it offers can be reused: keep as much information as possible between migrations.

Most, if not all, programming languages are able to serialize information, extract data structures from memory and store them, and deserialize it, restore the dumped information to memory. Through this mechanism, it is possible to obtain the *bytes* that represent data that the agent is working with and store them until the migration has been completed.

Zoe messages are formed by plain text, so sending bytes directly is not possible. Therefore the data has to be encoded so that it can be represented as plain text. This can be achieved by using the *Base 64* encoding which is designed to represent arbitrary sequences of octets in a form that allows the use of both upper and lowercase letters but that needs not be human readable [27].

Mixing data serialization and Base 64 encoding (in Python through the `pickle` and `base64` modules), data can be transmitted and stored as plain text through the Zoe messaging protocol, only needing a small change: the Base 64 alphabet defines the character `=` for padding the end of the data. Due to the Zoe protocol using that symbol in its key-value pairs, it is substituted with the character `[`, which **is not used in the Base 64 alphabet**, when sent and changed back at the time of deserialization.

This data (de)serialization is triggered by special messages that use the tags `travel!` when migrating and `settle!` when restoring information. An overview is shown in Figure 3.2, which omits the agent *Scout* that is explained later on. In the figure, the agent *outpostest* is notified of the migration, serializes its variables `a` and `b` and sends them back for storage. In addition, when the agent is started, it asks for its stored information and restores it.

In the Python implementation, this process is performed through four special methods that are added to the agent code when initialized:

- `__prepare__travel__()`: serializes the information as stated before. The developer may specify variables to store by using an internal list (`_scout_include` for inclusion and `_scout_exclude` for exclusion) or let use all known variables
- `__settle__outpost__(message)`: with the message specified extract the serialized information and set the variable values to the deserialized data. The lists mentioned before are taken into account to see what is going to be restored
- `__travel__()`: in the base implementation, this method simply executes `__prepare__travel__()`. Developers are expected to override this method should they need additional operations when migrating
- `__settle__(message)`: in the base implementation, this method simply executes `__settle__outpost__(message)`. Developers are expected to override this method should they need additional operations when restoring the agent after a migration

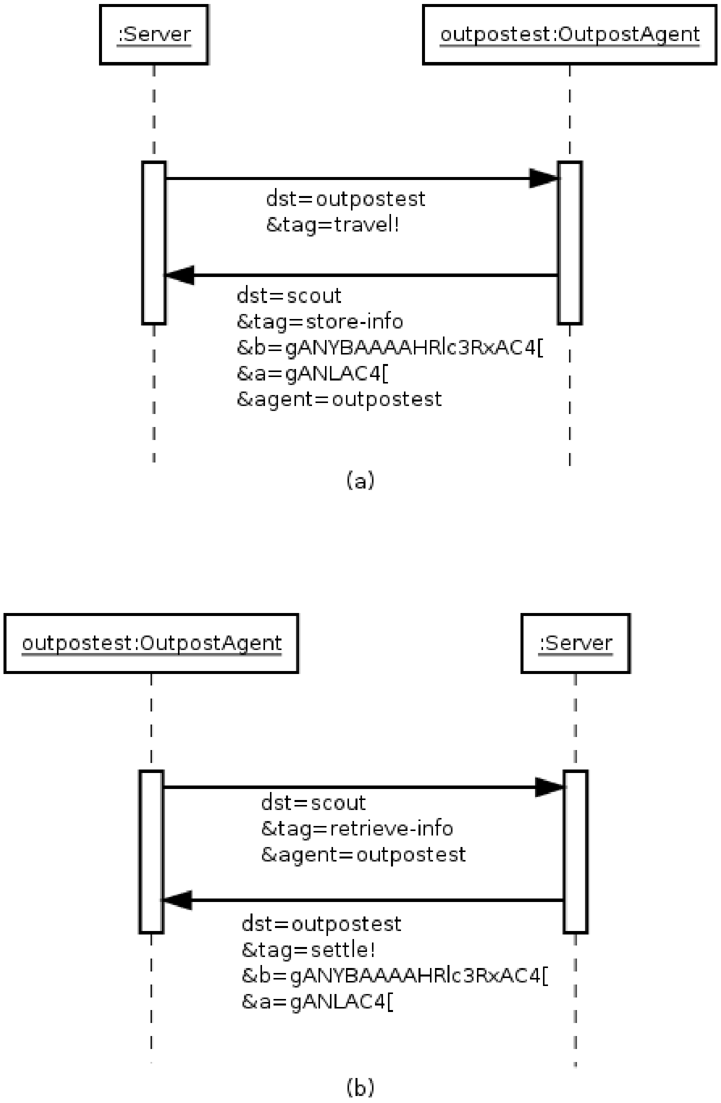


Figure 3.2: Data serialization and restoration; (a) serialization message, (b) restoration message

3.2.1.2 File migration

Another important factor in agent migration is copying files to the remote machine that will continue with the execution of the agent. This is done through a SSH connection, that is set up using public key authentication as to not store any type of password, and the `scp` utility, which allows copying files and directories through an established SSH connection.

In Python, connection is opened using the `paramiko` module and files copied using its `scp` plugin. It should be noted that this process is performed by the special agent *Scout* and hence does not need to be ported to other programming languages (e.g., the implementation is independent of other agents).

It is important to have a way of identifying the files that correspond to each agent in both local and remote systems. These files include those necessary for execution (i.e., scripts or binaries) and configuration files accessed during operation, as long as they are unique to the agent and not shared with any other agent/process. To that end, two files are considered in the protocol:

- `static`: each line of this file references the path to a file or directory that **never changes its content** (unless an user changes it explicitly), relative to the root directory of Zoe (aka "*ZOE_HOME*")
- `dynamic`: each line of this file references the path to a file or directory that **may change its content** (e.g., a database), relative to the root directory of Zoe

Given that paths are relative to the root of Zoe or the outpost, they can be easily identified in either case. Furthermore, separation of static and dynamic files increases performance of the system: the central server keeps a backup copy of the static files and these backup is copied to remote machines, but never from a remote machine to the central server, requiring only the dynamic files in this case, therefore saving bandwith and reducing the time needed to migrate.

An example of these files for the agent *outpostest*:

```
# static

agents/outpostest
etc/outpostest.conf



# dynamic
```

`etc/outpostest_dynamic.conf`

3.2.1.3 Message relaying

When agents are notified of their migration, they are given a short period of time (several seconds) to be able to complete any ongoing tasks before their process is terminated. During this grace period, they should not be able to process any message due to the imminent termination.

In order to make the downtime unnoticeable to the users, the protocol includes the concept of *message relaying*: when notified of the migration, the agent will relay any message addressed to it to the *Scout* agent without any modification. These messages are stored as is in a database, with their destination clearly identified, and delivered sequentially to the agent once migrated.

Taking into account that migrations usually do not take too much time to complete, relaying messages and delivering them after migration helps prevent availability issues without greatly affecting performance of the service.

Note that this process can be initiated manually at any time, but is supposed to be executed automatically when data has been restored after a migration (see Data migration).

3.2.1.4 Additional operations

Software that comprises the agents may have additional requirements, usually modules and/or libraries specific to the language. The convention for these additional components is to place them in the path `agents/AGENT_NAME/lib` so that they do not affect other agents, for instance with version mismatch.

These libraries can usually be installed without needing root/administrator access to the system through an additional tool. In the case of Python, the most used tool for this purpose is `pip`, which is able to download and install modules to the specified directory from the official Python Package Index (PyPI).

The main problem with dealing with such libraries is that, sometimes, they may require compilation of specific components, for instance C language bindings for some low level functionality. Such binaries cannot be directly copied to a remote machine that has a different architecture (e.g., x64 to arm64). Instead, the protocol allows the developer to specify additional commands or operations required when migrating an agent through two special files, similar to the *File migration* scenario:

- `premig`: each line of this file is a command that is executed in a shell, either locally or remotely depending on where the migration is performed, **before the files are copied**
- `postmig`: each line of this file is a command that is executed in a shell, either locally or remotely depending on where the migration is performed, **after the files are copied** (directory structures have been created)

This implementation is based on the mechanics employed by the *fumi* project, which is a simple tool for deploying remote applications [28] and uses a similar approach in its deployment file specification. The following is an example obtained from the `postmig` file of agent *madtrans*:

```
pip3 install -t ${ZOE_HOME}/agents/madtrans/lib requests
```

Note that the token `${ZOE_HOME}` is replaced accordingly depending on the root of the Zoe server or outpost. The previous line simply installs the `requests` module for Python in the `agents/madtrans/lib` directory when the directory tree has been created (after copying files from central).

3.2.1.5 Standard library

As mentioned previously, this protocol should work on top of an existing Zoe implementation without modifying internal components, such as the server. In order to achieve this, it was devised as an addition to the standard Zoe library that implements the relevant functionalities specified in the previous section as an *importable* module for the agents.

The downside of this approach is that agent developers have to modify their agents in order to use the outpost protocol. However, for the Python implementation, and hopefully for future language implementations, this fact has been alleviated by offering an interface that appears to be very simple.

Figure 3.3 shows two blocks of code: one for a *regular* Zoe agent (left block) and the other for the same agent using the outpost interface (right block). The agent is identified in the server as *outpostest* and recognizes three different actions/messages:

- `add`: adds 1 to the internal counter (`a`) and prints its new value to the log file
- `string`: prints the value of its internal string variable `b` to the log file

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import zoe
from zoe.deco import Agent, Message

@Agent(name="outpostest")
class Outpostest:

    def __init__(self):
        self.a = 0
        self.b = 'test'

    @Message(tags=['add'])
    def add(self, parser):
        self.a += 1
        print(self.a)

    @Message(tags=['string'])
    def string(self, parser):
        print(self.b)

    @Message(tags=['echo'])
    def echo(self, parser):
        msg = parser.get('msg')
        print('ECHO: ' + msg)
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import zoe
from zoe.outpost import OutpostAgent,\
        Message

@OutpostAgent(name="outpostest")
class Outpostest:

    def __init__(self):
        self.a = 0
        self.b = 'test'

        self._scout_include = ['a', 'b']

    @Message(tags=['add'])
    def add(self, parser):
        self.a += 1
        print(self.a)

    @Message(tags=['string'])
    def string(self, parser):
        print(self.b)

    @Message(tags=['echo'])
    def echo(self, parser):
        msg = parser.get('msg')
        print('ECHO: ' + msg)
```

Figure 3.3: left, regular Zoe agent; right, agent using outpost interface

- `echo`: prints the received message (`msg` key) to the log file

The code is very simple, meaning that it can be easily converted to an outpost agent by including the `static` and `dynamic` files (see *File migration*) and performing the shown modifications in the code (right block). Through these simple modifications the agent is now ready to work as part of the outpost schema:

```
from zoe.outpost import OutpostAgent, Message
```

The previous line indicates that, instead of importing the original Zoe library (`zoe.deco`), it should use the outpost one, which in fact inherits all the functionalities offered by the Zoe one (`OutpostAgent` instead of `Agent`).

```
@OutpostAgent(name="outpostest")
```

In a similar way to the import statement, the previous line simply replaces the original `Agent` class with the new `OutpostAgent` class that implements the data migration part of the protocol (see *Data migration*).

Lastly, the following additional line identifies internal variables that will be serialized in case a migration occurs (see *Data migration*), which are the only two variables present (`a` and `b`):

```
        self._scout_include = ['a', 'b']
```

Considering that every possible complex scenario cannot be accounted for in the library itself, developers have the possibility of creating their own additions to the migration process by overriding the internal methods `__travel__()` and `__settle__(message)`, as shown in the *Data migration* section. By doing so, they would have complete control over the behaviour of their agent in the migration process.

3.2.2 SSH tunneling

Considering that outposts are running on external/remote machines, a communication channel is needed for message delivery. Outposts relay messages from and to the agents (see *Outpost*) and should have a direct connection to the central server.

While this could be achieved by opening the port used by the outpost to the public (e.g., allowing connections from anywhere on the Internet) and configuring firewall rules to allow access from a specific IP address, the one where the central server is located, this solution does not escalate very well with high number of machines (higher number of IPs to add to the firewall) and usually requires intervention from an administrator.

Instead of relying on firewall for security, the outpost system employs SSH tunnels. SSH tunneling is a method that creates a virtual tunnel over the network, with client and server being the endpoints of the tunnel [29]. The main advantage of using tunneling is that all traffic between the machines is encrypted using standard security that does not modify the messaging protocol at all.

In the case of this implementation, the ends of the tunnel would be the central Zoe server and the remote machine running the outpost. This implies that outpost machines need to have a SSH server configured to be able to perform the connection of the tunnel.

In the practice, all network traffic that enters one side of the tunnel in machine *A* can be read through the other end of the pipe in machine *B*. As these ends are associated to specific ports in both systems, it is possible to launch a server that awaits connections from the corresponding port number. However, since attaching a server to a port implies that only one-way communication (i.e., receiving data) is available for that tunnel, the protocol expects two SSH tunnels for each outpost machine:

- *Tunnel 1* is attached to the outpost server in the remote machine and to a previously configured port number in the central machine
- *Tunnel 2* is attached to the Zoe server in the central machine and to a previously configured port number in the outpost machine

Moreover, by using the *localhost* domain, this integration can be extended to any number of machines while server and agents have the impression of being in the same machine.

These connections are managed by the *Scout* agent and use the command `autossh` instead of the regular `ssh` (both available in most GNU/Linux software repositories). This utility launches an instance of `ssh` for the connection, but is also able to monitor and reinitialize it when necessary, meaning that tunnels will only close when manually closed or there have been network issues.

Figure 3.4 shows how this architecture works: the Zoe server is listening on port *30000* of the central server (marked as receiving end of *tunnel 2*), the outpost is listening on port *30000* of the remote machine (marked as receiving end of *tunnel 1*) and port number *29999* is used in both machines as the end that forwards data to the other machine.

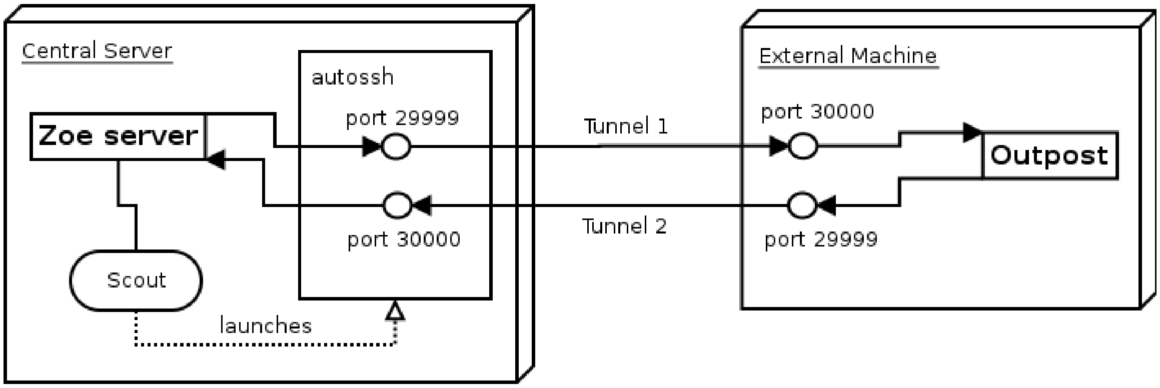


Figure 3.4: SSH tunneling architecture

3.2.3 Outpost

The outpost serves as a *gateway* between migrated agents and the central server. It is basically an asynchronous server that replaces the Zoe server in remote machines, using Zoe environment variables (host, port, etc.) to identify itself as the server the agents should be connecting to.

This, however, is not the only task performed by the outpost program. Apart from message and agent routing, it also satisfies special requests with specific operations named *actions*, which are triggered with a message addressed to the outpost itself using its unique ID.

All the functions and methods have been developed following a modular approach that facilitates the debugging and the implementation of more complex functions by means of simpler ones.

The following shows an overview of the outpost implementation.

3.2.3.1 Routing

When a message arrives, the outpost determines whether it should relay it to an agent it recognizes (in the same machine), to the Zoe server (through the tunnel) or if it contains a special action the outpost itself must perform (see Actions section).

Messages are relayed to the server if their destination is the server itself or the destination agent is unknown. The process to determine the destination of a message is as follows:

1. Read the destination tag of the message (`dst`) and discard the message if no tag

is present

2. Check the internal dictionary/hashmap maintained by the outpost. This map contains key-value pairs where the identification key is the name of the agent and the value is the port number where it is located
3. If the agent is not found in the map, the Zoe configuration file containing the mapping is checked (`etc/zoe.conf`). This file is used to populate the map on startup and to update the map when needed
4. If the agent is found after any of the previous two steps, the message is delivered
5. When the agent is not found anywhere, the outpost supposes it is located in the central machine and relays the message to the Zoe server

The aforementioned works surprisingly well with the Zoe architecture but has an additional issue: the Zoe server may resend a relayed message to the outpost indefinitely. Such case is observed when the outpost does not recognize the agent but Zoe still thinks that the agent is located in the outpost. In practice this may result in a possible, and unintended, saturation of the communication channel given that the message is sent back and forth *"forever"*.

To prevent this behaviour, the outpost performs a small modification to the message when relaying to Zoe. This modification consists on the inclusion of an additional tag `_outpost_replay=0` that is automatically ignored by the Zoe server, given that it is an unknown element, but remains in the message when the server sends it back to the outpost.

Its purpose is to identify how many times the message has been through the outpost, and its value is incremented by one by the outpost each time a message containing the tag is received. When the value of the tag reaches 5, **the message is automatically discarded** by the outpost.

3.2.3.2 Actions

When a message has the outpost as its destination, it may specify a special `action` tag that indicates the special operation to perform. Should this tag not be present, the outpost shall discard the message.

The following sections explain in detail each of the actions the outpost performs.

Gathering agent information

Triggered by the `action=gather-agents` tag, this is the most complex action implemented. The outpost measures the MIPS (*Million Instructions Per Second*) the agent is currently executing by means of the Linux *perf* tool (see *Resource gathering* section).

However, it is not up to the outpost to decide what to do with this information. Instead, it simply relays the data to the *Scout* agent, that will use it in the process of load balancing (see *Load balancing* section).

Refreshing users information

Triggered by the `action=refresh-users` tag. This action is implemented in order to increase compatibility with agents that access the information of Zoe users, for instance to check for permissions.

Periodically, the outpost is sent a serialized copy of the `etc/zoe-users.conf` file from the central server. When received, the data is deserialized and the contents of the file updated. By doing so, an administrator would only need to change the original file to have the changes replicate to the other machines.

Adding an agent to the outpost

Triggered by the `action=add-agent` tag. With the name of the agent and port included in the message, the outpost updates the internal map and configuration file (see *Routing* section) with the new agent.

This operation is required in order to recognize an agent for message relaying.

Removing an agent from the outpost

Triggered by the `action=rm-agent` tag. As opposed to the previous action, the agent is removed from the map and the configuration file so that the outpost will not try to deliver any message and relay them to the central server instead.

Cleaning agent files

Triggered by the `action=clean` tag. When an agent is migrated from the outpost to another machine, its static files (see *File migration* section) have to be removed from the system.

Given that the outpost is unaware of which files belong to the agent in order to reduce complexity of the implementation, the message contains a list of paths that should be removed sent by the *Scout*.

Launching an agent

Triggered by the `action=launch` tag. Starts an agent process.

The most common scenario where this action is applied is when an agent has been migrated to the outpost: its process is started and the data restoration process (see *Data migration*) begins.

Stopping an agent

Triggered by the `action=stop` tag. Stops an agent process.

Similarly to the previous action, this one is used when an agent is going to be migrated to a different machine. Note that at this point, all the data of the agent has been previously stored.

Reloading configuration

Triggered by the `action=reload` tag, the outpost will read the configuration files and update its internal variables, including the router. As it is not uncommon to perform maintenance or modify settings while the outpost is running, this action tries to alleviate the process by providing a way to update those values without having to restart the outpost (with the downtime that would involve).

Registration with Zoe

While not an action that can be triggered manually, this functionality is very important for the correct operation of the system.

Firstly, when an agent is started, it may send a *register* message to the server to indicate its name and port. It is the job of the outpost to intercept that message and update its internal routing map. However, the Zoe server must know where to find the agent, therefore the outpost builds a new message for the server in order to **register the agent in the tunnel**.

Taking into account that the outpost knows the port used by the tunnel in the central machine (set in the configuration), the resulting relation for Zoe would be that the agent can be contacted through the tunnel port number (e.g., *29999*) that connects directly to the outpost server.

Secondly, apart from registering its agents on startup, the outpost registers *itself* with Zoe (also with the tunnel port number). This allows addressing the outpost through the server by reusing its original communication architecture (see *Communication* section), greatly simplifying the implementation as a secondary communication channel between *Scout* and outposts is not required.

3.2.4 Scout

The Scout works as a regular Zoe agent (see *Agents* section): it receives messages from users, discarding them if the user does not belong to the *admin* group, and performs periodic management tasks.

As a normal Zoe agent that implements the original standard library, the Scout is able to access all the communication channels known to Zoe. This allows for communication by means such as Jabber/XMPP, Telegram or email. Such communication interface was implemented in a generic manner so that any future channel may interact with the agent.

The functionalities implemented in this agent are much more complex than those implemented in the outpost code, mainly due to the fact that the Scout has to manage all the outposts and knows where each *movable* agent is located at every moment. This management is performed on a *best-effort* basis, as each outpost may respond at different times and deadlines cannot be guaranteed in such architecture.

All the functions and methods have been developed following a modular approach that facilitates the debugging and the implementation of more complex functions by means of simpler ones.

The following sections detail both periodic and message-triggered tasks.

3.2.4.1 Periodic tasks

As shown previously (see *Agents*), Zoe agents are small asynchronous servers that execute tasks on demand. Introducing synchronous tasks to this architecture helps perform special operations without any intervention from the users. In the case of the Scout, these tasks are mostly related to communication with outposts and the agents that are able to migrate.

Gathering agent information

The purpose of this task is to gather the MIPS of each agent at *regular* intervals. The implementation is the same as the one in the outpost (see *Gathering agent information* outpost section) for agents located in the central machine, where the Scout is located.

However, this task is also in charge of triggering the same functionality in all the outposts configured by sending individual messages to each of them. Note that these messages expect no immediate response, and the result of the remote operation is parsed in a separate task.

Refreshing local information

Accounting for the possible scenario in which an administrator configures an additional outpost or installs a new *outpost agent*, the Scout periodically checks for changes in configuration files (outpost list, main scout configuration) and directories (agent static and dynamic rules).

This information is stored in a database that contains agent resources and relative locations, as well as status of each outpost, employing a relational architecture to that end.

Sending users configuration

As mentioned previously (see *Refreshing users information* outpost section), several agents require access to the users configuration file (`etc/zoe-users.conf`) to check for permissions and other relevant information. While this file could have been accessed by opening a SSH connection from the outpost, that would imply duplication of keys and higher complexity of the protocol.

Instead, and benefiting from the fact that both Scout and outpost are written in Python, the whole contents of the file are serialized and sent to each of the outposts known to the Scout.

Load balancing

With the MIPS obtained for each of the agents and knowing their current locations and capacity of each machine, the Scout is also in charge of performing the load balancing. In essence, this task is a proxy between the load balancing algorithms (see future *Load balancing* section) and the internal migration algorithm implemented in the agent.

In a real time system, the MIPS gathering would be performed just before performing the balancing to make sure the balancing is as accurate as possible. As this is not possible, the load balancing task is executed less frequently than the gathering one in order to work with recent enough information.

3.2.4.2 Triggered tasks

These tasks may be triggered by either an user, through any communication channel, by outposts replying to a request (e.g., gather agent MIPS), or by migrated agents (e.g., save/restore information).

Closing a SSH tunnel

SSH tunnels are automatically closed when the main Zoe server is stopped. However, there are times when an administrator may want to close a tunnel manually: the remote machine is going to be stopped, the outpost is to be restarted with new configuration, etc.

This task can only be triggered by a message from an administrator.

Holding an agent

Load balancing moves agents across all the machines automatically. While this behaviour is the one expected in a fully automated environment, it would be inefficient in scenarios where an agent has to stay in a specific machine, for instance to monitor that machine or to access a resource which is only available there.

For such cases, administrators can flag an agent as *held* in place. The Scout maintains a list of agents on hold and ignores them when the load balancing is launched, effectively keeping them in their current location. However, the MIPS of these agents are still considered for the hypothetical loads of the machines in which they reside and can, therefore, affect the balancing.

This task can only be triggered by a message from an administrator.

Launching an outpost

Once a SSH tunnel has been established, the Scout can proceed to launch the remote outpost. This process is done automatically for each outpost when the scout is started, but can also be triggered by an administrator when needed. The only requisite is that the SSH tunnel be open at the time of launch.

There is an additional consideration to be taken into account: as the *real* status of the outpost process is not known, the Scout **restarts the outpost server and the agents present there**. While this involves some downtime, it makes sure that all newest configurations are used when launched.

This task can only be triggered by a message from an administrator.

Making a backup

As shown in *File migration*, agent files are identified by *static* and *dynamic* files. A backup of the static files is always kept in the central machine so that they can be transferred directly to the required outpost (or copied if the agent is in central server). This backup is just a simple directory containing the tree structure for all the files belonging to the agent relative to the Zoe root path.

Said tree can be directly copied on top of the root Zoe/outpost path in order to *migrate* the files, and is recreated each time the agent is migrated from central to a different outpost. The reason for this is because an agent can only be updated/installed by means of the Zoe agent manager when it is in the central machine. That is why the directory tree is always kept updated when migrated.

This task may be triggered by a message from an administrator, but is also triggered automatically when, as specified before, an agent is migrated from the central machine.

Migrating an agent

The *Scout* is in charge of migrating agents to other machines. This process checks the origin and destination of an agent and performs the notification to the agent (to prepare for migration and stop), file copying and outpost notification (to add and launch the agent).

Notifications are done through regular Zoe messages, but file copying and special commands (see *Additional operations* section) are performed by means of a SSH connection (when dealing with remote migrations). After the migration, all the relevant databases and files are updated.

This task is triggered by either a message from an administrator wanting to migrate an agent to a specific location or automatically when performing load balancing.

Opening a SSH tunnel

Even though tunnels are automatically opened when the agent is started, administrators are also able to open them when needed, for instance after closing it to update its configuration.

Internally, this functionality calls the `autossh` tool to maintain the tunnel(s) open. The command executed has the following form:

```
autossh -L <LT>:localhost:<RP> -R <RT>:localhost:<LP> <user>@<host>
```

Where:

- `LT` is the port number to use as local tunnel end (in central server)
- `RP` is the remote port number to which the tunnel is connected (in remote machine, outpost port)
- `RT` is the port number to use as remote tunnel end (in remote machine)
- `LP` is the local port number to which the tunnel is connected (in central server)
- `user` is the username to use in the connection as configured in the remote system
- `host` is the IP or hostname of the remote machine

Note that the information is extracted from the outpost list `etc/scout/outpost.list` file and that the process results in two tunnels being opened for the outpost (read and write) as shown in the *SSH tunneling* section.

In addition, this process supposes that public key authentication has been properly configured.

Agent information retrieval

After an agent has been migrated, it asks the Scout for its stored information, if any. The Scout then looks for the serialized data in its database and sends a message to the migrated agent if there is information to deliver (see *Data migration* section).

In case no information is found, any message stored for delayed relaying is sent, as this process would be triggered by the agent after restoring its information. Given that there is no information to restore, the agent will only trigger that retrieval if the developer indicated it explicitly.

This task may be triggered by an administrator for testing purposes, although it should be triggered by agents in a normal execution.

Message retrieval

Similar to the previous case, this task delivers any pending messages the agent may have in order to complete the operations requested during the time it was preparing for the migration. Note that this supposes agent information has been previously restored.

This task may be triggered by an administrator for testing purposes, although it should be triggered by agents in a normal execution.

Showing agent locations

At a given time, administrators may want to check current locations of all the outpost agents. This task returns a list of locations and the agents present in them in a plain text manner that can be represented by all the communication channels supported by Zoe. The following is an example response:

```
# Agent locations

outpost_pi
---------


central
---------
outpostest
madtrans
fibonacci
dummy1
dummy2
dummy3
```

Showing agent status

This task provides information on the status of agents. The base implementation includes the following information:

- Whether or not the agent is *on hold*
- Current location of the agent
- Latest obtained MIPS of the agent
- Date and time when the information was updated

A response from this task would look like the following:

```
# Agent status

outpostest
---------
FREE
- Location: central
- MIPS: 0.008592
- Last update: 10-05-2016 17:54:53

madtrans
---------
FREE
- Location: central
- MIPS: 0.008264
- Last update: 10-05-2016 17:54:53
```

Showing outpost status

In the case of outposts, the status information delivered consists of the configured values in the outpost list file, if the outpost is running, when the information was updated and, in the case of the central server, the load balancing algorithm in use. An example response would be as follows:

```
# Outpost status

central
---------
ONLINE
- Balancer: none
- MIPS: 5217.933559
- Priority: 3
- Last update: 06-04-2016 20:42:43

outpost_pi
---------
ONLINE
```

```
- Host: 192.168.1.50
- Remote port: 30000
- Local tunnel: 29999
- Remote tunnel: 29999
- Remote directory: /home/zoe_outpost
- MIPS: 132.287850
- Priority: 2
- Last update: 06-04-2016 20:42:43
```

Stopping an outpost

As opposed to the previous case, this task stops an outpost **and all the agents running in it** on demand. Note that stopping an outpost does not close the tunnel implicitly, requiring another manual operation to do so.

This task can only be triggered by a message from an administrator.

Agent information storage

The Scout stores whatever information an agent sends serialized before its migration. This information is stored in a relational database and is only retrieved when asked for.

This task can/should only be triggered by a migrating agent.

Agent resource storage

Previously, it was stated that the Scout sent messages asking for agent MIPS to each of the outposts configured. This task receives and parses the response messages from the outposts and updates the database records with the new information.

This task can/should only be triggered by an outpost.

Agent message storage

As said before, when an agent enters the migration preparation phase, it ignores all incoming messages and instead relays them to the Scout. This task receives these messages and stores them in a database for further delivery when asked for.

This task can/should only be triggered by a migrating agent.

Freeing an agent

As opposed to *holding an agent*, an administrator can use this functionality to flag an agent as free, meaning that it will be moved automatically if needed in the load balancing process. As before, the list of held and free agents is updated after flagging an agent.

This task can only be triggered by a message from an administrator.

3.2.5 Resource gathering

Resource gathering in the system is performed through the Linux *perf* tool. This utility was introduced in the Linux kernel in version 2.6.31 [30] and is available in most kernels bundled with GNU/Linux distributions (kernel 4.5.5 was the latest stable version at the time of writing).

By using *perf*, both *Scout* and outposts can access several counters and events, although the only information used is the number of instructions executed by an agent in a given period. Such number is obtained by spawning a new process that will run the *perf* executable for each agent in execution that will attach to the agent PID (*Process ID*) for 5 seconds.

The PIDs for all agents are available in the `var/` directory in the Zoe root directory, meaning that no additional modification is required for this process, and once the *perf* processes terminate the MIPS value is calculated for each agent.

In this case, only the number of instructions performed by the processor and the time taken to complete the LINPACK benchmark were necessary. In order to obtain that information, *perf* was used to launch the benchmark and obtain the events and time. As this results in the total number of instructions and time in seconds, the MIPS have to be calculated through the following formula:

$$\text{MIPS} = \frac{\text{Instructions}}{\text{Time}_{seconds} \cdot 10^6} \tag{3.1}$$

Note that any other benchmark may be used for this purpose and the decision of which benchmark to use must be taken by the administrator(s) configuring the systems.

3.2.6 Dashboard

The dashboard is a small program, with a GTK+ based user interface, used as a secondary way to view current data managed by the *Scout* at a glance. The data that can be visualized in the dashboard is current agent locations and current resources stored for those agents.

This program may be executed in the same machine as the main Zoe instance (central server) or in a remote server, although it would require SSH credentials to fetch a copy of the databases and relevant files periodically. More details on how the dashboard is used can be found in the *Dashboard* section of *Appendix B*.

It should be noted that, while useful for easy visualization of locations, the recommended (and more efficient) way of interacting and viewing current data from the system is by using the commands implemented in the *Scout*. These commands are implemented as an internal component in the Zoe architecture and thus are fully integrated with the messaging system without requiring any additional workaround (such as registration with the server for the replies).

3.3 Load balancing

Load balancing is a functionality that was implemented expressly with the idea of being easy to extend. In the practice, the *Scout* is not aware of how the load balancing algorithms work and simply inputs specific information and receives the output of the algorithms indicating the new destinations of the agents.

On one hand, the information that is passed as argument to the algorithm function is a map which contains agents, grouped by their current outpost, their location, their current MIPS, the timestamp of the latest update to their information, whether they are on hold or not and the priority of the outpost. This is basically all the information the *Scout* has available for each of the agents. On the other hand, the output obtained from the algorithms is a map that groups agent names by their new destination.

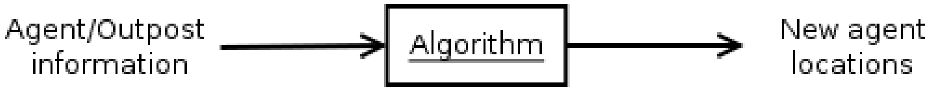


Figure 3.5: Load balancing algorithm

Even though algorithm implementations themselves are considered a *black box*, as illustrated by Figure 3.5, the *Scout* knows which algorithm to execute by an unique identification tag that is set in the configuration files of the agent. By completely separating the algorithm implementation from the rest of the agent code, other developers would be able to include new algorithms without needing to understand the rest of the software as long as they work with the parameters specified previously.

The work implemented includes two base algorithms, **equal balance** and **priority** (which will explained in the next sections), although as stated before, it should be possible to easily extend these algorithms or add new ones in the future.

3.3.1 Equal balance algorithm

The aim of this algorithm (identified by the tag *equal* but also referred to as *balanced*) is to try to maintain, as much as possible, an equal load among all outposts and central server. This process considers both agent MIPS and machine capacity (as measured with *LINPACK* or other benchmark) in order to calculate current load percentage of the machine the agent is located in and hypothetical load percentage of the machine to which the agent could be moved. The algorithm is as follows:

1. Parse MIPS of the agents and capacity of each outpost
2. Initialize a list of outposts with their current load percentage
3. Continue with next agent or exit if there are no agents remaining
4. If the agent cannot be moved, calculate the load percentage and add it to the load of its current machine. Continue from **3**
5. Order the outpost load list in ascending order (*min* to *max*)
6. Calculate the new hypothetical load percentage for the first outpost in the list and add the agent to the outpost in the resulting map. Continue from **3**

Calculation of the percentages is performed by the following operation:

$$\text{Load}_{percentage} = \frac{\text{MIPS}_{agent}}{\text{MIPS}_{machine}} \tag{3.2}$$

The performance of this operation greatly depends on the processor used to calculate it: considerable precision is required for the calculation as, depending on the MIPS capacity of the machine, the resulting percentage may be very small (i.e., *0.00070678* out of 1).

By sorting the hypothetical outpost loads in ascending order, the algorithm makes sure that the outpost with the lowest load is filled first. Even though the result is an approximation, given that the agents are balanced in a *first come first served* manner and without considering all the MIPS before the balancing, the result is deemed appropriate taking into account that Zoe agents do not usually have a very high load.

3.3.2 Priority algorithm

The priority algorithm (identified by the tag *prio*) uses the same information as the *equal balance* algorithm (agent MIPS and machine MIPS capacity) plus an additional setting that is found in the outpost list file (`etc/scout/outpost.list`) used to specify the priority of a machine for this algorithm.

Such setting consists of an integer value, ideally starting with 1, which translates into a **lower priority as the value increases**. For instance, a machine with a value **1** would have higher priority than a machine with a value **4** and will be considered first in the algorithm. The reason for assigning priority to values in reverse order is mainly so that lower priority levels can be easily added (simply increment the number) while keeping the highest priority intact (value 1).

The balance algorithm is as follows:

1. Parse MIPS of the agents and capacity and priority of each outpost
2. Initialize a list of outposts with their current load percentage
3. Sort the outposts by priority (ascending)
4. Continue with next agent or exit if there are no agents remaining
5. If the agent cannot be moved, calculate the load percentage and add it to the load of its current machine. Continue from **4**
6. Calculate the new hypothetical load percentage for the first outpost in the list.
7. If the load exceeds the limit, continue with next in the list until a proper destination is found
8. If all the hypothetical loads exceed the limit, send the agent to the central server by default (updating its load percentage)
9. Continue from **4**

As shown, this algorithm performs the same calculations as the *equal balance* algorithm but assigns the destinations by user-defined priority rather than current load. The **limit** mentioned in steps 7 and 8 is a value of **0.8** (80%) that was set as an algorithm parameter in order to prevent overload of the machines where possible.

Finally, the reason for sending agents to the central server if no suitable destination is found in the algorithm (i.e., no destination can take the load) is twofold: the central server is assumed to have more resources available, considering it has to deal with regular agents, and this way the *leftover* agents are easily localized.

3.4 Class diagrams

Figure 3.8 shows the main UML class diagram of the project. This includes the relevant classes inherited from the standard Zoe Python library and how each component interacts with the others.

Regarding the *Scout* and *outpost* classes, note that each of them has a set of additional modules contained in a package/library (*libscout* and *liboutpost* respectively) custom made for them. Each function and class in these modules was created to facilitate development and maintenance of the methods for both *Scout* and *outpost* classes. Therefore they will not be explained in further detail due to being building blocks that form said methods. A brief summary of each module would be:

- **libscout**: *Scout* helper library
  - *algorithm*: contains load balancing algorithm implementations
  - *book*: implements singleton classes to ease access to the local database
  - *db*: defines the database models used to store agent and outpost information
  - *log*: contains an utility function to configure logging for the agent
  - *messages*: contains functions that generate various Zoe protocol messages
  - *static*: contains static constants and objects (such as those defined in the *book* module) used throughout execution
  - *util*: implements functions used to compose *Scout* methods
- **liboutpost**: *outpost* helper library
  - *actions*: implements outpost actions
  - *log*: contains an utility function to configure logging for the server
  - *messages*: contains functions that generate various Zoe protocol messages
  - *util*: implements functions used to compose *outpost* methods and actions

There are, however, two components of the *libscout* library that require further explanation. The first of them is the *algorithm* module, shown in Figure 3.6, which defines a singleton class (`Balancer`) that can be easily extended to include new algorithms by adding a method and its identifier to the `_algorithms` dictionary.

The second module is the *db* module, which contains the database models and is shown in Figure 3.7. These models are used in the data migration process (see *Data migration*) message relaying (see *Message relaying*) and resource/MIPS storage:

- `AgentInfo`: stores serialized data of the agent
- `AgentMessage`: stores relayed messages due to migration
- `OutpostZone`: stores current status of known outposts
- `AgentZone`: stores current status of known agents

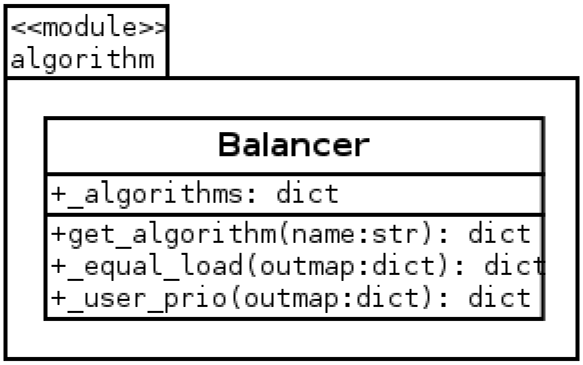

<<module>>
algorithm
Balancer
+_algorithms: dict
+get_algorithm(name:str): dict
+_equal_load(outmap:dict): dict
+_user_prio(outmap:dict): dict

Figure 3.6: *libscout algorithm* module

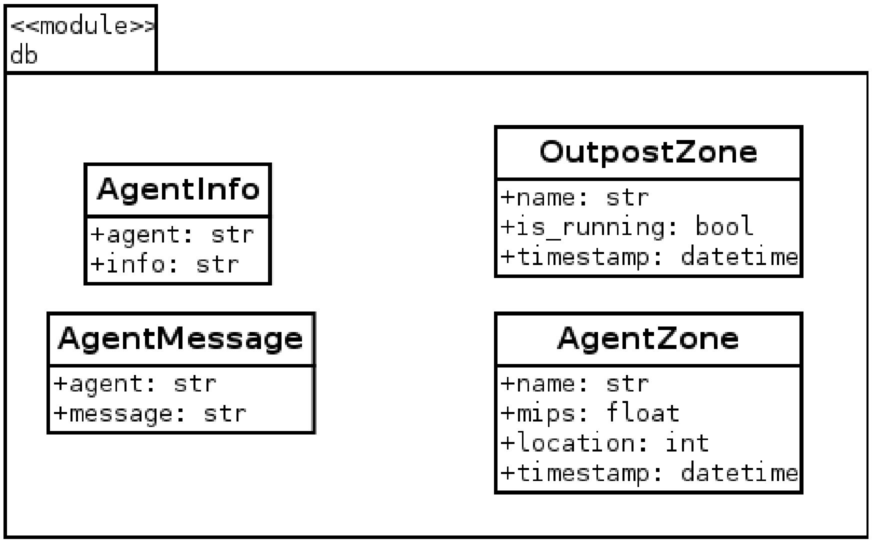


Figure 3.7: *libscout db* module

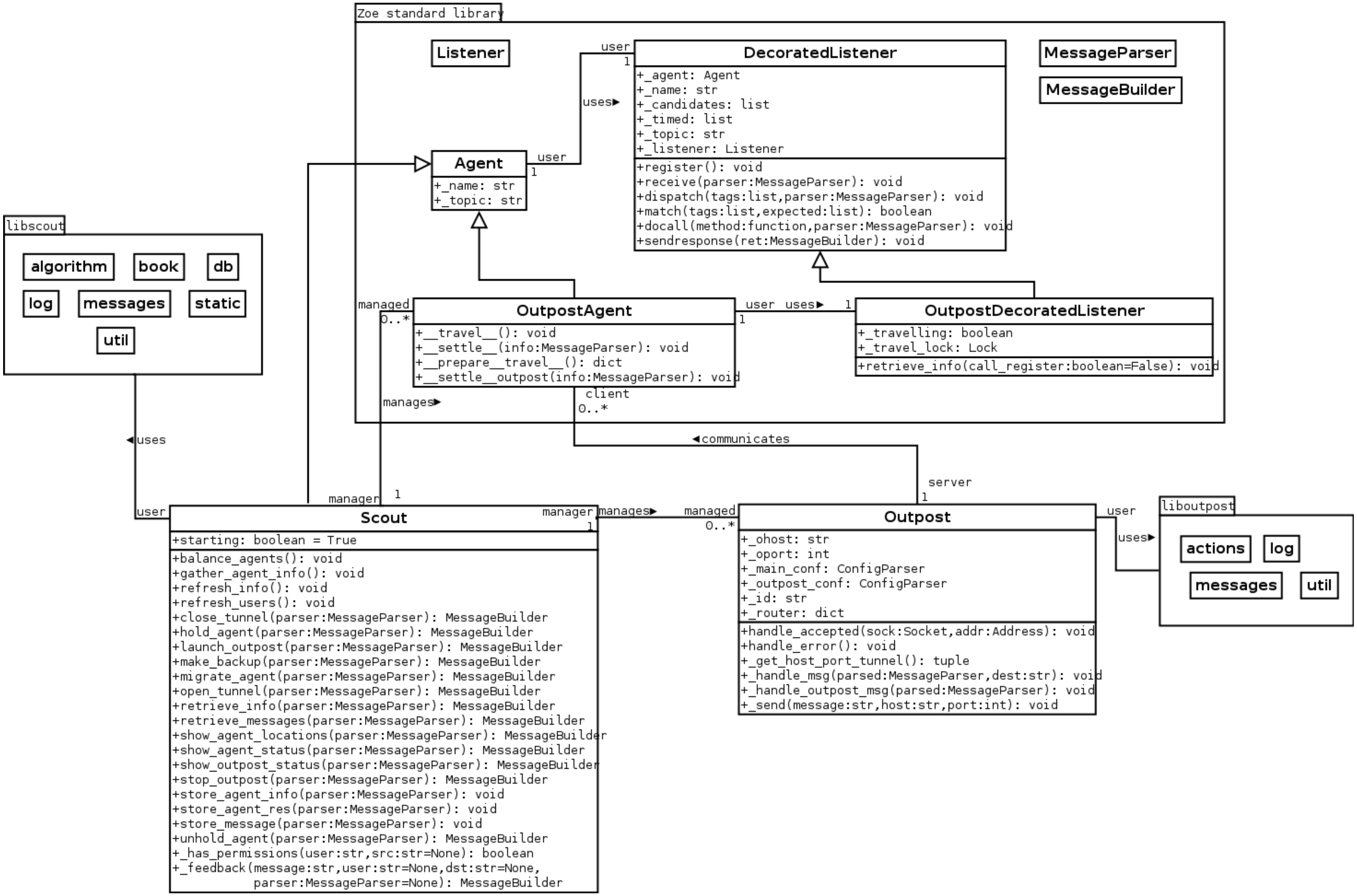

Zoe standard library
Listener
DecoratedListener
+_agent: Agent
+_name: str
+_candidates: list
+_timed: list
+_topic: str
+_listener: Listener
+register(): void
+receive(parser:MessageParser): void
+dispatch(tags:list,parser:MessageParser): void
+match(tags:list,expected:list): boolean
+docall(method:function,parser:MessageParser): void
+sendresponse(ret:MessageBuilder): void
MessageParser
MessageBuilder
user
1
uses▸
Agent
+_name: str
+_topic: str
user
1
OutpostAgent
+__travel__(): void
+__settle__(info:MessageParser): void
+__prepare__travel__(): dict
+__settle__outpost(info:MessageParser): void
managed
0..*
user
uses▸
1
1
OutpostDecoratedListener
+_travelling: boolean
+_travel_lock: Lock
+retrieve_info(call_register:boolean=False): void
manages▸
client
0..*
◂communicates
server
1
libscout
algorithm
book
db
log
messages
static
util
◂uses
user
manager
1
Scout
+starting: boolean = True
+balance_agents(): void
+gather_agent_info(): void
+refresh_info(): void
+refresh_users(): void
+close_tunnel(parser:MessageParser): MessageBuilder
+hold_agent(parser:MessageParser): MessageBuilder
+launch_outpost(parser:MessageParser): MessageBuilder
+make_backup(parser:MessageParser): MessageBuilder
+migrate_agent(parser:MessageParser): MessageBuilder
+open_tunnel(parser:MessageParser): MessageBuilder
+retrieve_info(parser:MessageParser): MessageBuilder
+retrieve_messages(parser:MessageParser): MessageBuilder
+show_agent_locations(parser:MessageParser): MessageBuilder
+show_agent_status(parser:MessageParser): MessageBuilder
+show_outpost_status(parser:MessageParser): MessageBuilder
+stop_outpost(parser:MessageParser): MessageBuilder
+store_agent_info(parser:MessageParser): void
+store_agent_res(parser:MessageParser): void
+store_message(parser:MessageParser): void
+unhold_agent(parser:MessageParser): MessageBuilder
+_has_permissions(user:str,src:str=None): boolean
+_feedback(message:str,user:str=None,dst:str=None,
parser:MessageParser=None): MessageBuilder
manager
1
manages▸
managed
0..*
Outpost
+_ohost: str
+_oport: int
+_main_conf: ConfigParser
+_outpost_conf: ConfigParser
+_id: str
+_router: dict
+handle_accepted(sock:Socket,addr:Address): void
+handle_error(): void
+_get_host_port_tunnel(): tuple
+_handle_msg(parsed:MessageParser,dest:str): void
+_handle_outpost_msg(parsed:MessageParser): void
+_send(message:str,host:str,port:int): void
user
uses▸
liboutpost
actions
log
messages
util

Figure 3.8: Proposed solution class diagram

3.5 Requirement analysis

Several requirements were set for the development of this project. Each of them are modeled from the point of view of a specific user role (developer, administrator, user), although there are cases in which a requirement may affect different scopes at the same time.

The following sections describe all the requirements grouped by role, details include: identification code (using format `X-YY`, where `X` is a letter and `YY` is a two-digit code), user story, any conditions necessary to accomplish the requirement and how the requirement was achieved in the implementation.

3.5.1 Developer requirements

These requirements are written from the point of view of a developer that creates new agents and functionalities for the Zoe platform. This implies direct interaction with the protocol and implementation.

D-01

As a developer, I want my existing code to work without modifications.

This implies that already existing agents should work regardless of the outpost system (including *Scout* agent) being present or not. The requirement is satisfied through design decisions: functionality of outpost agents library does not replace the original standard library, meaning that the developer must explicitly enable that functionality by updating their agent.

D-02

As a developer, I want to change less than 10% of my code to use the outpost functionalities.

Considering that the complexity of agent implementations varies greatly depending on their use case, *10%* would be an appropriate approximation for a breakpoint for this use case: simpler agents would not need to change too many lines while more complex agents may require custom migration algorithms.

Furthermore, and as shown in the *Standard library* section, the standard library is designed, and implemented, to be easy to *plug* into already existing code.

D-03

As a developer, I want my outpost agent to work without a Scout.

A scenario in which the *Scout* agent is not present had to be considered, as administrators may choose to disable that functionality. However, the library implementation **must** be present in the available Zoe libraries for the agent to work (e.g., they could be added to the basic Zoe distribution).

If the Zoe instance lacks the *Scout* agent, then the outpost agents would continue to work as normal agents, and the server should discard messages directed to the *Scout*.

D-04

As a developer, I want my agent to keep data across migrations.

Developers should be able to specify which data should be stored before migration and restored after migration is done. This is achieved through the *Standard library* implementation.

D-05

As a developer, I want my agent to serve all the requests it receives.

Even when migrating/migrated, agents should serve all the requests they have received at some point. In the system design, this would be achieved by relaying incoming messages to the *Scout* when migrating and retrieving them once migrated.

While that would mean requests would not obtain a response as soon as they would have in the original system, the implementation would make sure all of them are completed at some point, effectively dealing with the availability of the service.

In addition, the outpost server makes sure that all messages that have the agent as destination are correctly delivered from the central Zoe server.

D-06

As a developer, I want my agent to be able to migrate.

In order for an agent to be able to migrate, the following conditions must be met:

- The developer should include *static* and/or *dynamic* files in the distribution of their agent
- The developer should implement the standard library interface for the outpost

protocol (directly related to **D-02**)
- The developer should specify data to keep after migration (related to **D-04**)
- The agent should serve requests in any location (related to **D-05**)

D-07

As a developer, I want to develop my agents in any programming language.

By design, agents for Zoe can be developed using virtually any programming language. The solution and its implementation should also comply with this in order to be usable by all Zoe agent developers.

The solution to this requirement came in the form of the designed protocol (see *Protocol*). Even though the implementation was performed only for the Python programming language due to current agents using that same language, it was designed to be easily implemented for others by using mechanics, such as serialization, which have a counterpart in other languages.

3.5.2 Administrator requirements

These requirements are written from the point of view of an administrator that sets different configuration aspects of the Zoe system. This implies direct interaction with configuration files of the system.

A-01

As an administrator, I want to be able to choose whether or not to use the outpost system.

As the proposed solution is not a general use implementation, an administrator may choose to disable the added functionality. Such scenario would simply require moving the `agents/scout` directory to `disable-agents/scout` so that the *Scout* agent is prevented from launching.

All the agents that implement the outpost library would simply work as regular Zoe agents and the server would ignore outpost protocol messages.

A-02

As an administrator, I want to be able to configure remote outposts easily.

Administrators should be able to decide how the outposts are executed remotely, how the connections are made, and other variables that define the outpost configuration. To this end, the settings included in the implementation were made simple to understand and modify (by setting values in a plain text file), not imposing too many hard requirements on the administrator.

A-03

As an administrator, I want to be able to configure load balancing.

Following what was defined in **A-02**, automatic load balancing is available in the implementation and can be configured through a specific setting in one of the configuration files.

The module that contains the algorithm implementations was designed in such a way that it allows administrators to include their own algorithms while including several useful algorithms in the official distribution.

A-04

As an administrator, I want to be able to migrate agents manually.

In scenarios in which automatic load balancing is disabled, administrators may choose to move agents to specific locations. Some possible use cases where this requirement could be applied would be:

- Manage a mailing list in a remote server
- Perform intensive computations in a powerful node
- System monitorization

That is why a communication interface was implemented in the agent by using basic Zoe functionalities (e.g., communication channels, natural language, etc).

A-05

As an administrator, I want to be able to know where each agent is located.

While the **A-04** requirement specifies that an administrator should be able to migrate an agent manually, it does not deal with knowing the location of each agent and the available locations.

This requirement was satisfied in a similar manner to the previous one: a communication interface that showed current locations was implemented in the *Scout* agent.

A-06

As an administrator, I want to be able to know the status of outpost agents.

Again related to the previous requirements, an administrator should have a way of obtaining information on the outpost agents without having to connect to the central server machine and open the database files manually.

This was solved by including a natural language command that returned relevant information on known agents, including location and their current MIPS.

A-07

As an administrator, I want to be able to know the status of the outposts.

Similar to **A-06**, this requirement deals with the status of the outposts themselves. The natural language command that satisfies this requirement returns information such as whether the outpost is online or not, its host address and tunnel details.

A-08

As an administrator, I want to be able to keep an agent from migrating automatically.

The use cases shown in **A-04** can be applied to this requirement as well. In scenarios where automatic load balancing is enabled and the administrator needs/wants an agent in a specific machine, this agent should be ignored by the load balancing algorithm.

To accomplish this, a natural language command was introduced. This command flags an agent so that it would be ignored by the balancing algorithms.

A-09

As an administrator, I want to be able to allow an agent to migrate automatically.

This requirement is the opposite case for requirement **A-08**, and would give the administrator the means to undo the previously specified *hold* and allow an agent to be migrated automatically by the active load balancing algorithm.

Again, a command satisfied this requirement. In this case, the agent would be unflagged.

A-10

As an administrator, I want to be able to change configurations during execution.

The aim of this requirement is to reduce possible downtimes that would occur when an administrator changed configuration values related to the outposts in their Zoe instance. This downtime would have been due to the necessity of restarting the system for the changes to take effect.

In order to be as non-intrusive as possible in that sense, the *Scout* was designed to check configuration values periodically or when needed, always working with up-to-date settings instead of loading them on startup.

While this approach has the penalty of accessing the filesystem continuously for reading, the volume of data to read is not very high and it should not have such a great impact.

A-11

As an administrator, I want to be able to start an outpost manually.

Directly related to requirement **A-10**, should an administrator change outpost connection settings, or simply desire to start a previously stopped outpost, they should be able to do it.

As with previous requirements, this was solved by including a command which allowed administrators to start an outpost on demand by specifying its identification name.

Note that this requirement does not affect the tunnel that would connect the central server with the remote machine.

A-12

As an administrator, I want to be able to stop an outpost manually.

Directly related to requirement **A-10**. Scenarios in which an administrator may wish to stop an outpost include:

- To save resources in a remote machine (not being used currently)
- To change low level system configurations
- To restart it later on due to an error

As the counterpart of the previous requirement, a natural language command to allow this functionality was implemented.

Note that this requirement does not affect the tunnel that would connect the central server with the remote machine.

A-13

As an administrator, I want to be able to open a tunnel manually.

This requirement has a close relation to requirement **A-11**, given that the previous requirement does not include tunnels in its definition. Tunnels should be treated as a separate element in the implementation, although they should be identified by the outpost identification name as well.

This requirement was also assessed by implementing a natural language command to open the tunnel to an outpost using the configuration stored in the system.

A-14

As an administrator, I want to be able to close a tunnel manually.

Directly related to **A-12**. The administrator should be able to terminate the tunnel without affecting the outpost execution itself. This could be needed in cases such as system reboot or modification in tunnels.

Again, this was solved by introducing a natural language command.

A-15

As an administrator, I do not want to store any additional passwords in the system.

No additional password should be required by the outpost system, as that would require designing a secure storage for said passwords. As the only additional credentials this solution introduces are those required for the SSH connection, this requirement was satisfied by allowing only public key authentication for the connections.

That way, the administrator would only need to manage the private keys of their system as they would normally do, and may even generate a new key-pair specifically for this purpose.

A-16

As an administrator, I want the outpost system to work in any hardware architecture supported by my GNU/Linux operating system of choice.

This requirement is very important, as it takes into consideration portability of the solution, not enforcing a particular hardware architecture.

As explained previously, the solution (*Scout* and outpost) was implemented using the Python programming language, which provides the required portability as long as there is a Python interpreter available.

3.5.3 User requirements

These requirements are written from the point of view of a regular user that communicates with Zoe agents through a communication channel such as email or chat (Telegram, Jabber/XMPP, etc.).

U-01

As a user, I cannot tell whether outposts are used or not

Given the nature of the project, this requirement is easily satisfied with the implementation provided: all the code and protocol specification are located in the low-level architecture of Zoe, leaving user interaction with the agents intact and working in a transparent manner.

3.6 Validation tests

Taking into account all the requirements defined in the previous *Requirement analysis* section, it is necessary to prove that all of them are satisfied. In order to do this, the following validation tests were designed.

Each test is identified by a code using the format `VT-YY`, where `YY` is a two-digit code and has a short description of the test. A traceability matrix showing the correlation between validation test and requirements is included at the end of the chapter.

VT-01

Execution of a normal agent in the new system.

This implies running an original Zoe agent (in this case, `madtrans`) in a system where the outpost architecture has been implemented (in the central server).

VT-02

Convert a normal agent to an outpost one.

In this case, the agent `madtrans` is updated to be an outpost agent (capable of migrating), using the standard library addition, to see if it continues to work normally.

VT-03

Execution of the system without a Scout.

This test consists on running the Zoe system without the *Scout* agent being present but with at least one outpost agent (in this case, `outpostest`) to observe if the outpost agents behave as normal agents without a *Scout* in place.

VT-04

Migrate an agent to a remote outpost.

Perform manual migration of an outpost agent (in this case, `outpostest`) to a remote machine/outpost. This test effectively covers all the *File migration* and *Data migration* parts of the protocol.

VT-05

Execution of a load balancing algorithm.

In this case, the *equal-balance* algorithm is used to test the correct functionality of automatic migrations (as opposed to the manual migration performed in test **VT-04**).

VT-06

Execution of Scout commands through a communication channel.

In this test, the user communicates with the *Scout* through a communication channel (in this case, Telegram), to execute several commands implemented (see user manual in *Appendix B*).

VT-07

Execution of a load balancing algorithm with agents on hold.

As opposed to test **VT-05** where all the agents were free to migrate automatically, this time the `outpostest` agent was on hold in the central server while the `madtrans` agent was free to migrate.

VT-08

Add a new outpost agent during execution.

For this test, agent `dummy1` is added to the central server after the system has been started .

VT-09

Stop and start a running outpost manually.

This consists on executing the relevant commands through the communication channel (in this case, Telegram) to effectively restart an outpost.

VT-10

Stop and start a SSH tunnel manually.

This consists on executing the relevant commands through the communication channel (in this case, Telegram) to effectively restart a SSH tunnel.

VT-11

Implement the outpost in a hardware architecture different to the central server.

In this case, this is achieved by installing all the relevant software and files to a Raspberry Pi (armhf) while the central server is running on a different architecture (x64).

3.6.1 Traceability matrix

Table 3.1 contains the traceability matrix for the aforementioned validation tests, indicating which test validates each of the requirements specified previously. Note that there is also a column for requirements that are considered to be implicitly tested as part as complete operation of the system.

Table 3.1: Validation test traceability matrix

VT	D-01	D-02	D-03	D-04	D-05	D-06	D-07	A-01	A-02	A-03	A-04	A-05	A-06	A-07	A-08	A-09	A-10	A-11	A-12	A-13	A-14	A-15	A-16	U-01
Implicit									X								X					X		X
01	X																							
02		X					X																	
03			X					X																
04				X	X	X					X													
05										X														
06												X	X	X										
07															X	X								
08																	X							
09																		X	X					
10																				X	X			
11																							X	

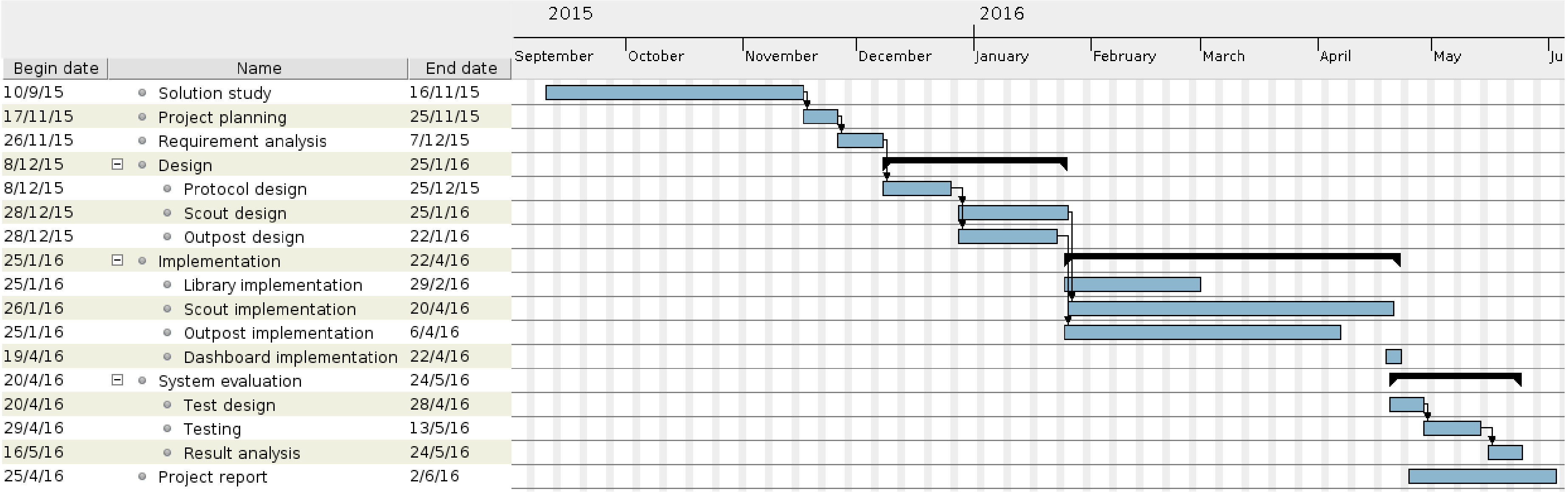
Chapter 4

Planning

This section shows the planning for this work. In particular, the time planning and cost estimation of the project are presented in detail.

4.1 Time planning

The Gantt chart for this project can be found in Figure 4.1. The next lines explain all the tasks created for the development of the complete project.



Begin date	Name	End date
10/9/15	Solution study	16/11/15
17/11/15	Project planning	25/11/15
26/11/15	Requirement analysis	7/12/15
8/12/15	⊟ Design	25/1/16
8/12/15	Protocol design	25/12/15
28/12/15	Scout design	25/1/16
28/12/15	Outpost design	22/1/16
25/1/16	⊟ Implementation	22/4/16
25/1/16	Library implementation	29/2/16
26/1/16	Scout implementation	20/4/16
25/1/16	Outpost implementation	6/4/16
19/4/16	Dashboard implementation	22/4/16
20/4/16	⊟ System evaluation	24/5/16
20/4/16	Test design	28/4/16
29/4/16	Testing	13/5/16
16/5/16	Result analysis	24/5/16
25/4/16	Project report	2/6/16

Figure 4.1: Project Gantt chart

- **Solution study**. Throughout the duration of this task, and considering that the problem to solve has already been defined, different solutions and their viability are studied in depth. This study involves building small implementations using the methods offered by each solution to see if they satisfy the requirements imposed on the project and can be further developed.
- **Project planning**. Once a solution has been chosen, it is necessary to plan the complete development of the project. This includes the Gantt chart in Figure 4.1 and the necessary steps to complete the project. Requires: **Solution study**
- **Requirement analysis**. Every system developed must comply with a series of requirements that define the functionality they should offer and how they work. These requirements can be found in the *Requirement analysis* section. Requires: **Project planning**
- **Design**. In the design phase, the different components of the solution are drafted, including their interactions and other details needed in the system. This meta-task includes three additional tasks: *protocol design*, *scout design* and *outpost design*.
    - **Protocol design**. This task defines the protocol (see *Protocol*) and low-level details such as the SSH tunneling, communication mechanisms and the library design. Requires: **Requirement analysis**
    - **Scout design**. Through the duration of this task, the *Scout* agent and its functionalities are designed, in addition to configuration files and databases. Note that this task and the *Outpost design* one are allocated in parallel in order to work on both at the same time, given the need to have common interaction between them. Requires: **Protocol design**
    - **Outpost design**. In the period specified by this task, the outpost server and its functionalities are designed, in addition to configuration files. As specified previously, this task is worked on in parallel to *Scout design*. Requires: **Protocol design**
- **Implementation**. In the implementation phase, all components previously designed are developed and their correct functionality tested at the same time (see *Functionality tests*). It includes four additional tasks; *library implementation*, *scout implementation*, *outpost implementation* and *dashboard implementation*. Note that all of them are developed in parallel.
    - **Library implementation**. In this task, the outpost standard library for

Python is developed (see *Standard library*). Requires: **Protocol design**

  - **Scout implementation**. In this task, the *Scout* agent for Zoe is implemented (see *Scout*), in addition to management of tunnels and outposts, which are considered internal functionalities of the agent. Requires: **Scout design**
  - **Outpost implementation**. In this task, the outpost microserver is implemented (see *Outpost*). Requires: **Outpost design**
  - **Dashboard implementation**. The dashboard is a very simple secondary way of visualizing current status information extracted from the *Scout* agent. As it uses the same database models as the agent, its implementation does not require much time.

- **System evaluation**. In this phase, the solution is tested in a *real scenario*. In particular, these tests deal with load balancing (see *Performance tests*), as the functionalities are tested in the *Functionality tests*.
  - **Test design**. Throughout the duration of this task, the different performance tests for the implemented load balancing algorithms are designed.
  - **Testing**. Once designed, the tests are performed in the time period of this task, storing the logs of the *Scout* agent, which contain all the information required to evaluate the results. Requires: **Test design**
  - **Result analysis**. In this task, the relevant information from the logs is parsed and extracted, including conclusions on the results in the project report. Requires: **Testing**
- **Project report**. The last phase of the project consists on writing the report of the work performed. This task can be worked on in parallel to the *System evaluation* one, as content from the previous tasks can be written before the test results and conclusions are available.

4.2 Cost estimation

This section covers the cost estimation for the resources required to complete the development of this project.

Taking into account that Zoe can be installed in less powerful machines (such as Raspberry Pi computers), the physical resources required for the project have the advantage of reduced cost when compared to other computational nodes.

Even though all the hardware was already available, Table 4.1 shows the cost projection for these resources, which include: personal (development) computer, Raspberry Pi computer bundles and the router to which all the machines connect for testing. Furthermore, the table includes hardware amortization values valid in Spain since 1st of January 2015 [31].

All the computers (personal computer and Raspberry Pi computers) are considered equipment for information processes (*equipos para procesos de información*) and have a maximum yearly linear coefficient of 25% of the cost over a maximum period of 8 years. The wireless router can be placed in the category of systems and software (*sistemas y programas informáticos*), and a maximum yearly linear coefficient of 33% of the cost over a maximum period of 6 years.

However, and given that the cost of the Raspebrry Pi computers and the wireless router are inferior to 300 €, these elements can be amortized freely [31], in this case 100% over the period of 10 months (estimated maximum duration of the project).

Table 4.1: Cost projection of physical resources

Concept	Num. units	Unitary cost	Total	Monthly amortization	Amortization (10 months)
Personal computer	1	649.00 €	649.00 €	2.083%	135.21 €
Raspberry Pi Model A bundle	2	32.00 €	64.00 €	10%	32.00 €
Raspberry Pi Model B+ bundle	1	69.95 €	69.95 €	10%	69.95 €
D-Link Cloud Router N300 (WiFi)	1	24.99 €	24.99 €	10%	24.99 €
Total			**807.94 €**		**262.15 €**

Regarding the Raspberry Pi bundles, each of them includes additional resources which are detailed below:

- **Raspberry Pi Model A bundle**: Raspberry Pi Model A board, case, 8 GB SD card, 5V 2A power supply
- **Raspberry Pi Model B+ bundle**: Raspberry Pi Model B+ board, case, 8 GB MicroSD card, 5V 2A power supply, HDMI cable, USB Wifi adapter

Table 4.2 contains the cost of fungible resources employed in the project. Such resources include energy consumption of the previously specified hardware, which are as follows (usual averages):

- **Personal computer**: 22.75 W
- **Raspberry Pi Model A**: 2.5 W
- **Raspberry Pi Model B+**: 3.0 W
- **Wireless router**: 6.0 W

Taking the previous values into account, two different periods have to be defined for calculating the cost of electricity. In the first place, development process requiring use of the personal computer, Raspberry Pi Model B+ and wireless router, which would amount to 31.75 W and estimated 490 hours dedicated to development. In second place, testing process requiring all the equipment shown previously (34.25 W) and testing hours specified in the *Evaluation chapter* (4.3 hours).

Table 4.2: Cost projection of fungible resources

Concept	Num. units	Unitary cost	Total
Ethernet cable (class 6 / 1 Gbps)	2	2.49 €	4.98 €
Electricity	494.3 hours	0.121 €/KWh	2.30 €
Internet connection	10 months	26.90 €/month	269.00 €
Total			**275.88 €**

The price for the Internet connection package has been obtained from an offer by *Telefónica de España* [32] and includes telephone calls (landline and mobile) as well as a 300Mb fibre optic Internet connection. The price for the electricity was obtained by calculating the average of the fees for different companies for year 2016 (including 21% VAT) [33]. Note that these prices are merely orientative, and do not include other fees such as line rent.

Table 4.3 shows estimated costs of human resources. This estimation considers the periods shown in the previous time planning, the different roles that participated in the development process (even though most of them were taken by the student). The social security percentage included is an initial estimation of the cost (extracted from [34]), given that the actual cost depends greatly on external factors of the company and are therefore considered out of the scope of this project.

Table 4.3: Cost projection of human resources

Role	Cost per hour	Hours	Social security	Total
Project manager	30.00 €	70	23.60%	2,595.60 €
Analyst	20.00 €	127	23.60%	3,139.44 €
Designer	20.00 €	45	23.60%	1,112.40 €
Programmer	20.00 €	490	23.60%	12,112.80 €
Tester	20.00 €	4.3	23.60%	106.30 €
Total				**19,172.84 €**

Finally, Table 4.4 shows the total estimated cost for the development of the project, including 15% increase due to risks, such as tax changes or indirect costs, and 20% increase due to project benefits. The total amounts include VAT.

Table 4.4: Total cost estimation summary

Concept	Total
Physical resources	262.15 €
Fungible resources	275.88 €
Human resources	19,172.84 €
Risks	2,956.63 €
Benefits	3,942.18 €
Total	**26,609.67 €**

Chapter 5

Regulations

This chapter briefly analyzes possible regulations and contexts in several fields that may be applied to the developed project.

5.1 Social

This project is aimed at developers and administrators of the Zoe ecosystem. Considering the free licence under which Zoe is distributed, anyone can benefit from both the original functionalities offered by the main project and the extended, and more specialized, functionalities offered by this outpost system.

5.2 Legal

Being designed as a generic protocol, and system, that tries to make no assumptions on the scenarios in which it could be executed, the project itself does not have knowledge of the information being transmitted during migration. Responsibility for what information is delivered lies with administrators and, to some extent, developers.

Furthermore, inter-machine connections have to be set up explicitly by an administrator, which means that they should have access/ownership/permissions necessary to use the machine as an outpost. Moreover, by using public key authentication, the administrator is the one responsible for generating and managing the key-pair used in the connection, having to make sure that the private key is stored safely and the SSH connection is correctly encrypted.

Regarding the source code of the project, the components previously explained are released under different licences:

- **Standard library addition**: which includes Python implementation of the custom `OutpostAgent` class and the data migration methods is released under the **MIT licence**[1]
- **Scout,Outpost and Dashboard**: which includes the source code of the *Scout* agent, the outpost server and the secondary visualization dashboard, as well as the utility libraries created for them, are released under the **GPLv3 licence**[2]

The **MIT licence** allows anyone receiving a copy of the source code to use it, distribute it, modify it (and redistribute modifications) and make commercial use of the software with the condition that a copy of the licence is included with the software and the original Copyright notice is kept in the code.

The **GPLv3 licence** grants the same permissions as the MIT licence (in addition to patent rights), although with more conditions apart from including a copy of the licence and keeping the Copyright notice in the code: the source code must be made available when distributing the software, modifications must be released under the same, or similar, licence and the user is required to list any changes made to the software.

In addition, both licences include a clause that the software is provided without warranty and the author cannot be held liable for any type of damage caused from the use of the software.

5.3 Economic

While it is legally possible to commercialize the proposed solution, as shown in the previous section, in practice it would only be possible when distributing the project as part of a complete Zoe installation. This is due to the hard requirements the solution has on the Zoe project (libraries, functionalities, etc.), as it is specifically tailored for that platform.

A possible commercial use case would be providing hosting for Zoe instances, which would be considered as providing a service and is a business model used by important companies such as Canonical Ltd. or Red Hat, Inc.

[1] https://opensource.org/licenses/MIT

[2] http://www.gnu.org/licenses/gpl-3.0.html

Chapter 6

Evaluation

This section presents the evaluation performed on the implementation of the solution proposed. After and during the development of said implementation, two different types of tests were carried out:

- **Functionality tests**: intended to test the correct functionality of an implemented feature in a controlled environment during development
- **Performance tests**: intended to evaluate the performance and correctness of the implementation using load balancing algorithms

6.1 Platform description

Table 6.1 shows the relevant hardware and software details of the systems used to perform the tests, in addition to the benchmarked MIPS for each system and their assigned priorities for the *priority* algorithm.

Functionality tests only required the first two systems (*central* executing the Zoe instance and *outpost_pi* the outpost) as to have a reduced and more *controllable* environment.

The two additional Raspberry Pi computers (model A) were configured as additional outposts for the **performance tests** in order to check the load balancing algorithms and correct functionality of the system with more servers connected.

MIPS results were obtained after 5 iterations of the HPLinpack benchmark with `perf` to obtain the average time and number of instructions executed. The last two machines *outpost_arco1* and *outpost_arco2* have the same MIPS considering that both had a clean installation of the operating system with exactly the same configuration steps followed.

Table 6.1: Hardware and software specifications of the test systems. Note: names for *outpost_pi*, *outpost_arco1* and *outpost_arco2* were shortened to *pi*, *arco1* and *arco2* respectively

	central	**pi**	**arco1**	**arco2**
Model	Asus K53SV	Rasperry Pi Model B+	Raspberry Pi Model A	Raspberry Pi Model A
CPU	Intel Core i7-2670QM	Broadcom BCM2835	Broadcom BCM2835	Broadcom BCM2835
Frequency	2.2 GHz	700 MHz	700 MHz	700 MHz
RAM	4 GB	512 MB	256 MB	256 MB
Operating System	Debian 9 (*stretch*)	Raspbian *jessie*	Raspbian *jessie*	Raspbian *jessie*
Connection	wireless	wireless	ethernet	ethernet
MIPS	5217.933559389	132.287849904	138.476783011	138.476783011
Priority	3	2	1	1
Outpost	no	yes	yes	yes

Regarding the connectivity of the devices, the first two were connected to the network using a wireless interface and the other two computers were attached through a wired connection directly to the wireless router. This setup helped test the performance when different communication methods are used, each with their own latency (wireless connection in Model B+ had more latency than the rest of the machines due to how the hardware is implemented).

Finally, the assigned priorities are interpreted as follows: both *outpost_arco1* and *outpost_arco2* will be filled first in the *priority* algorithm, then *outpost_pi* and finally *central*, unless the 80% limit is reached (see *Priority algorithm*).

Figure 6.1 shows shows the layout of the connections for the tests.

6.2 Functionality tests

Every single functionality was thoroughly designed to be easy to debug by introducing a custom logging format which identifies the most critical points of every method and function, allowing to easily follow the flow of execution.

Before modifying an already existing agent to test its adaptation to the new system, a

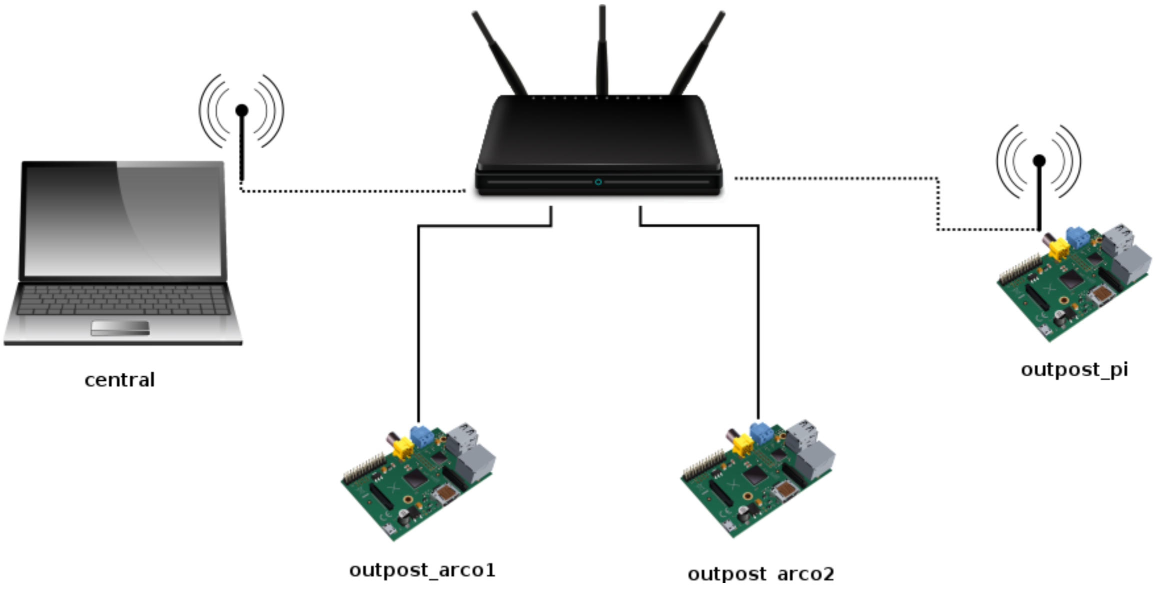


Figure 6.1: Equipment layout for performance tests

test agent named *outpostest* was implemented. Its purpose was to use all the features available in the standard library (see *Standard library*) and test the correct execution of both *Scout* and outpost methods.

These tests, triggered by the *Scout* commands, helped identify and correct errors in the implementation of the system, as the result of each operation was known beforehand and could be compared with the one obtained (e.g., file migration, message relaying, data storage, etc.).

6.3 Performance tests

Performance tests focus on the automatic migrations performed by load balancing algorithms and have no interaction from the user/administrator whatsoever. The aim is to check that agents are being migrated to all the different nodes according to the criteria specified in the algorithm with different configurations.

Two groups of four tests each (8 tests in total) were designed to this end. The tests included in each group are as follows:

- Execution of the **balanced** algorithm with all the agents free
- Execution of the **balanced** algorithm with some agents on hold
- Execution of the **priority** algorithm with all the agents free

- Execution of the **priority** algorithm with some agents on hold

By having different test cases for free and held agents it is possible to compare how this fact affects the balancing result in each case. Test runs have a duration of around 20 minutes in order to allow the system to perform several automatic migrations.

The difference between the aforementioned test groups is that the first group works with a *normal* load that could be found in a regular Zoe instance, while the second group is an *extreme* case in which there is an agent with a very high load. The common agents used in both tests are:

- **dummy1**: synthetic agent with a higher load than the rest that simply creates 20 threads which are kept in an infinite loop performing operations asynchronously
- **dummy2**: same as *dummy1*
- **dummy3**: same as *dummy1*
- **fibonacci**: agent that calculates specific elements of the fibonacci series on demand
- **madtrans**: agent that fetches information from the Madrid bus system
- **outpostest**: test agent for the outpost system

In addition, the second group includes the **overload** agent, which is a synthetic agent designed to place a lot of stress in the processor.

Results for these tests were extracted from the logs of the *Scout* agent, which contain all the information from the execution of the system.

The following additional considerations must be taken into account with regards to the tests:

- Load balancing algorithms are scheduled to be executed 5 minutes **after the last algorithm execution**. Such execution finishes after all automatic migrations have been performed
- MIPS measurements are scheduled to be executed every 30 seconds, although this time can only apply to the central machine, being slightly higher in the practice. This is due to the process of outposts receiving the request messages, measuring the MIPS of their agents and sending the information back

While testing time is 20 minutes for each test, some of the tests had to be repeated due to errors, hence the final estimated testing period is **260 minutes** (*4.3 hours*).

6.4 Result analysis

In this section the results obtained from the aforementioned tests and their conclusions are presented, indicating relevant information such as at which point migrations occur.

To this end, six figures are included, showing the performance evolution of each agent, or set of agents, over time. This evolution is represented by the amount of MIPS measured for the specific agent at a given sample number (i.e., each of the measurements done every 30 seconds).

6.4.1 Normal load tests

This first group of tests consists on the execution of the common agents previously explained in order to emulate a *regular* use case of a Zoe instance in which agents do not have an excessive load.

The four tests and their results are explained next.

6.4.1.1 Equal-Free scenario

This test consists on running the system using the *balanced* algorithm with all agents free and available for automatic migration.

The information found in Table 6.2 represents the calculated destinations **right before** agents are migrated. In the case of **migration 0**, all the agents are located in the central machine at the start of the test.

Table 6.2: Calculated agent destinations before migrations of the normal load *Equal-Free* test scenario

Migration	central	outpost_pi	oupost_arco1	outpost_arco2
0	dummy1 dummy2 dummy3 fibonacci madtrans outpostest			
1	dummy1	dummy2	dummy3	fibonacci madtrans outpostest
2	dummy1	dummy2	dummy3	fibonacci madtrans outpostest
3	dummy1	dummy2	dummy3	fibonacci madtrans outpostest

Table 6.3: Calculated hypothetical machine loads (percentages over 1) used for determining new agent destinations in normal load *Equal-Free* test scenario

Migration	central	outpost_pi	outpost_arco1	outpost_arco2
1	0.000930	0.036216	0.034872	0.000186
2	0.000843	0.031544	0.029465	0.000261
3	0.000845	0.032389	0.029772	0.000203

Table 6.2 shows the destinations calculated by the *balanced* algorithm as a direct result of the values shown in Table 6.3. The *dummy* agents are all placed in isolation given that they are the ones with a higher load, while the other three (*fibonacci*, *madtrans*, *outpostest*) can all be placed in a single machine (*outpost_arco2*), resulting in lower resource usage when compared to the rest of the machines.

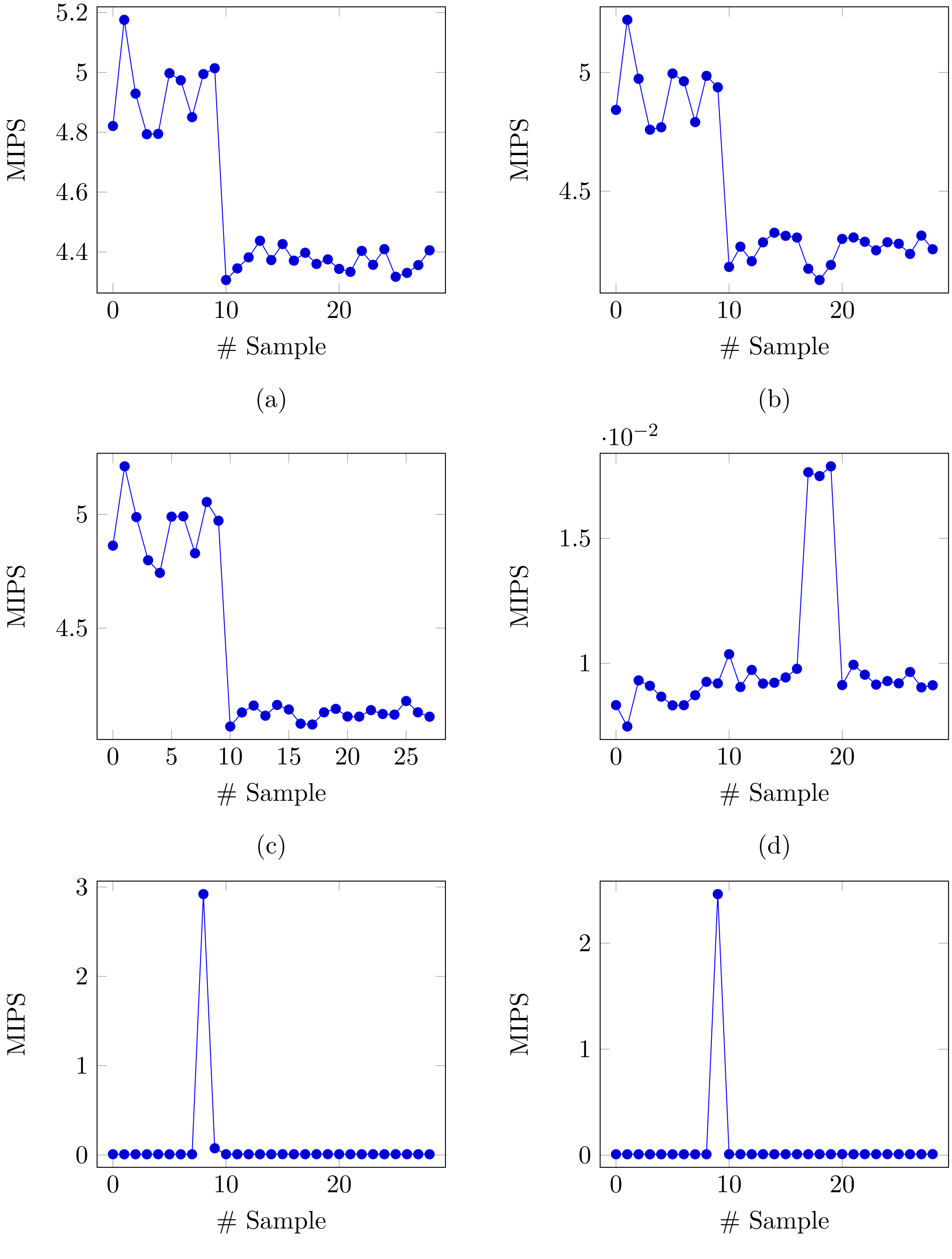
Figure 6.2 contains the individual evolution of the MIPS measured for each of the agents used in the test. While subfigures (d), (e) and (f) maintain their *constant* tendency throughout execution (with the peaks in (e) and (f) coinciding with their migration/data serialization process), they have a very low resource usage, which does not affect their performance regardless of the machine they are in.

However, it is interesting to compare the evolutions shown in subfigures (a),(b) and

(c) (*dummy1*, *dummy2* and *dummy3*). The three agents have similar MIPS values in the beginning, due to being present in the *central* machine, although they experience a drop around **sample 10**.

The drop in subfigure (a) is probably due to operating system policies (in particular, CPU scheduling), considering that it remains in the *central* machine throughout the duration of the test. Furthermore, the values for subfigures (b) and (c) after that point are lower than those in subfigure (a), which might be considered significant when taking into account that the values represent million instructions per second.

The conclusion reached from this test is that migration does affect the performance of the agents, although it does not have a great impact in cases where the agent does not make an excessive use of system processing.



(a) (b) (c) (d) (e) (f)

Figure 6.2: MIPS evolution in Equal-Free test scenario(normal load); (a) agent "dummy1", (b) agent "dummy2", (c) agent "dummy3", (d) agent "fibonacci", (e) agent "madtrans", (f) agent "outpostest"

6.4.1.2 Equal-Hold scenario

This test consists on running the system using the *balanced* algorithm with all agents free except for *dummy2* and *dummy3*, which are held in *central*.

The information found in Table 6.4 represents the calculated destinations **right before** agents are migrated. In the case of **migration 0**, all the agents are located in the central machine at the start of the test.

Table 6.4: Calculated agent destinations before migrations of the normal load *Equal-Hold* test scenario

Migration	central	outpost_pi	oupost_arco1	outpost_arco2
0	dummy1 dummy2 dummy3 fibonacci madtrans outpostest			
1	dummy2 dummy3 fibonacci	madtrans	outpostest	dummy1
2	dummy2 dummy3 fibonacci	madtrans	outpostest	dummy1
3	dummy2 dummy3 fibonacci	madtrans	outpostest	dummy1

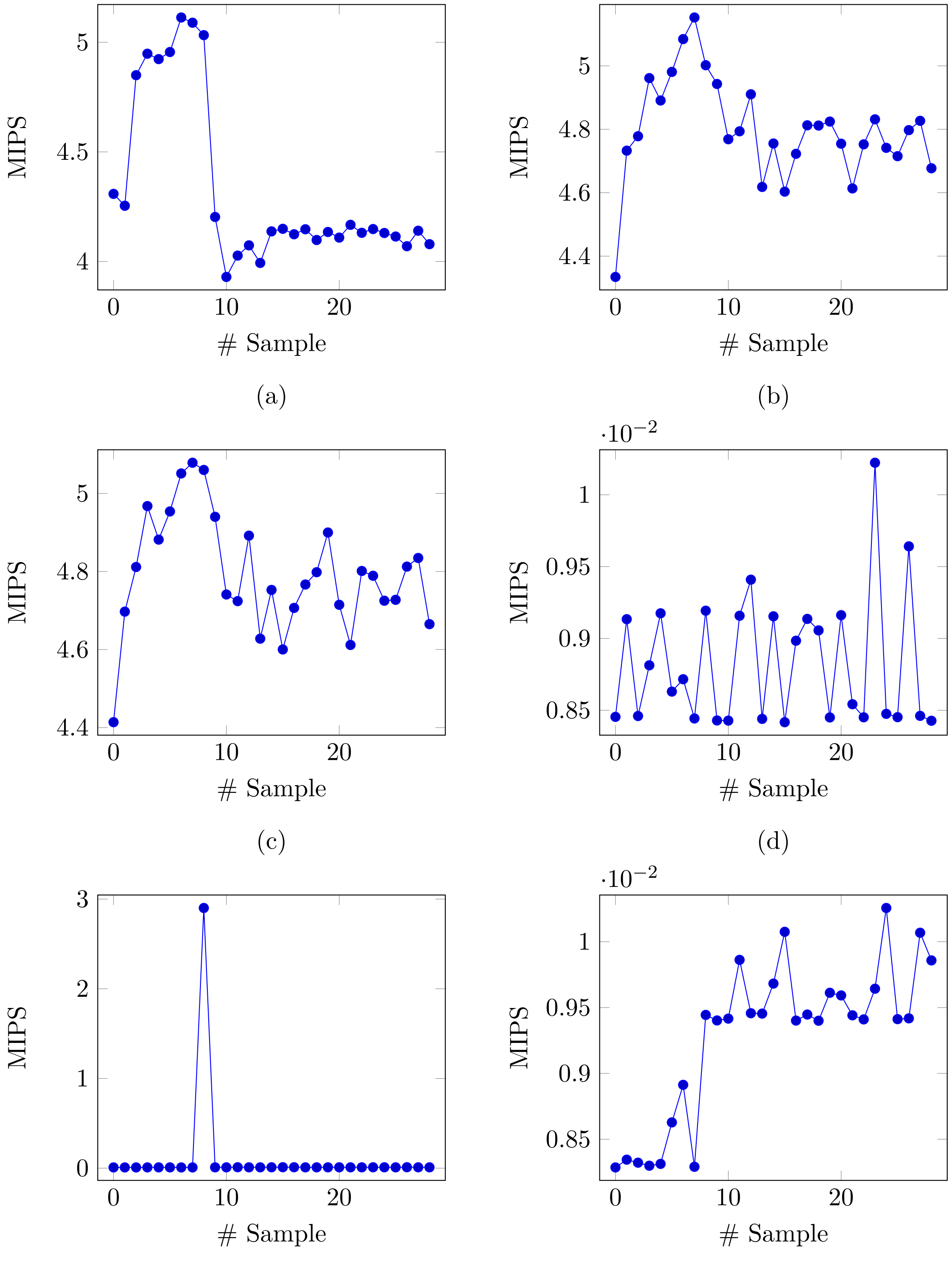
Table 6.5: Calculated hypothetical machine loads (percentages over 1) used for determining new agent destinations in normal load *Equal-Hold* test scenario

Migration	central	outpost_pi	outpost_arco1	outpost_arco2
1	0.001962	0.000063	0.000060	0.036752
2	0.001809	0.000079	0.000068	0.029783
3	0.001846	0.000073	0.000073	0.029955

Table 6.4 shows the destinations calculated by the *balanced* algorithm as a direct result of the values shown in Table 6.5. As opposed to the previous test, agents *dummy2* and *dummy3* are now forced to stay in the *central* machine resulting in both *central* and *outpost_arco2* sharing most of the load.

Figure 6.3 contains the individual evolution of the MIPS measured for each of the agents used in the test. Again, subfigures (d), (e) and (f) show very low values similar to the ones presented in the previous scenario (including the migration peak in subfigure (e)), although their calculated locations are different from the other ones, effectively reducing the load on *outpost_pi* and *outpost_arco1*.

In this occasion, however, subfigures (b) and C show very similar values due to being in the same machine and the difference with subfigure (a) is very close to the one presented in the previous case as well. Furthermore, the peak difference in subfigure (a) when migrated may be used to reiterate the previous findings.



(a)

(b)

(c)

(d)

(e)

(f)

Figure 6.3: MIPS evolution in Equal-Hold test scenario (normal load); (a) agent "dummy1", (b) agent "dummy2", (c) agent "dummy3", (d) agent "fibonacci", (e) agent "madtrans", (f) agent "outpostest"

6.4.1.3 Prio-Free scenario

This test consists on running the system using the *priority* algorithm with all agents free and available for automatic migration.

The information found in Table 6.6 represents the calculated destinations **right before** agents are migrated. In the case of **migration 0**, all the agents are located in the central machine at the start of the test.

Table 6.6: Calculated agent destinations before migrations of the normal load *Prio-Free* test scenario

Migration	central	outpost_pi	oupost_arco1	outpost_arco2
0	dummy1 dummy2 dummy3 fibonacci madtrans outpostest			
1				outpostest dummy1 dummy2 dummy3 madtrans fibonacci
2				outpostest dummy1 dummy2 dummy3 madtrans fibonacci
3				outpostest dummy1 dummy2 dummy3 madtrans fibonacci

Table 6.7: Calculated hypothetical machine loads (percentages over 1) used for determining new agent destinations in normal load *Prio-Free* test scenario

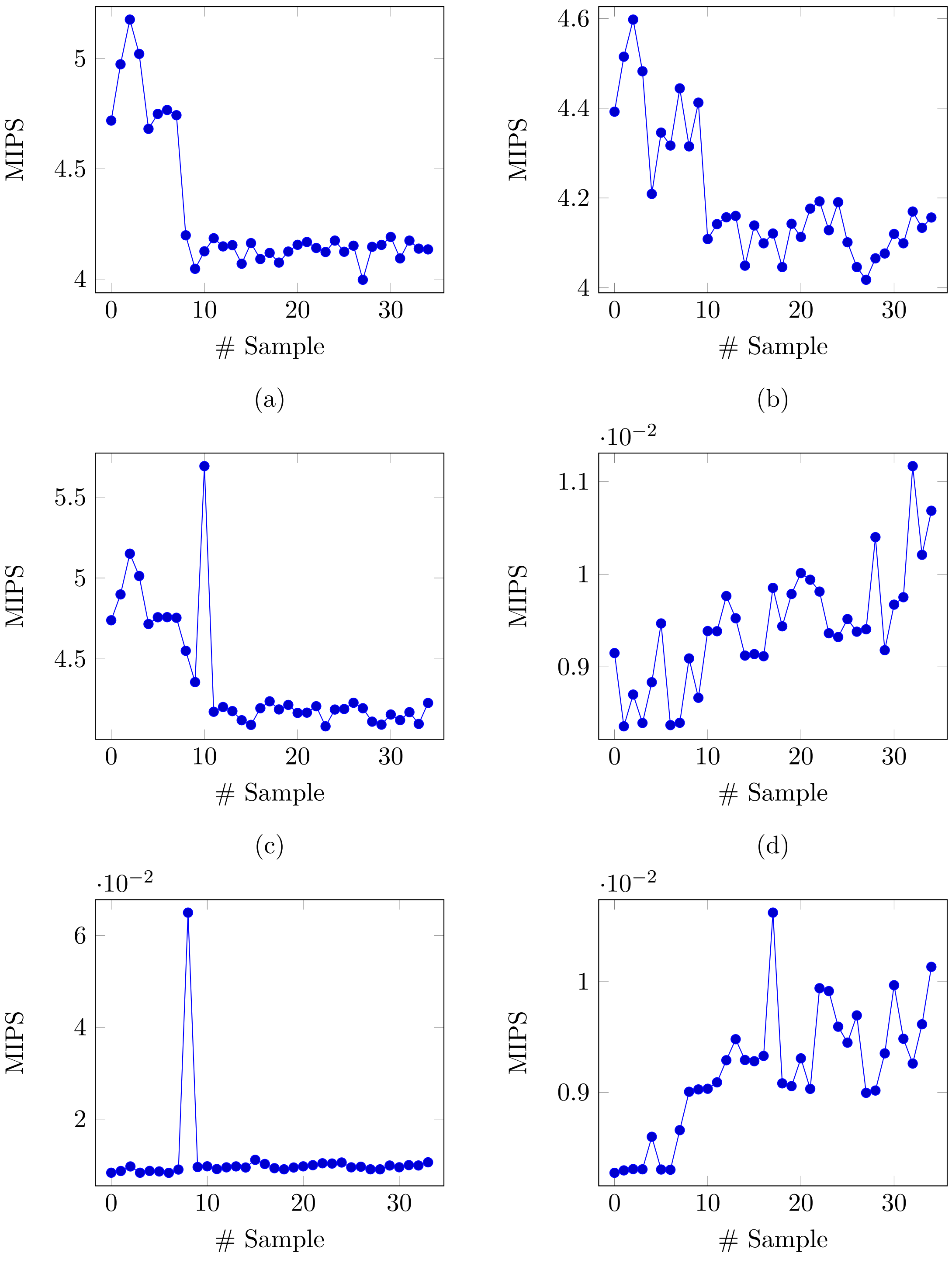
Migration	central	outpost_pi	outpost_arco1	outpost_arco2
1	0.0	0.0	0.0	0.100866
2	0.0	0.0	0.0	0.090347
3	0.0	0.0	0.0	0.089204

Table 6.6 shows the destinations calculated by the *priority* algorithm. Given the priorities specified previously, the load balancing algorithm will try to migrate all agents to either *outpost_arco1* and *outpost_arco2*, until their hypothetical loads exceed 80% (*0.8*), in which case it will try to fill *outpost_pi* and finally, *central*.

Considering the calculated hypothetical loads shown in Table 6.7, all agents would be migrated to either *outpost_arco1* or *outpost_arco2* due to the total load reaching a maximum of 10% (*0.1*). For this reason, agents remain in that machine throughout the duration of the test and the other machines have a hypothetical load of 0% with respect to the outpost agents.

Again, this change can be clearly seen in Figure 6.4. Although subfigures (d), (e) and (f) show very little difference due to their reduced loads, the loss of performance is more clear in subfigures (a), (b) and (d) as in the previous test scenarios.

Taking into account that this algorithm does not try to maintain balance between the machines, it is important to note that these results are a direct consequence of the priority assignment made beforehand which, in conclusion, would allow to shut down the *outpost_pi* and *outpost_arco1* machines, for instance, in order to save energy.



(a)

(b)

(c)

(d)

(e)

(f)

Figure 6.4: MIPS evolution in Prio-Free test scenario (normal load); (a) agent "dummy1", (b) agent "dummy2", (c) agent "dummy3", (d) agent "fibonacci", (e) agent "madtrans", (f) agent "outpostest"

6.4.1.4 Prio-Hold scenario

This test consists on running the system using the *priority* algorithm with all agents free except for *dummy2* and *dummy3*, which are held in *central*.

The information found in Table 6.8 represents the calculated destinations **right before** agents are migrated. In the case of **migration 0**, all the agents are located in the central machine at the start of the test.

Table 6.8: Calculated agent destinations before migrations of the normal load *Prio-Hold* test scenario

Migration	central	outpost_pi	oupost_arco1	outpost_arco2
0	dummy1 dummy2 dummy3 fibonacci madtrans outpostest			
1	dummy2 dummy3		dummy1 fibonacci madtrans outpostest	
2	dummy2 dummy3		dummy1 fibonacci madtrans outpostest	
3	dummy2 dummy3		dummy1 fibonacci madtrans outpostest	

Table 6.9: Calculated hypothetical machine loads (percentages over 1) used for determining new agent destinations in normal load *Prio-Hold* test scenario

Migration	central	outpost_pi	outpost_arco1	outpost_arco2
1	0.001822	0.0	0.035142	0.0
2	0.001792	0.0	0.029116	0.0
3	0.001731	0.0	0.030092	0.0

Table 6.8 shows the destinations calculated by the *priority* algorithm. Given the priorities specified previously, the load balancing algorithm will try to migrate all

agents to either *outpost_arco1* and *outpost_arco2*, until their hypothetical loads exceed 80% (*0.8*), in which case it will try to fill *outpost_pi* and finally, *central*.

In this scenario, as opposed to the **Prio-Free** one, agents *dummy2* and *dummy3* are forced to stay in the *central* machine. The result is that, as before, the algorithm will migrate the rest of the agents to the other machines, in this case *outpost_arco1*.

Regarding the MIPS evolution shown in Figure 6.5, similar conclusions to the previous ones can be reached: while the change is not noticeable in subfigures (d), (e) and (f), subfigure (a) has considerable drop when compared to (b) and (c) (which are always in *central*).

The main difference when compared to scenario **Prio-Free** is that the *central* machine has a higher load due to the agents held there. Therefore, agents on hold should be considered when choosing an algorithm to perform automatic load balancing in the system.

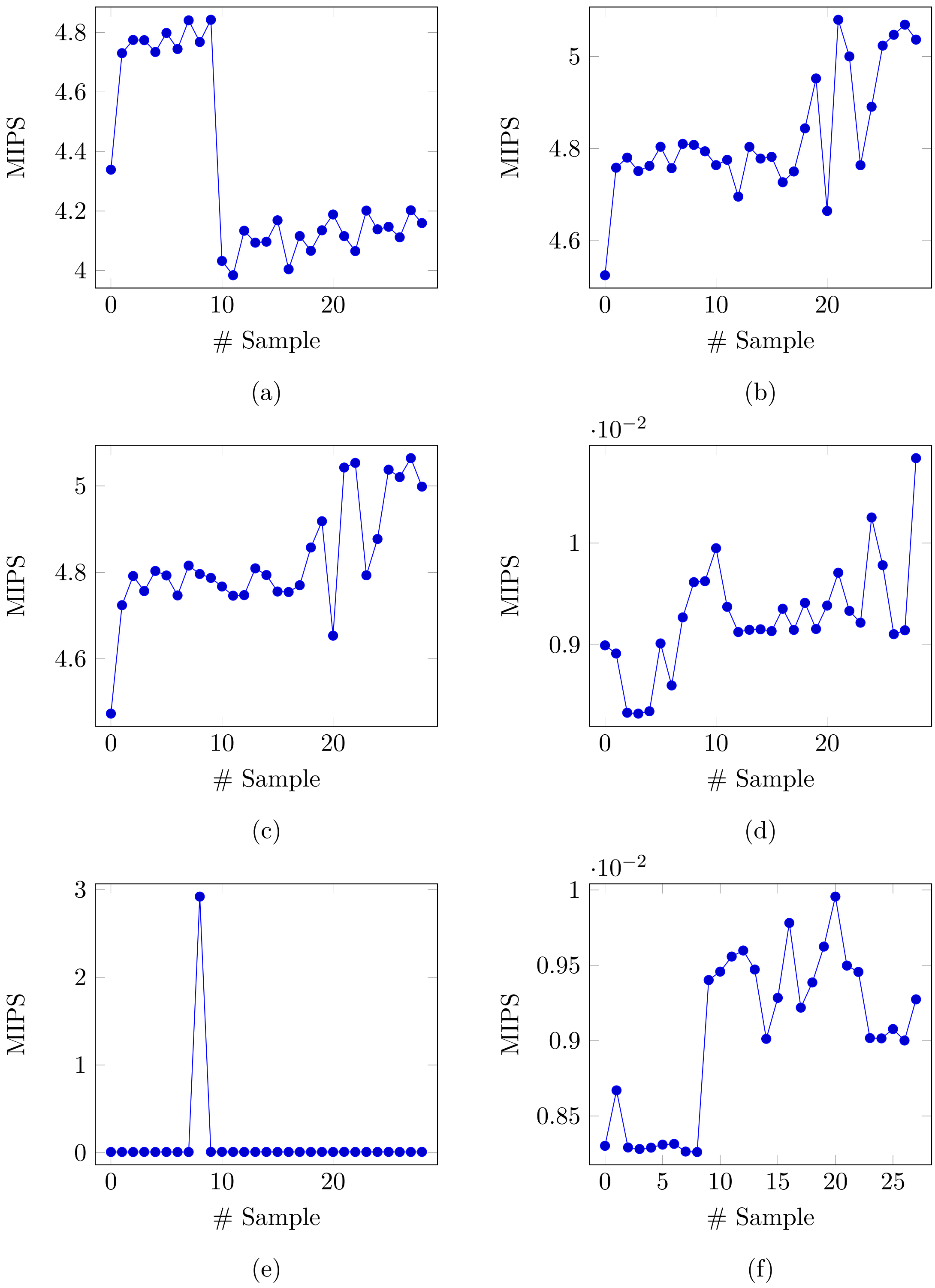


Figure 6.5: MIPS evolution in Prio-Hold test scenario (normal load); (a) agent "dummy1", (b) agent "dummy2", (c) agent "dummy3", (d) agent "fibonacci", (e) agent "madtrans", (f) agent "outpostest"

6.4.2 Overload tests

This second group of tests consists on the execution of the common agents previously explained in addition to the *overload* agent, whose only task is to waste CPU resources, allowing to see how the algorithms behave in such scenario.

The four tests and their results are explained next.

6.4.2.1 Equal-Free scenario

This test consists on running the system using the *balanced* algorithm with all agents free and available for automatic migration.

The information found in Table 6.10 represents the calculated destinations **right before** agents are migrated. In the case of **migration 0**, all the agents are located in the central machine at the start of the test.

Table 6.10: Calculated agent destinations before migrations of the overload *Equal-Free* test scenario

Migration	central	outpost_pi	oupost_arco1	outpost_arco2
0	dummy1 dummy2 dummy3 fibonacci madtrans outpostest overload			
1	madtrans overload	dummy1	dummy2	dummy3 fibonacci outpostest
2	dummy2 madtrans	dummy1	overload	dummy3 fibonacci outpostest
3	madtrans overload	dummy1	dummy2	dummy3 fibonacci outpostest

Table 6.11: Calculated hypothetical machine loads (percentages over 1) used for determining new agent destinations in overload *Equal-Free* test scenario

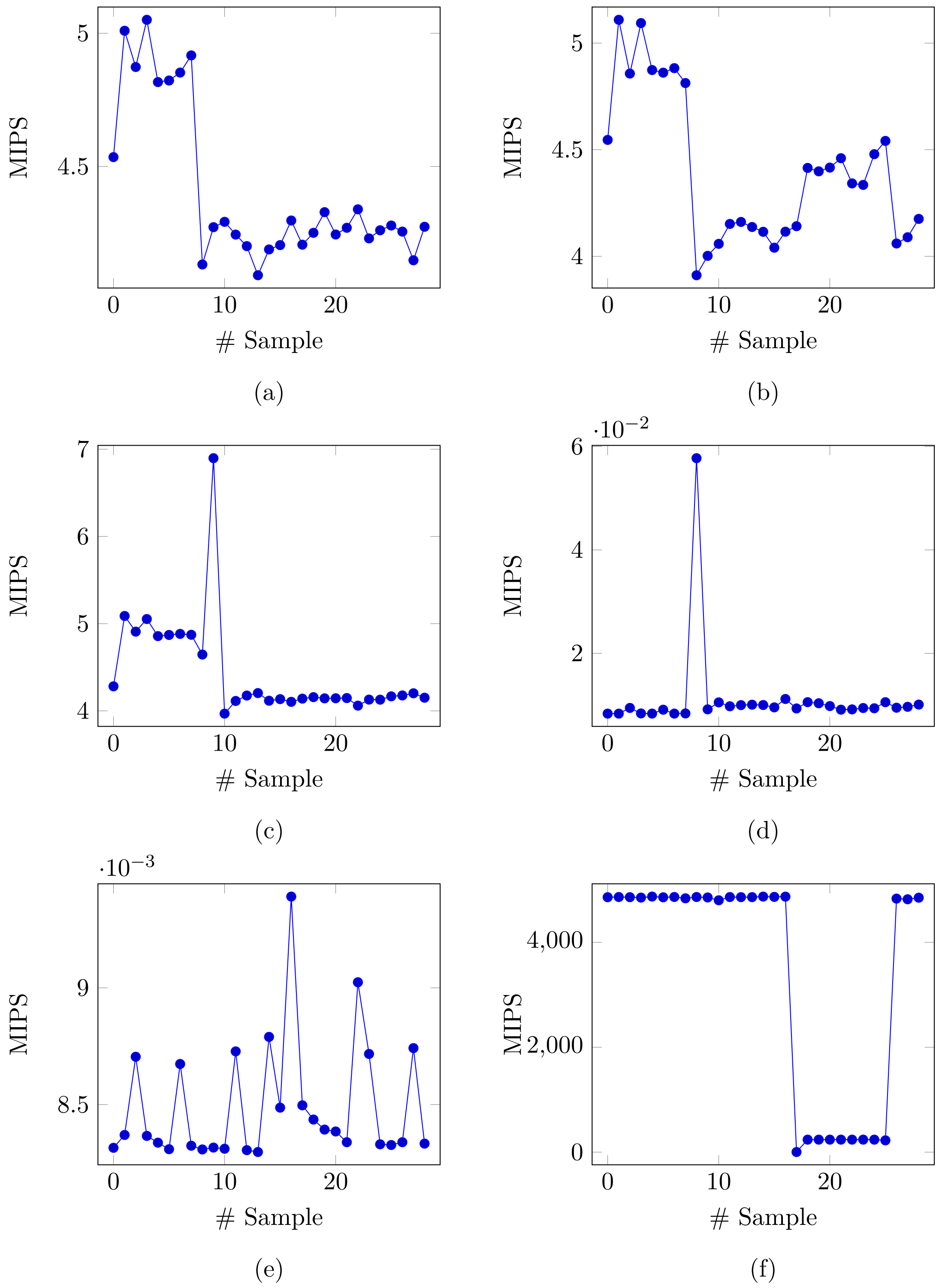
Migration	central	outpost_pi	outpost_arco1	outpost_arco2
1	0.927513	0.037168	0.034754	0.035306
2	0.000790	0.032485	35.178499	0.029781
3	0.045677	0.032208	0.032345	0.045677

Table 6.10 shows the destinations calculated by the *balanced* algorithm as a direct result of the values shown in Table 6.11. The newly introduced *overload* agent greatly affects the results when compared to the same scenario in the normal load test group.

At first glance, the previous table shows a value of 35.178499 (over 1) for the load of *outpost_arco1*. While this would seem impossible, the implementation of the algorithm (see *Equal balance algorithm*) must be considered before reaching that conclusion. In practice, the hypothetical loads that are used to determine the new locations are calculated with the MIPS of the agents **in their current location**. Therefore, the load value is not real and would be much lower after migration.

In Figure 6.6, the subfigures for agents *fibonacci* and *outpostest* have been merged into subfigure (d), considering that their loads were almost identical and both of them were placed in the same machine during execution. While most of the plots show similar values to those analyzed in the previous test scenarios, subfigure (f) shows very interesting results. The *overload* agent reaches its peak in MIPS when located in the *central* machine, suffering a great performance loss when migrated.

In addition, through the values of the tables it is possible to conclude that the algorithm will probably continue migrating *overload* between *central* and *outpost_arco1* due to the excessive resource usage done by the agent during its execution.



(a)

(b)

(c)

(d)

(e)

(f)

Figure 6.6: MIPS evolution in Equal-Free test scenario (overload); (a) agent "dummy1", (b) agent "dummy2", (c) agent "dummy3", (d) agents "fibonacci" and "outpostest", (e) agent "madtrans", (f) agent "overload"

6.4.2.2 Equal-Hold scenario

This test consists on running the system using the *balanced* algorithm with all agents free except for *dummy2* and *dummy3*, which are held in *central*.

The information found in Table 6.12 represents the calculated destinations **right before** agents are migrated. In the case of **migration 0**, all the agents are located in the central machine at the start of the test.

Table 6.12: Calculated agent destinations before migrations of the overload *Equal-Hold* test scenario

Migration	central	outpost_pi	oupost_arco1	outpost_arco2
0	dummy1 dummy2 dummy3 fibonacci madtrans outpostest overload			
1	dummy2 dummy3	fibonacci	madtrans overload	dummy1 outpostest
2	dummy2 dummy3	fibonacci	madtrans overload	dummy1 outpostest
3	dummy2 dummy3	fibonacci	dummy1 madtrans	outpostest overload

Table 6.13: Calculated hypothetical machine loads (percentages over 1) used for determining new agent destinations in overload *Equal-Hold* test scenario

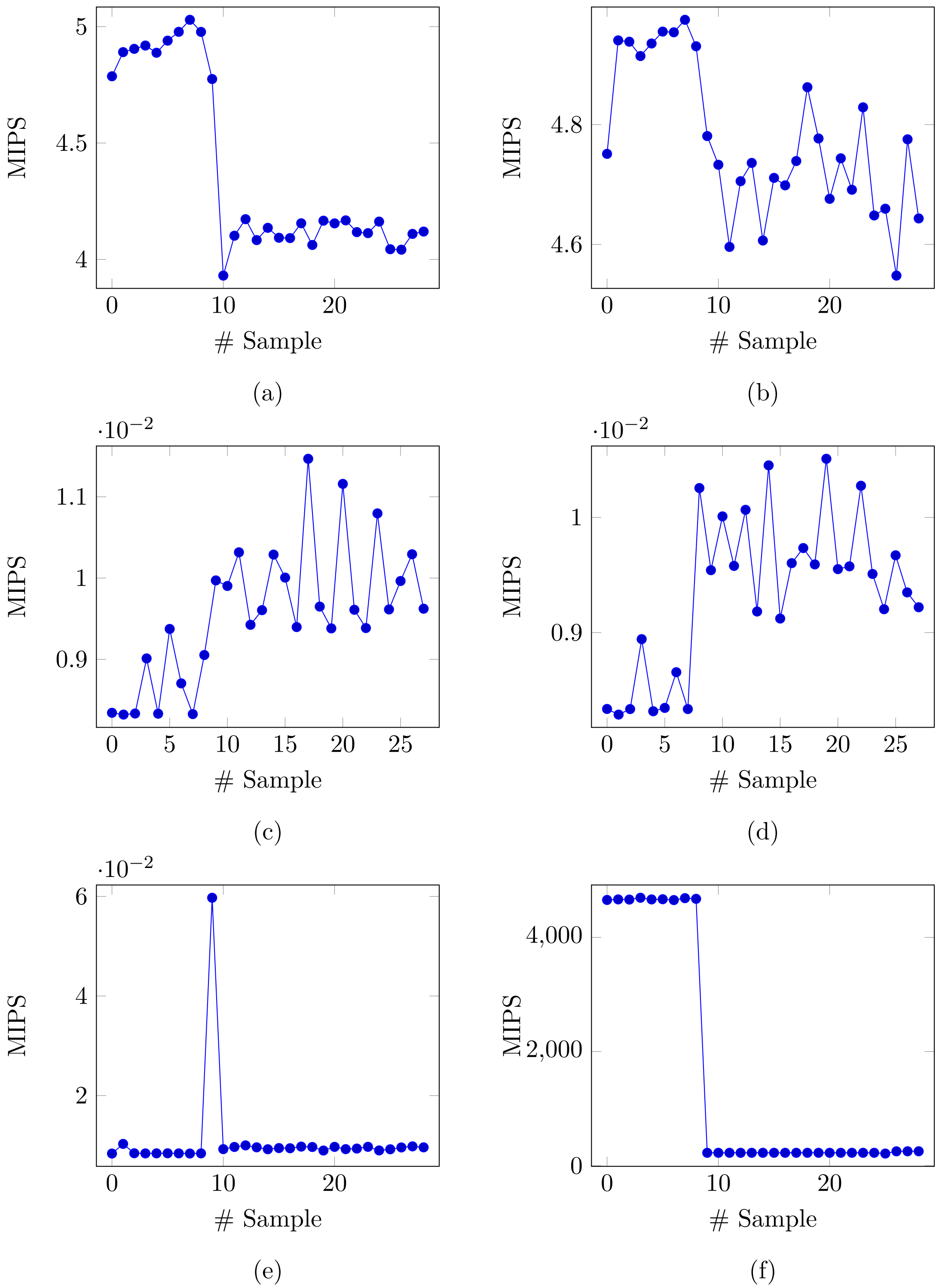
Migration	central	outpost_pi	outpost_arco1	outpost_arco2
1	0.001907	0.000063	33.843381	0.036377
2	0.001820	0.000071	1.642751	0.030078
3	0.001775	0.000082	0.030128	1.640552

Table 6.12 shows the destinations calculated by the *balanced* algorithm as a direct result of the values shown in Table 6.13. Due to agents *dummy2* and *dummy3* being on hold in the *central* machine, the algorithm migrates the *overload* agent to the outposts, greatly affecting its performance after the first migration.

A case of high load (33.843381 over 1) is found in a similar way to the **Equal-Free** scenario. However, the hypothetical loads for *outpost_arco1* and *outpost_arco2* exceed

1 in second and third migrations respectively. This is likely due to the loads being calculated by considering the total MIPS a machine is capable of through benchmarking, given that the benchmark results were the average values of 5 executions.

Regarding Figure 6.7, plots of agents *dummy2* and *dummy3* have been merged into subfigure (b) due to their similarity and location. Furthermore, and following the analysis of the previous tables, subfigure (f) shows the performance drop of the *overload* agent after the first migration, even though it maintains a regular load in both outposts.



(a)

(b)

(c)

(d)

(e)

(f)

Figure 6.7: MIPS evolution in Equal-Hold test scenario (overload); (a) agent "dummy1", (b) agents "dummy2" and "dummy3", (c) agent "fibonacci", (d) agents "madtrans", (e) agent "outpostest", (f) agent "overload"

6.4.2.3 Prio-Free scenario

This test consists on running the system using the *priority* algorithm with all agents free and available for automatic migration.

The information found in Table 6.14 represents the calculated destinations **right before** agents are migrated. In the case of **migration 0**, all the agents are located in the central machine at the start of the test.

Table 6.14: Calculated agent destinations before migrations of the overload *Prio-Free* test scenario

Migration	central	outpost_pi	oupost_arco1	outpost_arco2
0	dummy1 dummy2 dummy3 fibonacci madtrans outpostest overload			
1	overload			dummy1 dummy2 dummy3 fibonacci madtrans outpostest
2	overload			dummy1 dummy2 dummy3 fibonacci madtrans outpostest
3	overload			dummy1 dummy2 dummy3 fibonacci madtrans outpostest

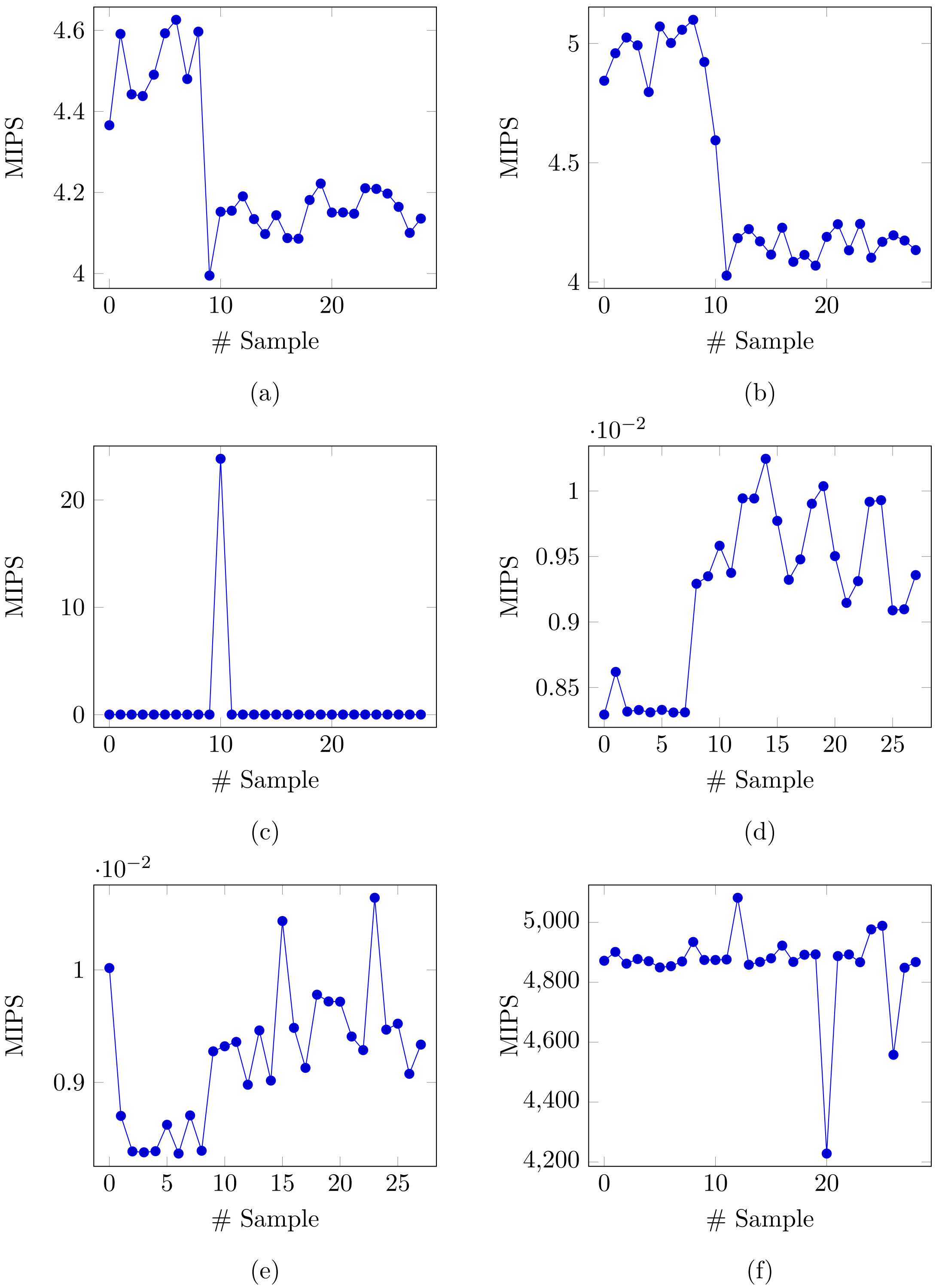
Table 6.15: Calculated hypothetical machine loads (percentages over 1) used for determining new agent destinations in overload *Prio-Free* test scenario

Migration	central	outpost_pi	outpost_arco1	outpost_arco2
1	0.9333203	0.0	0.0	0.105181
2	0.932935	0.0	0.0	0.089295
3	0.956055	0.0	0.0	0.090598

Table 6.14 shows the destinations calculated by the *priority* algorithm as a direct result of the values shown in Table 6.15. The first thing to note is that the *overload* agent remains **always** in the *central* machine. This behaviour is to be expected, considering that the algorithm determined the agent would have exceeded the 80% threshold parameter included in the algorithm (see *Priority algorithm*).

Therefore, while the rest of the agents are placed in *outpost_arco2* in a similar way to the normal load **Prio-Free** scenario, the *overload* agent is forced to stay in *central*, which is the default when all the machines exceed said threshold.

Plots for agents *dummy2* and *dummy3* have been merged, once again due to their similarity, into subfigure (b) of Figure 6.8. Overall, the plots show the results in line with the previously analyzed scenarios (e.g., *dummy* agents have a performance drop after migrated), with the *overload* agent maintaining a high load due to being located in the *central* machine.



(a)

(b)

(c)

(d)

(e)

(f)

Figure 6.8: MIPS evolution in Prio-Free test scenario (overload); (a) agent "dummy1", (b) agents "dummy2" and "dummy3", (c) agent "fibonacci", (d) agents "madtrans",(e) agent "outpostest", (f) agent "overload"

6.4.2.4 Prio-Hold scenario

This test consists on running the system using the *priority* algorithm with all agents free except for *dummy2* and *dummy3*, which are held in *central*.

The information found in Table 6.16 represents the calculated destinations **right before** agents are migrated. In the case of **migration 0**, all the agents are located in the central machine at the start of the test.

Table 6.16: Calculated agent destinations before migrations of the overload *Prio-Hold* test scenario

Migration	central	outpost_pi	oupost_arco1	outpost_arco2
0	dummy1 dummy2 dummy3 fibonacci madtrans outpostest overload			
1	dummy2 dummy3 overload			dummy1 fibonacci madtrans outpostest
2	dummy2 dummy3 overload			dummy1 fibonacci madtrans outpostest
3	dummy2 dummy3 overload			dummy1 fibonacci madtrans outpostest

Table 6.17: Calculated hypothetical machine loads (percentages over 1) used for determining new agent destinations in overload *Prio-Hold* test scenario

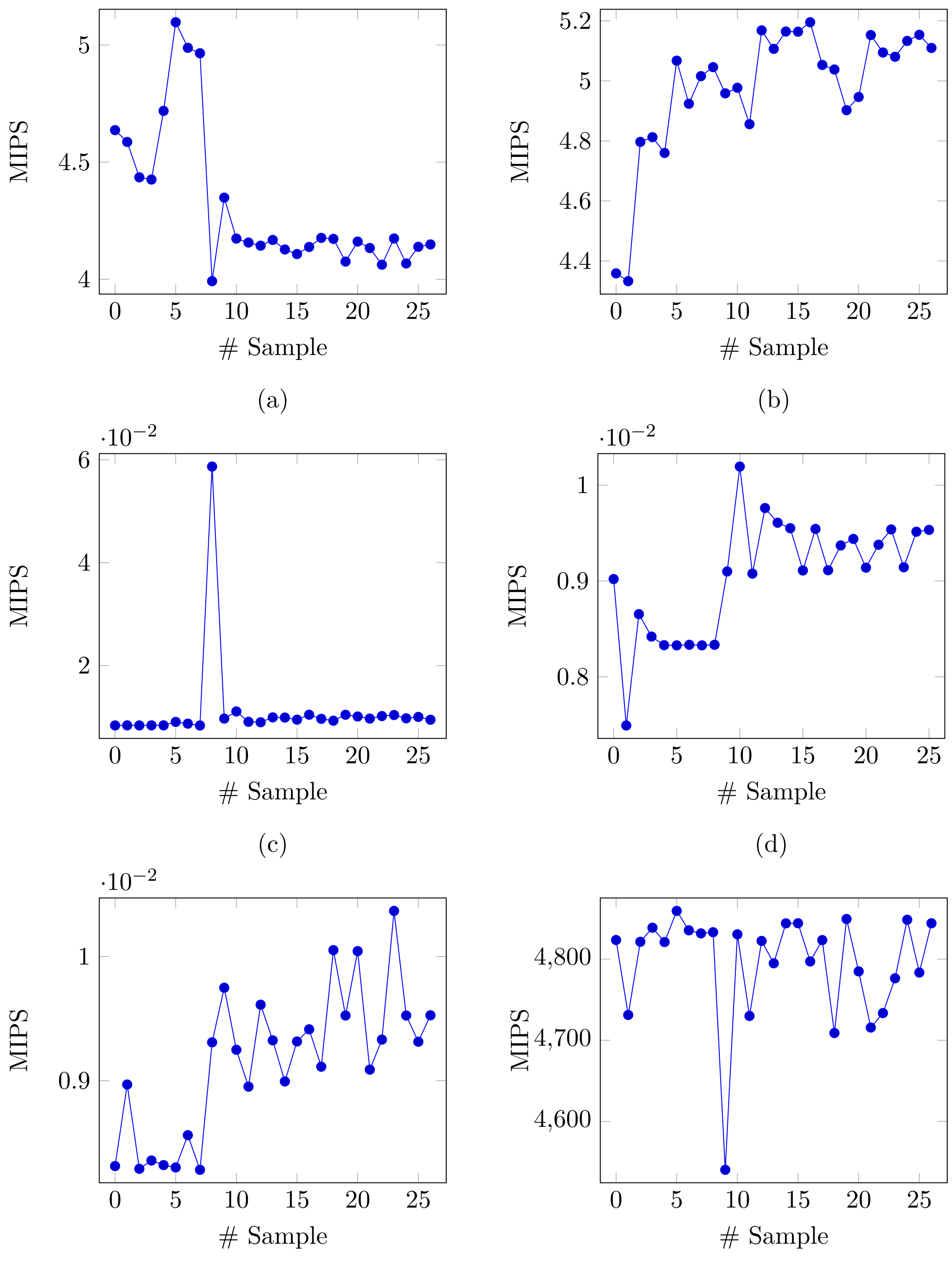
Migration	central	outpost_pi	outpost_arco1	outpost_arco2
1	0.925975	0.0	0.0	0.036034
2	0.919349	0.0	0.0	0.030091
3	0.929200	0.0	0.0	0.029579

Table 6.14 shows the destinations calculated by the *priority* algorithm as a direct result

of the values shown in Table 6.15. As in the previous scenario, *overload* surpasses the 80% threshold in all the outposts with higher priority, and is moved to *central* machine.

Similar to previous cases, in Figure 6.9, *dummy2* and *dummy3* (subfigure (b)) show a constant *high* load while *dummy1* loses some MIPS due to the migration. In addition, as before, the *overload* agent maintains a regular load due to being always in *central*.

As a conclusion, the priority algorithm has shown to be efficient for centralizing the load in as little machines as possible, which would allow to use outposts for other tasks or simply shut them down.



(a)

(b)

(c)

(d)

(e)

(f)

Figure 6.9: MIPS evolution in Prio-Hold test scenario (overload); (a) agent "dummy1", (b) agents "dummy2" and "dummy3", (c) agent "fibonacci", (d) agents "madtrans", (e) agent "outpostest", (f) agent "overload"

Chapter 7

Conclusions and future work

This chapter contains the conclusions reached after development of the project and the analysis of the solution, in addition to proposed further work in order to expand what has been implemented and discussed in this document.

7.1 Conclusions

The *Introduction* of this document listed the set of objectives that were aimed to be solved through the presented solution. These are analyzed below.

The main objective of this project was to **design an architecture to support agent migration** by extending the original Zoe project. This was achieved by taking advantage of the functionalities and mechanisms available in the Zoe architecture, which resulted in a *pluggable* system that is completely compatible (both backward and forward) with the original system.

This software based solution is possible due to the **design of a portable and efficient communication protocol**, which was another objective of this work. Achieving a generic an easy to implement solution is a complicated ask. While the alternatives initially considered were very powerful and would have solved the problem at hand (in particular, live migration of agents) directly, they would have required modifications to the original Zoe architecture or specific hardware support.

Instead of that, this proposal leverages mechanisms available in most, if not all, programming languages, such as plaintext handling or data (de)serialization, effectively making it appropriate for its use in any programming language and hardware architecture supported by those languages. The result obtained is a transparent system were agent developers do not need to worry about how the outpost or the *Scout* agent work, greatly reducing the complexity for them, and with a detailed specification for creating *migration-enabled* agents.

Even though such approach works well with Zoe agents considering their usual load, there are also limitations to what can be done. The most notable limitation is the fact that real *live migration*, such as the one offered by containers or virtual machines, cannot be achieved and there is bound to be downtime in the service offered by the agent.

In addition to migration capabilities, another requirement was to **design load balancing algorithms for an efficient hardware resource management**. To this end, the *equal-balance* and *priority* algorithms were implemented, although it is clear that they may not be fit for every use case. Therefore, algorithm implementations were decoupled from the main *Scout* agent routines.

With this decoupling, developers are encouraged to extend the system with their own algorithms, which was designed to be rather simple: algorithms receive current information on agents and outposts and must simply return the new location of each agent. This type of implementation gives freedom to developers, as long as they use the specified format for the output of their algorithms, without needing to know how the *Scout* agent works internally.

Finally, the last objective was to **implement and evaluate the system in an heterogeneous architecture**. As shown in the *Evaluation* chapter of this document, this was achieved by performing tests on machines with different hardware architectures and operating systems, in particular Debian/Raspbian for *x64* and *armhf* systems.

The *Result analysis* performed showed that the system worked correctly in the setup used. While there were evident decreases in performance when agents were migrated to a Raspberry Pi computer, especially in the overload test scenarios, these were as expected considering the difference in the computational power of each machine. Nonetheless, every agent continued to work as expected, proving that the system can indeed be implemented in heterogeneous environments.

All in all the implemented solution, which is formed by over **3000 lines of code** among all the components of the system, solves the problem considered at the beginning of this document and has allowed the student to experience first hand the process of design and implementation of a complex system by taking advantage of existing protocols and tools (such as *SSH tunneling* for inter-machine communication) and to apply knowledge gained throughout the years at the University.

7.2 Future work

As with every project, the proposed solution also has room for improvement. While the core functionalities and features have been implemented, there are some parts that could be enhanced or would be interesting to add to the system. The following lines express some ideas of future work for this project.

First and foremost, the protocol would need to be ported to **other programming languages**. Considering that it was designed to be as generic as possible, it would be possible to include the required integration in the standard library after new languages are added to the Zoe platform.

Moreover, it would be very beneficial to have a better integration with other core Zoe agents, in particular with the **Zoe agent manager**. Taking into account that the agent manager is the one in charge of all tasks regarding administration of agents (e.g. installation, removal, update, etc.), information exchange between the manager and the *Scout* could be used for cases such as trying to update an agent located in a remote machine.

Lastly, **new load balancing algorithms** could be implemented and existing ones improved. This would include energy saving policies and linear programming algorithms for the load balancing process. These new implementations would of course need to be tested, which might provide interesting results, depending on the case scenario, that might be applied to other projects.

Chapter 8

Glossary

8.1 Acronyms

Acronym	Meaning
GUL	Grupo de Usuarios de Linux (Linux User Group)
PDA	Personal Digital Assistant
IPA	Intelligent Personal Assistant
API	Application Programming Interface
IRC	Internet Relay Chat
SaaS	Software as a Service
RPC	Remote Procedure Call
REST	Representational State Transfer
SSL	Secure Socket Layer
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
GNU	GNU is Not UNIX
NFS	Network File System
KVM	Kernel-based Virtual Machine
LXC	Linux Containers
RMI	Remote Method Invocation
MIPS	Million Instructions Per Second
PID	Process ID
GTK	GIMP ToolKit

Appendix A

Installation manual

This section covers the installation and key configuration aspects of the Zoe virtual assistant and the Outpost addition (the work explained in this document).

A.1 Preparation

Installation steps shown here are performed in two different hardware/software configurations:

- Debian GNU/Linux testing (stretch) on an i7 2670QM @ 2.2GHz: main server
- Raspbian on Raspberry Pi: outpost

Both platforms have common software requirements and can be installed with the following command (supposing superuser access is enabled):

```
# apt-get install git perl openjdk-8-jre python3 python3-pip
```

This command installs git, Perl 5, Java 1.8, Python 3 and pip respectively.

Regarding the code of the outpost project, the following directory structure is supposed (inside a compressed file):

```
agents/     -- directory containing agents code (scout and outpostest)
cmdproc/    -- natural language commands for communication channels
mailproc/   -- natural language commands for email
lib/        -- outpost library code
outpost/    -- outpost server code
dashboard/  -- visualization GUI code
```

Note that the outpost system expects public key authentication when opening the SSH tunnels. This is done by enabling the `PubkeyAuthentication` option in the SSH daemon configuration file (in Debian `/etc/sshd/sshd_config`) and appending the public key from the user to `~/.ssh/authorized_keys`.

As it is out of the scope of this work, generation of secure key-pairs will not be covered.

A.1.1 Server installation

The code for the *Zoe startup kit* is publicly available at https://github.com/voiser/zoe-startup-kit and can be downloaded using git. For the sake of simplicity, code will be downloaded to `/home/zoe` for fast access. This directory is created with the command:

```
# mkdir /home/zoe
# chown -R USER:USER /home/zoe
```

Git will download all the code, and past revisions, from the remote repository. By default it works with the latest code in a branch called *master*. However, and given that the project is currently being developed, this document will work with the latest code committed at the time of writing (with id *340ef02483ceb0642e78f97d928f91717751ca62*). In order to do this:

```
$ git clone https://github.com/voiser/zoe-startup-kit.git /home/zoe
$ cd /home/zoe
$ git checkout -b outpost 340ef02483ceb0642e78f97d928f91717751ca62
```

The previous commands make sure the working code is the one the tests were performed in. To enable outpost capabilities, the following actions have to be performed (suppose `SRC` directory is the base directory that contains the outpost source):

```
$ cp -r SRC/agents/* /home/zoe/agents/
$ cp -r SRC/lib/*   /home/zoe/lib/
$ cp SRC/cmdproc/* /home/zoe/cmdproc/
$ cp SRC/mailproc/* /home/zoe/mailproc/

$ mkdir -p /home/zoe/etc/scout/rules
$ touch /home/zoe/etc/scout/scout.conf
$ touch /home/zoe/etc/scout/outpost.list
```

In addition to copying the outpost-related files, an additional command creates the directory structure used to store both scout configurations (`/home/zoe/etc/scout`) and outpost **agents rules** (`/home/zoe/etc/scout/rules/`).

While not necessary, it is recommended to explicitly identify the *scout* agent in the Zoe configuration file (`/home/zoe/etc/zoe.conf`) by adding the following lines:

```
[agent scout]
port = <PORT_NUMBER>
```

Where `<PORT_NUMBER>` could have a value such as **30008**. This way, the server knows how to address the *scout*.

Furthermore, it is necessary to install the Linux *perf* tool for accessing hardware counters and *autossh* for the tunneling. Usually this can be performed with the command:

```
# apt-get install linux-perf autossh
```

If the *linux-perf* package is not available, static builds of the *perf* binary would also work in this case.

Finally, the *scout* agent needs some additional Python modules for its operations, the `pip3` tool can install these to the *lib* directory of the agent without needing superuser permissions:

```
$ pip3 install -t /home/zoe/agents/scout/lib paramiko scp peewee
```

A.1.2 Outpost installation

Supposing the outpost machine has the same `/home/zoe` directory with the commit specified in the previous section, installation has minor differences compared to the one for the main server. File copy is as follows:

```
$ cp -r SRC/lib/* /home/zoe/lib
$ cp -r SRC/outpost/* /home/zoe/

$ rm -rf /home/zoe/zoe /home/zoe/server
$ rm -rf /home/zoe/agents/*
$ rm -rf /home/zoe/disabled-agents/*
$ rm /home/zoe/etc/zoe.conf && touch /home/zoe/etc/zoe.conf

$ mkdir -p /home/etc/outpost && touch /home/zoe/etc/outpost/outpost.conf
```

As the `linux-perf` package was not available for Linux kernel 4.1 (stable version of Raspbian at the time of writing), a static build downloaded from the Internet was used in its place.

A.2 Configuration

This section covers the main configuration aspects of the outpost system. Details specific to Zoe are available online[1].

By default, both server and outpost will listen to address `localhost:30000` as defined in `etc/environment.sh`.

A.2.1 Server configuration files

There are two important files that configure how the outpost system works. The first one is located in `/home/zoe/etc/scout/scout.conf` and contains information for both the scout itself and the machine it is running on. Its structure is as follows:

```
[general]
perf_path = /usr/bin/perf
balance = none
priority = 2
mips = 5217.933559389

[agents]
free = madtrans outpostest fibonacci dummy1 dummy2
hold =
```

Regarding the *general* section:

- `perf_path`: specifies the path to the Linux `perf` utility
- `balance`: specifies the balance algorithm to use (*equal* or *prio*), disable with *none*
- `priority`: specifies the priority of the machine in the *prio* balance algorithm, higher numerical value equals lower priority
- `mips`: Million Instructions Per Second the machine is capable of as measured with the Linpack benchmark

[1] http://zoe.readthedocs.io/en/latest/index.html

Regarding the *agents* section:

- `free`: list of agents that can be migrated automatically when balancing
- `hold`: list of agents that cannot be migrated automatically when balancing but can be migrated manually

By default, new agents are added to the *free* list by the scout.

The second configuration file specifies connection settings and additional details for the remote machines/outposts. It is located in `/home/zoe/etc/scout/outpost.list` and has the following structure:

```
[outpost outpost_pi]
host = 192.168.1.111
remote_port = 30000
local_tunnel = 29999
remote_tunnel = 29999
directory = /home/zoe_outpost
priority = 1
mips = 132.287849904
```

Each section must start with *outpost* followed by the unique name given to the outpost, with the following settings:

- `host`: IP or hostname used to connect to the outpost
- `username`: optional, username to be used in connection, defaults to current local user
- `remote_port`: port number the remote sever is listening at (in the Raspberry Pi)
- `local_tunnel`: local port number used to establish the SSH tunnel connection
- `remote_tunnel`: remote port number used to establish the SSH tunnel connection (in the Raspberry Pi)
- `directory`: base directory of the outpost in the remote machine
- `priority`: specifies the priority of the machine in the *prio* balance algorithm, higher numerical value equals lower priority
- `mips`: Million Instructions Per Second the machine is capable of as measured with the Linpack benchmark

A.2.2 Outpost configuration files

As opposed to the scout configuration, the outpost is easier to configure. There is only one configuration file located in /home/zoe/etc/outpost/outpost.conf which has the following structure:

```
[central]
host = localhost
port = 29999
tunnel = 29999

[outpost]
id = outpost_pi
perf_path = /home/zoe_outpost/perf
```

As the outpost does not have to worry about additional details, it contains only a section with information on the central server:

- host: host in which the server is running, as the outpost works with SSH tunneling, this is usually *localhost*
- port: port number from which the central server can be accessed (tunnel port)
- tunnel: port number of the tunnel to which the outpost is connected (in the central machine)

And a section with information on the outpost:

- id: unique ID (same as configured in outpost.list in central) used to work with messages directed to the outpost itself
- perf_path: path to the Linux perf tool for measuring performance

Appendix B

User manual

This section covers the usage of the outpost system for Zoe administrators, in particular the commands used to communicate with the *Scout* and the dashboard, which can be used as a secondary way to visualize simple data obtained from the *Scout*.

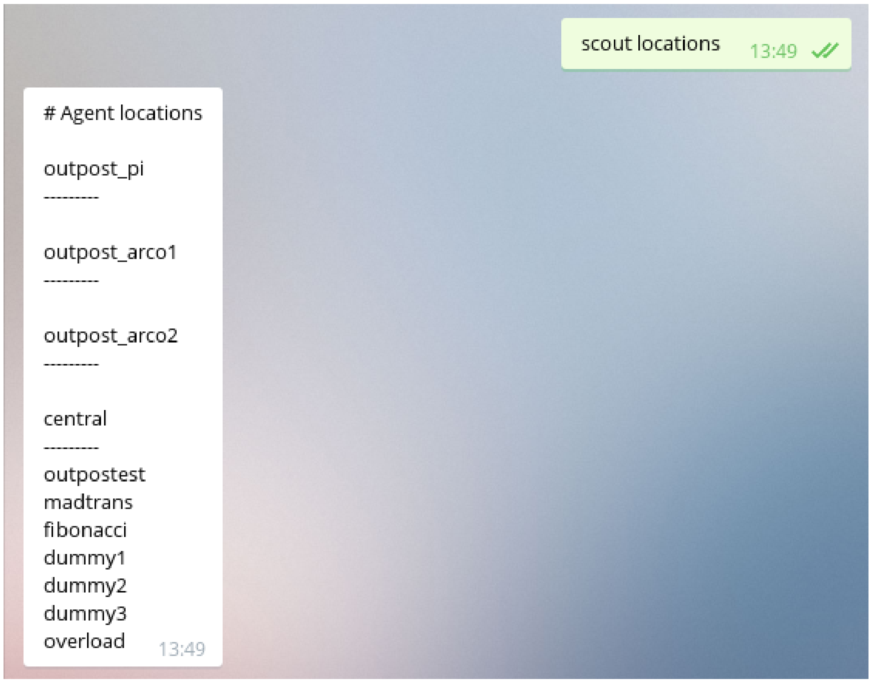
B.1 Commands

Scout commands were designed to resemble a classic command line interface, with the difference that they are executed through a messaging service. The implementation, for instance, was tested using Telegram, Jabber and email messaging, although any communication channel available to Zoe should be able to work with the commands. Figure B.1 shows an example of the Telegram interface by executing the `locations` command.

While regular Zoe commands employ natural language recognition, such feature could be potentially dangerous for critical commands like the ones implemented for the Scout. Instead, the exact matching implemented by David Muñoz is used [35] to provide the functionality.

It should be noted that email commands are slightly different to the other channels (Telegram, Jabber, etc.). In order to recognize an incoming mail as a scout message, the subject of the mail should be `scout`, and the command should appear in a single line of the body.

In the following sections, all the commands available will be explained in detail.



scout locations

```
# Agent locations

outpost_pi
---------

outpost_arco1
---------

outpost_arco2
---------

central
---------
outpostest
madtrans
fibonacci
dummy1
dummy2
dummy3
overload
```

Figure B.1: Example command through Telegram

help

This command displays a list of available commands to the user. It can be executed by sending the following message to Zoe:

```
scout help
```

backup

The *backup* command may be used to regenerate the backup directory of an agent (see *File migration*). It can only be executed when the agent in question is present in the central server and will discard an already existing backup directory.

Note that the backup is already recreated automatically when an agent is migrated from central, therefore this command is only implemented to be used in special cases.

The syntax for the command is as follows:

```
scout backup <agent>
```

Where `<agent>` is the name of the agent to back up (for instance, `madtrans`).

close-tunnel

Closes a SSH tunnel to the specified outpost (see *SSH tunneling*). The tunnel should be opened for this command to work (the *Scout* will check automatically).

The syntax for the command is as follows:

```
scout close-tunnel <outpost>
```

Where `<outpost>` is the name of the outpost for which the tunnel is being closed (for instance, `outpost_pi`).

hold

This command flags an agent as *held* in place so that it cannot be migrated by the load balancing algorithm (see *Holding an agent*). The agent should not be already on hold, although the *Scout* will check automatically.

The syntax for the command is as follows:

```
scout hold <agent>
```

Where `<agent>` is the name of the agent to hold (for instance, `madtrans`).

launch-outpost

Starts (or restarts) an outpost server **and all the agents located in that outpost**. Note that the behaviour of this command is to restart everything given that the *Scout* does not know the actual state of the server (see *Launching an outpost*).

The syntax for the command is as follows:

```
scout launch-outpost <outpost>
```

Where `<outpost>` is the name of the outpost to launch (for instance, `outpost_pi`).

locations

Returns a table-like list of the available outposts and the agents that are present in each of them. The presented information is based on the last known location of an agent as stored in the internal database.

The syntax for the command is as follows:

```
scout locations
```

migrate

Used to manually migrate an agent to a specific outpost. This triggers *Data migration* and *File migration* processes and informs the user when the migration has been completed.

In addition, the user will be notified in case an error in the migration.

The syntax for the command is as follows:

```
scout migrate <agent> <outpost>
```

Where `<agent>` is the name of the agent to migrate (for instance,`madtrans`) and `<outpost>` is the name of the new destination (for instance, `outpost_pi`).

open-tunnel

Opens a SSH tunnel to the specified outpost (see *SSH tunneling*). The tunnel should be closed for this command to work (the *Scout* will check automatically).

The syntax for the command is as follows:

```
scout open-tunnel <outpost>
```

Where `<outpost>` is the name of the outpost for which the tunnel is being opened (for instance, `outpost_pi`).

retrieve-info

Forces stored agent information retrieval (see *Data migration*). This command is executed automatically and should only be used in special cases.

The syntax for the command is as follows:

```
scout retrieve-info <agent>
```

Where `<agent>` is the name of the agent for which to retrieve information (for instance, `madtrans`).

retrieve-msg

Forces stored agent message retrieval (see *Message relaying*). This command is executed automatically and should only be used in special cases.

The syntax for the command is as follows:

```
scout retrieve-msg <agent>
```

Where `<agent>` is the name of the agent for which to retrieve messages (for instance, `madtrans`).

status

This command offers two different variants: it can either obtain status information for agents or for outposts.

The agent mode is executed using the syntax:

```
scout status agents
```

And returns the following information:

- Whether the agent is on hold or not
- Its current location
- Current MIPS of the agent
- Timestamp of the last update of the information

The outpost mode is executed using the syntax:

```
scout status outposts
```

And returns the following information:

- Whether the outpost is running or not
- Host address
- Ports used for server and tunnel
- Remote outpost directory path
- MIPS the machine is capable of
- Priority of the outpost for *prio* load balancing algorithm
- Timestamp of the last update of the information

Furthermore, the outpost mode also returns MIPS, priority and timestamp of the central machine in addition to the current balancing algorithm in use.

stop-outpost

Stops an outpost server **and all the agents located in that outpost**.

The syntax for the command is as follows:

```
scout stop-outpost <outpost>
```

Where `<outpost>` is the name of the outpost to stop (for instance, `outpost_pi`).

unhold

This command flags an agent as *free* so that it can be migrated by the load balancing algorithm (see *Freeing an agent*). The agent should not be already free, although the *Scout* will check automatically.

The syntax for the command is as follows:

```
scout unhold <agent>
```

Where `<agent>` is the name of the agent to unhold (for instance, `madtrans`).

B.2 Dashboard

In order to use the dashboard, Python bindings for GTK and additional Python modules must be installed in the system.

In Debian systems, the bindings can be installed with the following command:

```
# apt-get install python3-gi
```

While the additional modules can be installed with:

```
$ pip3 install blinker paramiko scp peewee
```

B.2.1 Configuration

The dashboard must be configured to obtain the information from a Zoe installation. Two different operation modes are considered: local and remote.

In **local mode**, the data is obtained from the same machine. It requires the following parameter:

- Path to the root of the Zoe instance

In **remote mode**, data is obtained from a remote machine through SSH and relevant files are copied to the machine where the dashboard is being executed. It requires the following parameters:

- Host address of the remote machine
- Path to the root of the remote Zoe instance

B.2.2 Usage

Once configured, the dashboard periodically fetches the data from the Zoe instance and works with a copy to display the information in order to prevent affecting the original one used by the *Scout*.

Figure B.2 shows the **Outposts** view, which offers a simple visualization of where each agent is located. In the image, all the agents are currently running in the *central server*, and the list can be scrolled vertically and horizontally as needed.

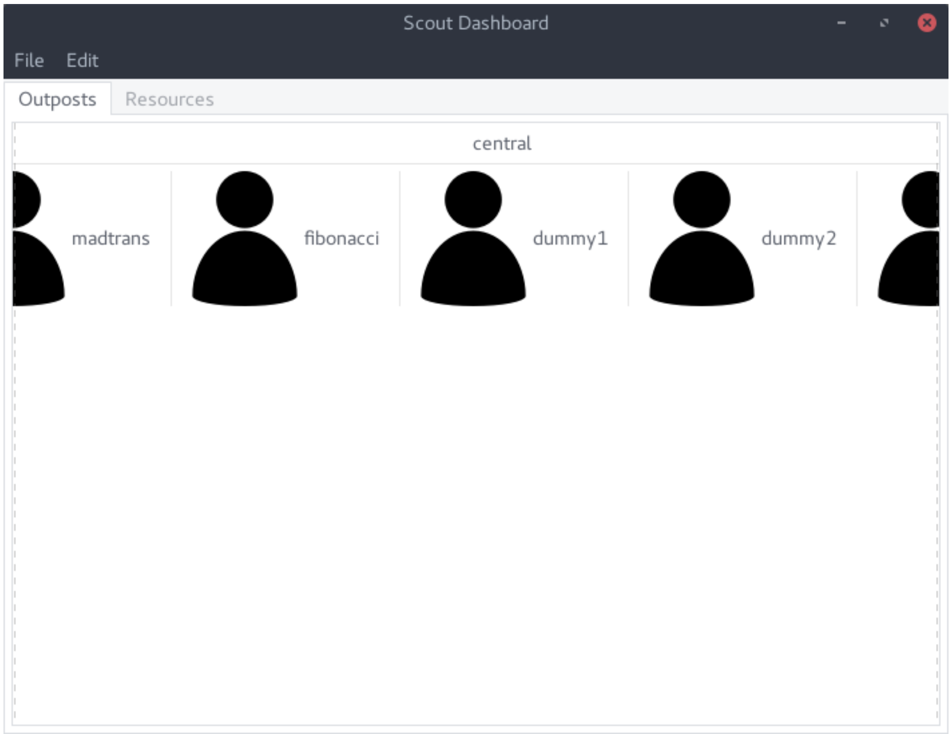
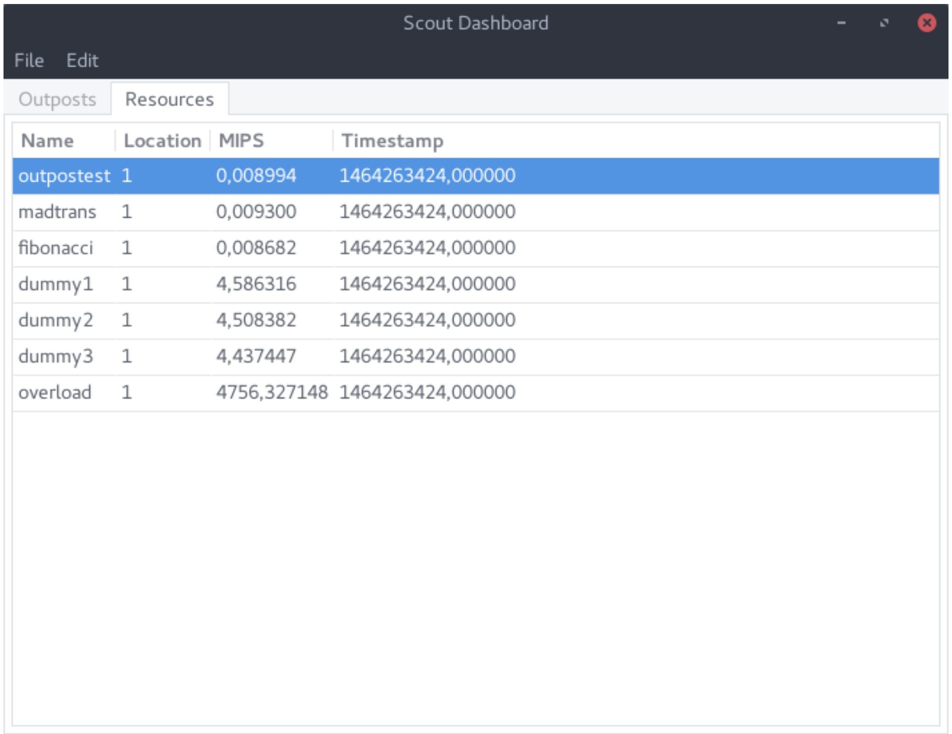


Figure B.2: Dashboard *Outposts* view

The **Resources** view shows a small table with the ID of the outpost the agent is currently in, its latest MIPS and the timestamp of the last update. In the future, this view could be expanded to include other resources that may be taken into account in the system.



Name	Location	MIPS	Timestamp
outpostest	1	0,008994	1464263424,000000
madtrans	1	0,009300	1464263424,000000
fibonacci	1	0,008682	1464263424,000000
dummy1	1	4,586316	1464263424,000000
dummy2	1	4,508382	1464263424,000000
dummy3	1	4,437447	1464263424,000000
overload	1	4756,327148	1464263424,000000

Figure B.3: Dashboard *Resources* view

References

[1] R. H. Wiggins Iii, “Personal digital assistants,” *Journal of digital imaging*, vol. 17, no. 1, pp. 5–17, 2004.

[2] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and others, “Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 223–238.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and others, “A view of cloud computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[4] Lucas Mearian, “WD ships world’s first 10TB helium-filled hard drive.” http://www.computerworld.com/article/3011142/data-storage/wd-ships-worlds-first-10tb-helium-filled-hard-drive.html [Online; Accessed 07-May-2016], Dec-2015.

[5] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[6] J. Thones, “Microservices,” *Software, IEEE*, vol. 32, no. 1, pp. 116–116, 2015.

[7] F. Martin, “SSL Certificates Howto,” *The Linux Documentation Project LDP*, 2002.

[8] J. Naude and L. Drevin, “The adversarial threat posed by the NSA to the integrity of the internet,” in *Information Security for South Africa (ISSA), 2015*, 2015, pp. 1–7.

[9] T. Ylonen and C. Lonvick, “The secure shell (SSH) protocol architecture,” 2006.

[10] E. S. Raymond, *The art of Unix programming.* Addison-Wesley Professional, 2003.

[11] I. Habib, “Virtualization with kvm,” *Linux Journal*, vol. 2008, no. 166, p. 8,

2008.

[12] Red Hat, Inc., “Virtualization Tuning and Optimization Guide.” https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html-single/Virtualization_Tuning_and_Optimization_Guide/index.html [Online; Accessed 1-Jun-2016], 2016.

[13] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “kvm: the Linux virtual machine monitor,” in *Proceedings of the Linux symposium*, 2007, vol. 1, pp. 225–230.

[14] R. Chamberlain, L. Invenshure, and J. Schommer, “Using Docker to support reproducible research,” Technical report, Technical Report 1101910, figshare, 2014. http://dx. doi. org/10.6084/m9. figshare. 1101910, 2014.

[15] J. Fink, “Docker: A software as a service, operating system-level virtualization framework,” *Code4Lib Journal*, vol. 25, 2014.

[16] Docker, Inc., “What is Docker?” https://www.docker.com/what-docker [Online; Accessed 1-Jun-2016], 2016.

[17] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, “Design and implementation of the Sun network filesystem,” in *Proceedings of the Summer USENIX conference*, 1985, pp. 119–130.

[18] J. Forcier, “Paramiko homepage.” http://www.paramiko.org/ [Online; Accessed 16-May-2016], 2016.

[19] J. Bardin, “scp.py homepage.” https://github.com/jbardin/scp.py [Online; Accessed 16-May-2016], 2015.

[20] C. Leifer, “peewee documentation.” http://docs.peewee-orm.com/en/latest/ [Online; Accessed 16-May-2016], 2016.

[21] J. Kirtland, “Blinker homepage.” http://pythonhosted.org/blinker/ [Online; Accessed 16-May-2016], 2016.

[22] G. Foundation, “PyGObject homepage.” https://wiki.gnome.org/action/show/Projects/PyGObject [Online; Accessed 16-May-2016], 2016.

[23] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK users’ guide*. Siam, 1979.

[24] J. Gruber, “Markdown homepage.” https://daringfireball.net/projects/markdown/ [Online; Accessed 16-May-2016], 2016.

[25] J. MacFarlane, “Pandoc homepage.” http://pandoc.org/ [Online; Accessed 16-

May-2016], 2016.

[26] J. Waldo, “Remote procedure calls and java remote method invocation,” *Concurrency, IEEE*, vol. 6, no. 3, pp. 5–7, 1998.

[27] S. Josefsson, “The base16, base32, and base64 data encodings,” 2006.

[28] R. Medina, “fumi documentation.” http://fumi.readthedocs.io/en/latest/ [Online; Accessed 16-May-2016], Sep-2015.

[29] A. Burande, A. Pise, S. Desai, Y. Martin, and S. D’mello, “Wireless Network Security by SSH Tunneling.”

[30] V. M. Weaver, “Linux perf_event features and overhead,” in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, 2013, p. 80.

[31] BOE, “Ley 27/2014, de 27 de noviembre, del Impuesto sobre Sociedades.” *BOE*, no. 288, pp. 96939–97097, 28 nov 2014.

[32] “Tarifas Internet.” http://www.movistar.es/particulares/tienda/comparador-tarifas-internet/ [Online; Accessed 1-Jun-2016].

[33] “Comparativa tarifas eléctricas 2016.” http://tarifasgasluz.com/faq/comparativa-tarifas-electricas-2016#cual-es-precio-luz-2016 [Online; Accessed 1-Jun-2016].

[34] “Bases y tipos de cotización 2016.” http://www.seg-social.es/Internet_1/Trabajadores/CotizacionRecaudaci10777/Basesytiposdecotiza36537/index.htm [Online; Accessed 1-Jun-2016].

[35] D. Muñoz, “Exact match in Zoe commands.” http://voiser.org/post/100898543602/exact-match-in-zoe-commands [Online; Accessed 16-May-2016], Oct-2015.