Building blocks of scalable applications

Ciprian Crăciun
Scientific adviser: Prof. Dr. Dana Petcu
West University of Timişoara, Romania

9th July 2012

Abstract

Today's most demanding applications are social networking or social media sites, which are built in the emerging cloud computing environment, using the newest up to date techniques and technologies. Some of these applications are built on top of specialized software platforms, hosted by the cloud providers, called `PaaS`.

Therefore the current work tries to propose such an open source `PaaS` solution. It starts with an analysis of the business and economic environment surrounding these applications, then presents a set of case studies for both some popular applications, but also a few promising commercial platforms. Following that it expresses the requirements distilled from what has been observed, then an overall architecture and design of the proposed platform. It concludes by describing the experiences gathered while implementing a prototype of the platform, as part of the European FP7 research project mOSAIC.

Abstract

Cele mai populare aplicaţii ale zilelor noastre sunt de regulă cele de "*social media*" sau "*social networking*", construite în general în contextul domeniului emergent al "*cloud computing*". Multe dintre acestea sunt de fapt construite utilizând platforme specializate, oferite de furnizorii de servicii "*cloud*", numite `PaaS`.

În acest context lucrarea de faţă îşi propune să descrie şi să implementeze o astfel de platformă, disponibilă ca şi proiect "*open source*". Pentru început lucrarea prezintă o scurtă analiză a mediului economic ce guvernează aceste aplicaţii, urmând apoi prezentarea câtorva studii de caz, atât al unor aplicaţii populare, cât şi a câtorva platforme comerciale. În contextul expus, se trece la identificarea cerinţelor software ale platformei finale, apoi la descrierea arhitecturii şi a designului general al acesteia. În final sunt prezentate experienţele practice acumulate încercând implementarea unui prototip al acestei platforme, activitate conexă proiectului european de cercetare mOSAIC.

Contents

Chapter 1

Introduction

Today the most popular applications, and therefore economically very profitable for their owners, besides enterprise or commercial off-the-shelf software, are mainly those viral social networking or social media sites, or by generalization, those revolving around social interactions and relationships, the main examples in this category being Facebook, LinkedIn, Twitter, YouTube, and their myriad of clones. Then to these we can add those from the entertainment business, like online multiplayer games, then from the retail business, like EBay or Amazon, and, less viral thus less profitable than the previous categories, the professionally oriented or utilitarian ones, like GitHub, BaseCamp, or WikiDot. And last, but not least, we could add the free and open Wikipedia site, together with all the accompanying sister projects sponsored by the WikiMedia foundation.

What do all of these have in common? First is their presentation to, and interaction with the final user, which almost all of the time is through a web-browser, as a Web 2.0-based or `RIA`-based application. Then a large part are also available as mobile applications on today's popular platforms like Android or iOS, and only very few have classical desktop based interfaces. Then they have in common their operational requirements: 24/7 service to their users, uninterrupted regardless of any technical difficulties; followed by the necessity of efficiently handling varying user demand, either of predictable, but most important unpredictable, nature. Viewed from a technical perspective: their overall architecture falls into a small number of well known patterns; and all use quite the same or at least related backing technologies, like storage or communication, `OS`s. But the most invariant characteristic is their infrastructure solution, which implies either exclusively self hosting, or exclusively outsourced hosting, as very few mix self hosting for core parts and outsourced hosting for those have high usage variance.

Why do we care about these applications? Because these types of applications became so popular in recent years — up to the point where we have questioned ourselves if this is not the new DotCom bubble — that not only the big enterprises like Google, but even startups, began creating such applications by the dozen in the hope they will hit a critical mass and become viral. But this is only our incentive to study them, as they pose quite a few difficult technical problems, which, as it happens to be in our line of work, we try to our best knowledge to analyze, solve on paper, prototype, and, if we are not to bold hopefully, solve in practice.

Purpose Therefore the current work revolves around applications, like the ones presented above, which could be defined only by their characteristics: Internet accessible, close to zero downtime, sustaining dynamic user demand, easily if not automatically operated, based on conventional technologies, within affordable costs, usually developed by startups.

Unfortunately, we could easily observe that some of the previous examples do not fit well into this definition because they do not match the last two characteristics, specifically affordable for startups. But this is not true, for starters because most of these companies were in their turn startups, and grew, usually over night after reaching the viral status, into the corporations they are today. Then it is possible for startups to achieve this goal because there are solutions for incremental infrastructure procurement as they increase in size and profits, the infrastructure costs being one of the main factors hampering initial success.

As such, as hinted before, we set ourselves the goal to offer to these startups an acceptable technical solution, that would allow them to at least jumpstart their prototypes, skipping the effort of developing non-business related modules, by applying our proposed techniques, and reusing our developed tools.

Getting closer to today's technical realities and context — that of the "*cloud computing*" era, near its pivotal moment — we could rephrase our endeavour, and by sprinkling it with today's buzzwords, we would obtain: creating an open-source, `IaaS` independent, automated, and efficient `PaaS` for today's scalable applications; or in a couple of words an "*universal panacea for all technical problems*".

Thus if we focus on our purpose's last definition, we must admit that at the current state of art, speaking about such a product, of providing yet another `PaaS`, is, as the saying goes, like "*beating a dead horse*", because there are countless such alternatives out in the open begging for users and luring them with free offers. These are ranging from well matured solutions like Heroku, App Engine, or Joyent — some being provided by the same enterprises presented in the beginning, powering their viral applications — going through late entry companies trying to catch a piece of the market, like Cloud Foundry, OpenShift, or DotCloud; even followed by those rebranding their old solutions as "*cloud enabled*"; and at the other end of the spectrum with prototypes and toy projects from various individuals that want to experiment with new programming languages and technologies.

Therefore setting this as our main goal would most likely be a failure as the market is already saturated and overflown with such solutions. It follows then that our main focus should be in designing and implementing a technically sane solution, that although its immediate goal is to enable the building of such a `PaaS`, it could also be easily retrofitted to serve other purposes: like for example migrating from a classical multiple tier, manually deployed application, to a more dynamic and automated one using the same technologies; or by taking apart its modules and selectively reusing some of them in isolation just for their primary features.

In conclusion we reformulate the purpose of the current work, as that of providing a set of modules that by stitching them together we might end up with a `PaaS` suitable for the previously described applications, but if treated as Lego blocks, we can obtain other interesting usage scenarios. ◇

Contribution As presented in the previous section the topic of building a PaaS is not such a new topic, being amply covered by both the industry and the academia, although in different directions. As such the author's contribution is not related with the creation of new theoretical concepts, techniques, or algorithms, at most it could be that of documenting and mixing them. Nonetheless the author's contribution is that of dissecting a possible design of such a PaaS, of course influenced by already existing designs, followed by a proof-of-concept prototype based on reuse of other open-source solutions as much as possible.

Two of the designed and prototyped modules, namely the component hub and the credentials service, are, to the author's best knowledge, unique solutions, at least in the cloud computing environment, as their inspiration came from technologies a decade old.

But as the current work was written during the author's involvement in the European FP7 research project mOSAIC, most of the topics covered herein being conceived by the entire team and then published as research papers, it is hard to draw a line of ownership. To this end the current work tries to take a step further, at least in the design area, and present a, hopefully, improved version of the platform, based on the experience gathered during the project's unrolling. ◇

Roadmap In order to fulfill the goals expressed above, we first try, in §2 "IT ecosystem", to understand which is the context within which such applications, and therefore platforms, are created and operated. This context is actually a set of perspectives that give us the principles and constraints that are present in all these instances, and thus if our solution would have any success, it must follow them closely. The perspectives looked from are: §2.1 "Cloud computing", presenting the overall technical environment; §2.2 "Economical context", laying the basic operational constraints; §2.3 "Development methodologies" and §2.4 "Developer roles", explaining the methodologies we must adapt to; §2.5 "Target applications", providing a more detailed view of the applications we are pertaining to; and finally §2.6 "Miscellaneous remarks" for clarifications about some used concepts, but also a reminder of the basic mistakes we can easily fall pray to when developing in a highly distributed environment.

Then in §3 "Case studies", we continue with §3.1 "Application case studies", an overview of some real-life applications like §3.1.1 "Reddit" or §3.1.2 "LinkedIn", ones of the few that have publicly accessible in-detail blueprints, but also abstract scenarios from another platform provider, §3.1.3 "RightScale patterns". We then continue to look closer at some of the today's platform providers like §3.2.1 "Heroku" or §3.2.2 "Cloud Foundry", which fortunately provides a lot of publicly accessible internal details, or §3.2.3 "App Engine", which although very secretive about their internals, provides a very extensive developer documentation from which we can infer at least the "*do's and don'ts*".

The most important part of the work is §4 "Design", which provides a better description of the current's work background in §4.1 "Inspiration". Followed with §4.2 "Concepts", pinning a concrete meaning on some highly overloaded terms, after which we proceed with §4.3 "Requirements" to distill the previous chapters into concrete requirements that we shall refer back from further sections. Starting with §4.4 "Architecture", §4.5 "Life-cycles" and ending with §4.6 "Sub-systems" we get deeper into the solutions design. We then continue with a detailed presentation of those modules the author had the most involvement in, namely §4.7 "Component hub and controller" and §4.8 "Credentials service". We conclude with an in-depth analysis, §4.9 "Security", of the platform's security aspects and providing solutions.

For the developed prototype, in §5 "Implementation" we limit ourselves to present, in §5.1 "Guidelines", the main guidelines for technical solutions, and then, in §5.2 "Backing technologies", the most important ones selected. We conclude with §5.3 "Outcomes" by presenting the main prototypes the author has written.

The last part of the work, §6 "Conclusions", gives some last remarks about the work in §6.1 "Summary”, followed by future directions in §6.2 “Open problems and future work”.

Last but not least, in §6.3 “Acknowledgements”, we give credit where credit is due, as previously explained the current work was greatly influenced by other people’s work, and supported by various institutions and individuals.

Before moving to the matter at hand, we must add that in some parts the work is redundant, by summarizing or repeating what has been explained in other parts of the current work, especially when related with concepts and terminology, to allow the reader to jump directly into various chapters, without reading it from cover to cover. But regardless of the reading choice, two sections are crucial for the understanding, specifically §4.2 “Concepts” and §4.4 “Architecture”. And if the reader wants to focus only on the most interesting parts of the current work, then maybe these are, in order of originality, §4.7 “Component hub and controller”, §4.8 “Credentials service” and §4.9 “Security”. ◇

Chapter 2

IT ecosystem

In order to accomplish our goal of building a useful platform for startups to build upon, as we started to describe in §1 "Introduction", we first try to understand the environment in which such a solution would run, but also the non-technical factors, all of which will greatly influence its requirements, design and implementation.

Unfortunately the task at hand is completely a moving target, because every day countless ideas, techniques or solutions spawn over the Internet in various blogs or social media sites, and new products are pushed to the market in a fast pace, all these continuously shaping the technological and economical landscape. However because we can't cover in depth every such topic, we try to summarize at least those having the greatest impact on our domain.

Therefore in §2.1 "Cloud computing", closer to our domain of interest, we start by looking at the technical, mostly infrastructure related, context which greatly influences the design and operational tasks. Still related to the previous topics we move next to the economical aspects, in §2.2 "Economical context", where we describe the factors influencing the operational costs, but also the general economical climate of our domain. Then we jump in §2.3 "Development methodologies" and §2.4 "Developer roles" to talk about the development and operational tasks, focusing on the current trends and contrasting them to the classical approaches. And finally we slowly introduce in §2.5 "Target applications" some aspects of our problem, followed in §2.6 "Miscellaneous remarks" by observations pertaining to the subject.

2.1 Cloud computing

As previously stated the contents of this work lays in the context of the recently emerging "*cloud computing*" environment and it revolves also around distributed, scalable, highly available and fault tolerant applications. As such an acquaintance with cloud computing would prove useful in order to grasp the positioning of the statements and assertions that will follow.

2.1.1 Characterization

Unfortunately the concept of cloud computing is somehow still fuzzy, in that no succinct, and thus clear and precise, definition was given, although different standardization or research groups have tried to define, if not formalize, it. Defining cloud computing is a very difficult task, for at least two reasons: • firstly because it is a very hot research topic which has just started to take shape, and it will take some time until the industry and the academia is able to explore all the directions and crossroads with other fields; • and secondly because it caught the general industry's attention as a potential revenue driver, both as a technological leverage — allowing them to optimize or cost-efficientize their internal operations or product offerings — but also as a marketing buzzword — by stamping the "*cloud enabled*" label on anything from mere load balancers, clustered application containers, replicated databases, to virtualization technologies, just as the "*grid*" term started being stamped on not so quite grid-ish technologies like workload management systems, like `SGE` and others. As such most statements related to this emerging technology should be taken with a grain of salt, by carefully identifying the facts and weeding-out the hype.

But before citing any definition it would be better to start off by citing a warning:

> Cloud computing is still an evolving paradigm. Its definitions, use cases, underlying technologies, issues, risks, and benefits will be refined in a spirited debate by the public and private sectors. These definitions, attributes, and characteristics will evolve and change over time. (see [11])

Thus, if searching for a canonical or definitive definition may be impossible, we can at least settle with one generally accepted, both by its broad coverage, and by its proponent position and acceptance, the definition given by `NIST`, which states:

> Cloud computing is a model for enabling convenient, on demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. (see [11])

Which is then followed by the essential characteristics that summarize the definition:

> [...] on demand, self service, broad network access, resource pooling, rapid elasticity, measured service [...] (see [11])

Another recent definition that shares the spirit of the previous one, but which clearly points out the scalability requirement, business model, and resource contracts, is provided in [12], together with at least a dozen citations of earlier definitions, moreover the cited paper also represents a broad and useful overview of the subject:

> Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and / or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized `SLA`s. (see [12])

Because so far we have seen "*what cloud computing is*", it might prove useful to have a definition of "*what cloud computing is not*", which we have chosen to cite because it captures a very important aspect of cloud computing automatic scalability:

> If the system requires human management to allocate processes to resources, it is not a cloud: it is simply a data center. (see [9])

Unfortunately if we search deeper into the subject of defining what cloud computing stands for we would find even more diverging definitions and reaching a conclusion would be quite difficult, as for example some proponents see it as an embodiment of utility computing, while others see it as rather a version of the old and tried grid computing [12] [10]. As a consequence the best, and maybe most accurate, definition we could provide for it — as in general to all things in real world — would be to analyze the way we can or should use the cloud computing environment, and then try to decide on such a definition. Such an approach was already tried in [42], achieved with a community driven editorial process.

But the scope of the current work is not to pursue such a task, that of defining cloud computing, but instead to use it as a technical environment where, once we have identified suitable target applications, we can find common useful patterns, and in the end implement reusable building blocks which fit in such an ecosystem.

2.1.2 Models

Before moving further, and because the term "*cloud*" is so overloaded with semantics, we must note that there are multiple intended beneficiaries of such offerings:

- **consumer centric** — with products such as Google's GMail or GDocs, Apple's iCloud, Microsoft's Office 365, and most mobile applications, targeted at final non technical users as means to replace similar desktop based installable software, most of the time this is termed as `SaaS`, but depending on the marketing strategy it could also be labeled as `PaaS`;
- **integrator centric** — with offers such as Amazon's `AWS`, Google's App Engine, Microsoft's Azure, and others, targeting companies which want to either outsource their infrastructure, thus named as `IaaS`, or using managed platforms, named as `PaaS`, to build applications, which in most of the cases are in their turn sold as user centric cloud applications.

These examples, in a formal manner, are categorized as "*delivery models*" or "*service models*", and are thoroughly presented in [11] and [25], thus we just summarize them here for completeness:

- `IaaS` — where the provider rents to the beneficiary virtualized infrastructure like virtual machines, virtual networks, network attached storage, or anything raw in nature that can be built upon; we shall detail this in §4.2.3 "Infrastructure";
- `PaaS` — where the provider offers to developers run-time environments and development tools, allowing them to focus on the business perspective of the application, the platform providing an abstraction and, most importantly, automatizing for the lower layers;
- `SaaS` — where the offering consists of fully fledged, highly customized, applications ready to be used by non-technical persons.

As hinted before, these service models are usually stacked, some providers focusing on `IaaS`, some others building upon these and focusing on `PaaS`, while at the top of the food chain some focus on `SaaS`. One such example is Diaspora [27], a social networking site served as free `SaaS`, which uses Heroku, one of the main `PaaS` we shall focus in §3.2.1 "Heroku", which in its turn uses Amazon's `AWS` for acquiring virtual machines and storage.

And as a final note we must stress the fact that adopting a cloud-ish approach to our applications, doesn't exclude the possibility to move back to an on premises environment. In fact quite the contrary, as architecting for the cloud imposes constraints which make our applications more robust and scalable, and quite a few companies started deploying their application in a cloud environment but then gradually, and most of the time just partially, backed out to a more classical model [31] [53].

2.2 Economical context

As previously stated the cloud computing movement has generated a lot of stir in the `IT` related economy, involving large enterprises and small entrepreneurs alike, by promising, beside the technical benefits, a lot of advantages in terms of costs. Therefore in the current section we shall analyze some of the main changes that have happened in the `IT` world once the cloud computing gained traction, and for each of these factors we try do draw conclusions pertaining to our goal, that of building an application platform.

“*Pay as you go*” Maybe the major selling point of cloud computing is the fact that you don’t need any down payment to get started and use it, as for most cloud providers all you need is a credit card, even a personal one, which you must provide when registering a new account. After that you can acquire and release resources at your will and the amount you pay is simply computed by multiplying the quantity of consumed resources with an hourly quota dependent on the quality, thus the “*pay as you go*” name. In enterprise terminology this means a reduction in `CAPEX`, actually the infrastructure costs being close to zero, which then translates into a smaller risk if the product does not catch its audience in becoming viral, and thus the expected revenue does not come in.

But, something that most cloud providers forget to mention, is that while your initial down payment becomes almost zero, your monthly payment increases, which in economical terms, called `OPEX`, means that over a long period of time you start to catch up with the initial benefits, as hinted in [53].

Looking from this perspective we deduce that our proposed platform should be compatible which such an usage model, that is it should accept dynamism in what concerns the infrastructure, enabling the beneficiary to increase or decrease the committed resources. ◇

From outsourcing to automation of secondary concerns In general companies, usually from medium to large, try to outsource most of the activities which are not their main business, thus getting a better control of the budget, and hopefully optimizing costs and externalizing risks. Initially non critical activities were outsourced, like office space, cleaning, maintenance, etc.; then more sensitive ones, like customer relations, infrastructure, software maintenance, security audits, server administration, and others.

This outsourcing didn’t change the process much, as the way things were done remained the same: you wanted a new server you created a ticket on the hosting company’s support site, and later an employee prepared one for you; you wanted a new web-stack or a particular software component to be installed on one of your servers, you again created a ticket and someone later made it so. This process evolved on the road to cloud computing, but only in two very important aspects:

- **automatizing** — your request is, usually immediately, executed by a software component, thus eliminating the latency, and obtaining a more consistent and deterministic result each time, as the outcome didn’t depend on the skills of an employee;
- **programmability** — as the request is fulfilled by a computer, you can now have access, besides a `GUI` or `CLI`, to a programmatic `API` which in turn allows you to automatize your operations further.

But maybe the most important aspect of this transition is in the cost reduction of the offered services, which in turn means that the entry bar is lowered even further, thus smaller companies, the startups or even individuals, could afford such services.

For us the takeaway for our platform, is that like the cloud provider gives us programmatic `API`’s for managing the infrastructure, so should we offer to our beneficiary the needed tools to manage his application. ◇

Offering competition Like in any emerging domain, cloud computing became an attractive field especially for the industry, initially trying to acquire market share by providing a variety of products, often initially for free, ranging from high quality ones to bare prototypes. This competition meant that providers had to drastically reduce their effort invested in designing, implementing and testing their products, in favour of releasing earlier their technologies, in enterprise terms "*reducing the time to market*", labeling them as "*beta*" versions and hoping to finish the final stages of testing in the field. The good side, is that developers get their hands early on these solutions, thus allowing them to design their products for future technologies, and at the same time providing feedback to the providers about missing or superfluous features. At the same time it allows companies to compare their products with their competitors' alternatives, thus fueling both innovation, but on the flip side of the coin, driven by marketing data-sheet inflation, a lot of needless or corner case features appeared.

Now looking from the entrepreneur's perspective, as the technology infiltrates in almost all of our daily activities, and our virtual self becomes more coupled with our physical self, and as the cost of initial development and operation becomes smaller, thus affordable, plenty of new and promising products are released weekly, to the point that for each conceivable idea there are at least a couple of solutions out there. This in turn puts pressure on the product life-cycle and forces teams to "*put something out-there*", with minimum investment, to see if it is a viable solution or not, thus minimizing the potential risks.

From this we could learn at least a couple of things: • first that we should analyze the the solutions of our enterprise competitors, learning from their experiences and thus extracting some good ideas and implementation techniques, as we try to accomplish in §3.2 "Platform case studies", because we don't have the necessary resources to invest in market studies and extensive research; • second that we should not wait for ever in making our work public, but we should clearly inform and warn our users about the quality of the provided prototype; • that we should carefully select only those features that make sens for the potential users, and not treat them as user bait; • but in all these actions we need moderation, as "*putting something out-there*" without a minimum level of quality will hurt us in the long term, as potential users might be easily disappointed by our prototypes. ◇

IT commoditization Unlike one or two decades ago, when programming was a profession requiring highly skilled and seasoned persons, or at least with a touch of genius, today's landscape looks very dissimilar, as many popular applications are conceived by smart, young, and very open minded individuals which experiment all kind of technologies until they settle for a particular solution; and stories like these can be found all over the place from Google or Facebook to the yet to be famous small startups.

This tells us that our solution should be as simple as possible, thus allowing potential users to get started easily without requiring lots of insight into the bowels of our platform. But at the same time as they get acquainted with the technology and their application gets more demanding, they should be able to understand, tune, or even change our solution to fit their needs. A very good example are the `LAMP` stacks — which, a decade ago, when provided as an "*hosted environment*", as the terminology was at the time, represented the de facto `PaaS` relative to now — that allowed developers, most of the time high school students, to create, from toy applications to semi-professional sites, with little computer science background. ◇

2.3 Development methodologies

Because discussing about development methodologies we are getting into a very delicate software engineering topic, we don't even intend to summarize them, but instead to make a few observations that would prove useful in our endeavour of creating a platform.

Over the years the industry has moved from the classical waterfall model, through prototyping, and recently to a set of related development models that raise our interest. As such we focus on the later instances, not treating them individually, but as a conglomerate, especially because in practice they are often mixed or adapted. Thus their main principles are given below:

- **short release cycles** — as pushed by the agile methodologies like Scrum or Kanban, where in each cycle, usually lasting from two to four weeks, the development team is tasked to take a reduced set of features from the idea stage, and move it through analysis, design, development and testing, so that at the end of the cycle it is incorporated in the new delivered product;
- **large coverage tests** — as pushed by the `TDD`, usually in the context of another agile methodology, which requires that all developed code should have unitary tests, that are periodically run through `CI` tools;
- **automatizing tasks** — as most methodologies require a fast development pace, any repetitive task should be automated, as seen above from testing, to packaging; some even propose "*continuous delivery*" where each artifact passing all tests is automatically deployed into a staging environment.

Thus the takeaways for us could be summarized as: • we shouldn't hamper the developer's productivity, streamlining as much as possible the involved tasks such as packaging, deployment or configuration; • because of the test driven approach, we should provide the developer with adequate tools to run integration tests on his components, if possible in an environment as close as possible to the real platform; • automation is key for all possible possible tasks, allowing the developer to focus less on our platform and more on his business domain.

2.4 Developer roles

Maybe one of the most important factors driving the adoption of a particular technology is the human one, as we shouldn't forget that in the end the first users of any technology are the developers which build applications based on it, and then those that manage them.[1] As any non-trivial and long-running application — be it a web-based application, classical desktop client connected to a back office server, or everything in between — has a complex life cycle, involving persons with a certain set of skills, playing various roles, in the current section we try to identify which are these roles, and how they have changed in the last few years, as the skills have migrated from one role to another, especially in the light of the previously presented methodologies and in the context of cloud computing.

We try this in the hope to better adapt our platform to the needs of various individuals interacting with it, thus increasing its adoption likelihood.

[1]We acknowledge that the final users are those that give any real value to the application, but the technologies that lay at its foundation are the ones enabling such an opportunity.

Classical roles Although not the case anymore, the classical roles, which follow closely the waterfall model's stages, currently being found mostly in legacy enterprise environments, are important at least as a starting point for further discussions, as their basic responsibilities have remained the same, at most migrating from a new role to another:

- **analyst** — the one person that understands both the client's business, but is also able to translate it into concrete and clear requirements written with from a software perspective;
- **architect** — which based on the requirements should identify the best suited technologies and conceptually mix-and-match them into a puzzle where all the pieces fit together; it is without saying that such a person must know by heart all the pieces, how they work, how they fit together, and non the least, what their limitations are;
- **designer** — although sometimes it is the same person as the architect, he must take each of the parts previously identified, and in isolation decide how their implementation should be best proceeded; this implies using various modeling techniques, such as `UML`, to clearly state how his intentions should be put in practice; he usually also decides which programming languages should be used and which components should be reused, customized or written from scratch;
- **developer** — most likely a team thereof, where each member takes one part and, by obeying the designer's choice expressed through the various models and diagrams, he materializes it by writing the needed code; and then building and packaging it into a software artifact;
- **tester or `QA`** — takes the software artifact built by the developer, and closely following the analyst's requirements he lays down a test plan, and thoroughly follows it through certifying that the product matches what it is supposed to do; of course if any problems are encountered he should file an issue back to the developer, and preferably if he has the technical skills and understands the code, also an initial assessment of the probable cause;
- **administrator** — finally once the product is ready to be put in production, it is his job to closely follow the documentation provided with the software product, and once installed monitor and manage the running instance.

We must add that especially in the enterprise environment, involving complex software products, these roles are split over different teams, if not even outsourced, most of the time the analyst or designer in one team completely separated by the developers and testers; certainly the `OaM` team is totally disjoint and usually composed of the beneficiaries employees.

But sometimes, for example as described in the `J2EE` tutorial [51], these distinctions are greatly exacerbated, and roles are split following fine border lines, thus what we have summarized as the developer role, it is described in that `J2EE` tutorial as four different ones, "*enterprise bean developer*", "*web component developer*", "*application client developer*" and "*application assembler*". We can not say if this is needed or not, but it makes a good example for contrasting in the following sections. ◇

The DevOps movement In the last decade or so, a lot has changed regarding the developer's responsibilities, especially, as described in §2.3 "Development methodologies", with the adoption of the agile methodologies, driven in part by budget reductions or market constraints, and part because `IT` became a mainstream profession. Therefore the developer started gradually playing more roles, initially that of the tester, then that of the designer or even analyst, this being even more so in the case of startups.

But around 2007, maybe closely related with the boom of the cloud computing era, and coupled with the explosion of development opportunities in the emergent mobile device and especially smart phone markets, the developer started gaining also the role of operations, thus a new trend emerging, that of DevOps. The idea was pretty simple, because for most emerging startups there were only a few people with lots of different tasks to accomplish, they couldn't afford to have a narrow specialization on either marketing, development, or operations. Therefore those technically inclined individuals which were usually the developers also managed their applications into production, thus merging the developer role, which already included design and testing, with that of operations.

Of course this has also solved one of the biggest problems with the classical approach, that of "*eating your own dog food*". More exactly even when a product was used inside the same company that built it, once the product was over and ready to be shipped out, a completely new team had to install and manage it. But of course as nothing in this world is perfect, the operations team had the time of their life in completing these tasks, and unfortunately they couldn't congratulate the developers about their top notch work, because most of the time the development team has moved to new adventures, its place being taken by a junior-grade one. Therefore, because now the developer has also to manage the application he has created, there is less impedance mismatch between the two roles, and the design and implementation is more sensitive to the issues related with running the system. A similar position, not related with the DevOps movement, was also taken by Amazon, when they migrated all their internal systems to a `SOA`-based architecture, and imposed that the same team developing a service should be the one managing it, a move which in retrospective proved quite efficient.

Concluding, this has led to a few things: • first of all better support for running the product in production, with shorter times for issue resolution, and a better overall quality, that leads to user confidence; • better designed products, more suitable for integration with other existing infrastructure and platforms; • and maybe the most important of them all, automatizing most tasks, fueled by the developer's professional nature, scripting away all these non-essential tasks, wanting to lose as little time as possible.

All this is very important for us, as a platform's implementers, because we need to acknowledge that in such a context, not only should our solution appropriately solve the problems on both sides of development and operations, but that the one managing the application is the same developer, thus requiring us to give him proper tools for such a task. By proper tools we mean not only `CLI`s or `UI`s, but also exposing actions as `API`s matched by software libraries enabling the developers to build their own tools. ◇

The NoOps movement Related with DevOps, seen as an evolution thereof, and fueled by the prevalence of `PaaS` offerings, another even more extreme trend has emerged, that of NoOps, which implies the complete elimination of `OaM` tasks from a startup's responsibilities, and delegating them to the `PaaS` provider. And all this, to a certain extent, is already accomplished by the auto-scaling features offered by platforms such as App Engine.

Because this is a very fresh idea, only time will tell if it is a good one, or even if it is completely possible. Certainly we should take note of it, and try to make our platform imply the least amount of management; but unfortunately we can't eliminate it completely, as in our case the developer gets a software artifact and not a hosted service, thus he is the one having to manage his platform instance. ◇

2.5 Target applications

As stated in the §1 "Introduction", the current trends, for quite some years actually, for applications, and especially for those needing heavy interaction with final users, has shifted from locally installed ones to so called web-based ones — over the time they have been called either Web 2.0 or `RIA` depending on what frameworks or platforms they have been built upon. The main factors that drove such a shift were: • quick user access, as the he doesn't need to install anything, all that is needed is either already installed like the web-browser, or it will be automatically downloaded on request; • ease of `OaM` tasks, as the operators, or even the developers themselves as pushed by DevOps movement, can upgrade the code base without taking the entire application down, or can easily monitor all aspects of the application run-time, from performance metrics to logs.

Therefore in the current work the focus is on these kind of applications, but it does not mean that the other kinds can not benefit from what is proposed herein. For example as the `SOA` movement is quite mature now, and many applications, with or without locally installed `UI`s, consume countless web-service endpoints `WSDL`-based over `SOAP`, pointing to a centralized infrastructure, the only difference between them and those targeted by the current work being how the final user interacts with the applications. We can also observe that even web applications rely on hidden web-services, though this time mostly `REST`-based, to fulfill user's requests.

We need thus, before moving to concrete case studies as provided in §3.1 "Application case studies", to understand their general structuring, architecture, used technologies and hosting, but with a glimpse towards their history as it helps us understand the surpassed limitations, and because most aspects haven't changed too much in essence.

2.5.1 Generic layers

In general applications, regardless of their domain, complexity, backing programming language or technology, follow a multitier approach composed of the following described layers, although not necessarily completely stacked.

Presentation layer The most visible of them all, generating and managing what the user sees and how it interacts with the whole system, which has been the focus of a lot of frameworks in almost any conceivable programming language, thus it can be seen as a very mature layer. As examples we can enumerate the most elaborate solutions: `JSF`, as a modern enhancement, or better said replacement, of the old `JSP`, although not enjoying the initial envisioned success; `GWT` powering Google's most prominent applications, like GMail; `ERB` as part of the `ROR` framework which triggered a revolution in the web development world; and countless other templating engines big or small. ◇

Domain layer Being entrusted with the business logic of the application, deciding "*what*", "*how*", and "*by whom*" can be done in terms of application entities. In contrast with the previous layer, here the developer has to do almost everything from the ground up, as each application having its own specific requirements and rules, of course there are specialized libraries which solve parts of the problem like authentication and authorization or various standard algorithms, but the developer has to adapt and glue them together among other things. Although usually the developer writes code in conventional high level programming languages like Java, C# or Python, there are initiatives that tried to allow business specialists to describe the rules, like `BPEL` or `BPMN` which are highly coupled with `SOA` and `WSDL` web-services. ◇

Data layer Tasked with storing and indexing all the data needed to fulfill the user requests. The classical definition of this layer relates to `RDBMS` and `SQL`, but in later years the `NoSQL` databases have emerged as suitable replacements in various situations; moreover although the classical definition includes here only the database engine, `ORM` libraries have started replacing `SQL` as the primary tool to interact with the data, as such we include these libraries at this layer, prominent examples are Hibernate for Java or .Net, or DataNucleus only for Java, and countless other libraries for most of the programming languages, especially those in the scripting category like Python or Ruby. ◇

Analytic layer Although part of the domain logic, in recent years this activity has attracted much attention as more and more data started piling up, and the requested reports increased in complexity; the best example is the map-reduce technique introduced by Google, which had many implementations, Hadoop standing out as the most mature and feature-full. This direction has led to other approaches as moving the code closer to the data, even inside the database engine as proposed by another European FP7 research project, Vision, or tightly coupled with the data as proposed by the BOOM project; ◇

Background processing layer Also informally called `cron`-jobs, as a resemblance with the way in which maintenance tasks are run in the UNIX world through the `cron` tool, most on-line applications, especially those needing to consume large data pushed by the user, don't process all the inputs immediately, but instead store the data in a staging area, give back a quick answer to the user, and process the data asynchronously, thus increasing responsivity and optimizing resource usage. As this activity became important only as Web 2.0 applications got more complex, there aren't many mature de facto frameworks out there, each application usually building its own solution on existing building blocks, mostly message queues, distributed file-systems, or distributed databases. ◇

We can observe that the first three layers, presentation, domain, and data, are the classical ones found in many applications, while the rest could be merged into one of the first three, but they are kept distinct as their importance increased more and more over the time.

In conclusion to what has been presented so far we can see that the general architecture hasn't changed much from the classical approach, as was hinted at the beginning of the section, instead new libraries and frameworks have been created to aid development, and as requirements shifted, some activities have gained importance.

2.5.2 Generic architecture

After having presented the generic layers of today's applications, next we try to present how these are mapped onto running machines, and highlighting the transformations, and their causal factors, that have occurred with time. We underline once more that the focus is mostly on web applications, as being the most in-use today, but because of the close similarity with other types, it enables us to easily transfer the knowledge and apply the same techniques devised later in the current work.

The architecture evolution As said, to better understand the context, we start with a short history from the early days of web applications when there was only one physical machine, "*the server*", dedicated to the entire application. Here we had crowded both the HTTP server, CGI scripts or binaries, and a single database server. But as the demand grew, and this happened naturally as more people gained access to the Internet, one single machine wasn't able to cope with the load, as such it was split into two distinct machines: • one hosting the web server and CGI, called "*the web server*"; • and one housing only the database, naturally called "*the database server*". Some big applications actually had multiple web servers, each serving a different role, like user interface, reporting, or static data, but still only one server per role, including the database.

Again as user demand continued to increase these web servers started crawling under load, and the infrastructure suffered another transformation meant to increase the application scalability — although at that time it was called "*load balancing*", the term "*scalability*" emerging once with the appearance of cloud computing. The solution, widely in use today, was to clone each web server into a fixed number of identical machines serving the same role, and user requests being redirected, usually in a round robin fashion, to one of these clones. The redirection mechanism varied from simple DNS manipulation, HTTP reverse proxies, generic TCP load balancers, to complicated network layer solutions, or even hardware load balancers. But still the presentation layer and the domain layer were collocated on the same machine, and the database was still hosted on one single machine — we don't take into account stand-by replicas that allowed fault-tolerance.

The next natural step was to separate the two layers collocated in the same web server onto their own machines, but usually the code didn't suffer major modifications, as the domain layer instead of directly calling the domain procedures, it called proxies which through RPC forwarded requests over the network to the domain layer code. But as the SOA movement gained traction the binary RPC was replaced with web-services, mostly WSDL. Therefore the new challenge was how to bind the presentation layer to the domain layer, and the solutions again ranged from DNS, proxies, networking, to service repositories like UDDI.

As seen so far the two upper layers scaled horizontally by increasing the allocated machines, but keeping constant their performance — by constant we mean as time passed new and more powerful hardware was used, but at the same relative cost, or performance / cost ratio — but the data layer scaled vertically by replacing one weaker machine with an increasing powerful and more expensive one. The consequence was that the new bottleneck became the data-layer, thus itself needing a horizontal scalability, which was achieved by various tricks — we call them tricks because the techniques were applied outside of the database, and into the applications themselves, thus changing not how the databases used to work, but in how they were used — such as introducing read-only replicas, machines which were used exclusively for reading, taking into account the possibility of stale data; or data partitioning, "*sharding*", which implied having multiple read-write databases, each having a disjoint part of the entire data. Although these tricks solved, at least in part, the problem they have increased the complexity of the overall applications.

From a networking perspective usually only the load balancers are exposed directly to the Internet, being part of two networks: public and private, meanwhile the other machines are only part of the private network. Of course depending on the complexity and security requirements of the application, each group of servers, having a well specified role, can be isolated in their own private network, and strictly controlled firewalls are enabled on all machines, exposing only the minimum attack surface.

Thus concluding the big picture so far consists of a small number of machines serving the role of load balancers, which redirected the requests, usually HTTP, to a dedicated set of presentation servers, sometimes one set per role for example user interface or reporting, that delegated the actual work, usually through RPC or web-services, to a third set of identical machines, again sometimes clustered by role, meant to validate and process the data or execute complicated procedures, but which, based on a complicated scheme of data partitioning, relied on a cluster of machines dedicated to database servers; and to all this there were added caching systems to ease even more the load on the application servers. ◇

And in latter years, as each layer was better analyzed, and new and specialized tools or systems appeared, we add to the previous picture: • reliable asynchronous messaging systems like `JMS` or `AMQP`; • various specialized database servers like `LDAP`, key-value stores or columnar databases; • various specialized reusable sub-systems for authentication, email.

But we leave the discussion about modern architectures for a later section, §3.1 "Application case studies", mostly because they have not changed much from the last picture we have described, instead the effort went into replacing one technological product with a better one, the principles remaining the same. Moreover there is a limit up to which a prescribed architectural pattern can be applied, and as we shall see in the mentioned section, each concrete application having its own specific architecture.

2.5.3 "*The right mix*"

Before moving on we should also look at some developments that took place in parallel with the architectural changes.

For example looking at the `ROR` framework which starting from 2005 took off and is currently one of the leading web frameworks, influencing or sparking other such projects like the `JVM` based Grails, the Python based Django, and a couple others. The important point here is the fact that even though it didn't have the amount of publicity or corporate backing as other competing technologies like `JSF`, a successor of the established `JSP`, or even `ASP`, it did surpass them in popularity, as it had the "*right mix*".

This "*right mix*", which is also related with the changes in development roles and methodologies earlier presented, included among multiple other things, the following principles that we should take into account for our platform:

- **convention over configuration** — by taking common sense developer's expectations and translating them into facts, like for example naming conventions for classes, source code file-system layout, `URL` mapping over controllers, or relational entities naming for the built in `ORM`;
- **built in coherent features** — it provided the developer with an integrated framework, providing a good built in solution from an `ORM` to a template engine, which most importantly fit together nicely;
- and although not part of the `ROR` framework, it came bundled with a simple dependency management system and deployment tools.

In essence none of these features would have the same success in isolation — as for example in the Java world we have many alternatives for any of them although not quite as simple or straight forward — but all of them provided as a single coherent package was the key to success.

Therefore in our attempt to implement a useful platform suitable for the kind of applications described above, we need to carefully choose the right combination of features, and their degree of flexibility, as too much complexity doesn't always pay off.

2.6 Miscellaneous remarks

Distributed applications Regarding the concept of "*distributed applications*" already used in the introduction and more so in the following chapters, we want to clarify its meaning in the scope of the current work without getting to deep into the details.

Thus by this term we refer mostly to applications which are split into orthogonal modules, or components as we shall name them, each one having a specific responsibility, and which by collaborating, mainly through choreography than through orchestration, fulfill the requirements of the whole application they are part of. Thus the "*distributed*" term refers to the network based distribution and communication aspects of an application, than to the concepts from the academic fields of distributed algorithms or distributed systems.

But this does not preclude that such, more formal, concepts and algorithms are not embedded in the resulting platform, on the contrary most solutions presented hereafter are heavily based on such theoretical and technical solutions of such nature; it is just that the focus of the current work is more on the practical or engineering aspects than on the formal ones. ◇

Fallacies of distributed computing As a prologue to the previous sections where we have spoken mostly about distributed applications, we must recollect what has been described in [29], so that we don't slowly slip into an ideal world, and based on false assertions, be forced to start over. This is especially true as all cloud providers' marketing leaflets give so many promises, that our common sense is "*clouded*".[2]

> The network is reliable.
> Latency is zero.
> Bandwidth is infinite.
> The network is secure.
> Topology doesn't change.
> There is one administrator.
> Transport cost is zero.
> The network is homogeneous. (see [29])

In conclusion we can also add Joe Armstrong's words of wisdom:

> To make a fault-tolerant system you need at least **two** computers. (see [16])

◇

[2]Unfortunately many cloud-related projects, especially from the academia, have fallen pray to these assumptions.

Chapter 3

Case studies

After having provided in §2 “IT ecosystem” the overall technical and economical landscape of the problem we intend to solve, that is, as stated in §1 “Introduction”, a software platform enabling startups to create applications in a cloud computing environment, we continue by analyzing some concrete examples of both the targeted applications, but also similar commercial or open source products.

Before describing our case studies, we must mention the difficulty of getting our hands on enough detailed blueprints of such instances. In the case of commercial applications we admit that these schematics are trade secrets, and making them public would ease the life for the competition, but we are grateful for any piece of information they have opened. On the other side in the case of open source applications, like Reddit, the developers have not invested much effort in the formal documentation, but instead they blogged here and there about various aspects, thus making our task to search and put the pieces back together. Fortunately our task was made much easier by various sites, like the InfoQ or High Scalability communities, wherein most of the references in the current chapter were found, either published or covered by the community members.

We intend to present first a birds eye view, but from a technical perspective, of some real world popular applications, even viral in the sense presented in §1 “Introduction”, like for example §3.1.1 “Reddit” or §3.1.2 “LinkedIn”, based on scarce internals’ descriptions found over the Internet, as said previously mostly from blog posts. As the third case study we present a few generic patterns, §3.1.3 “RightScale patterns”, although mostly classical in architecture they provide a bridge between the statically deployed applications to the new cloud environment.

Afterwards we continue with some existing, commercial or open source, platforms that would enable applications like the ones previously mentioned, although none of these platforms are used for such purposes, but instead, like §3.2.1 “Heroku” or §3.2.3 “App Engine”, are used by lots of startups, and one of them, §3.2.2 “Cloud Foundry” has just exited the beta status.

3.1 Application case studies

3.1.1 Reddit

About Reddit starting with 2005, provides a social news site, where users are able to submit, comment and rank various news, blogs and sites. According to [47], in December 2010 it served about 829 million page views, rising by January 2012 at 2 billion page views, and according to [35], for about 7 million registered users. As an interesting fact, initially it was written in Common Lisp and later rewritten in Python, thus being a very good example of a popular application being implemented in a non mainstream programming language. ◇

Architecture As explained in the beginning of the chapter, although Reddit has been fully open source, at `https://github.com/reddit/reddit`, architectural or technical documentation is hard to find, the current section being based mostly on their blog posts or other Internet sources: [35], [46] and [47].

We can summarize their architecture — which we have included in our work for completeness as §3.1 "Reddit high level architecture" — and their principles as:

- they employed a somewhat classical architecture where after the load balancers it comes directly the web servers, running Python code that handles both data management and page rendering;
- caching, through Memcached and MemcacheDb, is essential for providing acceptable latency;
- even though they use PostgreSQL, with master slave replication, the schema is similar with that of a key value store;
- some data, especially for the new features, is instead stored in the columnar distributed database Cassandra, while at the same time slowly migrating old code onto the new data store;
- all things that can be done asynchronously with the user's request, or in background, are done through message queues backed by RabbitMQ, or periodically triggered by `cron`.

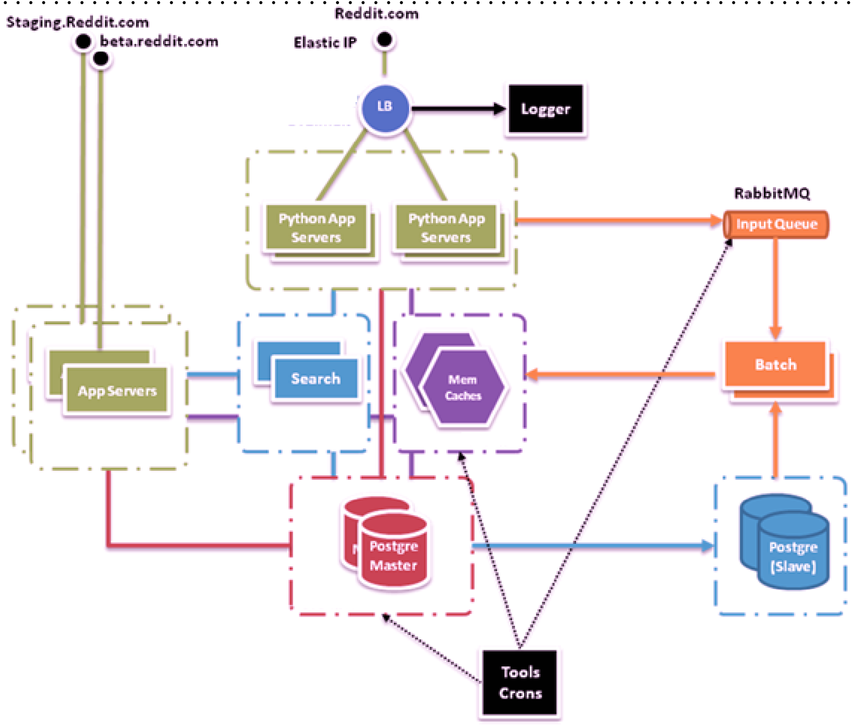

Figure 3.1: Reddit high level architecture
Borrowed from [46].

◇

Infrastructure It seems that such an architecture allows them to easily scale horizontally as data and load increases, running at the beginning of 2012 about 240 servers, according to [47].

In the beginning they have managed themselves the hardware for running the site, but starting with 2009 they have migrated to Amazon's `EC2`. As a side note in 2011 when Amazon hand a serious infrastructure meltdown, Reddit was also heavily impacted having to resort to a read-only version of the site for a short period. Moreover, according to [47], they had to renounce to `EBS` in favour of non persistent disks, because of performance reasons.

They also seem to use Amazon's `S3` for delivering static content. Unfortunately more details about how they use the cloud based infrastructure is currently unavailable. ◇

3.1.2 LinkedIn

About Starting with 2003, LinkedIn is a professional networking site, allowing persons to "*connect*" with current or former colleagues, to obtain professional recommendations from employers, join in virtual groups based on their interests, and lately it has become an indispensable tool for recruiting. Therefore we don't even have to mention how valuable is the collected data for profiling or commercial usage[1], consisting in 2008 about 22 million members, according to [38], and serving about 40 million page views each day. ◇

Architecture As in the previous case of Reddit the information is even more scarce, only two presentations, [38] and [39], giving some hints about their architecture, thus we can only try our best to guess the actual solutions and involved technologies. What is for certain is that they build almost entirely on Java and related technologies.

We can summarize as follows the information we could gather about their architecture — which is also depicted in the figure §3.2 "LinkedIn high level architecture" that we have borrowed from them:

- a `SOA` architecture plays a central role in their architecture;
- caching definitely used, but with customized graph-aware caches;
- `RDBMS` are still the norm, by using either Oracle or MySQL;
- at the same time a `NoSQL` store, Voldemort, is in use which was created by them;
- asynchronous workflows are based on message queues backed by `JMS`;
- a custom `ESB`, called Databus enables real time event distribution in a publish-subscribe pattern;
- map-reduce through Hadoop is also employed for analytic tasks.

As a side note, they have also mentioned the traps encountered when developing distributed applications, as we have reminded ourselves in §2.6 "Fallacies of distributed computing".

Another insight is given by the figure §3.3 "LinkedIn message delivery workflow", also borrowed from them, which depicts an asynchronous workflow based on a mix of `JMS` and web services.

Although not architecture related, as anticipated in §2.3 "Development methodologies", it seems that integration testing automation is an important aspect in their workflow. ◇

[1]Not to mention the issues caused by the security breach in mid 2012 when LinkedIn lost about 6 million unsalted passwords.

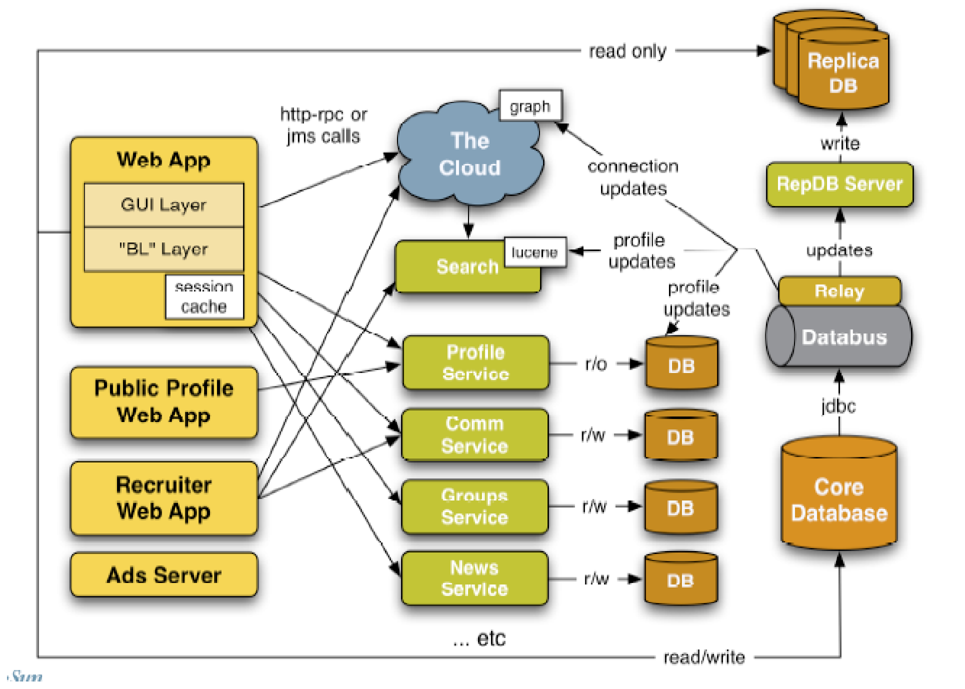

Figure 3.2: LinkedIn high level architecture
Borrowed from [38].

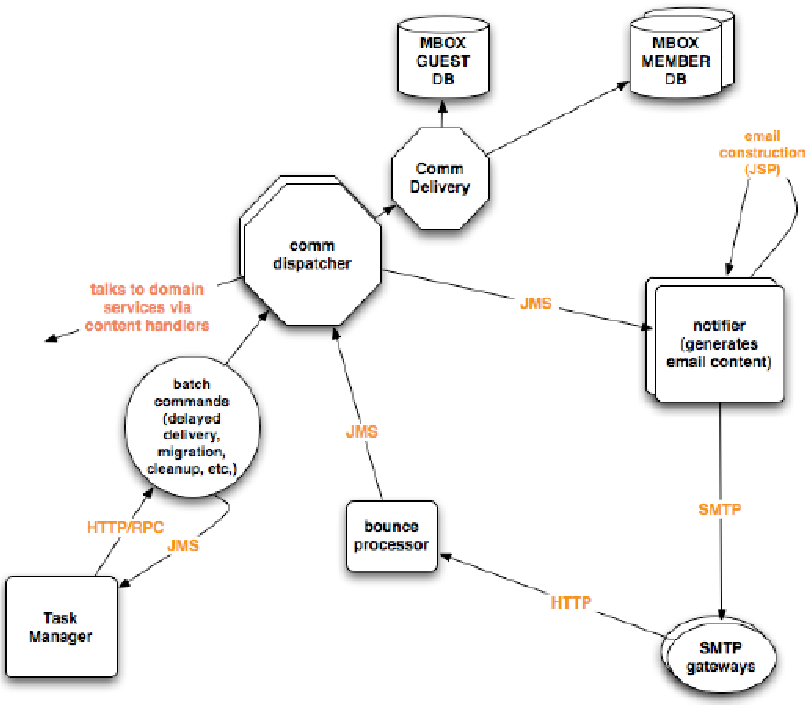

Figure 3.3: LinkedIn message delivery workflow
Borrowed from [39].

Infrastructure If the architecture related information was limited, about the infrastructure all we have are hints from the above sources, that is they use Solaris based machines, and only for caching they have about 40 servers. Unfortunately no further information has leaked out so far. ◇

3.1.3 RightScale patterns

RightScale is not an application, but instead it is a `PaaS` provider built on top of Amazon's `AWS`, providing support for hosting classical applications, in the sens described in §2.5 "Target applications", together with monitoring and auto scaling features.

Our interest is towards their very good guide [50], providing useful insight on how to architecture your application so that it can benefit from the cloud promises. The two main use cases that are particularly important for our platform, as depicted by the two figures §3.4 "RightScale scalable multi tier architecture" and §3.5 "RightScale scalable queue based architecture" that we have borrowed from them, are the 3 tier web application, and the extension with background jobs.

The main points mandatory for the success of a scalable application, thus mandatory for us to offer in our platform, are summarized below:

- `DNS` is the first layer of load balancing, splitting traffic evenly between the actual load balancers;
- load balancers are mandatory, but they could be deployed either on independent machines, or on the same ones where the frontend runs;
- the application logic must be split in independent layers, in the current case we have only three: the frontend handling user input and rendering, the backend handling most likely the entire business logic, and the background tasks;
- although not immediately required, the data layer must be scalable, thus we should either go directly for distributed `NoSQL` databases, or if we must remain with `RDBMS` then we should use master slave replication schemes and make our code use the correct target for reads or writes.

3.2 Platform case studies

3.2.1 Heroku

About According to their first blog post [33] dating from October 2007, Heroku started as a Ruby only, mostly `ROR`, `PaaS` allowing developers to easily deploy and control such applications.

This provider is of particular interest for us, not only because it had one of the earliest offerings not backed by big companies, but also because it set a trend for future developments in the `PaaS` domain — sufficient to say that most current platforms, as of 2012, are very similar, if not clones — coming up with a simple and coherent scheme to handle issues ranging from development, configuration, deployment, and management, which has slowly distilled into [52].

As such we shall spend a little more time describing their platform, applying some of these concepts to other case studies or referring back to them from later chapters, not only because they have cared enough to prepare such clear and highly qualitative documentation, but mostly because they are moving in the correct direction for a `PaaS` offering, a path similar to the one taken by us. ◇

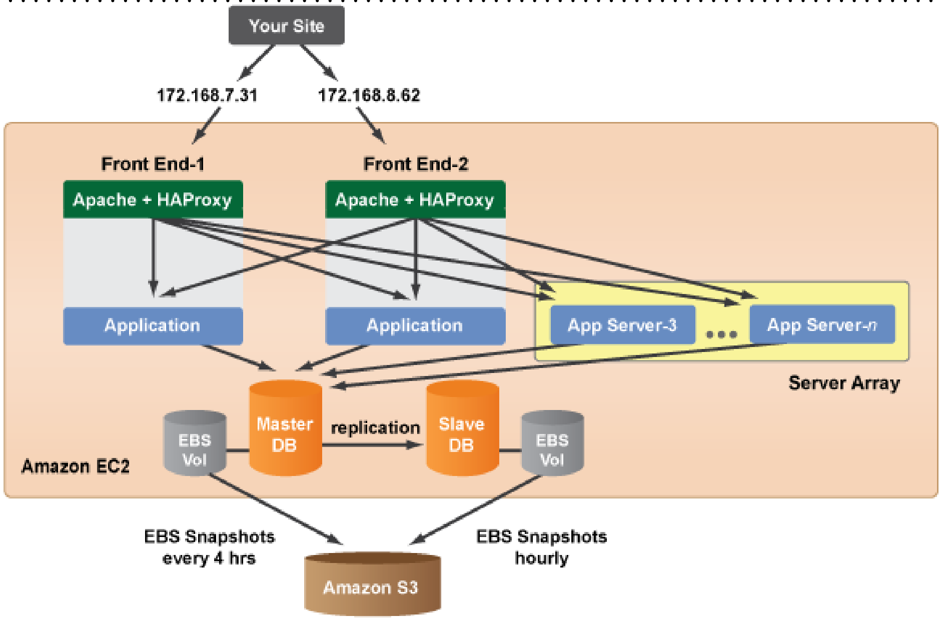

Figure 3.4: RightScale scalable multi tier architecture
Borrowed from [50].

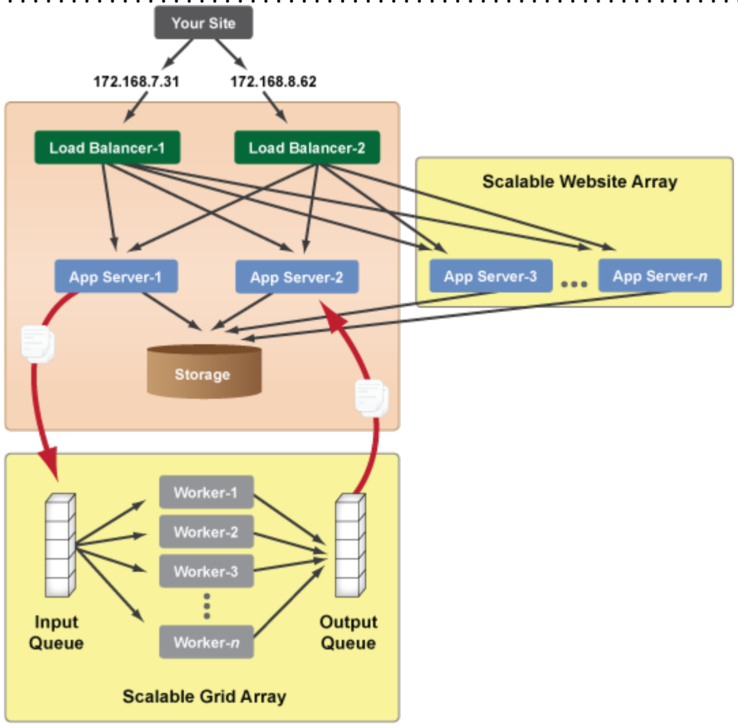

Figure 3.5: RightScale scalable queue based architecture
Borrowed from [50].

Concepts As previously mentioned one member of the Heroku's development team has written a nice and to the point self contained document [52], clearly stating which are the main concepts used within this platform, and proposing to others to use the same:

- **"*application*"** — any self sufficient software product, in line with the platform's requirements and restrictions, composed of services and consuming resources;
- **"*service*"** — the main building block of an application, embodied as an isolated operating system process, or a group thereof, implementing the application's logic, by collaborating with other services part of the same application, or consuming resources; this is usually a web-service, or a background worker;
- **"*resource*"** — any external network accessible service that is used by the application's services; for example a database, or other external web-services, maybe part of different applications hosted on the same platform;
- **"*code base*"** — the source code and other static files, needed to compile, package and run the application's services;
- **"*repository*"** — a network accessible service exporting the code base; usually a version control system, in Heroku's case it is a Git repository.

We must mention that although the terminology is similar with our own, proposed in §4.2 "Concepts", the mapping is not always one to one, the main differences being clearly stated in that section.

Although not present in their document previously cited, but very important in the platform's economy, there is the concept of "*add-ons*", that is hosted services that the developer buys in a pay as you go model, which play the role of resources. Also the packaged applications are called "*slugs*". ◇

Principles Continuing to extract the useful information from [52], we present their main principles:

- **a single code base** — all the application code should live in the same repository, and if it does not fit it should be split into multiple applications; it is encouraged however to have the application run as a collection of related services all coming from the same code-base;
- **explicit dependencies** — the developer should not assume anything about the available software packages, except those described in the platform's documentation, and should explicitly detail all applicable dependencies with their exact version;
- **explicit configuration** — all runtime parameters that should, or could, be tweaked don't belong into the source code, especially endpoints for external resources like databases, or credentials for `AWS`; at best there could be some default values hard coded like cache size or timeouts;
- **external resources** — anything that is not part of the application should be treated like an external resource and configured or used accordingly; this includes databases, web-services, including those developed by the same team;
- **separation of application phases** — which is split in a number of non-overlapping sequential phases like development, packaging, and execution;
- **concurrency** — if multiple instances of the same service are to be run at the same time, then they must do so without direct communication like files or sockets, but only through shared resources like databases, web-services, or message brokers;
- **disposability** — all services should be stateless and all non-temporary data should be stored inside resources;

- although not having a concrete term, another principle requires that services should not assume any "*sticky*" behaviour from the load balancers, that is requests originating from the same user are likely to be forwarded to different service instances.

It is obvious that some these principles have already been seen in the previous section §3.1 "Application case studies", especially concertized in the applications' architecture. ◇

Implementations During the five years since inception, Heroku went through some serious changes, gradually switching three "*stacks*", following an Ubuntu-like release naming scheme, and as described in [34] we have:

- **Argent Aspen** — the initial release, circa 2009, of their platform supporting only Ruby 1.8 applications, based on Debian 4.0; this first version was completely centered around Ruby, with the internal architecture geared towards this technology;
- **Badious Bamboo** — the second release, circa 2010, still Ruby only but offering two environments, Ruby 1.8 or 1.9, based on Debian 5.0; it was an enhancement of the previous version, adding alternative Ruby interpreter implementations and versions;
- **Celadon Cedar** — the current release, as of 2012, switching from Debian to Ubuntu as the `OS` distribution, and making other dramatic changes in their implementation.

We are interested especially in the latest version, Cedar, because it is closer to our view of a platform, but also because it introduced a lot of new features, currently not found in other solutions:

- **polyglot platform** — as described in [32], it was one of the first `PaaS` that truly offered a large variety of programming languages, from their initial Ruby, to Python or NodeJS, and `JVM` based Scala or Clojure; moreover it also supports arbitrary executables either as binary or scripts, the only requirement being the compatibility with the host `OS` and not needing special permissions; unfortunately it is not straightforward to mix multiple programming languages in the same application;
- **advanced `HTTP` routing** — until this release the developer had at his disposal only the request-reply pattern with known, or small, payload size; but with the latest routing code, which it seems is implemented in Erlang, the developer can now use more up to date techniques like long pooling or chunked encoding; unfortunately there is no support for WebSockets yet;
- **advanced process models** — allowing the developer to describe arbitrary process hierarchies to be run as services, and supporting "*one off*" commands for administrative purposes.

As such in this regard Heroku is indeed an advanced platform raising the bar for its competitors, one of them being presented in the next section, §3.2.2 "Cloud Foundry". ◇

Architecture Unfortunately we can't say much about Heroku's internal architecture, as we have only what is available on their marketing web-site, and what can be deduced from their usage documentation.

In a few words it seems that: • once the developer has pushed his code through Git in their hosted repository, a packaged container image, called a "*slug*", is built from the code together with the declared dependencies, and if successful a new release of the application is issued; • then the developer can request that one or more instances of a particular service to be started; • some of these services play the role of web-services or web-applications, accessible either at an `URL` with a `FQDN` owned by Heroku or delegated to them by the developer, therefore when the final user accesses any `URL` mapped to this application, the routing layer finds an available instance of the service and forwards the request. ◇

Technologies First of all Heroku has provided a few of their tools and systems as open source, beside contributing to other open source projects used by them. For example their logging system Logplex, or a Paxos implementation Doozer are available on their GitHub account at `https://github.com/heroku`.

Getting back to the technologies we tried to compile a small list by searching through their documentation and blog:

- **Linux** — obviously, as they use either the Debian or Ubuntu distributions;
- `LXC` — for isolating the services' processes from one another and also for constraining resource utilization; it is unclear what other Linux features are used in conjunction to this one;
- **Erlang** — it seems to play an important role for them, because at least their logging system mentioned earlier, Logplex, but also their Cedar routing layer are implemented in this programming language;
- **Ruby** — another obvious choice, because Heroku is first and foremost a Ruby based platform; certainly the developer tools are built with Ruby, but so could other internal systems;
- **Go** — another non-mainstream programming language, which seems to slowly find its way into their tool belt, as for example Doozer is implemented in it;
- **Git** — used to upload the code on the platform, benefiting from a lot of already made tools, and nonetheless for its efficiency; this code deployment mechanism seems to have caught on, as more and more platforms are adopting it.

Unfortunately we can not find more about other technologies, or even how the current ones are used, because the internals of most Heroku's systems are not open. ◇

Infrastructure Regarding the infrastructure they are very open in declaring that Heroku is a `PaaS` built on top of an `IaaS`, that is Amazon's `EC2`, thus one of the few examples fully embracing the cloud model. There are advantages also on the developer's side because he is able to consume other `AWS` services, or third party ones hosted on `EC2`, without paying an extra cost for communication, and at the same time keeping the latency to a minimum. Moreover the developer could choose to build part of his application with Heroku, and the other part directly on top of `EC2`. Unfortunately there are no other details related with their infrastructure. ◇

Critique Although the Heroku team has done a very good job, and their product is very similar to what we would like to achieve, we have a few observations, that we hope to take into account ourselves:

- **real polyglot platform** — as seen in the previous paragraphs, although Heroku allows the developer to choose from a wide range of programming languages, it is actually impractical to mix multiple languages in the same application without loosing some of the platforms features like dependency management and such;
- **live upgrades** — one of the biggest issues with running an in production application is upgrading it; and Heroku does give a great help to the developer by providing a release history, thus enabling him to rollback in case of a major issue in the newly deployed code; but unfortunately during the upgrade all the application's services are stopped, and from the outside it looks like the application went down; we acknowledge that this is not a particularly easy task, especially when there are schema migrations and such, but at least in simple cases of changing the code it should be made possible;
- **fine grained routing** — in case of small applications, where all the web code fits inside the same service, everything works nicely with the routing layer; but if one would like to split his code into multiple services, each handling a disjoint set of `URL`s this becomes impossible in the current version, as there must be only one type of service handling `HTTP` requests; of course the developer could implement himself a service that acts as another router for some background services but this would defeat the purpose of a platform;
- **console access** — because they use `LXC` to handle the service execution, it would have been possible to allow the developer to open a remote shell executing within the same container, thus helping him debug or better inspect a running service; unfortunately this is not the case and once a service is started the only way in which the developer can inspect it is through the log;
- **local testing environment** — although they suggest that the developer creates a separate application for testing or integration purposes, it would have been much more easy for the developer to have a local virtual machine behaving in a similar way as the real platform;
- **non `HTTP` protocols** — although technically this would be quite hard to achieve, because in order to support arbitrary protocols one would need a dedicated `IP` address for each different application and each different protocol, thus for one server multiple network addresses should have been attached, which could be impossible to obtain from `AWS`; but they could have at least exposed a dynamically selected port for each application and protocol pair, this way one `IP` address could have been reused for about 60 thousand instances, more than enough to saturate one server.

Concluding we are certain that some of these issues will be solved in the near future together with other expected features, making Heroku one of the leaders in `PaaS` providers. ◇

3.2.2 Cloud Foundry

Although Cloud Foundry is available as a fully open source project, hosted on GitHub at `https://github.com/cloudfoundry`, their internal architecture documentation is quite lacking, and for detailed information we have to shovel through their Ruby source code.

Fortunately we have found some useful information on their and a few other enthusiasts' blogs, and we were able to create an impression about their overall design. As such we came to the conclusion that although the terminology and the implementation technologies differ from that of Heroku's, their solution must be quite similar, and as such instead of thoroughly describing Cloud Foundry's choices, we shall make a parallel with Heroku's, thus eliding most details.

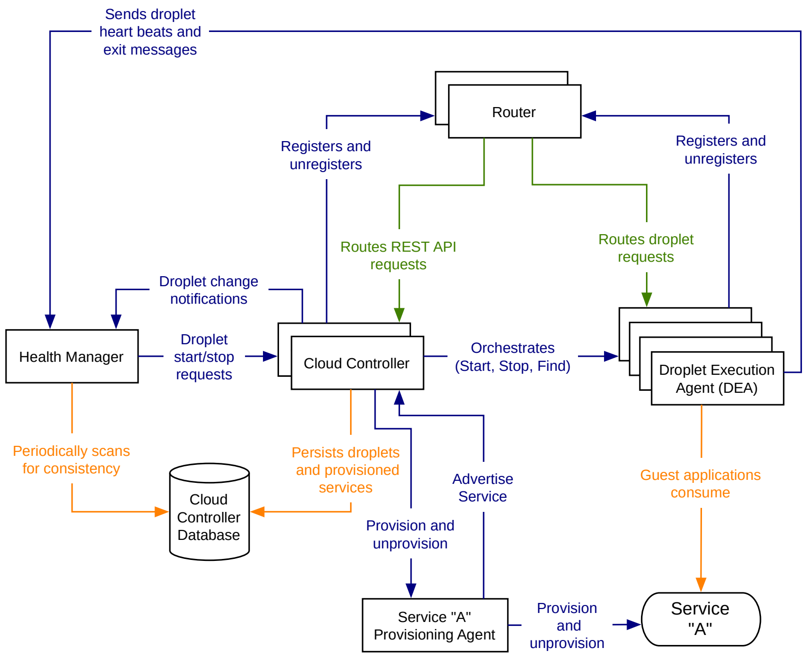
Sends droplet heart beats and exit messages
Router
Registers and unregisters
Registers and unregisters
Routes REST API requests
Routes droplet requests
Droplet change notifications
Health Manager
Droplet start/stop requests
Cloud Controller
Orchestrates (Start, Stop, Find)
Droplet Execution Agent (DEA)
Periodically scans for consistency
Cloud Controller Database
Persists droplets and provisioned services
Advertise Service
Provision and unprovision
Guest applications consume
Service "A" Provisioning Agent
Provision and unprovision
Service "A"

Figure 3.6: Cloud Foundry high level architecture
Borrowed from [26].

As seen from §3.6 "Cloud Foundry high level architecture", the overall architecture is composed of:

- **router** — which serves exactly the same purpose as Heroku's one, by accepting `HTTP` requests and forwarding them to the right handler, called in case of Cloud Foundry "*droplet*", and in Heroku's "*service*";
- **droplet execution agent** — which takes care of the developer's code packaging, deployment, execution, monitoring, logging and a few other responsibilities;
- **service provisioning agent** — that provides droplets with the needed resources, like databases, message brokers, and others, called "*services*" by Cloud Foundry, and "*add-ons*" by Heroku;
- **cloud controller** — that handles the provisioning and monitoring of virtual machines, provided by VMware's infrastructure in case of the commercial platform, or other compatible solutions; because Heroku does not provide an architectural diagram we don't have their alternative for this one;
- **health manager** — that continuously polls all the running pieces and ensures that they are alive, and if not takes needed actions; we have no Heroku alternative as in the previous case.

If we believe that Cloud Foundry's architecture is similar to that of Heroku's, we are certain about that in what concerns the features and limitations of the platform, thus all that has been expressed in §3.2.1 "Critique" applies also in this case.

Even though we have compared Cloud Foundry with Heroku, there is a definitive advantage relative to the later, that of the open source code availability, which means anyone can install, operate and even modify and resell solutions based on it. Thus in this respect Cloud Foundry is by a large margin ahead of its competition.

3.2.3 App Engine

If in the previous two cases we hand either some blog posts or video recordings from various conferences detailing the internal architecture, or being more fortunate and having in our hands the actual open source code, in case of Google's App Engine we have almost nothing leaked out, except some high level hints found in a presentation recording [37], detailing more how to develop applications for their platform.

As we could have guessed the general architecture is similar with the previous case studies, that is internally Google has a set of load balancers forwarding the `HTTP` requests to one application server, where an instance of the handler is already running or is scheduled to get started.

Thus if we can't peek inside their solution, why are we interested in their platform? Mainly because this `PaaS` solution was ground-breaking, and some could say even pushing it to the extreme, in what concerns constraints and limitations imposed to the developers. And because they still have users[2] it means that these requirements were correctly identified and haven't endangered the success of the hosted applications.

As such we only list these constraints, both in because we intend to contrast with the previous providers, but also as a proof of how much can the `PaaS` concept be pushed without losing the market[3]:

- **constrained run-time environment** — although the developer programs using Java, Python or Go, the actual libraries available to the user are very constrained, offering only a reduced set of what is available in the standard version; for example threading or even file-system write access are forbidden;
- **zero external un-intermediated contact** — applications not only are not allowed to make arbitrary network connections to the Internet, but even `HTTP` requests must be made through a sanctioned library;
- **only columnar data stores** — all data must be stored in a columnar database, most probably backed by a BigTable instance, and even though the developer has a query language resembling `SQL`, it is very limited in what concerns filters and it doesn't support joins;
- **limited request timeout** — maybe one of the features receiving the most complaints, is that one request should be handled within 60 seconds or less, the previous limit being only 30 seconds.

There are of course many other aspects common with other `PaaS` solutions that we haven't described here, like for example the stateless nature of handlers or lack of request "*stickiness*".

[2]A few days ago they have announced `GCE`, similar with `EC2`, which means that currently the market is happier with an `IaaS` approach than to a fully fledged `PaaS` one.

[3]Although not described in the current work, but the mOSAIC project offers as one of its main features a similar constrained programming environment, which is referred to as "*cloudlets*" in its documentation.

Chapter 4

Design

Finally after having reviewed in §2 "IT ecosystem" the overall technical and non-technical context, and in §3 "Case studies" real world examples of both applications and platforms, we describe in the current chapter the design of such a solution. We start, in §4.1 "Inspiration", by mentioning previous projects or technologies that greatly impacted the current work; and before continuing with the actual design, we first give, in §4.2 "Concepts", the abstract descriptions of a few concepts, frequently used in the current work, anticipating some design aspects presented in later sections. Then in order to avoid scope creep, we attempt to informally summarize, in §4.3 "Requirements", the most important requirements, thus helping us focus on a set of concrete goals, to which we shall refer back throughout the current chapter in order to be sure we have covered the intended scenarios, and to try to keep the solution as simple as possible.

Then, in §4.4 "Architecture", we give a high level overview of the architectural patterns applied in our solution, followed by the description of various entities states and transitions in §4.5 "Life-cycles".

Because, as anticipated in §1 "Contribution", the current work took shape while the author was involved in an European FP7 project, mOSAIC, the section §4.6 "Sub-systems", which describes the proposed platform, only gives a coarse grained explanation of the involved sub-systems, following in-depth details being provided in §4.7 "Component hub and controller", §4.8 "Credentials service" and §4.9 "Security", as in these areas the author was heavily involved in.

4.1 Inspiration

The previous chapters provided a good insight into either: • how the current `IT` landscape looks today or which are the general expectations and future trends (see §2 "IT ecosystem"); • how the current applications are designed to withstand the waves of users generated by technology commoditization (see §3.1 "Application case studies"); • which are the current products and solutions that enable such applications to become reality (see §3.2 "Platform case studies"). The impact of these topics on the current work can mainly be seen in the requirements for any solution that would be useful in such a context (see §4.3 "Requirements"), but also in the selection of the backing technologys (see §5.2 "Backing technologies").

Also, as the current work was mostly written during the author's involvement in the European FP7 research project mOSAIC, its turnout was to a large extent influenced by the project's context, decisions, design (see [24] or [6]), and implementation; moreover especially by the collaboration with various team members both from the academia and the industry. Although there are important concept and design overlaps between this work and the project, there are also important deviations, which we'll highlight in the following sections, especially in the overall goal and focus, design decisions (see §4 "Design").

Although the previous influence sources were heavily focused on the target applications, another important impact came from a very unexpected source, specifically the Erlang [16] programming language and its `OTP` [43] run-time environment — which also happens to be the keystone for a large part of the enabling technologies (see §5.2.5 “RabbitMQ”, §5.2.4 “Riak and Cassandra”, or §5.2.4 “CouchDB”). This is because its main principles found their way into the design and implementation of the proposed platform: • communicating stateless processes — closely related to the actor model or `CSP` — influenced the way applications are decomposed (see §4.2.5 “Component”); • supervision hierarchies and monitoring impacted the way components are managed and relate to each other (see §4.3.1 “Control”); • seamless distribution and network-transparency influenced the way in which components interact (see §4.6.1 “Component hub”).

And nonetheless, many of the current concepts, designs, prototypes, and — hopefully in not such a so far future — finalized platforms, appeared and took shape as the consequence of countless, verbal or electronic, conversations with various smart people, either current or former colleagues, computer enthusiasts, and others to which I'm in dept (see §6.3 “Acknowledgements”).

4.2 Concepts

In the current section we try to summarize some of the primary concepts used throughout the current-work. Although the scope is summarizing, in some cases, we can't refrain from an in-depth explanation, especially for those details that are keystones for understanding either the concept, its context, or the main motivation behind a certain decision — but in neither case we won't focus on technical details, as these will be described in later sections (see §4.4 “Architecture”, §4.6 “Sub-systems”, or §5 “Implementation”). Furthermore, as explained in §4.1 “Inspiration”, most of these concepts have already been described, as part of the mOSAIC FP7 project, in other published works such as [24], [6], [5], [1], or [4].

4.2.1 Developer, user, and provider

Although the generic concept of “*developer*” needs no explanation whatsoever, in the context of cloud computing and especially within the emerging DevOps movement, this position has engulfed other roles as well, as previously explained in §2.4 “Developer roles”. Thus the current work uses this term to refer to any person that has to deal with any technical and / or business aspects of a software product, regardless of any particular organization structure. Furthermore, in order to simplify the discourse, we pretend that any software artifact is analyzed, designed, implemented, tested, deployed, managed, even marketed by a single person, the “*developer*”.

Again, especially in the cloud computing context, the developer itself could be the beneficiary of a solution “*provider*”, thus he plays the role of an “*user*”. But at the same time the developed product has, in its own turn, users that benefit from the final application, thus overloading the term with different semantics depending on the point of view. Therefore in order to avoid any confusion the term “*user*” refers to the later case, that of a final beneficiary, the target of the product.

4.2.2 Application

Similar with the approach described in [33] and detailed in §3.2.1 “Heroku”, an application is a self sufficient, self contained, and automated software product that solves a particular need. But unlike Heroku's (and other similar platforms) approach, which encourages the developer to split a complex logical application into multiple platform applications, we underline the “*self-sufficient*” characteristic.

To be more specific, if we have on our hands a quite complex application, like a social networking site, or similar to those described in §3.1 “Application case studies” — for example exposing a `WUI`, an `API`, both being backed by a set of background services, all which can be decoupled and reused independently — Heroku's suggestion is to split them into multiple applications (for example one for each of the three categories) that interoperate with each-other using a well-defined private[1] `API`. If we also add the constraint that each of these should be written in possibly different programming languages, then the suggestion is elevated to a mandatory requirement for most platforms (including Heroku, Cloud Foundry, App Engine). But regardless of the technical details, and a few advantages (like development or deployment simplification), such a constraint has more negative impact on the `OaM` aspects, as we have to deploy, control, monitor and upgrade multiple distinct applications.

Moreover, in our case, the concept of "*application*" is also an umbrella for all those building blocks which stitched together constitute the final product. Anticipating what we shall describe in §4.4.1 "Layers", these blocks, to some extent, can be grouped together in a layered structure:

- **infrastructure layer** — consisting physical or virtualized hardware (see §4.2.3 "Infrastructure", §4.2.11 "Node", or §4.2.12 "Cluster"), and hosted resources (see §4.2.6 "Resource");
- **platform layer** — providing core functionalities to the application (see §4.2.4 "Platform", §4.2.8 "Service", or §4.2.9 "Component hub") or to the developer (see §4.2.13 "Tool");
- **application layer** — embodied by the developed components, that provide the specific behaviour of the final product (see §4.2.5 "Component");

As such, depending on the context, the term "*application*" relates to either: • only the components created by the developer, usually referred to when the context relates to development, building, packaging); • the entire conglomerate, including infrastructure, platform, and developed components, referred to in the context of `OaM` tasks.

4.2.3 Infrastructure

Everything hardware related like machines, network-attached like `NAS`, `SAN`, or `CDN`, obtained from a cloud provider like `AWS`, or in general not under the direct control of the developer (and managed by the platform) is regarded as "*infrastructure*". The layers main role is providing raw low-level resources like computing, storage, or communication, with explicit, usually fixed and non-negotiable, characteristics, known as `SLA` like capacity, `QoS`, or cost.[2]

In general the infrastructure elements are either: • owned by the developer (as in the form of existing data-centers), or in cloud-computing terms "*private cloud*"; • leased by the developer from external entities, again in cloud computing terms, from the "*cloud provider*" thus using "*public cloud*"; • or a mixture of the two cases called "*hybrid cloud*". But regardless, we refer to the resources' owner as the "*provider*". It is also possible for the developer to use elements from multiple providers[3], but for the purpose of the current work we neglect this aspect and treat all the infrastructure elements as provided by the same entity.[4]

[1]Although we think of such an `API` as being "*private*", in case of Heroku (and other similar platforms) we must be conscious that it is actually **very "*public*"**, exposed to the external network, in the sens that anyone from the Internet can access it, thus with serious security implications, like authenticating our processes to each-other, or providing confidentiality and integrity. This is due to the fact that there is no concept of "*linked*" applications.

[2]Many research projects assume that in the near future such characteristics could be negotiated between the provider and the developer. But unfortunately so far none of the existing providers show any sign in this direction, all having fixed offers, most of the providers not even describing them in structured format (except the case where `HTML` counts as such a data format). We must admit that `AWS` does provide some sort of bidding facility for virtual machines under the name of "*spot instances*".

[3]Again by looking in the current research topics, such a multiple provider approach is a very hot subject, found under the name of "*cloud federation*" or "*cloud brokering*". But as in the case of negotiation none of the existing providers seem to show interest in such a direction.

[4]By neglecting such a multi-tenant aspect, we could make false assumptions, as warned against in §2.6 "Fallacies of distributed computing". But as we consider such an approach impractical, or at least for the current state-of-art, we consider it a complication that can easily distract our attention from the actual foreseen target applications.

4.2.4 Platform

This one is very easy, and by applying set theory: if we take all those elements that comprise the entire application (as described above in §4.2.2 "Application"), and we subtract everything that fits under the infrastructure category (see §4.2.3 "Infrastructure"), we are left with the "*platform*". Specifically all those software modules that the developer obtains from us, and builds upon a complete solution; as noted in their respective sections, these can be either components, resources, services, but also software libraries, software packages, etc.

These usually fulfill the following roles:

- **execution** — firstly, it must provide a suitable run-time environment in which the components are able to execute — this includes isolation, dependency management, life-cycle control, or failure detection; (see §4.6.3 "Component controller" or §4.6.4 "Container controller";)
- **abstraction** — shield the developer from the low level, technical, details when interacting with raw resources or infrastructure elements;
- **support** — provide additional and useful assistance, especially in `OaM` tasks, like monitoring, or logging; but also give the developer the necessary tools to develop, bootstrap and control his application; (see §4.6.7 "Miscellaneous services";)
- **integration** — taking various general purpose products (like caches, proxies, or databases) and wrapping them accordingly, so that they are exposed to the developer as an integral part of the proposed solution; (see §4.6.7 "Protocol gateways".)

4.2.5 Component

Following a `SOA`-like[5] model we push the developer to split his application into multiple independent modules, called "*components*", each one fulfilling a single, concrete, well-defined, and with a proper granularity, role — thus respecting the `SRP` principle. These interact with other peer components, either by using the platform's provided mechanism, the "*component hub*", or by communicating directly over network channels, and even through shared network resources.

But the term "*component*", depending on the context, could have multiple related meanings, one being that of "*component types*", the logical group of all those executing modules having exactly the same role and behaviour; the second meaning, "*component instances*", being bound to one particular executing module.

Possible examples range from simple web-services, data crunching background services, databases, or esoteric gateways.

This term is somewhat similar with Heroku's concept of "*process*" [23]; "*droplet*" in case of Cloud Foundry; or "*handler*" for App Engine; except that in all these cases it refers strictly to either web-services or various background-services, thus our definition is broader, allowing the developer greater flexibility, and opening new possibilities.

Also, another important differentiating aspect, between our proposed solution and other platform providers, like those we have exemplified earlier, is related with the versioning of the source code underlying the components. In our case one component type is associated with exactly one source code version, different versions being seen by the platform as totally different component types. Therefore we give to the developer yet another degree of movement, by allowing him to

[5]We use the expression "*SOA-like*" — instead of plain "*SOA*" or "*SOA-based*" — because we want to emphasize the reference to the `SOA`-principles, like componentization, loose coupling, or interoperability, rather than the `SOA`-backing technologies, like `SOAP` or `WSDL`, although most of the times the implementation is compliant with one of these technologies.

either gradually upgrade his code, or to test new code in production[6].

A more in-depth description for this concept, and the next one, has been published in [5], as part of the mOSAIC FP7 project.

4.2.6 Resource

As previously mentioned, components — while focusing on their narrow concerns, usually business-logic related, or of an algorithmic nature — need to externalize certain tasks to other networked peers[7], some of which we call "*resources*". The most eloquent examples in this category are databases, caches, distributed-file-systems, or messaging middleware, but also more targeted solutions like email gateways, multi-media processing services, data catalogues, logging systems, or `DNS`.

Most of these are obtained directly from the infrastructure provider or third party partners — especially in the context of `PaaS` solutions, where in general this is the only viable option — and in other cases they are deployed and managed by the developers themselves, either as part of the application itself, or as standalone instances, maybe shared by multiple applications.

Because the purpose of such solutions have a wide range, and their focus goes from broad-reusability to a narrow domain applicability, there is a very thin line between components and resources — especially because in our work the implementation for some resources are components themselves. As such we apply the following heuristic in dividing components from resources:

- certainly any services provided directly by the providers themselves, or by external partners, are considered resources (and thus maybe part of the infrastructure);
- certainly any software solution that is not managed (i.e. deployed, or monitored) as part of the application is considered a resource;
- (from here onward we are left only with "*components*" that run as part of the application itself;)
- those components that are generic in nature, and don't embody any business logic, should be developed with reusability in mind, and could be considered resources (for example protocol gateways, multi-media processors);
- all the rest are considered "*components*";

This term is similar with Heroku's concept of "*backing service*" or "*add-on*"; or "*service*" in case of either Cloud Foundry or App Engine. Further details are given in §4.6 "Sub-systems", and have been previously described in [5].

[6]Incremental deployment is the custom for big enterprises like Google, Facebook or Twitter, that for each new feature deploy the new source code only on a fraction of their infrastructure, in the hope than any potential problems are identified before they impact a large number of users.

[7]We can't stress enough the importance of "*networked*" interaction between such peers, especially in the cloud computing context.

4.2.7 Management and service interface

It is important to observe than any non-trivial software product has two main interaction modes (the naming below is our own, as no largely adopted one exists):

- **the service interface** — In a word, the normal one, involving operations related to the domain model. For example in a `RDBMS` this is the case of `DML` statements (like `SELECT`, `INSERT`, `UPDATE`, or `DELETE`) and, maybe, equate to more than 99% of the interactions between a client and the server. In the case of a web-server, this would be the `HTTP` request routing to handlers; and in the case of an `AMQP` broker the message consumption and publishing.
- **the control (`OaM`) interface** — In general this is a system interface, as it relates more to the system / server internals, than to what the system actually does. Clear examples here would be the statistics gathering interfaces for resource consumption (like memory, CPU, or disk), performance counters (like requests per time interval), or other run-time information (like active operations), as exposed by the majority of servers. Other easily identifiable cases are management interfaces related to configuration, user creation, access management, again commonly exposed by servers. Then there are the border-line cases — which although relate to the domain model are rarely used, and most of the time they require elevated privileges (i.e. "*root*" or "*admin*" roles) — as `DDL` statements in the case of `RDBMS` (like `[ CREATE | ALTER ] [ DATABASE | TABLE | INDEX | USER ]`); exchange and queue declaration and binding in case of `AMQP` brokers.

Unfortunately most of the time these two major types of interfaces are exposed through the same endpoint, and using similar syntaxes, thus commingled with the natural operations, therefore sometimes this feels unnatural, and most of the times it poses real security issues. As such in the proposed platform we try to at least identify the two types of interfaces, and if possible to demultiplex them into proper channels.

4.2.8 Service

All the provided modules that are needed for either the platform or the application to run, and which in general are not seen by the developer [8], are named as "*services*". Their purpose is to:

- manage, enhance or provide core services for the operating system (like containers, firewall, watchdog, or `DNS` resolution);
- provide core platform services (like communication, deployment, or execution);
- provide auxiliary support (like monitoring, auditing, logging, or scaling);

These are usually embodied as daemon processes that can be accessed through well defined `API`s (like JSON-RPC-based), usually over local transports (like UNIX domain socket).

[8] Some services are certainly used by the developer, without him even realizing, as is the case of `DNS`, or logging. And of course some of these could even be exposed to the developer, just like resources, as components if they could be used by the developer.

4.2.9 Component hub

Although it is one of the core services in the proposed platform, the "*hub*", plays a major role and is visible to the developer especially `OaM` phases. It's similar with an `ESB` and it focuses on the management interface described above, providing the following roles:

- **interaction** — by providing logical (scoped within the application) addressing, it allows components to securely exchange messages, regardless of networking details;
- **discovery** — allowing components to organize themselves in groups, or to register themselves under well-known names, and then providing means for others to query these structures;
- **introspection** — in the sens that components can provide various state values (similar with `SNMP`), and are able to react to changes by observing various event (like component state change, failure, or group membership);
- **centralization** — by collecting various information (like monitoring or logging) from components and routing them to corresponding sinks;

4.2.10 Controller

As we shall see in later sections, especially in §4.6 "Sub-systems", the platform abounds with "*controllers*", as we have a separate controller for almost every other entity. The reason for this variety is the fact that each entity, like node or component, has its own specific life-cycle, and mixing the responsibilities together, although it would simplify the platform as seen from the outside, it needlessly couples the modules, reducing their flexibility and reusability.

In general the responsibilities of such a controller is twofold: • once to manage the entity's life-cycle, executing or delegating actions that would translate into transitions from one state no another, then monitoring the conditions required for that particular state; • and then to wait external requests, as from the developer itself or other sub-systems, validate, and translate them into queries or actions as previously described.

4.2.11 Node

In the end any software product must execute on some machines, either physical hardware, or virtualized one, which in turn needs to have an operating system and an accompanying set of system services. Each individual pair of hardware (physical or not), operating system and services, we call a "*node*". Usually such a node is obtained by installing our provided software on already owned hardware, or in case of a cloud-computing environment, a virtual machine is leased from the `IaaS` provider and instructed to boot our provided image.

We have chosen this term, because it is already customary in the distributed and parallel computing communities, and we wanted to avoid already semantically overloaded terms like "*virtual machine*", "*server*", etc.

We must remark that in most `PaaS` solutions this and the next concept are hidden from the user as he shouldn't care where his processes are run. But because the proposed solution is an open-source self-deployable system, the developer must be aware of these technical details.

4.2.12 Cluster

As previously said, the application executes on one or multiple nodes all under the control of the platform. All these nodes together are called a "*cluster*".

Furthermore this term is not only a synonym for "*a bunch of machines*", but designates a logical entity, and an administrative domain which is the subject of certain policies. For example in the case of owned hardware a cluster is usually matched by a `VLAN` for local networking and a designated router with security software installed (like firewalls, or `IPS`); and in the case of a cloud provider, depending from one to another, it is found under certain terms (like "*security groups*", or "*private cluster*"), but still serving the same roles as previously described.

At the same time it designates a network scope (or domain), termed "*cluster-local*", which implies certain expectations and constraints (like expected low latency, high bandwidth, or trusted environment).[9]

4.2.13 Tool

In order to allow the developer to build, interact with, manage or monitor his application we provide him with a set of executables (either `CLI`-, `WUI`-, or `GUI`-based), or plugins for various `IDE`'s (like Eclipse) or standard tools (like Maven). These can be run either on his local workstation, on remote machines, or even provided as remote accessible `HTTP` services.

4.2.14 Sub-system and agent

As observed from the previous sections, the application is split into a number of relatively independent modules, like services, resources, and components, the first two being provided by the platform, and the last one most likely by the developer.

If we group all the modules based on their roles, like for example all those services managing the nodes into one single group, all those related to discovery or `DNS` in another group, each resource into its own, all those components of the same component type, and so on, we obtain what we call "*sub-systems*", that is loosely coupled, but cooperating modules with orthogonal roles. But in the current work, by sub-system we refer only to those grouped modules provided by the platform, because the developer is mostly concerned with components, we'll refer to their grouping as component types.

Defined at the macro level, the sub-system is a logical entity, having well defined, but cohesive, responsibilities, that is reflected as one or multiple cooperating agents running on different nodes, but in the same cluster, thus the sub-system most likely falling under the category of distributed systems or multi agent systems. At the micro level one sub-system's agent is defined as one, or multiple related and cooperating `OS` processes, running on the same node, therefore an agent.

[9] Other such nested scopes are "*node-local*" (or "*loopback*"), "*cluster-local*", "*provider-local*" (or "*region*" / "*availability zone*" for some providers), and the "*Internet*". Anything which is node-local or cluster-local is called "*intra-cluster*" and is considered trusted; everything else should be considered as possibly hostile.

4.3 Requirements

After having described in previous sections the general use cases, and having presented the main concepts, we try in the current section to summarize the main emerging requirements. We have selected to use only plain English and present the broad aspects — instead of more established approaches such as `UML` use-cases — mainly because, in principle, the expected behaviour is quite simple; moreover where deemed necessary small examples have been provided, hopping to clarify the issue in question.

"*Integration*" – the most important requirement As the heading suggests, maybe the principal goal, applying to any of the sub-systems but also as a "*meta-requirement*" for the requirements themselves, is that everything should behave in a similar way, by having high cohesion between the building blocks. This way the developer has a smooth experience with the overall solution, making its job easier, and increasing the likelihood that such a solution would be adopted.

This aspect should be reflected throughout the entire system, from developer facing frontends, to the hidden services, for example: • there should be only one flavour (as in same technology, layout, naming scheme, or theme) of an `UI`, be it `WUI` or `GUI`, and all those tools needing an `UI` exposing such an interface; (this doesn't preclude the existence of a secondary `CLI`-based interface, needed mostly for `OaM` tasks;) • configuring sub-systems should be in a similar manner, especially the existence of a common syntax and terminology;[10] • as the solution is composed of multiple remote sub-systems, most of them export network `API`'s, thus all should use the same technology (like `REST`-based, or one `RPC`-solution); • exposed logging information and its format should be similar regardless if it originates from a Java, NodeJS or C process; etc. Unfortunately this uniformity is very difficult due to the fact that some sub-systems are only wrapped and adapted for the platform, but the way to communicate and configure them is still through their native techniques; but where possible, and in especially if the developer has to be involved, adapting to a common way is preferable. ◇

4.3.1 Functional requirements

This category is comprised of those features that need to be provided by, thus being mandatory for, the final platform. In general these are quickly identifiable as concrete sub-systems, tools or `API`'s, thus being the most concrete requirements.

Control The resulting platform should allow the developer to apply fine grained control over all running sub-systems, from own developed components to the platform's services. Such control directives could be: • starting and gracefully stopping or restarting any of the sub-systems; • forcibly terminating any unresponsive sub-system; • providing, prior to startup, a configuration mechanism to fine tune the sub-systems behaviour or execution parameters; • enabling observation for the controlled entity, by exposing events for various situations and metrics for quantitative characteristics; • (optionally) by intermediating generic simple requests, such as `OaM` actions.

These facilities should be provided both as `UI`s, as they could be manually initiated by the developer, but also pragmatically through an `API`, allowing thus automation. ◇

[10]Maybe the biggest, yet unsolved, problem in the Linux world is that of configuring various applications, and although the community has worked toward a common environment (`LSB`), no common solution for managing configuration has emerged.

Discovery Especially for components, resources and services, there must be a facility allowing interested sub-systems to: • register themselves to logical groups, possible more than one at a time, and then deregister at will, based on declared purpose; • query the membership of such groups, to find similar or dependent peers; • observe the membership dynamic, to detect alternative peers; • provide support for proximity-based selection or querying;

A good use-case would be in a frontend component finding the needed backend component or resource, without the need of hardcoding any endpoints into its configuration. Or a component experiencing poor performance with regard to a needed resource, might see that another one has just come up (maybe due to scaling) and decide to switch to that particular one. Furthermore, due to the proximity-based selection, a component could find the "*nearest*" resource — a feature not easily found in other solutions, but widely known and used low-level, high performance networking. ◇

Interaction As the reader has observed, in the previous section related to discovery, we haven't said anything about how two sub-systems should communicate, but only that they should be able to find one another. Thus based on the previously discovered information, the platform should allow sub-systems to seamlessly exchange messages, by providing the following (many of these patterns were inspired from [55]): • addressing should be possible either based the discovered information, or by the group identifier itself (thus allowing more complex patterns); • unicast (or anycast when using groups) "*request-reply*" or "*fire-and-forget*" patterns, useful for synchronous "*one step at a time*" procedures; • only with groups, multicast (maybe based on proximity selection) or broadcast, useful for information gathering; • publish-subscribe patterns useful for event notifications;

Obviously such a feature should be limited to the management interface of a sub-system (not the service one, as previously described in §4.2.7 "Management and service interface"), because the goals here are network transparency, complex communication patterns, interoperability. Thus we explicitly list the following as non-goals of such a facility: • performance, in the sens that due to routing schemes and possibly self describing data formats, the number and size of messages should be kept small; • reliability, as a certain message is not guaranteed to be delivered, or again due to routing it might be delivered twice;

Other possible facilities, but harder to implement, and most likely impractical to use under a general format, could be instances of well-known distributed algorithms for: • leader election, or consensus [13], useful in master-slave replication scenarios, singleton enforcing, or maintaining consistent global information; • distributed timestamps [21] or vector clocks [20], useful for "*happened before*" ordering or eventually consistent global information; ◇

Logging The platform must provide central, unique — not necessarily "*centralized*" — logging solution that should: • be exposed to targeted programming languages and environments through common `API`'s (like Log4j or Slf4j in case of Java, or syslog in case of `POSIX` compliant applications); • convey richer information than the usual "*a sequence of characters*", including operation contextual information (like the `MDC` in Java), possibly allowing the developer to link events happening on different components; • support multiple sinks, possibly storing the events for off-line batch-processing, or forwarding them further to other aggregators (like Loggly.)

Because such a topic can be treated lightly, we want to underline its importance, especially in the context of distributed systems. Although logging techniques are as old as programming itself, and maybe just because of that, logging facilities haven't progressed much from the `printf("got here with value '%d'...\n", some_variable)`, as most of the logging infrastructure, from filtering, searching to storing, still treats events as single lines of text, which makes it almost impossible to correlate events. As such for aiding the developer better solutions need to be found and integrated, as the `CEE` workgroup tries. Moreover as noted in [41] another stumbling block is that most of our industry treats logs as files and not as event streams, thus needlessly hampering their usefulness. ◇

Monitoring In order to aid the `OaM` tasks, but also to detect fault conditions, and possibly to aid automated scalability, the platform is required to: • gather various metrics from the infrastructure layer (like virtual machine resource utilization, or network latency); • gather metrics from within the node itself, possible with the aid of the `OS` (like `CPU`, `RAM`, disk, or network utilization); • provide to other sub-systems the possibility to export such metrics themselves (like requests per time unit); • provide a mechanism through which abnormal situations can be described, and by using the collected data, triggering such alerts; • as in the case of logging, providing support for multiple sinks, storage or forwarding (like NewRelic). ◇

Scheduling Because we have adopted an approach where on the same node we run multiple sub-systems (components, resources or services), and we could have heterogeneous node types, the platform must provide a solution to optimize the placement of these instances so that: • the resource consumption pressure is evenly distributed throughout the existing nodes; this implies a mix-and-match of various consumption patterns (like low `CPU` and `RAM` but high network bandwidth, with possibly high `CPU` and disk needs, but very little network `IO` activity); • ensure that in the case of node failures, there is a low probability that all instances of a sub-system disappear, thus not to cluster instances of the same type together; • based on the interaction patterns provide better proximity, thus putting related sub-systems close to each-other. ◇

Security Although running, and more so, designing, or developing a secure system is a very difficult task — one for which in itself we should have separate analysis and requirements — we at least try to scratch the surface by stating a few guidelines, which to our best should be followed in the following stages: • communication confidentiality and authentication, by employing well-established technical solutions like `SSL`, especially for the management interfaces; • extracting and consolidating the credential management for various services (like `AWS`), under a common sub-system; • repository authentication, so that any stored and retrieved archive is guaranteed to be created or trusted by the developer; • sub-system isolation, disallowing arbitrary, or intrusive interaction between sub-systems. ◇

Isolation Again because on the same node there are multiple sub-systems running, and in order to solve some of the security requirements, we mandate that each sub-system must run in isolation of the others. This requires isolation from at least the following perspectives, all related with the hosting `OS`: • **file-system** — in that newly created or updated files are not visible from one sub-system to another; • **network** — disallowing arbitrary endpoint creation, and constraining communication between sub-systems only as previously described by the developer; • **processes** — restricting each sub-system's view and interactions only with those processes spawned by itself; • **resources** — by confining to a bounded set of resources, to eliminate the risk of resource depletion in case of misbehaving sub-systems. ◇

Packaging and deployment As the developers code must run in the end on the platforms machines, thus needing there both executable, libraries, and data files of his components, we must provide him with the necessary tools to: • take the various files, assembling them into a special self contained archive; • upload them into a special repository of such archives; • extract these archives on the target node, in order to prepare the component's file-system. ◇

Automation Overall, in order for the platform to actually ease the life of the developer, most of its aspects must be either automatized (as in already working in an autonomous manner) or automatable (as in open to programmatic control by the developer himself). In order to achieve this the platform must expose its functionalities through one of these methods (in order of preference, but they could also be stacked, one based on another): • a clearly defined network `API`, using established technologies like `REST`-, `WSDL`-based web-services, or miscellaneous `RPC` solutions; • a programmatic `API`, maybe built on top of the network `API`, usable directly in automation tools; • a `CLI`-based tool, in the UNIX spirit, allowing developers to script their actions. ◇

4.3.2 Architectural requirements

Different from the previous set of requirements that focused on concrete end-goals, without implying a certain solution, at most just alternatives, the current one presents overall general applicable rules, that should guide the concrete design and implementation choices.

Scalability The resulting platform should be scalable in the classical sens of the term, that is if demanded by higher levels of load, increasing the infrastructure capacity, which usually translates in adding new nodes to our cluster, should translate to keeping the performance characteristics constant, or at least decaying linearly. Of course this applies only if the hosted application is itself scalable, which also implies that the selection of resources we provide integrated into the platform should pertain to such a requirement.

At the same time it mandates that our platform should also behave appropriately in the opposite direction, that is to work with a constrained and smaller infrastructure. For example it should be possible to run the entire platform inside a single virtual machine for testing and show casing purposes. Moreover the overall overhead incurred should be extremely low, otherwise the cost advantages are diminished drastically.

At the same time we could consider the perspective proposed in [28], which implies that features, especially those described herein, shouldn't hamper simplicity, flexibility, or the ease with which one is able to understand, modify and adapt our platform to suit his own custom requirements. Unfortunately this "*downwardly scalability*" aspect is often missed by most of our technological products, by trying to do many things at once, but none with an acceptable level of quality, increasing the complexity up to the point where everything looks like a complete unintelligible mess. ◇

Fault-tolerance The employed solutions should exhibit a coherent behaviour in unexpected cases (as those described in §4.3.4 "Unreliable environment"), by adopting some solutions like: • replication in either "*quorum*", "*master-slave*" or "*hot-spare*" modes; • continuous backup of critical data, either to cloud provider local storage mechanisms like `S3` in case of `AWS`, or external solutions; • on-demand complete dump of resource data, especially for those resources hosted inside the platform as components or services; • at the same time reliance on external (even fault-tolerant) solutions should be avoided as they could quickly become a problem in case of network connectivity problems. ◇

"No special roles" If possible, especially due to simplicity, and due to the unreliability and churn of nodes, employed algorithms should be master-less, or at least being able to automatically switch to a new master.

The same statement should also apply to the nodes themselves, granted that they could be assigned different roles, as such those performing the same role should behave as peers. ◇

Loose coupling In order to increase the flexibility of the overall platform, each sub-system's external dependency should be implemented in terms of a generic networked `API`, leaking as few details as possible, allowing us to easily upgrade implementations, or port our solution to other infrastructures or technologies. ◇

Reusability This is a two edged sword. For once we should reuse as much as possible already made solutions, wrapping them in to allow integration with our sub-system architecture, but as previously said, leaking few implementation or technology specific details, to allow future replacements.

On the other side each of our own sub-systems should be designed with reusability in mind, because if we manage to implement a modular platform, many of the solved problems are generic enough so that our solutions could be reused in other contexts. ◇

4.3.3 Non-functional requirements

The current section focuses on "*nice-to-have*" features, that are not critical for the overall platform, but are certainly important for the developers experience, thus making the difference between a successful or failed product [54].

Integration As stated in §4.3 ""*Integration*" – the most important requirement", and in order to ease the developer's job, we must provide ready-made solutions, either as services, resources or components, for various generic needs, such as: • popular `RDBMS` implementations like PostgreSQL or MySQL; • various `NoSQL` solutions, especially distributed ones, like Riak, MongoDB, CouchDB, or Cassandra; • staying in the same category, miscellaneous types of data stores, as caching solutions like Memcache, or data structure servers as Redis; • messaging middleware implementing either the `AMQP` protocol like RabbitMQ, or following the `JMS` model; • gateways for the most common Internet protocols like `HTTP` or `SMTP`; • solutions for offloading static content.

In general if there is a generic software solution that could be easily wrapped and provided by our platform, and that is needed for applications then we should proceed so. ◇

Determinism In order to gain and keep the developer's confidence the platform must exhibit a consistent behaviour, which could be translated into: • employing as few non-deterministic algorithms as possible, especially in the context of distributed algorithms, thus easing the reproduction of fault conditions for debugging; • following the principle of "*least surprise*" by diverging as little as possible from established procedures or concepts, as previously hinted in §2.5.3 ""*The right mix*""; • by keeping things simple, we reduce the likelihood of bugs, thus making the overall solution stable and dependable.

This requirement is very important, as if the developer has a bad previous experience with a particular feature, he is less likely to try it again in further releases — even though we promise that this time we got it right — and worse he tries to find workarounds that makes his tasks less pleasant, thus getting more disappointed by our solution. This applies regardless if the feature has a bug, or is only half backed. ◇

4.3.4 Constraints

And last but not least, some, again non-critical, guide-lines focusing the work towards selected concrete targets, or making technical decisions clear and consistent.

"*Cloud*" compatible The resulting platform should be suitable for deployment in cloud environments, thus imposing certain constraints on what infrastructure features we can rely on. These should be the common denominator of the most prominent cloud providers like `AWS`, Rackspace or GoGrid, and can be summarized to: • network features should be as much as possible constrained to unicast using common protocols like `TCP` or (if strictly necessary) `UDP`, as most cloud providers don't support multicast or broadcast, nor protocols like `SCTP`; • the choice of operating systems is mostly limited to Linux, and in some cases only to certain distributions; • storage layout is most of the times dictated by the cloud provider, forcing fixed schemes like standard partitioning, or available file-systems; • for easing the deployment, we should provide to the developer already-made virtual machine images for the selected cloud providers. ◇

"*Bare-metal*" compatible But at the same time the platform should be able to run on existing hardware, either installable into existing deployments, or providing itself the required environment, which implies: • broad compatibility with selected operating systems (see below) and various of their distributions; • straight-forward deployment and configuration procedure; • minimal intrusion or conflicts with the selected environments; • for completely dedicated hardware, we need to provide a custom integrated bootable system. ◇

Linux focus Because most cloud providers are almost Linux exclusive — many at best providing experimental support for BSD-based or Windows-based systems, with lower performance by using plain virtualization instead of paravirtualization — and because most used technologies have the better support for, or compatibility with Linux, we shall focus the design and development around such operating system. Thus: • if faced with a design or implementation problem that already has an established solution on Linux-based systems we should adopt or build around it; • in case of a choice (either for performance, security, simplicity, or other reason), the best suited option for such an environment should be used; • if possible the platform should be able to run on various, but recent enough, "*vanilla*" distributions, especially the most popular ones (like Ubuntu, Debian, or CentOS), as in some cases we won't have the possibility to provide our own base distribution; • related to the previous observation, if some required features are only available on the latest versions (either of the kernel or the base system), we should work around them and find technical solutions that work on such older versions.

We must underline the fact that although we should focus on Linux environments, and because the enhancement rate is quite high, making it a moving target, we can't always rely on the latest features being available. As such, a rule of thumb is to use whatever is available on the more conservative of distributions, like CentOS or Ubuntu `LTS`. ◇

POSIX compatible At the same time, and somehow opposing to the previous constraint, when we can maintain `POSIX` compatibility we should do so, thus keeping open the possibility to port the platform to other `POSIX`-compliant operating systems. But this compatibility shouldn't harm other platform characteristics like performance, security, simplicity, etc. ◇

Unreliable environment As the platform targets both (more) reliable (self hosted or dedicated hardware) but also less reliable cloud providers, and taking into account the warnings presented in §2.6 "Fallacies of distributed computing", we must ensure that system withstands various failure scenarios: • the simple case of sudden node failure, as explicitly warned against by cloud providers; • unreliable or flaky networking, especially in cloud environments; • arbitrary network partitioning, or complete temporary disconnection, especially in self hosted situations where redundant networking equipment is unavailable; • lack of disk persistence, either due to hardware malfunction in self hosted scenarios, or because of cost in case of cloud providers; • unreliable performance levels, or temporary "*freezes*", mostly when virtualization is involved; • dynamic, but moderate, churn for nodes, for example due to cloud provider failure and restart of nodes. ◇

4.4 Architecture

4.4.1 Layers

Summarizing the essence of those expressed in §4.2 "Concepts", and as depicted in §4.1 "Application layers", the overall solution obtained by the developer has the following layers:

- **infrastructure layer** — composed of nodes, grouped into a cluster, and those resource's obtained directly from the provider;
- **platform layer** — composed of various sub-systems, exposed either as services or resources to the application;
- **application layer** — composed of components.

4.4.2 Inter-sub-systems architectural style

As previously hinted in §4.2.2 "Application" or §4.2.14 "Sub-system and agent", the overall architectural style, between different sub-systems, is a `SOA`-like one, where each one fulfills a single well-defined role, delegating and coordinating tasks to other specialized sub-systems.

In order to fulfill the explicit requirements §4.3.2 "Loose coupling" or §4.3.2 "Reusability", but also §4.3 ""*Integration*" – the most important requirement", and because we intend to implement these sub-systems in various, but most suited for the task at hand, programming languages, we have chosen a well established technology for the communication, that of `REST`-ful web-services, discovered directly through `DNS`. But where necessary, like for performance reasons in the case of the component hub, described in §4.6.1 "Component hub", we break this implementation decision and choose the best suited solution.

Moreover because one sub-system is composed of multiple agents, each running on a different node, we have chosen that all these agents should be equivalent, each one providing the same

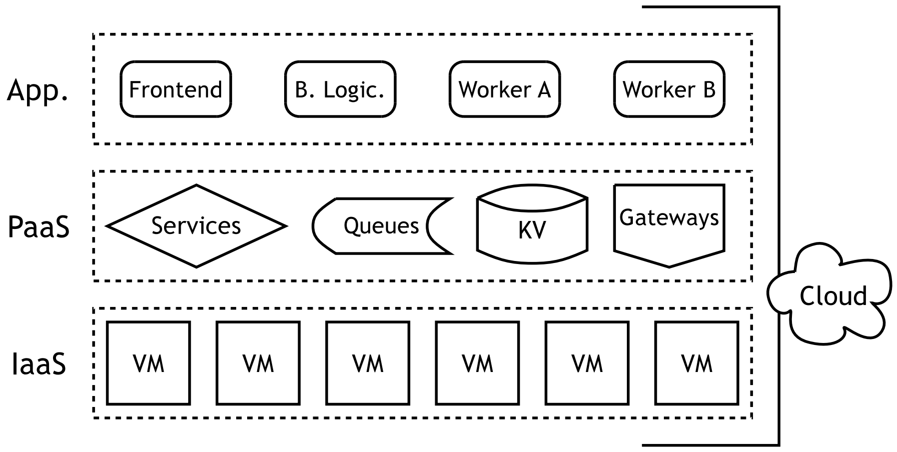

Figure 4.1: Application layers

functionality, thus helping in solving the §4.3.2 "Scalability" requirement.

These decisions give us the following benefits: • **flexibility** — by allowing us to replace one sub-system implementation for another, thus porting from one environment to another would become simpler; • **testing** — by easily replacing a real sub-system with a mockup one in order to test the behaviour of other sub-systems in various scenarios; • **debugging** — because we can easily trace all the calls between sub-systems by simply changing the `DNS` information and redirecting all requests through proxies; and also because we can manually interact with the sub-systems even with rudimentary tools like curl or even Web 2.0 tools.

4.4.3 Intra-sub-systems architectural style

On the other hand, inside the same sub-system, the agents follow a "*peer to peer*" architectural style, thus fulfilling the requirement §4.3.2 "Fault-tolerance". This also forces us to select certain algorithms, that usually exhibit characteristics demanded by the requirements §4.3.2 "Scalability", §4.3.2 "Fault-tolerance", or §4.3.4 "Unreliable environment".

As an example, anticipating what we'll explain in §4.7 "Component hub and controller", the component hub and the component controller use an adaptation of the Dynamo system [14], thus allowing the sub-systems to work even in case of node failure, or even worse cluster partitioning.

About the employed implementation techniques, because each sub-system has its own specific requirements, and because we reuse various already developed frameworks, like riak-core in the previous case, we impose only minimal restrictions on the used technologies: • they should only use `TCP` or unicast `UDP` for communication, but only inside the cluster as we shouldn't expect Internet access, because the application could be used only inside an intranet; • if discovery is necessary it should either be done through `DNS`, or through the platform's provided mechanism, the discovery service, which in the end is also responsible for maintaining the `DNS` information; • preferably should use `SSL` with mutual, both client and server side, authentication, as described in §4.9 "Security", or at least provide some minimum security, like communication channel authentication, because although the node's firewall is controlling access from the outside, it's harder to protect from within the node itself where other modules run, especially the developer's components.

4.5 Life-cycles

In order to facilitate the understanding of future sections, we shall describe which are the phases through which a few essential entities pass during the life time of the platform and the hosted application. We model these phases as simple `FSM`s, giving short descriptions for the states, and the actions or events that trigger transitions between states.

Unfortunately because we are referring to highly complex entities, like the platform which doesn't even exist as a concrete element but is a conglomerate of various elements, or the nodes whose life-cycle depend on the provider, the running `OS`, and our own code, therefore we can not be very thorough in our definition, thus the informal and a rather informative contents of the current section.

We underline the fact that these are not related with the product or development phases, as described in §2.3 "Development methodologies", but with the run-time behaviour of the resulting artifacts.

4.5.1 Application and platform life-cycle

Although neither the application nor the platform are tangible entities, being only umbrella concepts for various sub-systems, but because they are perceived by the developer as concrete entities, we decided to treat them as such. We first start by observing that the life-cycles of both the application and the platform are linked and synchronized together, because the platform is perceived as a subset of the former, but at the same time it hosts the application.

The states and transitions, depicted in §4.2 "Application and platform life-cycles" are the next ones, in chronological order:

- ***"none"* / *"prepared"*** — when the developer has already developed, tested, and packaged his components, but not yet deployed anything;
- ***"bootstrapping"*** — entered after the developer initiated the "*start*" operation, during which: • the initial infrastructure is obtained, either through a cloud provider, or manually in case of owned hardware; • after which, the nodes startup and finish bootstrapping themselves as seen in the next section; • then the core sub-systems, like discovery service, container controller, component controller and component hub start;
- ***"starting"*** — only pertaining for the application, during which the components and the required resources are started, and begin to handle the application's needs; during this the platform itself is already fully functional and in the "*executing*" state;
- ***"executing"*** — entered automatically once all the previous steps are finished; and only now we can speak of a fully functional application, suitable to be used by the final user;
- ***"stopping"*** — like in the case of "*starting*" it is appropriate only for the application, during which all the running components and resources are destroyed; at this time the platform is still in "*executing*" state, but we can't speak of a fully functional application; this transition must be triggered by the developer;
- ***"destroying"*** — entered either after the previous phase has finished, or when the developer, or the platform itself, initiates a "*destroy*" operation, forcing the platform to be shutdown; during this time: • any non-core sub-system still running is forcibly destroyed; • some of the core sub-systems are stopped, like the component hub or container controller, but some other, like the node controller or discovery service, are never stopped, but instead just disappear when the last node is destroyed; • as the last step the infrastructure is decommissioned;
- ***"destroyed"*** — entered automatically when none of the infrastructure elements, especially the nodes, exist.

We must note that these states and transitions are not explicit throughout any of the sub-systems, as they are obtained from the coordinated behaviour of all the comprising sub-systems.

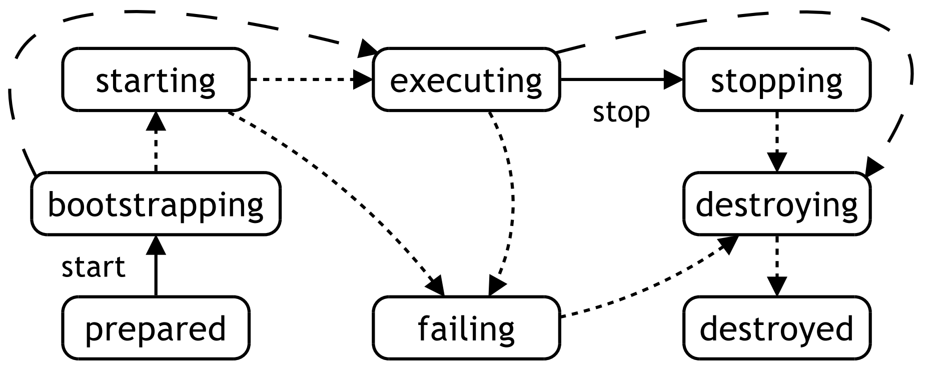

Figure 4.2: Application and platform life-cycles
The normal lines are transitions initiated by the developer. The dotted lines are automatic transitions. The dashed lines are transitions only for the platform.

Thus their management should be delegated to a new specific sub-system, the application controller, as described in §4.6.7 "Application controller". But it would prove to be a difficult task as it must work in an unreliable environment, especially in case of a cloud computing environment, and therefore this application controller should have as little in common with the platform and especially with the infrastructure, or else we'll be in a "*chicken and egg*" kind of problem.

4.5.2 Node life-cycle

Before starting we remark that most cloud providers have their own states and transitions for the virtual machines they provide, but unfortunately there is no common or standardized schema. As such in the current section we try to propose a schema adapted for our platform consisting only of those phases essential for us.

Thus as depicted in §4.3 "Node life-cycle" we have:

- "***none***" — a fake state where the node doesn't even exist, but we use it as an initial state for our `FSM`;
- "***acquiring***" — initiated usually automatically by one of the controllers, which implies contacting the cloud provider through one of its remote `APIs` and requesting the lease of a new virtual machine; if the lease is not allowed then the state becomes "*acquiring-failed*"; in the case of owned hardware we don't have this state;
- "***preparing***" — transitioned automatically when the provider accepted the lease and started the provisioning; or in case of owned hardware this is transitioned when the physical machine started booting the `BIOS` or equivalent firmware; if this fails we end up in "*preparing-failed*";
- "***initializing***" — entered automatically when the `OS` and the initial controllers have started; during this state the core sub-systems are started;
- "***available***" — entered automatically when all the core sub-systems have started;
- "***unavailable***" — entered by an "*mark-unavailable*" action triggered manually by the developer when the node shouldn't be used by the platform, because of unspecified reasons; to exit this state the developer should manually trigger "*mark-available*";
- "***offline***" — entered automatically when the node is not accessible from the network or when the core sub-systems are not accessible node; this is a temporary state, that happens due to unexpected conditions, like virtual machine destruction or reboot, power surge, etc.; after a

certain timeout if the node still remains in this state it transitions to "*failed*";

- **"*destroying*"** — entered usually after the node was already in the "*unavailable*" state, and after all the applications components have been stopped or migrated, leaving only the core sub-systems running on the node;
- **"*destroyed*"** — entered when the machine was completely stopped and un-leased from the cloud provider;
- **"*failed*"** — entered automatically when the node is not behaving accordingly, for example core sub-system failure, hardware failure; this state, in the case of a cloud computing environment, should imply that the virtual machine was released.

We should note that this life-cycle, as in the case of the application's or platform's as described above, is maintained only through coordination of various agents, both running on and off the node in question, as for example the "*offline*" state can't be determined from within the node itself.

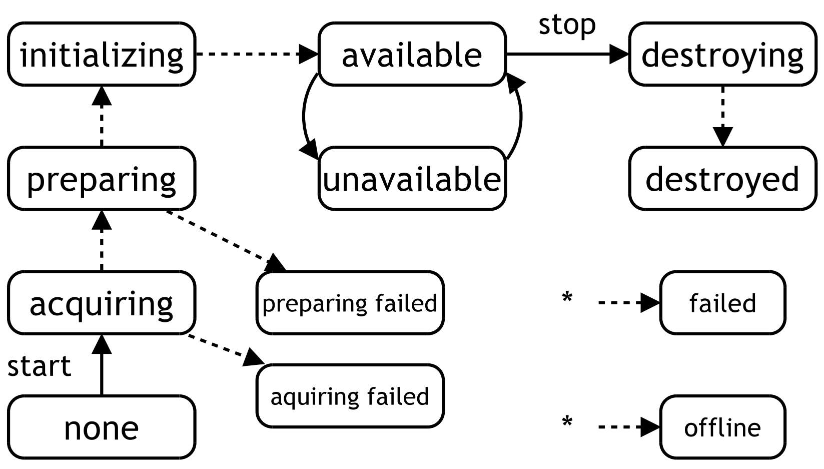

Figure 4.3: Node life-cycle
The normal lines are transitions initiated by the platform. The dotted lines are automatic transitions.

4.5.3 Agent, component, service and resource life-cycle

As hinted in §4.2.14 "Sub-system and agent" and §4.4.2 "Inter-sub-systems architectural style", some sub-systems are composed of multiple agents forming a peer to peer network, which in the current section we call "*clustered agents*", while other agents are independent of each other, thus called "*independent agents*". Moreover only for the scope of the current section, we consider as agents also the developer's components.

For both the "*clustered*" and "*independent*" agents we have the following states and transitions, depicted also in §4.4 "Independent module life-cycle":

- **"*none*"** — a fake state where the agent doesn't even exist, but we use it as an initial state for the next transition;
- **"*preparing*"** — initiated by the "*start*" action, during which requests are coordinated towards various sub-systems, but none of the actual processes are yet running;
- **"*initializing*"** — automatically entered when the first process of the agent has started running;

- **“*available*”** — automatically entered when the agent has finished initialization and notified accordingly the platform;
- **“*unavailable*”** — usually triggered by the agent itself to mark the fact that although it is alive, it shouldn’t be used as it is temporarily unable to fulfill requests;
- **“*stopping*”** — triggered by the “*stop*” action, during which the agent is still alive and starts relinquishing resources;
- **“*destroying*”** — entered automatically after the agent has finished its shutdown procedure, and during which any `OS` resources are released;
- **“*destroyed*”** — entered automatically after all traces of the agent have been cleared.

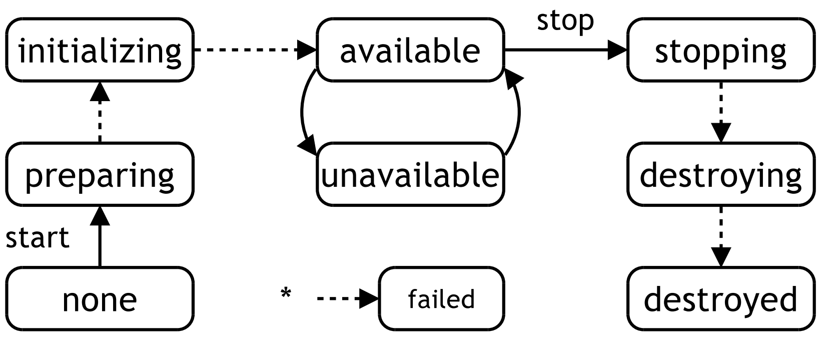

Figure 4.4: Independent module life-cycle

Only for the “*clustered*” agents we have additional states related with their peer to peer behaviour, also depicted in §4.5 “Clustered module life-cycle”:

- **“*joining*”** — just after “*initializing*” and before “*available*” during which the agent tries to connect to its peers; if it fails it could move to “*stopping*”;
- **“*leaving*”** — just before “*stopping*”, during which the agent executes the reverse of the “*joining*” operation;
- **“*offline*”** — as in the case of node’s life-cycle, the “*offline*” state is only determined from outside the agent itself, for example in situations when one can not contact that particular agent.

We should add that for simplicity we force that an agent is allowed only once to join and leave its peer to peer network, although it could do this behind curtains and mark itself as “*unavailable*”. Also if we are not interested in its peer status, we should consider “*joining*” as “*initializing*”, and “*leaving*” as “*stopping*”, these being actually sub-states of the ones mentioned.

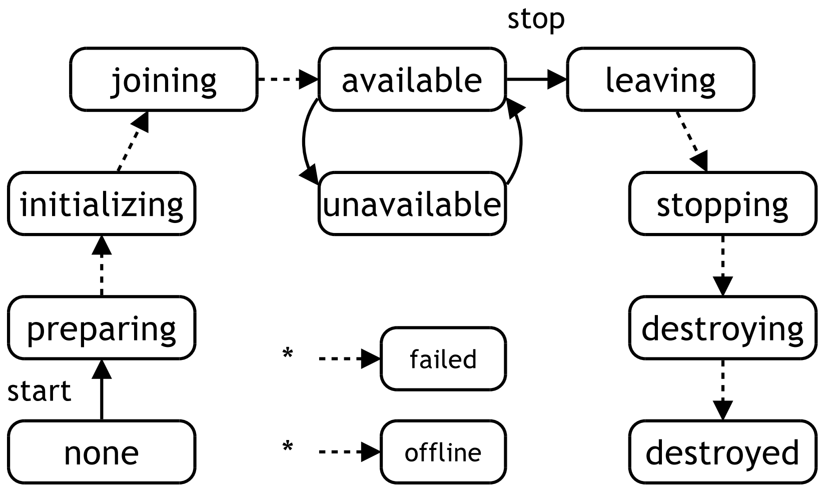

Figure 4.5: Clustered module life-cycle

4.6 Sub-systems

We continue what has been started in §4.2.14 "Sub-system and agent", by providing a deeper view into the sub-systems provided by us as part of the platform, enabling the developer to build his application in abstraction of the underlying infrastructure. Some of these sub-systems are completely hidden inside the platform from the developer, like the container controller or node controller, others are clearly visible and exposed as services but mediated through the component harness, like the component hub or the logging service, while a minority, which are only seldom used, are exposed as services and must be explicitly used by the developer inside his components.

In the next sections we'll superficially describe most of them, focusing on their purpose and relation with others, but also on design and some implementation aspects or hints. Also, as the topic of the current section is the design of the platform, we'll limit ourselves only to those sub-systems that heavily impact its outcome, ignoring those less complex — but not necessarily less trivial or unimportant, in particular tools, UIs, data-stores, or external services, but also third party products that have been integrated into the platform to offer to the developer a "*ready-to-go*" solution. We must observe that some of the requirements presented in §4.3 "Requirements" are identifiable with one of these sub-systems, thus another reason why we'll not insist much on their role, but point to the respective sections.

In further sections we'll take the most important ones and present them in more detail, especially those where the author's contribution is substantial — see §4.7 "Component hub and controller" and §4.8 "Credentials service". Also as stated in §4.1 "Inspiration", the current work was mostly done under the umbrella of the mOSAIC FP7 project, therefore we'll frequently refer to its current design and implementation, either to put in context a sub-system's role, for pointers to more in-depth conceptual or especially technical explanations, and even to contrast divergent directions presented in the current work from that of the project. The main reason why the presented design differs, in places even largely, with the project's technical reports, like [24], is due to the difference between the two phases in which they were written — the project's report being among the first deliverable, when only proof-of-concept prototypes have been available, meanwhile the current work was written close to the end, when the major parts have already been stabilized, most having either alpha-ready or prototype implementations. A secondary reason is because the purpose of the current work is to take a step forward[11], as it present an enhanced variant of the such a platform, having in common mostly the principles, but not always the exact details. Therefore where deemed necessary we'll point out the author's contribution, or lack thereof, and the ways in which the two designs diverge.

4.6.1 Component hub

As described in §4.7 "Component hub and controller", the component hub mainly facilitates component interaction by providing all those listed features, thus fulfilling the requirements described in §4.3.1 "Discovery", §4.3.1 "Interaction", and §4.3.1 "Logging". In essence the component hub is nothing more than a glorified message broker like RabbitMQ, or distributed message bus like ZeroC Ice, with a custom semantic and a distributed implementation. But as we'll describe in depth all the design and technical aspects in §4.7 "Component hub and controller", we limit ourselves in the current section only to the interactions with other sub-systems.

Thus in order for the component hub to route messages or notifications between components, it communicates over a network channel directly with each component harness, which in its turn relays these over a local channel with the component itself; for this purpose a special RPC-like protocol was needed.

As additional dependencies the component hub also requires a local, not necessarily persistent, database tuned for reads of small size, writes being seldom, used to store various information like component membership status, related meta-data, and others.

4.6.2 Component harness

Although the purpose for the component harness's existence is purely a convenience one, all its facilities being implementable inside the components themselves, or collected in a programming-language dependent library, we see it as mandatory for the overall success of the platform. This is because it eases both the implementation of components and the adaptation as components of well-behaved software — by "*well-behaved*" we mean those software products following the principles described in [17] — like for example RDBMSs as PostgreSQL or MySQL, J2EE containers as (Apache) Tomcat or Jetty, or Python / Ruby web-services using standard frameworks like WSGI, respectively Rack.

Mediator role In essence the role of the component harness is that of the mediator between the component itself and the rest of the platform — that is other services, resources or other core sub-system's agents — allowing thus the component to have a simplified view of the entire platform, all the needed operations being exposed by this component harness through a unique local channel. Therefore we won't insist on the particular exposed operations, summarizing them as those exposed by the mediated sub-systems. Although the component harness changes the communication channel, and the final destination of messages, it doesn't change in any way their syntax or semantic, at most providing enveloping, which is helped by the fact that services use a common interface.

For example even though messages and notifications are routed through the component hub, each component sends and receives them through the channel established towards the component harness; the same in case of logging messages, they are sent over the same channel, being the job of the component harness to relay them to the actual logging service.

We can easily observe that by adopting, from the point of view of the component, such a centralized approach, we could also benefit in other areas like: • simplifying testing, and especially

[11]Sometimes taking a step forward implies actually taking a step behind and looking at the big picture, ascertaining what fits, and weeding out what doesn't.

automated testing as adopted in `CI` movement, by allowing the developer to create a simplified component harness, that doesn't interact with the platform but fulfills the requests locally by following an apriori designed script; • increasing the flexibility of the overall platform, by allowing us to change from certain technical aspects, like communication channels, or authentication mechanisms, up to moving the responsibilities from one sub-system to another, all as long as we keep the syntactic and semantic contract of the local channel between the component harness and the component; • porting components to other platforms offering similar services, just by translating and adapting the requests in terms of the targeted platform; this being achievable even in less-similar ones, by reimplementing inside the component harness the missing functionality; • allowing migration from a dynamic environment like our platform provides, to a more static, configuration driven, environment like the classical approaches presented in §2.5 "Target applications". ◇

Adapter role As expressed in the beginning of the section, the component harness, or better said a specialized variant, could be used to facilitate the integration and adaption of non-component, but well-behaved, software solutions, which we call "*legacy components*", and exposing them at the platform level as full-blown components.

This is achieved by investigating the requirements, especially those `OaM` related, of various targeted software products, examples having been provided in the introduction of the current section. We could summarize these needed facilities as:

- **resource resolution** — in general some need to access external resources, most likely some kind of database like a `RDBMS` or a message-broker like RabbitMQ; these resource's endpoints should be hard coded into a specific place in the configuration file; therefore the component harness should resolve these dependencies via the component hub and then fill the configuration file accordingly by using a templating-engine;
- **resource health checking** — because most of these products were designed with a stable environment in mind, which according to our requirements is not the case, the component harness should periodically check if the resources are still operational, and if not either update the configuration file accordingly and restart the legacy component, or signal an erroneous situation and terminate;
- **endpoint management** — most of these legacy components require one or multiple network endpoints in order to directly communicate with their clients — as opposed to using a message broker like RabbitMQ — which implies that the component harness must handle the dialogue with the container controller to obtain the endpoints' specifications and, depending on how the software was written: • in most cases, like with the resources hard code the specification in the correct places inside the configuration file; • only with recent software that uses the socket activation technique, open the endpoints and transmit to the process the associated descriptors;
- **endpoint registration** — thereafter these endpoints should be registered via the component hub so that other components are able to resolve them;
- **logging collection** — as explained in §4.3.1 "Logging", most software solutions take the rudimentary approach to logging, that is lines of text; thus in order to have a unified and useful logging system, the component harness would have to transform these lines to proper logging events and forward them to the logging service, by, again depending on how the legacy component was written: • reading from `stderr`, plus from `stdout`[12], and based on specific regular expression patterns create the structured logging events; • reading from various log files[41], again structuring the messages, and then truncating these log files like logrotate; • implement the syslog protocol thus intercepting the messages, and treating them as in the first case;

- **life-cycle management** — although the life-cycle of a component is quite simple, some of these software solutions require certain preparatory steps before being usable, like for example in the case of a `RDBMS` initializing the needed files; thus the component harness should allow scripting such stages in order to adapt the complex life-cycle to our simplified one;
- **operation translation** — some of these software products also allow limited interaction, especially for `OaM`, through `OS` signals, `CLIs`, simple `IPC`, or if in luck even D-BUS; therefore the component harness should allow minimal support to export such mechanisms, via the component hub, thus enhancing the integration of these legacy components with the rest of the platform's components.

We remind that other dependencies like the various software libraries, data files, or `OS` resources, except the ones mentioned above like the endpoints, are handled directly by the container controller itself, via the descriptors placed inside the legacy component's bundle, as explained in §4.6.4 "File-system management". ◇

4.6.3 Component controller

Before introducing yet another controller, we recall the big picture presented in §4.4 "Architecture", where we can see that the platform is built like an onion from layers of abstractions: • first there is the infrastructure from where we carve out a cluster for the platform; • the cluster is then decomposed into nodes; • each node hosts sub-systems, where one such sub-system manages multiple containers; • each container may in turn hold a component, a service or a resource; (these could have been run directly as containers or sub-systems themselves, and some of them certainly are, but its better to run them as plain components for the sake of simplicity and flexibility;) • if we put together all these components, services and resources, we obtain the application.

Thus each of the previously described controllers had targeted a particular layer: • each node controller agent took care of an individual node, the distributed sub-system managing the entire cluster; • on each node we had an independent container controller managing the containers; • and now we need a sub-system to take care of the component layer.

Therefore the "*component-controller*" sub-system is tasked with managing all the components running inside the platform and constituting the application, thus fulfilling the requirement presented in §4.3.1 "Control". Furthermore in order to fulfill the supplementary requirements presented in §4.3.2 "Fault-tolerance" and §4.3.4 "Unreliable environment" we must make the sub-system and its agents take a distributed system approach.

Responsibilities In essence the component controller has the following responsibilities:

- **maintain a component (type) repository** — a collection of component types available for instantiating, together with information about them, like for example: • the associated bundle, • required `OS` resources for the container, • (default) configuration data for the component harness, • (default) configuration data for the component itself, • audit information about the registration, source, etc.;
- **maintain a component (instance) registry** — similar with the above, but a collection of actual instances of a particular type of component, associated with information like: • obviously the component type, • the actual configuration data for the component harness, • the actual configuration data for the component, • the node and container where the component is hosted, • `OaM` information as the various active endpoints, • security information as the

[12]Maybe one of the worst ideas in "*modern*" programming languages — especially in Java's Log4j, Slf4j, but the list could be continued with NodeJS, Ruby — was to use `stdout` for logging purposes, making it impossible to discern the real output from the logging messages.

certificate, • audit information about the instantiating, the initiator (either automated by the component scaler or manual by the developer;
- **manage the component's life-cycle** — by keeping track of the current state, delegating the execution of creation or destruction operations to the right container controller, and delegating intermediary life-cycle transitions to the component harness;
- **coordinating with the component scheduler and component scaler** — by delegating the decision of creation, together with the targeted container controller, or destruction, together with the targeted component instance, originating either from manual or automatic actions;
- **expose a network `API`** — allowing tools to maintain, respectively initiate, the previously mentioned data, respectively operations.

We can easily observe a resemblance with the component hub, both in the scope of the responsibilities, that of managing or connecting components, but also in the upcoming design and implementation, which is a topic we shall better argument in §4.7 "Merging reasons". ◇

4.6.4 Container controller

Although not described in §4.2 "Concepts", a "*container*" is the `OS`-level environment in which any component — and possibly a resource, a service, or even another sub-system's agent — executes in complete isolation of other peers. This relates to all aspects visible by an `OS` process, including: • constraining file-system layout and contents; • constraining available network endpoints; • confining interaction with other processes, only to those belonging to the same container; • resource quota like `RAM`, `CPU`, disk `IO`; • authorization through capabilities and privileges. As such by exposing the facilities described below, it resolves half of the requirement described in §4.3.1 "Packaging and deployment", and fully those in §4.3.1 "Control" and §4.3.1 "Isolation".

The sub-system agent binds on a network endpoint and accepts requests from potential clients, secured as described in §4.9 "Security", using an enhanced version of JSON-RPC, exposing the following operations:

- **container initialization** — which implies preparing a fresh version of the container's file-system, and allocating, or in most cases just logically reserving, the required `OS` related resources like `RAM`, network endpoints, temporary file system;
- **container control** — once the container has been initialized, the client can request starting the necessary processes inside the isolated container; after some time it could also request for its graceful or forced termination;
- **container monitoring** — by providing those metrics available for the container, obtained for `OS` level resources like `RAM`, `CPU`, and others, but also providing various notifications related to the container's life-cycle, like initialization, execution, termination; other fine-grained metrics, like for what exactly the `RAM` is actually used, should be provided by the executed processes themselves.

Although the client perceives the operations as an atomic actions, most of these tasks are actually delegated to other agents, thus in these cases it plays the role of a coordinator.

File-system management As presented previously, one of its main roles is to take care of the file-system of the containers, initializing, monitoring, and destroying them accordingly. At the current state of art, each container's file-system is correlated with a "*bundle*", an archive containing all the necessary files to execute the component — we underline that "*all*" actually mandates that any file not existing in the mentioned bundle won't be available to the component, including plain interpreters or tools which must be explicitly provided by the developer as part of the bundle.

Although at first this might seem like a waste of storage, it needn't be so, by employing the right technology. As such on each node, and for each container type — as defined all those using the same bundle — already initialized, it keeps a template file-system which is then cloned for each new instantiation.

This can be achieved by using one of the following techniques (the first being the one currently in use, the others being given as valid alternatives): • using Btrfs, each template is kept as an individual sub-volume, and when needed cloned as a `COW` snapshot; (this technique is similar to what Joyent is doing with their Solaris-based SmartOS, by using `ZFS`); • as Btrfs is still regarded as experimental, and because a mature file-system with similar features doesn't exist on Linux, we could apply the same technique but at a lower layer, specifically using `LVM` to create a logical volume for each template, and then writable snapshots for each instance; the only disadvantage is that we must decide how much space to be allocated up-front to each volume or snapshot, thus making optimistic over-commit impossible; (although this restriction has been lifted in very recent Linux versions by using "*thin-provisioning*"; • another, rather obscure, solution would be to use either Vserver's `COW` hard-links, or an overlay file-systems like aufs; • and if none of these methods work, we could resort to a read-only "*bind*" mount point of the template file-system folder, and providing a single instance of `tmpfs` usually mounted on `/tmp`. Each of these techniques have their advantages and disadvantages, but the number of options clearly proves that such a goal can easily be achieved, and without performance costs.

The container controller doesn't need to manage these file-system entities itself, but instead it could delegate these tasks to the node controller, thus the only thing remaining from this operation is filling the file-system contents. ◇

Process management The second important role of the container controller is managing the processes confined inside the container, and thanks to the implementation technique presented next, it also manages resource quota, capabilities and privileges.

Until a few years ago, in Linux there were little options in isolating processes and treating them as a single compact logical unit; meanwhile other operating systems like the BSD family had the concept of jails, or Solaris had zones. But recently there is `LXC` which plays a similar role, which actually is only a set of tools that compose some interesting kernel features: • `OS` namespaces that provide the actual process isolation, allowing each set of processes to have disjoint file-system mount points (the `CLONE_NEWNS` flag of the `clone` system call), limited process visibility (`CLONE_NEWPID`), private networking stack (`CLONE_NEWNET`), and private `IPC` domain (`CLONE_NEWIPC`); • the `cgroup` facility that provides resource quota, priority, accounting and monitoring; • and capabilities [40] that provide special actions (like networking configuration, module management) authorization, regardless of the user identity.

Security-wise, unfortunately, as the documentation states, `LXC` (and the underlying mechanisms) is more an application virtualization framework, than an enforcement one. As such although it provides similar facilities with jails, it does not offer the same guarantees. If such is the case then `LXC` could be swapped with a security-focused alternative, namely Vserver (and the corresponding grsecurity patches), but this implies running a non-standard kernel. A second alternative would be to combine the `LXC` isolation with one of the `LSM` implementations available, like AppArmor, enforcing certain policies. And for completeness we could remind that maybe a lighter, finer grained approach, to policy enforcement would have been the Capsicum framework available in FreeBSD [8].

Getting back to the process management, after the container controller prepares the file-system and the isolated environment, it starts an initial `init`-like process, that in the case of a component is the component harness, which is responsible of starting and managing other needed child processes. It provides this parent process with relevant configuration, through various simple techniques like writing it in a temporary file, accessible to the process, and giving its path as an environment variable or command argument, or even better by giving the process a readable pipe descriptor, again specified as in the former case, and passing the contents through it.

The concrete resource quota and priority, capabilities, `init`-like process and configuration specification, and other run-time information, is also described accordingly in the bundle. ◇

As stated in the beginning of the section, we don't go in depth, partially because this sub-system is currently under heavy development in a post-prototype, but pre-alpha phase, thus more concrete information would be out-dated before the ink dries even; secondly because it was not covered by the author, but being designed and developed by another member of the mOSAIC FP7 project.

4.6.5 Node controller

The purpose of the node controller is to manage the node itself in the context of the existing infrastructure — thus the underlying machine and its environment, be it a virtualized or a physical hardware — and in the context of the running host OS or host distribution, by providing common operations with an unified API, independent of these particularities. In a certain way it could be seen as an OS level services and resources API, a meta-OS if we want, translating operations to the lower level OS APIs, or even coordinating a mix of local ones with those exposed by the cloud provider — concrete examples are provided as we go through the responsibilities. As such it covers the requirements presented in §4.3.4 "“*Cloud*” compatible" and §4.3.4 "“*Bare-metal*” compatible".

The node controller could also be seen as an equivalent of the container controller, but for nodes, and in a certain sense these two sub-systems could have been merged into a single one, as some operations needed by the container controller, like the file-system or networking management, are actually delegated to the node controller. But there are differences that make the separation important: • the container controller is focused on container virtualization, thus targeting exactly one of the enabling technologies, like LXC; but it is independent on how the actual node exists, as physical or virtualized hardware; • not all sub-systems necessarily need containers, and some of these sub-systems might be hosted on dedicated clusters; for example a map-reduce system like Hadoop, or a workload management system like Condor, which are highly specialized software applications themselves, having their own isolation, scheduling, and recently even scaling mechanisms, and expecting a dedicated cluster, where they are in complete control; • depending on the actual type of the node, like physical hardware, Xen virtual machine, EC2 virtual machine, there are non-standard methods to fulfill certain tasks, like obtaining monitoring information, disk attachment, and thus for each of these we could need specialized node controller code; • privilege requirements could also be different, as the node controller need full access to the underlying OS or infrastructure, meanwhile the container controller at most needs some limited capabilities to bootstrap the container's required resources; as such by separating the privileges, preferably requiring that the container controller delegates all privilege tasks to the node controller, we considerably reduce the the attack surface of the platform thus increasing its overall security characteristics.

Getting back to the node controller itself, its responsibilities are described below — splitted based on importance, as there are quite a few tasks that could be fulfilled by such a sub-system, some crucial and some just as "*nice-to-have*" features — together with hints of their possible implementation. Unfortunately we elide the actual details, because, as in the case of the container controller, the actual sub-system is in early design and prototype stages, and its implementation is handled by a different person than the author; furthermore these details are very technical in nature, involving fine-grained accurate explanations in order to make them useful, thus making them inappropriate for being described in the current work. Nonetheless some aspects presented here diverge quite significantly from the current state and direction of the implemented sub-system, making them a valuable contribution to the platform's big picture.

Primary responsibilities The following are the most important tasks, thus mandatory to be implemented, that must be provided by the node controller:

- **sub-system management** — by handling the life-cycle of all the necessary sub-system's agents — including starting, stopping, and if necessary restarting — both in an automated manner, by taking into account the various interdependencies, but also on-demand as requested by the developer or other sub-systems; this could be easily achieved by leveraging current service management systems like systemd, Upstart, or lightweight alternatives like runit;
- **sub-system health checking**[13] — periodically running a set of tests, like making simple `HTTP` requests in case of agents exposing such an interface, sending heartbeat packets, or simply by checking that certain expected network endpoints are active, all of this besides checking that all the expected processes still exist, thus determining for each sub-system if its agent is behaving as expected; if this isn't the case it should proceed to forcibly restarting the faulty sub-system agent; this could be easily achieved by leveraging already existing monitoring software like Monit;
- **sub-system monitoring** — by collecting and exposing certain limited `OS` level metrics related solely to the sub-system's agent's processes, most likely only those exposed through the `getrusage` system call; again like for the container controller finer grained metrics should be exposed by the sub-system itself; moreover it should also provide notifications for the various sub-system agent life-cycle events, like those enumerated in the management paragraph, or notifications for health-check failures and recoveries;
- **node intrinsic monitoring** — by exposing all the `OS` level metrics related to the entire node as a whole, like `OS` load, total usage of `RAM` or `CPU`, total disk usage or `IO` activity; most of these values being available through the `/proc` synthetic file-system, or collected through delegation to a specialized tool like collectd;
- **node extrinsic monitoring** — in general only within a cloud environment, similar with the above, but reporting metrics obtained from the cloud provider, most likely through a certain `API`, like Amazon CloudWatch;
- **node discovery** — subscribing the node via the discovery service under multiple groups, based the node's role, making them easily resolvable by other agent's and visible for the developer.

◇

[13]Although in other contexts this should have been called "*monitoring*", in the current work the term "*monitoring*" relates to collecting various metrics and distributing them to interested parties. Therefore we were forced to use this, maybe even better suited, term of "*health checking*".

Secondary responsibilities Besides the previous ones, we could invest the node controller with other useful tasks, that would otherwise be implemented by the sub-systems themselves, thus possibly leading to effort duplication, lost focus, and even security implications as most of these require elevated privileges:

- **package management** — by installing, and possibly automatically upgrading, all the software packages needed to run a certain sub-system's agents; this could be achieved by delegating the actual work to the host `OS`'s native package manager, then exposing an abstract interface, independent of the actual distribution; furthermore because some frequently used packages don't have a common name over all of the important distributions, it must also provide such a unified naming scheme, or at least, name mappings;
- **network access management** — because on the same node there are many sub-system's agents, components, services and resources, executing at the same time, each one of these possibly requiring a network endpoint, there is a need to control which of these endpoints are visible from the Internet, which from other nodes in the same cluster, and which are private to the current node; this can be solely achieved by controlling the local node's firewall;
- **private network management** — although not described in the container controller section, there is also the possibility for each container to get its own private `IP` address; but in this case that `IP` address would be visible only to the node itself and not outside; thus depending on the actual running context (in cloud, on owned hardware) this can be solved either through static or dynamic routing, tunneling, `VPN`s, or even through `PNAT`; unfortunately the exact details are far too technical in nature, and involve a high level of detail, to be described in the current work;
- **file-system management** — as presented earlier, some sub-systems, like the container controller, might need special file-system storage, which could be managed directly by the node controller; the technique was already hinted in the section §4.6.4 "File-system management";
- **developer console** — although the platform should be fully automated, sometimes the developer might need to manually intervene, especially in case of misbehaving sub-systems; this could be easily achieved by using `SSH`, the node controller just needing to fill the right keys, and provide the useful tools to query and control the executing sub-system's agents.

◇

Miscellaneous responsibilities Although unlikely, in addition to ones previously described, another set of tasks could be implemented inside the node controller, especially if we employ a very customized host `OS`, depending on how tight we want to be able to manage the node:

- micro-managing each and every resource itself, like configuring the network interface `IP` addresses, configuring the `CPU` frequency scaling;
- micro-managing resources by delegating the tasks to various already existing tools.

◇

4.6.6 Credentials service

We start by defining the concept of "*credential*" as a pair of "*credential data*" and associated "*credential operations*", where:

- ***"credential data"*** — is split into two parts: • ***"shareable credential data"*** — usually information identifying the credential, like user account identifiers, application access keys, X.509 certificates, or information describing the credential, like attached email account, service endpoint; in general any information related with the credential that if made public wouldn't heavily impact the security of the platform or the running application; but just because this information could be public, it doesn't imply that we should treat it carelessly, but instead we should try our best not to be leaked; • ***"confidential credential data"*** — usually information used in authenticating and authorizing requests made under the identity of the credential, like user account passwords, application secret keys, or X.509 private keys; the general rule is the reverse of the above, meaning that any information which if leaked would considerably harm the security, should be treated as part of this category; therefore we must always handle these with utmost care, never storing or communicating them in plain text;
- ***"credential operations"*** — algorithms that are executed in the context of credential data, especially those needing access to the confidential credential data, like request authentication and authorization.

Thus the "*credentials service*"'s role is to fulfill part of the security requirements, as described in §4.3.1 "Security" and following the guidelines from §4.9.1 "Beliefs", more specifically that of decoupling the credential operations user of the credential data, especially the confidential credential data. A concrete example, assuming we are in an `EC2` cloud environment, is that of the node controller which in order to obtain information from the Amazon CloudWatch service must authenticate its request against the credentials of the developer paying for the infrastructure; in this case the node controller doesn't get the actual "*access-key*" and "*secret-key*", but instead builds the request, requests the credentials service to sign it under the requested credential, and forwards the request and the signature to the Amazon CloudWatch service.

Further design and technical details are given in section §4.8 "Credentials service", but we anticipate it by saying that our work closely follows that of Plan9's Factotum system described in [18].

4.6.7 Miscellaneous services

Component scheduler and scaler As previously expressed in §4.6.3 "Component controller", all the decisions related with matching components with the right execution node are delegated by the component controller to the component scheduler, which based the `OS` resources' requirements described for the component type in question, and corroborated with information obtained from the monitoring service regarding the nodes' status, is able to take an informed decision, thus fulfilling the requirement expressed in §4.3.1 "Scheduling".

At the same time it's counterpart, the component scaler as it continuously gathers information again from the monitoring service, but related with the components' status, and based on the expressed relation between component instances, it should be able to decide which component types, corresponding to a particular application layer such as frontend or resources, should be scaled up or scaled down depending on the demand, thus fulfilling the requirements expressed in §4.3.3 "Determinism" and §4.3.2 "Scalability". At the same time we almost get for free fault tolerance because the component scaler can see that certain component instances are unavailable, most probably due to a bug or an unreliable node, thus deciding its replacement, and such marking another requirement expressed in §4.3.2 “Fault-tolerance”.

The interaction between these three, plus a few other involved, sub-systems, as hinted before, is simple and we summarize it for completeness.

- **scheduling** — • the component controller initiates a component creation operation, • it contacts the component scheduler to obtain the appropriate node where to place it, • it then delegates the execution to the container controller;
- **scaling** — • the component scaler monitors the components, and thus the overall application status, • if it decides that a component type is under-loaded / overloaded it contacts the component controller to take action, • the component controller again delegates the selection to the component scheduler and forwards a request to the component harness / container controller.

A concrete algorithm, evaluation, and prototype have been presented in [2], and are completely in-line with the aspects described herein. ◇

Discovery service Although our various components discover themselves and interact via the component hub, the other sub-systems’ agents, using standardized protocols, like web-services, need a much lower-level, and very robust, mean of discovery, as explained in §4.3.1 “Discovery”; which taking into account the used technologies, and corroborated with the aspects presented in §4.9 “Security”, point towards the `DNS` as such a solution.

For example a component controller agent would need to know which are the available container controllers’ agents available for execution; this could be easily achieved by querying all the `DNS A` records attached to a particular `FQDN` — a configuration parameter for the platform itself established at bootstrap — and connect to them, assuming that all these agents bind on the same port; if this isn’t the case we could use the same technique as before, but querying `DNS TXT` records containing the pairs of `IP` and ports of these agents. The same approach could be used by the component controllers’ or component hubs’ agents to discover themselves and form a distributed system. We could even extend this technique to public, non-sensitive, and non-constant, configuration parameters that could be delivered in this way.

The main advantage of the `DNS` is that it’s a global, distributed, fault tolerant, and recently secure, database, which lays the foundation of the current Internet, and whose implementation is mature and ubiquitous — nearly all programming languages provide support for such `DNS` queries.

But the main defect for `DNS` is the lack of proper tools and `APIs` to manage this database, most `DNS` servers accessing information contained in textual configuration files, making it nearly impractical to programmatically update the data.

Therefore this sub-system should mainly resolve the management of a customized `DNS` server, by exporting a network `API`, of course web-service based and secure, that allows such updates. A proposal for such an implementation was already described in [3], as part of the mOSAIC FP7 project. ◇

Protocol gateways In most cases the final application doesn't sit in isolation from the real world, but quite on the contrary, the purpose of most applications, like the ones presented in §2.5 "Target applications" or §3.1 "Application case studies", is to offer to the final users useful services, so that in the end the developer would obtain some benefits, either material or moral in nature. Thus the developer, in order to reach the largest part of its intended audience, must use ubiquitous technologies, like for example those comprising the Web 2.0 stack.

Therefore it needs support from the platform — and this is also done by all the commercial platforms described in §3.2 "Platform case studies" — to ease the development process, as was anticipated in §4.3.3 "Integration", but at the same time it makes his application more robust.

As such we should at least provide ready made components, which we'll call "*protocol gateways*", for the following:

- `HTTP`, both for inbound (or reverse proxy in the correct `HTTP` terminology), but also for outbound (or proxy), for cases when the application must heavily access resources from the Internet; we need this outbound scenario, because most external services have complex behaviours which must be handled appropriately (like redirects, authentication, temporary failures) or common anti-`DoS` techniques (like throttling);
- WebSockets, as a counterpart of the `HTTP` solution;
- `SMTP`, again both for sending (`MTA`) but also for receiving (`MDA`), again for similar reasons as above;
- (optionally, thus less likely) support for streaming protocols like `RTSP`, `MMS`;
- (again optionally) although not strictly related with the current topic, but serving a similar purpose, support for various `CDN` interaction.

Related with the implementation aspects, we could firstly debate if we, as the platform's implementers should provide such features, which was addressed in the beginning of the section; and secondly we could debate if we shouldn't have provided these features as standalone `VM`s instead of components. Our reason to treat them as components is twofold, once because it gives an integrated feeling, as expressed in §4.3 ""*Integration*" – the most important requirement", and then because it is easier for us to implement and maintain.

We also underline the fact that we don't need to reimplement the protocols ourselves, but instead we could adapt already existing software, like Nginx or Mongrel2 in case of `HTTP`, and Postfix for `SMTP`, or even a `SaaS` solution as `SES` or SendGrid for outbound `SMTP` and CloudMailin for inbound `SMTP`. The adaptation would consist in making these software solutions use a standard structured data format (like `JSON`) and standard communication channel, for example a message broker like RabbitMQ — the term "*standard*" means a solution chosen by us to be used inside the platform by the developer. Of course, like in the case of some resources, these gateways running as plain components, would need an adapted life-cycle and interaction, for control and `OaM` operations, with the other components via the component hub.

The only difficult part is integrating such an approach with the way the Internet works, that is, in such a client-server architecture how would the remote clients find these gateway instances, playing the role of servers. In summary it boils down to the fact that the client uses a `FQDN`, usually entered by the user, and a "*standard*" network port to connect to the server — here the term "*standard*" means universally accepted and is usually hard coded. But our gateways, being plain components and obeying the rules of the platform, obtain from the container controller private network endpoints, thus confined to the cluster; as such we need to export them to match the criteria described before. For this there are limited options, two common ones being:

- add the `IP` addresses of the containing nodes to the `FQDN`s, and make the container ports available on the standard ports; this has several disadvantages, the main one being the fact that if a node goes down, this is very visible to the final users, and the second one is related with the involved technical complexities in achieving this;

- put a set of, hopefully more stable, customized `VMs`, acting as network layer load balancers, like HAProxy, which redirect traffic to live gateways; and make the later register themselves under a well-known `DNS TXT` record, interrogated periodically by the former to update their configuration data; (this `DNS` interaction could be delegated to the discovery service as explained in §4.6.7 "Discovery service").

Although we could argue that such a technique unnecessary complicates and slows down things, it is not true, because as demonstrated in [19] and through the common practice of most cloud providers, we don't incur a large overhead, and we certainly increase the security and robustness of the resulting application. ◇

Logging service Just summarizing what we've already expressed in §4.3.1 "Logging", we need a robust, and most likely distributed, sub-system that provides support for collecting, forwarding, storing and querying logging-events[14].

Unfortunately we haven't dug deeper into this subject, but we could point to some interesting technologies as potential solutions, like Scribe, Flume, Graylog2, or Logplex; but none of these solutions fits perfectly the structured logging events requirement. ◇

Monitoring service Similar with the logging service, and as expressed in §4.3.1 "Monitoring", we need a sub-system that would allow the collection, storage, query, and rule-based assessment, of monitoring events coming from our various sub-systems' agents and components, but it isn't tasked with measurement, as this is the responsibility of each sub-system.

Again we haven't yet got to completely identify and prototype such a system, neither did we investigate the existing solutions. ◇

Application controller We began in §4.5.1 "Application and platform life-cycle" explaining the need of an "*application controller*", which would be tasked with the tracking of the overall application's and platform's life-cycle. Indeed such a sub-system is mandatory because all the previously described ones have a narrow focus on their responsibilities and don't have the big picture of the overall application.

Such a controller would be required to monitor and coordinate the initial infrastructure acquisition, the platform's bootstrapping process, followed various services and the actual developer's components creation and configuration according to a specification describing the entire architecture and parameters of the application.

At the same time it should expose to the developer an integrated `API`, and possibly `UI`, providing an unified view of the whole platform.

But like in the previous two sub-systems, we considered this to be less important at the current stages of design and development, but we acknowledge that for the final developer this would represent an essential element, being the one he would interact most with. ◇

[14]We once more stress the difference between "*logging messages*", that is textual strings, from "*logging events*", which are data structures.

4.7 Component hub and controller

As anticipated in §4.2.9 "Component hub", §4.6.1 "Component hub", and §4.6.3 "Component controller" we shall try to detail the design of these two sub-systems, and as already accustomed, focusing mostly on the practical side, pointing only to the theoretical aspects, and giving hits about the implementation details. Also we shall proceed right into the mater at hand, without describing in detail the sub-systems' purpose and responsibilities, as they have already been covered in the mentioned sections.

Merging reasons First of all we must argue why we have decided to design and then to implement them together and not as independent sub-systems. Although it is quite possible to treat them in isolation, the underlying details are to a large extent common, from the manner in which we solve the fault tolerance or scalability issues, or the common communication mechanism and reliance on the component harness, to their common run-time environment and required libraries. ◇

4.7.1 Dynamo's model

Summary We shall make a small detour from our current goal to briefly present an important theoretical and practical work from a few Amazon's employees, which based on the peer to peer distributed data stores, namely Chord, and the `CDN` theoretical outcomes of consistent hashing, have come up with a scalable, fault tolerant, eventually consistent, distributed data store, Dynamo [14]. Although our two sub-systems are not storage related as the described solution, we shall ignore this aspect for now, until the similarity will be obvious.

Thus faced with the problem of storing large amounts of data in a reliable, but eventually consistent manner, they turned towards one of the existing solutions coming from highly distributed peer to peer systems, Chord. The idea was simple: • we first assume that the key for finding a record is immutable, which implies no renaming, thus by importing the term from `SQL` it usually must be a synthetic key, like for example an `UUID`; • then we assume that we have a deterministic hashing function taking this key and yielding an integer number between 0 and an constant upper limit, which should be extremely large, usually with a few orders of magnitude larger than the potential stored keys; in Dynamo's case this was the cryptographic digest function `SHA1`[15]; • each peer part of the system chooses a random value inside this "*ring*", that ranges from 0 up to the hash limit, and finds out which is the value of any other peer smaller than his; • thus each peer is responsible for all the keys that hash from the previous peer's value up to his own; • the Chord algorithm proposed a method, which based on random connections between the peers, would route requests to the proper peer; • thus to find any key you could have asked any of the available peers.

The only problem with this solution was that it did not have acceptable upper limits for the routing mechanism, in the worst case being proportional with the average latency between peers multiplied with $log_2(n)$, where n is the total number of peers. This was not acceptable because Dynamo's authors wanted to reduce latency within a small upper bound.

Thus they have chosen a variant of Chord and adapted it as follows: • the ring is split into a number, which must be a power of 2, of equal ranges called "*virtual nodes*"; • each peer, had to take responsibility for an equal number of virtual nodes; • as new peers were added, they would randomly select virtual nodes from existing pairs and request their handoff; • and the key stone of the entire process, as the number of virtual nodes was kept low enough that the mapping of each virtual node to its owner peer was kept in memory, each lookup involved in most cases a single

[15] The number of potential values of a `SHA1` digest is 2^{160}, which is roughly 4% smaller than the total number of atoms that compose our planet, or at least so according to `http://education.jlab.org/qa/mathatom_05.html`.

network hop.

Therefore Dynamo proposed a simple and efficient mechanism to store records based on immutable keys, all within tight latency constraints. We must add that the cited paper also proposed solutions on how to handle data replication so that peer failures did not affect the data availability, and as a consequence how to handle potential conflicts in such an eventually consistent system. ◇

Adoption So how did we use this storing technique to control and communicate with processes? We remember that we have set ourselves, in §4.3 "Requirements", the goal to build a fault tolerant platform, these characteristics applying to its sub-systems, and if possible in a peer to peer manner.

In general for such a goal there are only two related solutions, one implying the use of a master-slave architecture where the master would take decisions and forward them for durability to slaves, the other involving consensus algorithms like Paxos [13]. The link between the two is the fact that even in case of a master-slave architecture we need leader election to select the master, which in turn is obtained through such consensus algorithms.

On the other hand our solution is adopting the more simpler technique proposed by Dynamo, and adapting it as follows:

- for each component we assign a non reusable, universally unique identifier which coincides with a `SHA1` digest; we skip the hashing because we retain the uniform distribution properties ourselves; thus we treat our components, or at least their meta data, as records;
- although the cited Dynamo paper describes a "*preference list*" for each record, consisting of those peers handling the first n virtual nodes after the record key's digest, we designate the first one as the primary component controller, and the rest as alternative ones; the total number being a configuration argument influencing the high availability and consistency characteristics;
- the primary controller shall be the one taking "*definitive*" decisions and executing or delegating the actions; by "*definitive*" decisions we mean those permanently impacting the component's life-cycle such as starting or stopping;
- any of the other controllers, including the primary one, should store the meta data about the component and could be used for less impacting actions or message exchange.

Although we do not achieve complete fault tolerance, especially when faced with dead nodes or network partitioning, the other functions related to a component can still be fulfilled without the primary controller. Moreover by adopting the recovery techniques proposed by Dynamo we could recover even from such a failure, as our meta data is already distributed but not necessarily consistent.

Thus to have the complete picture of what is proposed: • each component controller has dynamically assigned a few ring ranges, each component being statically assigned to a particular range; • for each component, the controller that was assigned to its range, becomes its primary controller; • the following few controllers having assigned the next virtual nodes, become its alternative ones.

We add one last but important note, that the adoption of the Dynamo model was greatly influenced by the fact that the people working at Basho, the ones having written the key-value store Riak, have already implemented most of Dynamo as the open source library riak-core, that we have reused in our implementation. ◇

4.7.2 Component hub role

As previously described in §4.2.9 "Component hub" and §4.6.1 "Component hub", the component hub is a central part of the proposed platform, its main role being that of mediating message exchange between components and enabling discovery through group membership.

Inspiration So far we have mentioned the fact that the component hub has its underlying design and implementation heavily rooted in the approach taken by Dynamo, but there are also two other important inspiration sources, Erlang's gproc [15] and D-BUS [36], from which we have borrowed the following concepts, using for now the original terminology, which we'll translate and adapt later into concepts and features:

- **process registry** — inspired from gproc, which allows Erlang processes to register themselves either under a local, inside the same Erlang interpreter, or a global, inside the same Erlang distributed application, name either as the unique or shared holder;
- **process interaction** — again inspired by gproc, which allows processes to send messages to those registered under the previously mentioned names, either to one of them or to all sharing the same name;
- **process monitoring and linking** — half inspired by Erlang itself which allows processes to be notified by the death of another process, or it allows them to declare a hard dependency between themselves and other processes, called links, which chain process destruction in case of faults; the other half from gproc itself which allows a process to wait for a certain name to be assigned, or observe the dynamic of the process registration under that particular name;
- **property management** — a nice feature of gproc which, similar with the process registry, which allows attaching values to certain names, and as names can be arbitrary terms it means that we can form arbitrary hierarchies;
- **property monitoring** — again from gproc, in a similar manner with process monitoring, it allows processes to observe the way values attached to certain names are changed; this also resembles to what `SNMP` provides, but unlike gproc at a larger scale;
- **object addressing** — inspired by D-BUS, where one connection to a bus is able to expose multiple objects, providing one or more interfaces, each having one or more methods;
- **object signals** — again from D-BUS, one's ability to register to an object's address and receive notifications from it, enabling thus a pub-sub pattern.

And the list of borrowed ideas could continue with other techniques, mostly from newer technologies like `AMQP`, ZeroMQ, some even from `REST`, and so on. ◇

Concepts Based on the previous mentioned inspiration sources we have designed our own solution, but for starters we present our definitions that shall be used in our design, these concepts being a distillation of the original ones:

- **"*target*"** — a logical entity that is the subject of our actions, keyed by a universally unique identifier, like has been presented in §4.7.1 "Dynamo's model"; it can be concertized by either a component or a group;
- **"*component*"** — as it was already defined in §4.2.5 "Component", but we extend it to any other process that speaks the component hub's protocol; as hinted, this extension is useful to allow services or other sub-systems to present themselves as components even though they don't have the appropriate life-cycle;
- **"*group*"** — a logical entity, that provides the components with the possibility to register themselves under well known names, based on what roles they fulfill, as described in §4.3.1 "Discovery"; one component being allowed to be part of multiple groups at the same

time, and possibly join or leave as it sees fit, although we have suggested in §4.5.3 "Agent, component, service and resource life-cycle" that for simplification it should do so only once;
- "***object***" — each component is allowed to present itself as a collection of individualized elements, at least one, the root representing the component itself; this is similar with `SNMP` model, or D-BUS model, and could be useful for example for a message broker like RabbitMQ which hosts multiple exchanges, queues and bindings, thus being able for a client to interact with these as objects; we add that the objects are not organized in a hierarchy or a graph, instead they form a flat namespace, but due to the naming scheme they could be perceived as such;
- "***operation***" — similar to a method from object oriented programming, it represents an action or query that can be requested to a component's object to execute;
- "***attribute***" — the equivalent of an object, but instead of having operations, it has a value, and it is maintained and handled by the component hub itself on behalf of the component; a possible usage is for various performance metrics as a hint about the load;
- "***correlation***" — a unique token that is sent inside each request by the client, which is used to match the response; it should be globally unique, by globally we mean no other requester connected at the same hub should use the same token for any pending request, and afterwards it must not be reused at least for some acceptable time; this could be achieved by the selection at each client's startup of an `UUID`, and then concatenating a sequential counter for each request, resetting the UUID when wrapping happens for the counter;
- "***selector***" — when sending a request to a group, the selector could be used by the hub to forward the request to one or many particular components, depending on the specified criteria; for example we could allow things like closest component to the client, the current leader if we decide to implement leader election for groups, the least loaded one, and so on; currently we don't have a clear semantic for it, except "*any*" or "*all*";
- "***attachment***" — inspired by `MIME`, we allow certain requests to have binary attachments, so that certain use cases are easier to implement.

◇

Features Extending what has been described in §4.2.9 "Component hub" and §4.6.1 "Component hub", we intend to offer the following actions through the component hub's web-service `API`, but also through its channel with the components:

- "***group register / unregister***" — as already explained allows components to join or leave groups; the request consists of a set of groups, and the response lists the number of members including itself at the time of action's execution; although this semantic requires consistency it could be easily fulfilled by applying the same concept of primary component hub as in the case of component controller; the number of members could then be used by some clustered component to decide if it is the first one part of the cluster and should behave accordingly, or if there are other members and it should contact and join them instead; but corroborated with unreliability this is only a shortcut and not full proof solution;
- "***group select***" — obviously it allows querying the formed groups, and based on the selector it can return only a part of the members;
- "***group monitor***" — as inspired by gproc, it enables one to observe the dynamic of group membership;
- "***call*** **and** "***cast***" — allowing a requester to make an `RPC` like call or fire and forget signal, to a specified target and object pair, also specifying an operation, set of inputs, and possibly also attachments; the response consists of either an error, or a set of outputs and again possibly attachments; we remind that by using a selector the requester could actually trigger

operations on multiple components, and thus should expect multiple individual replies;
- "***attribute update***" — allowing each component to update, initialize, or remove the value for one or many attributes attached either to its own target, or that of any group it is member of; here "*update*" refers to actual replacement of the old value and not to an additive or subtracting semantic; furthermore no other guarantees are given like atomicity, consistency or even persistence, as these values should be used only for informative purposes;
- "***attribute select***" — it permits obtaining the values attached to some attributes for a particular target;
- "***attribute monitor***" — it enables observing the dynamic of values for some attributes of a particular target.

All these requests also have an expiration time, which means that if the component hub is unable to fulfill the request in that amount of time it should give up and return an error; at the same time it could be used by a component to know if, it should execute the operation and send a reply or not, because if the timeout elapses the client might already be gone. But this is just a hint and doesn't require the component hub or the component to either respect this timeout if it already began executing and can't back out, or it is complicated to do so.

At the same time we underline that none of these requests or responses are reliable, meaning that the component hub delivers them on an best effort basis, being the requester's and component's responsibility to implement other stronger semantics, such as retry or transactions.

As observed most of these operations involve keeping some important meta data, which we'll briefly describe in §4.7.4 "Meta data". The way in which we achieve these goals is described in §4.7.4 "Design" as the design is tied with that of the component controller role, and the same observation applies also for the actual exchanged payloads described in §A "Component hub and controller payloads". ◇

4.7.3 Component controller role

We assume that by now the component controller role is clear, especially in what concerns its requirements and responsibilities, and we reuse the same concepts and we apply similar observations as described for the component hub.

Summarizing the features, we have the following actions exposed through its web-service based `API` or through the channel towards components:

- "***component type register / unregister***" — enables the registration of a new component type, specifying the associated bundle, additional information needed by the component scheduler or the component scaler, and default configuration to be given to the component instance at startup;
- "***component instance create***" — given a component type, and possibly overriding configuration for services and the component itself, it triggers the creation of a new component; it implies generating an identifier for that component, delegating to the component scheduler the decision on which node to place it, and then delegating the startup of such a component to the container controller running on the designated node, that in turn will start the component harness; once the component signals the component harness that it has initialized successfully, the component controller answers to the original "*create*" request;
- "***component terminate***" — which sends a signal to the component notifying it that it should terminate; but it has an optional nature, as the component is allowed to refuse such a request; within the request we can specify optional inputs that could influence the shutdown behaviour;

- "***component destroy***" — which is the mandatory variant of the previous action, that involves contacting the component harness and instructing it to shutdown, then after completion or after a certain period of time, contacting the container controller and requesting the forcibly destruction of the component's container;
- "***component monitor***" — provides the means by which an interested party could observe the component's life-cycle dynamic, especially termination, and availability or offline statuses; this feature could be useful when one component depends on the services of another, and in case the dependency terminates, it could reconfigure itself to be able to continue its work;
- "***component link***" — inspired by Erlang process links, allows another component to tie its existence with that of another component; unlike the previous action, it is not always easy to reconfigure the dependency to another component, as such this action provides a more brutal, but certain approach, that of terminating the dependent component, in the hope that the component scaler would restart a new one if necessary.

We observe that some of these actions, like terminating, destroying or monitoring, could use the selector feature, by applying the same action to many targets at the same time. But because the selector is not yet well defined, and most likely such a feature would have a non-deterministic behaviour, as mandated by the requirement §4.3.3 "Determinism", we have decided to allow clients to specify just components as targets for now.

As in the case of the component hub role we continue the design in the next section and present the actual messages in §A "Component hub and controller payloads".

4.7.4 Design

Virtual nodes So far in §4.7.1 "Dynamo's model" we have said that for each component we have one primary component controller or component hub and alternative ones keeping track of meta data or acting as backups. But as Dynamo implies that each peer should handle multiple virtual nodes, does it imply that on the same node we run multiple instances of our two sub-systems? No, because we run only one instance of the sub-system's agent, inside which we have independent actor, each taking care of exactly one virtual node. We use the term "*actor*" because it is not biased towards a particular implementation technique, such as independent `OS` process, thread, asynchronous or event driven handler, but at the same time it implies a highly concurrent nature.

Therefore as said, we propose for each virtual node to run one actor, but a separate one for each different responsibility of each virtual node. Thus we have the following agents, which when implementing would translate to relatively independent modules: • the one handling component type management like registering, unregistering, or querying; • one for handling the component life-cycle like creation, destruction, or monitoring; which together with the previous one fulfill the component controller role; • another one for handling the component group membership, queries and monitoring; • then one for handling the attribute management and monitoring; • another one for call and cast operations towards targets.

Of course that we can adopt the same technique for other sub-systems, especially the monitoring service or the logging service which would so benefit from all Dynamo's characteristics, especially scalability and fault tolerance. In was in fact our initial goal to also merge these two responsibilities into the hub, but as we haven't invested much time in thinking about the implications, we have opted to leave them independent for the time being. ◇

Component clustering As we intend to use Dynamo's approach, which implies adopting the virtual node model of clustering entities, but uniformly dispersed among them, all based on the element's identifier, but because the number of our elements is extremely small in comparison with the entire available space, and because of the implication described below, we have chosen not to generate these identifiers as digests of keys, but instead to carefully craft them by respecting the overall principles and suiting our specific needs, and at the same time embedding additional information in these identifiers.

As described in [2], clustering components based on their types allows for simpler and efficient scheduling and scalability techniques, but also enables migration or communication as we shall see in what follows.

Although migration is not a concern for the current platform, we ponder about the following scenario: • what if all the components part of the same virtual node, are hosted to the same node; but at the same time one node is able to host multiple virtual ones; (here the term "*node*" represents the virtual machine where the component runs, and not the agent where runs the actor responsible for the virtual node as implied by Dynamo' model;) • then what if all the components supporting migration would, upon receiving a migration signal, dump all their state on the file-system and terminate; afterwards at a later time when they startup, and receive a special argument to restore, they read the data back from the file-system without reinitializing it; this is applicable especially to resources like data stores, message brokers and others needing persistence; • now by using the feature of disk detachment from one virtual machine and reattachment to another virtual machine, like Amazon provides through its `EBS`, we could easily migrate all components at the same time; if such a feature is not provided we can still stream the contents across the network.

As a side note, we acknowledge that there are countless other migration techniques, but to our best knowledge, the one proposed herein is the most efficient as we transfer zero data between the source and destination virtual machine, all happening as a switch in the infrastructure level. Then one might also wonder why not to apply the same technique for each component instead of clustering them; this is because on a virtual machine we can attach only a couple of disks, therefore this approach.

We could extend this clustering to form also the basis for assigning private `IP` addresses to our containers so that we can establish our own `VPN` between the nodes.

But getting back to our clustering, we propose to map the components onto virtual nodes as follows:

- all the components scheduled to run on a particular node should be part of the same virtual node; but on the same node we might have assigned multiple virtual nodes, therefore we can choose any of these virtual node; as previously described this helps in migration;
- all the component instances of the same component type should be confined to only a fraction of the available virtual nodes; this helps in scheduling as components of the same type have similar behaviour, thus on a particular node we have a smaller mix of different behaviours, which makes predictions more accurate; on the other hand because we don't know beforehand the number of component types, we allow multiple of these instances to be mixed inside the same virtual node, thus we allow overlaps.

◇

Identifiers Getting back to Dynamo' model, with its constraints such as lack of namespaces for records, usage of identifiers in virtual node mapping and request routing, and our clustering requirements presented earlier, we propose to craft the identifiers as follows:

- as Dynamo's usage of virtual nodes is inspired by Chord, the prefix of each identifier designates the primary virtual node assigned to this entity; thus about 12 to 16 bits would give us more than enough room to scale, as we can't have more nodes than virtual nodes, but at the same time we would also like to keep the routing table small enough to fit into memory;
- because we need to have unique identifiers for any piece of information, from component types, component instances, groups, and many other elements unforeseen for now, we propose to embed inside the identifier a few bits representing these categories; thus maybe about 12 to 16 bits allowing more than enough categories;
- because some elements could contain sub-elements, like for example components expose objects or attributes, we would like to leave room for the possibility to directly address these entities with their own identifier; therefore we designate an amount of bits, maybe about 32, to represent these entities; the target has all these bits set to zero;
- the remaining bits are used for target identification, and according to our figures it remains about 96, of the 160 bits that a `SHA1` digest has.

But this composition does not yet solve the problem of keeping the components of the same type clustered in the same virtual nodes. To achieve this we apply the following technique: for each component type, depending on how many virtual nodes we have and how many virtual nodes we want to assign per type, we designate a fixed value for those bits in the middle of the virtual node prefix, selecting thus only a small portion of the virtual nodes. We could have chosen the first bits but it would have interfered with the way Dynamo assigns virtual nodes to peers by creating imbalance. ◇

Communication As another piece of the puzzle we describe how the component actually interacts with the component hub or the component controller through a dedicated channel, as for an external client we expose a web-service based `API`.

As previously presented the component talks with the component harness, which is the one actually communicating with the component hub. Therefore the component harness first makes a `DNS` resolution, or uses the discovery service, to find out the endpoints of one or more component hub agents. It then establishes communication with one, or even multiple of these, declaring its identity and sending or receiving messages through any of these. The authentication used is in line to what is presented in §4.9.4 "Approach".

Inside our agent we have a specific actor, that we call the "*router*", which keeps track of these connections and informs both the primary and alternative virtual node actors that it is able to communicate, or on the contrary that it ceded to communicate, with that specific component. Therefore the component's assigned actors are able to know where to send messages to, so that they reach the component. ◇

Meta data Although we have elided so far to mention how we intend to store the required meta data, we briefly mention that we can rely on what Dynamo is actually good for, thus reusing the record storage techniques proposed by the original work. Also because Dynamo provides only eventually consistent guarantees, where the possibility of conflicts is an expected situation, we point to `CRDT` data structures [7], that provide simple and efficient techniques to handle such situations. This is also possible because our meta data requires only simple data structures, like sets for group membership, dictionaries for attributes, monitors or links, all these being covered by the previously cited paper. ◇

4.8 Credentials service

As promised in §4.6.6 "Credentials service", we shall present our proposal for handling credential data, without once more describing the responsibilities or involved concepts, as they have already been expressed in the mentioned section.

Inspiration The main inspiration for our proposal was Plan9's Factotum described in [18], where the authors were faced with a similar problem of authenticating users but keeping the needed confidential data safe. As such they came up with a simple but secure solution, about which we spend a few words to describe in what follows.

First the actual problem, which was twofold: • for once how would a user safely store his secret credential data, without needing to remember all passwords, except a master one maybe, or without reusing the same secrets for different credentials; • and, the most important issue as the previous one could easily be solved, how to allow a tool make use of this secret data, to authenticate itself on behalf of the user, without ever seeing the actual needed data, thus preventing the potential leak in case of compromised or badly written tools.

Their solution was quite intuitive, and implied creating a unified agent, that had access to all the secret data, protected by a master password, and allowed tools to request authentication as the owning user, without giving away the actual data. This agent implemented the most common authentication protocols like the one for `SSH`, `CHAP`, or a few others specific to Plan9, and exposed an `API`[16] that allowed tools to execute these protocols.

For example taking the `CHAP` authentication mechanism — used in many email protocols like `SMTP`, `POP3` or `IMAP` — when the email client connected to the server, it received a challenge, that was forwarded by the client to the agent, which in turn created a challenge response based on the secret data, and returned it to the client together with the login name, that just forwarded them to the server, thus completing the authentication without our client actually seeing any password used to create the challenge response. ◇

[16]The Factotum's `API` was a very peculiar one, as it was actually based on a synthetic file-system exposed through 9P, which enabled any tool to interact with Factotum by just reading to and writing from a couple of files.

Providers Thus we have taken a similar path, and started investigating various cloud provider's authentication mechanisms, in the hope that we would find similarities with the Factotum's use case.

Unfortunately overall we were quite disappointed, as there were four major authentication techniques, any of them having major shortcomings:

- **X.509** — like in Amazon's `SOAP API` case, similar to what we have described in §4.9.4 "Approach", but instead of using `HTTPS` with client side authentication, or instead of signing the whole message by using a standard mechanisms such as `CMS` or `SMIME`, it relied on the WS-Security extension for `SOAP` based web-services; this made it impractical to be implemented in most programming languages other than Java and a few mainstream ones;
- `HMAC` — signing the actual request based on a shared secret key, like in Amazon's `REST API` alternative [22]; an approach providing adequate security, but was non-standard in the way the `HMAC` was chained over various information;
- **time based token** — like in the case of GoGrid, which implied generating a time constrained token by applying `MD5` over a string, obtained by concatenating a shared secret and a timestamp; and although this token is valid for a couple of minutes, this technique which opens a great deal of security issues, especially because the token is independent of the request, and because `MD5` is currently considered broken;
- **pre-authentication** — like in the case of Rackspace [44], which implied that before issuing requests, we first have to authenticate and obtain a token granting further access for a certain large time frame; which had the same issues as the previous approach;
- and the weakest possible solution, requiring to send the secret data for each request.

But we don't want to get further into details about the provider's authentication mechanisms, as it suffice to say that they have chosen non standard and diverging techniques, sometime lacking proper security characteristics. ◇

Design As such our proposal is along the following lines: • we delegate the storage and usage of credential data to the credentials service described herein; • for each provider's mechanism we enable the developer to write specific code, that exposes a set of operations, which by using the credential data is able to provide authorization signatures or tokens for requests; • then the developer is allowed to load these related operations, constituting what we call a module, into the credentials service, where the code would run and access the credential data on demand; we note that the code is run in a sandboxed environment, thus it can't interfere with other modules, leak the sensitive data to the outside, or use data to which it does not have rights to; • then when a component or other service needs to make a request to a particular provider, it first generates the request message, then it calls the credentials service specifying which operation to be applied on which credential data, and obtains the needed information to authenticate or authorize the request, which is then forwarded to the provider together with the request.

We observe that, as described in §4.9.4 "Approach", it is mandatory for the clients of our credentials service to authenticate themselves. Furthermore only if permitted by the credential usage policies, are they allowed to execute certain operations against the sensitive data.

We also acknowledge that applying such a schema, it does not increase the underlying mechanism's strength, as it depends on the provider's choices, but instead it at least eliminates security concerns on our part, as we do all we can to keep the secret credential data safe. ◇

4.9 Security

Although the concept of "*security*" is very broad, in the current work we relate it primarily with authentication and confidentiality, as the other aspects of the general term, like isolation, authorization, quota, auditing, credentials, integrity, and the like, are addressed and solved by other sub-systems, which are referenced here for completeness:

- process, file-system, networking isolation, and quota is achieved by the container controller (as described in §4.6.4 "Container controller"), by employing `OS` level partitioning (`LXC` in our case), confining each sub-system's processes to a private `OS` namespace;
- network access authorization is achieved by the container controller in cooperation with the node controller (as described in §4.6.5 "Node controller"), again with the help of the `OS`, by enforcing firewall policies;
- data integrity is achieved by using each sub-system's own X.509 certificate (detailed in the current section) to sign and / or encrypt the sensitive data;
- auditing is achieved by describing each sub-system's important actions in a proper structured and signed format, forwarded to the logging service; both these events and the logging service's own records could be protected against (post facto) tampering by employing linked hashing or merkle trees (as described in §4.6.7 "Logging service");
- credentials management is handled by the credentials service (as described in §4.6.6 "Credentials service").

Whenever the term is used it should be clear from the surrounding context which of the two meanings is used — the narrow one described herein, related to certificates, or the broader one. Moreover some other more theoretical aspects related to security, like algorithm or implementation correctness, are completely unaddressed in the current work.

Therefore for the rest of the current section we restrict ourselves to the narrow semantic, detailing: • the principles that guide our decisions (see §4.9.1 "Beliefs"); • some facts that we take for granted and assume are always true (see §4.9.2 "Assumptions"); • which are the available technical options (see §4.9.3 "Options"); • which is the best suitable choice (see §4.9.4 "Approach"); • what are the challenges in adopting such a solution (see §4.9.5 "Challenges"); • some concrete technical solutions for the identified problems (see §4.9.6 "Solutions").

4.9.1 Beliefs

As seen from the previous sections all pointing back here, and as described in §4.3.1 "Security", security is a very important aspect of the proposed platform, and a hard constraint for any chance of success. As such we have decided not to treat it lightly and carefully assess and select the available options, guided by the following beliefs:

- **simplicity** — one of the basic rules in security and cryptography is to keep things as simple and transparent as possible, because the power of any such technique doesn't lay in its obscurity, but in its provable properties;
- **"*battle proven*"** — as we don't focus on security research, we don't want to come up with our own algorithms and techniques, but instead to use those that are already well established and scrutinized by specialized people;
- **reutilization** — because bugs are so easily introduced, lurking in silence until they pop out at the wrong moment, we intend to write code as little as possible for this purpose, and reuse the most mature implementations;
- **minimum exposure** — sensitive information should be exposed to as fewer agents as possible — like processes, services, or even persons — for the minimum amount of time, and if preferably they should be confined to a single sub-system;

- **maximum volatility** — all sensitive information (especially in unencrypted format) should almost always be stored and exchanged through the most hard to inspect and volatile stores or channels;
- **minimum sharing** — in order to reduce the potential implications of an information leak, all employed techniques should require the minimum amount of sensitive common knowledge, like shared secrets;
- **minimum trust** — especially when the stakes are high, like in the case of infrastructure provisioning, any sub-system must treat the others, including stores or communication channels, like potential adversary or compromised entities; thus we require that each retrieved or exchanged data must be thoroughly checked against syntactical or semantic discrepancies, and authenticated against the claimed source's identity.

4.9.2 Assumptions

Before getting into the details we need — like Archimedes' "*point of support*" — to take some facts for granted, thus creating a foundation to build upon:

- **trusted core hardware** — in case of deployments directly over physical hardware, we consider it to be tamper proof while online; meaning we assume its `RAM` can't be remotely accessed, and it's non persistent (after a fair amount of time) [17]; the same for `CPU` itself and caches;
- **untrusted peripheral hardware** — however we assume that any attached disk devices, network cards, and other `IO` peripherals, physical or virtualized, are tamperable even while online; for example we assume that disks are readable and writable at any moment by an intruder, that network cards mirror packets to, or allow injections from, an intruder;
- **trusted core virtualization** — like in the case of physical hardware, we assume that `RAM` assigned to the virtual machine by the hypervisor, is inaccessible by any other virtual machine[18], including the hypervisor, and that before it's freed and reused it is zeroed;
- **secure `OS`** — similarly we assert that the `OS` operates as specified; especially the aspects related with access control, like file-system permissions, process isolation, or capabilities enforcing, but also those facilities used in cryptographic applications like `/dev/random`;
- **flawless algorithms** — we consider that the selected algorithms — provided that currently they aren't broken, and are used correctly — are perfect, in the sens that their properties are exactly as promised (like cryptographic hash irreversibility);
- **secure implementations** — the same for algorithm implementations, we assume that their implementation reflects perfectly the algorithm, and that they don't leak information through side channels like swapped or unwiped `RAM`, timing;
- **informed developer** — and maybe the most important assertion of them all, and most of the time 100% wrong, is that the developer knows what he is doing; meaning he at least chooses his passwords carefully, he doesn't run untrusted third party code inside components, and that he has basic security and cryptographic notions[19].

[17]It is already common knowledge that such an assumption is completely wrong, given enough resources. For example as early as the 60's the `USA` government was able to surmount such attacks, and were developed under the codename `TEMPEST` (see `http://www.nsa.gov/public_info/_files/cryptologic_spectrum/tempest.pdf`).

[18]Again such an assumption is completely false, as each hypervisor had countless issues and incidents. But regardless we can't consider this problem as the center of our focus.

[19]This lack of knowledge, or carelessness, is spread all over the `IT` landscape, from highly technically inclined persons (like the `kernel.org` operators), prestigious global enterprises (like LinkedIn, Sony, and others), to governmental institutions (like the `UK`'s `NHS`).

We must note that, any seriously security inclined person, after reading the previous statements, would immediately dismiss the current section as based on unfounded myths, and sadly, this would be completely true. But unfortunately we must keep our focus on other tasks, as we can't fix every known, or unknown, problem in the computing universe, or else we'll never even start tackling the real problem at hand. As such we underline the fact that although these assertions might be wrong, any mismatch should be considered as a fault or bug in its respective source, and dealt with in a proper manner through issue reporting and patching or upgrading.

4.9.3 Options

And because the primary usage of such security techniques is for authentication and confidentiality of communication between the platform's sub-systems, we must take into account the implementation particularities:

- `HTTP` **compatible** — because most of our sub-systems expose `WS` based `API`'s, the selected solution should be easily (or preferably already) integrated into the current `HTTP` stack;
- **run-time availability and diversity** — as the language of choice for implementing sub-systems ranges from Erlang, through Java, and ending with Python, any selected solution should be available in such programming languages.

Therefore the obvious, and maybe single, solution given the constraints and requirements, is `SSL` with X.509 certificates, as it could be used for either mutual authentication, confidentiality, even for storage, and there are many mature implementations with bindings for most programming languages.

But before getting into details, we should note that there are other mature options, but which unfortunately aren't as ubiquitous as `SSL` combined with X.509:

- for mutual authentication we could have used a Kerberos based solution, which is the de facto method in enterprise for service to service authentication; unfortunately it is poorly integrated into existing `HTTP` stacks of various run-times, and it doesn't help in case of confidentiality;
- a better replacement for X.509 certificates would have been OpenPG compliant solutions, easing at least the generation and management of certificates; but as in the previous case it lacks almost any integration with `HTTP` stacks, and availability in most run-times.

4.9.4 Approach

As suggested in the previous section our choice was to adopt X.509 certificates for mutual authentication, potentially with existing or custom extensions (like MyProxy for delegation), plus various other related technologies like `SSL` for transport security, or `CMS` (previously `SMIME`) for storage.

Thus for each individual sub-system's agent we'll need an unique X.509 certificate, clearly stating the following information:

- the application (or platform) instance wherein the agent runs; that should be mapped onto the `DN`'s `DC` attribute of the certificate's `Subject` property;
- the sub-system of which the agent is part of; mapped onto the `DN`'s `O` attribute;
- the identity of the agent, which most likely would be an universally unique non deterministic (`UUID`-like) token; mapped onto the `DN`'s `UID` attribute;
- the identity of the node where the agent resides, which in our case would be the `FQDN` of the node; mapped onto the `DN`'s `OU` attribute, although in general this would have been mapped elsewhere;

- the network endpoints where the agent is expected to listen, as it might expose multiple incompatible protocols, which would be a list of `URL`'s composed of the expected application level protocol name, transport level protocol name, and in general, another `FQDN` — obtained by concatenating the agent's identity and the node's `FQDN` — plus a port; mapped both as a `URI` values of the certificate's `subjectAltName` property, but for integration with current application stacks, also as `DNS` values of the same property containing only the concatenated `FQDN`;
- the capabilities available to this agent, with a format and under a property yet unspecified.

We must note that there was a potential open problem concerning the integration of such a scheme with some technologies, like those based on `SSL` as `HTTPS`, because on the same node, having exactly one externally usable `IP` address, we'll have to host multiple agent endpoints on different transport layer ports, thus the endpoints being identified by the `FQDN` and the port. But even though all the agents have an individual `FQDN`, they all resolve to the same `IP` address, and therefore one endpoint has multiple aliases — taking a particular agent as fixed, then any other agent's name (from the same node) plus the fixed port would resolve to the fixed agent's endpoint. But because `SSL` is always used in conjunction with `FQDN`'s, and because tools verify the target name with the one from the certificate, one agent can't impersonate another one — which would be the case if we wouldn't have had unique `FQDN`s for each agent.

As observed from the previous paragraph we have mentioned the existence of a set of capabilities declared for each agent. These are declarations about what actions should the agent be allowed to request, and should be verified by the receiver of the request. For example — anticipating what is described in the next sections — when the certificate manager receives a request for a new certificate generation, it should verify that the requester is allowed to request certificates for the target sub-system. For now we propose to limit ourselves to just role based authorization.

4.9.5 Challenges

Even though the average technical person considers `SSL` — actually the implied, hidden X.509 infrastructure as a foundation, which is actually the problematic part — as a panacea for all security problems, it can't be furthest from the truth, and as anticipated in the previous section it poses a great deal of problems to get it right[20]:

- **`CA` necessity** — the most problematic part, both from a security, management, and even cost, point of view, is the need of a centralized `CA`, which must first verify and then sign each new certificate; the security aspect is related to how we are certain that a `CSR` is actually coming from the designated requester, moreover it seems impossible to automatize this task (without lowering the certainty); the management aspect refers to the fact that now the platform would have to run a secure and automated `CA` instance; that is because of the cost, we can't rely on external `CA` services, as in such a case — even if they would skip the presumed thorough validation, which is actually their main role — each certificate would cost in the range of tens of dollars;
- **certificate generation** — because X.509 uses `PKI`, the usual case being RSA, generating a new certificate implies generating a new private key of an adequate length — which according to [45] that is about 2048 bits — and this in turn requires a high quality random data source, like `/dev/random`; unfortunately such a source has a rate of a couple of bytes per second, which rises the total generation time to at least a few seconds (not taking into account that other applications might use such a data source); (see §4.9.6 "Generation";)

[20]Actually we don't try to say that we got it right, but instead we describe our solution which, to our best knowledge, is valid.

- **certificate signing** — the next step is generating a `CSR` and forwarding it to the `CA` for signing; but because our `CA` is automated we need a way to authenticate the `CSR` itself; (see §4.9.6 "Signing", §4.9.6 "Initial handoff" and §4.9.6 "Subsequent handoff";)
- **certificate disposal** — when an agent is terminated (gracefully or abruptly) or when an agent is suspected of being compromised — determined either by the developer himself, or through various heuristics — we need to mark the issued certificate invalid and ensure it won't be trusted anymore; (see §4.9.6 "Certificate disposal";)
- **reliance on `DNS` in case of `SSL`** — that implies an almost perfect synchronization between the `DNS` registry and the issued certificates;
- **private key manipulation** — in the end the most sensitive element of a certificate is its private key, which in order to be used by any agent must be handed out in a secure manner, and in general this private key is embodied as a `PEM` file; naively we could say that such a file should be encrypted with a secret key, but then handing down this secret key would become a similar problem, and so on; but regardless of the hand out, once loaded it will be held in the agent's `RAM` in plain text, thus susceptible to leakage through core dumps or intrusive `RAM` inspection; (see §4.9.6 "Private key handling").

4.9.6 Solutions

As previously seen there are sufficient road blocks in implementing such a solution. But still, in the current section we present some of the employed solutions, starting with the easiest ones, and moving upwards[20].

Generation The problem of adequate entropy collection rate, can easily be solved through a compromise, putting in balance the generation speed and the obtained security, thus replacing `/dev/random` with `/dev/urandom`. Unfortunately the analysis of the consequences of such a decision are out of the scope of the current work, but based on the fact that a similar approach has been adopted by others, we believe it is secure enough to suit our purposes. We add in our defense that such certificates aren't to be exposed directly to the Internet, but to be used inside the cluster, thus the main purpose is protecting against very unlikely malicious actions from the provider, or against unlikely malicious components. ◇

Signing `CSR` authentication could easily be solved by applying the following observation: any sub-system's agent comes into existence as the consequence of another sub-system's agent's action, which must at least trust the authenticity of the executed code, and thus know its indented purpose. For example, a component is executed by the component harness, which in turn is started by the container controller, which in turn is started by the node controller, which in the end is among the first services running on the node. As such we propose the following:

- each sub-system that is able to spawn another one's agents, should have the appropriate capability described in its own certificate, including which are the types of sub-systems that it's allowed to spawn;
- we introduce another type of service called "*certificate manager*", that should be started on each node among the first processes; its main purpose being the generation and signature delegation to the CA service for new certificates, as requested by other allowed sub-systems' agents;
- at startup both the certificate manager and the node controller obtain via some secure mechanism (more about this below) their certificates to be used in further interactions;

- each time a sub-system's agent is about to spawn another one's agent, it should first determine its type, and request to the certificate manager a new certificate of the designated type, receiving in return a non deterministic secret token to be passed to the newly created agent;
- when an agent is started, after it receives the token, it contacts the certificate manager to obtain its own certificate; an operation which could be repeated multiple times.

◇

Initial handoff All that remains to be solved is bootstrapping the node controller and certificate manager with an adequate certificate, which could be pragmatically (and secure enough) solved by applying the following steps:

- most cloud providers allow when starting a new virtual machine for the client (in general through an `API`) to specify some limited amount of arbitrary textual data, called "*user data*", accessible from within the node (again through an `API`); thus using this channel we can transmit two non deterministic secret tokens — one for the node controller and another for the certificate manager — plus the certificate and the endpoint of the CA service; we note that it is not required for this data to be confidential; (if not using a cloud provider, but for example physical hardware, we could use other means to convey this data, like for example the kernel's parameters if booting through `PXE`, `DHCP` custom options, or even `DNS` by attaching this data to `TXT` records at a well known, but unique for each node, `FQDN`;)
- after retrieving this data, the two agents generate their private key, and a create the `CSR`;
- then each agent contacts the CA service — over `HTTPS` ensuring that the server's certificate is signed by the implied CA service — and transmit the token and the `CSR`; in return it obtains the actual certificate;
- immediately after retrieval the CA service invalidates the two tokens to prevent further such requests;
- (optionally, see afterwards for rationale) after retrieval, and a small delay, the CA service could try to contact back the two services on the designated endpoints (specified in the certificates just delivered) to see if they are actually in possession of the two certificates, and if not immediately revoke them.

Unfortunately, like in the case of any security related system, the bootstrapping process, and especially an automated one, is prone to vulnerabilities, as in the current case where there is a small time window, between the time the virtual machine is started, and until the proper agents come and try to claim their certificates based on the respective tokens. It could happen that a malicious entity — maybe the cloud provider, another cloud user that was able to break the provider's infrastructure, or a malicious process running on the virtual machine — could intercept the token and request for itself the certificate. This could be mitigated in the following way (only required for the two initial certificates): when the CA service creates the two certificates it sets their validity starting from that moment plus a small validation timeout; if the agents don't request their certificates, or the CA service is unable to verify that the certificates reached their proper owner until the specified timeout, the CA service revokes the two certificates.

The disadvantage of such a scheme is that we can choose a longer validation timeout, so that a slow starting node would be able to receive the certificates in time, but a fast starting one would have to wait until after the validation period has passed to use its certificates. This too can be solved by allowing the CA service to reissue certificates, valid from the moment of the verification, and pushing them to the services, but this would complicate things.

In a similar manner we could ask ourselves what happens if the CA service itself gets impersonated — which requires altering the bootstrap data. We could say that this isn't such a big problem, because if all agents require that the CA service that signed their certificates is the same one that signed the requester's ones, then the two peers won't be able to authenticate each other, thus aborting and logging the issue. ◇

Subsequent handoff Similar, but simpler than, in the case of initial handoff, when a sub-system is about to instantiate another one, it first contacts the certificate manager and makes a request for a new certificate on behalf the new agent. The certificate manager, after completing the generation and signature delegation, stores the new certificate (composed of private key, public key, and other meta data), and returns to the initiator a non deterministic secret token. The token is then passed by the parent agent to the newly created one, together with other configuration items, for example through environment variables, command line arguments, or inside a well known configuration file.

Before we describe how exactly does the newly created agent obtain the actual certificate from the token, we motivate why we haven't chosen to send back directly the certificate itself, thus simplifying the whole process. Unfortunately, as described in the beginning of the section, we must ensure that all sensitive data is handled in a secure manner, and all the previously mentioned methods of handoff are unsuitable. For starters, using the command line arguments, we leak the information to any other process on the same node through a simple invocation of the `ps` utility; the same in case of environment variables by inspecting `/proc /$pid/environ` (or `/proc /$pid/cmdline` for the former)[21]. By using files we make things even worse, as not only are they as visible as with the previous mechanisms, but they are also persistent beyond the lifespan of their owner process. Moreover the former methods (through environment variables or arguments) are unsuitable because the size of the certificate could easily take a few kilobytes.

Beside the previous disadvantages, transmitting a token instead of the certificate itself, also has the advantage of decreasing startup time: instead for the certificate manager to wait for the CA service to sign the new certificate, which is a remote operation, it could instead just start the whole process, generate the token associated with the outcome, and return it immediately. Thus until the new sub-system's agent starts and actually needs the certificate, the whole process could be over; also it opens other possibilities like certificate renewal.

Afterwards obtaining the certificate is quite simple, as the new agent just contacts the certificate manager over a well known channel — like a UNIX domain socket, maybe specified via the same mechanism as the token — presents the token and obtains the certificate. But maybe a better technical solution, one that is consistent with storing and handling the certificate, is by leveraging the Linux's key retention service as presented below. ◇

Private key handling According to the security maxim which states that "*the strength of any security system is at most as the strength of its weakest link*", our main attack vector is actually the manner in which we store and use the sensitive information for the lifetime of the agent. And unfortunately the underlying `OS`, and most existing libraries or tools don't help much in this respect. For example we would say that at least 90% of the all software products handling X.509 certificates expect them to be delivered as paths to files on a file-system, a method whose problems we have described in the previous section. Furthermore we are unable to determine if we can safely remove those temporary files or not, because its unspecified if the used software doesn't need maybe to access the certificate again.

A second, as important, problem is that most software usually loads the certificate's private key into `RAM` at startup, even though it sparely uses it — and most of the time it could be leaked

[21]Only recently (as of January 2012) we have a feature for Linux (`hidepid=[ 1 | 2 ]`) that allows us to hide all information about processes not owned by the current user (see `https://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=0499680a42141d86417a8fbaa8c8db806bea1201`).

through swapping because they don't use `RAM` pinning like `mlock` on Linux.

As such, as a solution to this problem we propose to use Linux's key retention service[30], which allows us firstly to decouple sensitive information initialization (in our case handoff), and secondly to store it securely while unused. The whole mechanism is quite simple: • the Linux kernel provides the concept of keyrings, list-like structures that point either to keys or other keyrings; in fact it provides multiple alternative ones scoped at different levels — one for each user, one for each session, or one for each process or thread; • a key is nothing else than an arbitrary blob of data which is to be handled in a secure manner;[22] • then it provides the `request_key` system call which allows a process to query the existence of a certain key, and if affirmative allows it to retrieve the attached data; • if the key doesn't exist, it also provides for a transparent method in which an external program (`/sbin/request-key`) initializes this key without the calling process even noticing.

We could thus reuse this facility as follows: • we arrange that the newly created sub-system agent when in need of the certificate, it tries to resolve the corresponding key, by using the `request_key` system call with the token as part of the query argument, key which has attached as data the contents of the certificate; • previous to that, we arrange that the `/sbin/request-key` tool knows how to interrogate the certificate manager, forwarding the token in question and initializing the required key; • if needed we could make the key expire after a certain amount of data, forcing upon subsequent usage its reinitialization, thus allowing usage tracking, revalidation or recreation.

Unfortunately we can't always apply the previous method, because as stated, most software solutions expect the certificate to be specified in terms of a file-system path. But even for this problem we have a workaround, meaning we could easily implement a module that exports a synthetic file-system exposing the contents of such keys as files, thus satisfying the requirement of most software. (The feasibility of such an approach was debated with one of the Linux's developer in a short email based dialogue[49].) ◇

Certificate disposal And as a final touch, we must also solve the certificate disposal problem, either as result of agent's death, or compromise. Luckily the X.509 standard provides two options: `CRL`'s and `OCSP` responders, which unfortunately aren't widely used. But regardless, playing on the safe side, we run both solutions in the hope clients use it.

Furthermore, by making the lifespan of a certificate small enough, like say a couple of hours, we obtain a similar effect as having used `CRL`'s, but on the down side we need agents to cooperate and request new certificates before the lifespan elapses. ◇

[22]Actually the model is more complex, a key having a well defined type, name, search description, protection, expiration time, and attached structured data, which in the case of "*user keys*" it just happens to be an arbitrary blob of data.

Chapter 5

Implementation

So far having focused mainly on the theoretical aspects of the proposed platform, as presented in §4 "Design", especially within topics such as §4.3 "Requirements", §4.6 "Sub-systems" and §4.9 "Security", in the current chapter we concern ourselves mostly with practical problems. But unlike the mentioned chapter, where the design was distanced from that of the European FP7 research project mOSAIC, in the current one we stick closely to our experiences gathered during the implementation phase of the project. Therefore the reader might find discrepancies between the details presented herein, and the proposals in the initial design; but we remind that those details are actually an extension or small divergence from the project's course, thus any details presented here are still applicable in implementing the envisaged platform.

Because we believe, by slightly adapting the Chinese proverb, "*a few lines of code say more than a thousand words*", we won't get too in detail, but as already accustomed we just present the overall picture, insisting upon the important issues that we have encountered, or interesting less known facts. About the source code itself, the reader is pointed towards §A "mOSAIC platform repositories" for further details.

Thus we start, in §5.1 "Guidelines", by presenting some general rules and principles that guided our choice of technologies and then the platform implementation. We then proceed, in §5.2 "Backing technologies", to present the main technologies that we have chosen to use, insisting less on their description, and more on useful features, and potential problems. In §5.3 "Outcomes" we give a high level overview of exactly what has been developed as part of the mOSAIC project, and especially those sub-systems where the author had an important contribution.

5.1 Guidelines

Before starting we remind that this section is about practical or technical choices and not about functional ones, as the later have already been presented in §4.3 "Requirements".

As such we summarize our guidelines, and which are the constraints they translate to, as follows in no particular order, but admitting that some of these guidelines are dependent or derive from one another.

Low overhead Any employed technology should use an acceptable level of resources, especially related to `CPU` or `RAM`, as although we can buy many virtual machines, the characteristics of one particular instance can only be stretched so much; this implies that in those places where the development process is not hampered, low level languages like C / C++, or even Go and other similar ones, should be used. Although this may seem as an "*premature optimization*", this choice would deem to be right once we have a dozen of sub-systems crammed inside the same node.

This also applies to the required data files, which should be kept to a minimum both in number and in size, to allow quick deployment of our sub-systems from archives obtained over the network. ◇

Bounded consumption An extension of the previous guideline, if possible we should use technologies where either we are able to accurately estimate the maximum used resources, especially `RAM`, better yet where these resource consumption is constant, or which at least allow us to set a cap on the usage. This is important as in an automated environment we want to be able to accurately determine if there is room for one more agent, or we should request a new node.

Indeed a solution to this problem could be limiting the maximum resources at the `OS` level, but this would just serve as a fail safe mechanism, and would forcibly terminate any process exceeding its quotas, thus not helping much in solving the problem. ◇

Minimal dependencies Because we intend to have a lightweight platform, suitable to be run either in a cloud environment, on owned hardware where we don't have virtual images, but also in a local testing environment, the resulting `OS` footprint should be minimal, which in turn translates to as few installed packages as possible.

Another way to look at this constraint is to imply that our sub-systems shouldn't depend on packages or other artifacts existing in the host `OS`, because sometimes we are unable to choose our own `OS` distribution. As such we could try to have either self contained run-time environments or statically linked binaries.

Related to the previously mentioned idea, we could also require that sub-systems are relocatable, that is it should not matter where on the file-system the files live, as long as they are kept together; this is based on the observation that various Linux distributions have different folder paths for similar purposes. ◇

Robustness Because we run in a potentially unreliable environment, any used technology should expect to encounter and properly handle the following situations: • network connectivity problems, when the sub-system is unable to find its required services; • network fuzzing, when a sub-system receives from the network unexpected, malformed or semantically incorrect messages, as they could originate from a broken peer, or possibly a hostile; • unexpected node termination or reboot, because especially in a cloud environment, the backing virtual machines are not guaranteed to stay alive, and because of various reasons outside the control of the developer, could be destroyed or restarted.

This should also imply that the programming language chosen to implement a certain sub-system is mostly suitable for such scenarios. ◇

Modularity and simplicity We once more remind what has been presented in [17], that a sub-system should be always split into smaller cooperating processes, each one doing only one task well.

And the second suggestion from the same book, that helps modularity, is keeping things as simple as possible, and complicating them only when needed. The reverse usually happens when over engineering a certain problem, especially induced by the nature of chosen programming languages.[1]

◇

Bounded tool suitability And last but not least, a rule we always need to remember, that each one of our tools is suitable for only a limited set of related tasks, thus it follows that we can't solve all our implementation problems with only one programming language, or only one technology.

As such diversity is key in easily completing our tasks, and at the same time obtaining an efficient outcome. Therefore in what concerns the mOSAIC project, we could say we have accomplished this, as we mix a wide palette of technologies, as presented next, of course each in its proper place.

◇

5.2 Backing technologies

In the current section we try to list some of the most prominent technologies keeping on their shoulders the entire platform and the hosted application. We do not intend to give an in depth description, as this could be easily obtained either from their own sites or for a general overview from Wikipedia. Instead we describe how we have used them and which were the main features that mandated their usage, but at the same time we also intend to document the issues encountered while using them, and if possible the actions we have taken to mitigate them.

We must mention that the current section reflects mostly what has been implemented so far as part of the European FP7 research project mOSAIC, by the author or in cooperation with other project's team members. In some cases, but make explicit in the respective paragraphs, the described technology was not yet used, but because of its qualities we either intend to use it, or to slowly migrate existing code to it, in the near future.

5.2.1 Programming languages

Erlang As presented in §4.1 "Inspiration", Erlang [16], plays a major role in our platform, both by influencing some of our design decisions, but especially by backing some sub-systems like for example the component hub and component controller, and as presented in next sections many other solutions embedded in our platform.

In essence Erlang allowed us to easily build a network enabled, highly concurrent and fault tolerant sub-system, by leveraging its run-time environment `OTP` [43], the features having already been presented in §4.1 "Inspiration".

Unfortunately on the down side using Erlang, actually `OTP`, in a distributed manner proves to be challenging in a dynamic and constrained environment, like the one in our platform. For example when hosting Riak as a component inside a container, and if we want to build a cluster of such components, they need an Erlang specific discovery process called `EPMD` that exposes an `UDP` port; the problem is that all Erlang processes that are part of the same cluster must use the same `EPMD` port; moreover this problem appears only when on the same node we host multiple instances of the same component, and part of the same cluster, as each tries to run its own `EPMD` instance, all fighting over the same `UDP` port. This leaves us with three possible solutions:

- we run `EPMD` on each node, one for all possible Erlang based solutions; which unfortunately

[1]Java, and most mainstream programming languages, are in general pushing the developer into over engineering.

means that one component would be able to impersonate another one running on the same machine;
- we run one `EPMD` on each node but handling only those component of the same type and part of the same cluster; this implies a special role for this service, which complicates the platform needlessly;
- we run inside each component's container its own `EPMD` instance; but this implies that each container has an unique `IP` address routed throughout the cluster.

A forth alternative, one that we have investigated, was to update the Erlang's `OTP` code to allow us to plug in an alternative resolver, one using our component hub. This has been concertized in an `RFC` and proposed patch sent to the Erlang community [48], which until now has remained unanswered. ◇

NodeJS NodeJS is a new programming language, slowly becoming mainstream, as for example its GitHub repository had the second place in order of popularity at the beginning of 2012. Because it is based on Google's V8 JavaScript virtual machine, and because it is a fully asynchronous programming language, we have mainly used it for building various frontends.

At the same time, almost half of our showcase application, presented in §5.3.3 "mOSAIC "*Real time feeds*" application", has been built with this language, because it allowed us a rapid prototyping pace, and yielded acceptable performance levels.

Moreover because of the same reasons as in the previous case, the initial prototype of the credentials service was also prototyped in NodeJS.

We don't have many negative words against NodeJS, except maybe the immaturity of some libraries, the fact that only recently it started to slowly stabilize its interfaces. ◇

C and Go Very few parts have been written in this programming language, mostly just those very close to the `OS` such as the component harness, but we intend to slowly migrate these to Google's new programming language Go.

As said Go is not currently in use, but it seems to be a promising alternative to port those tools currently written in C.

In either of the cases a major role for choosing them has been played by the fact the applications written in both of them can be statically linked into an executable requiring absolutely no other run-time dependency, thus ensuring that our most important tools would continue to run regardless of the `OS` distribution, version or installed packages, provided that the `OS` kernel and architecture remained the same. ◇

Python Although not favoured by the author, but heavily used inside the mOSAIC project, the Python language is the language of choice for non-critical sub-systems, like for example the container controller or discovery service.

The author's main grudge against it is the lack of self containment, as seen in the case of other programming languages like Go, and even Erlang. The main problem is that the interpreter can't be statically linked, thus making the binary executable usable only on the `OS` and distribution where it was compiled.

All in all, it still provides a good choice for prototyping and implementing even those sub-systems close to the `OS` as it is mature enough and provides a wide range of libraries, one could argue even wider than any of the previous presented languages. ◇

Bash Yet another author's favourite pet peeve is the Bash scripting language that powers almost the entire mOSAIC platform build system, and most bootstrap scripts for various sub-systems.

Although a very good choice for rapid prototyping, orders of magnitude quicker for development than any of the previous languages, Bash shows its teeth only when it comes down to debugging, maintenance, extension or corner cases.

Because Bash lacks almost any form of error handling features, it makes the worst choice for any task involving an unreliable environment. But after investing enough time in trial and error iterations, it becomes an almost reliable option.

We could conclude that maybe the time spent maintaining and fighting with debugging, negates the advantage of rapid prototyping. Therefore it could be an option that in the near future to replace its usage with Python, Ruby, even Lua, or any other sane scripting language. ◇

5.2.2 OS, distributions, virtualization

Linux, distributions Beside the fact that the mOSAIC team was acquainted with Linux better than with other UNIX-like alternatives, it seems the de facto choice when it comes to cloud computing, thus mandatory for us to adopt it. But in the current section we won't give details related with the kernel itself, but instead about the issues encountered with its distributions, cloud providers, and other strictly technical aspects.

One of the platform requirements is fast response time, even in the case of its startup. But unfortunately there are two great bottlenecks which prevent us from obtaining a fast starting system: • provisioning a virtual machine from any cloud provider usually takes from a couple of tens of seconds to a couple of minutes, and unfortunately we can't do anything about it; • today's mainstream distributions like those descending from Debian or RedHat, take up to a minute to start, and this is without additional platform related sub-systems.

Because we could have influenced only the `OS` startup procedure, one of our first decision was to investigate a possible solution for such a speedup; and at the same time we wanted to reduce as much as possible the size of the initial virtual machine image, and moreover to be able to boot such an `OS` directly over the network in a `PXE`-enabled environment.

Unfortunately our conclusion was that none of the previously mentioned main stream Linux distributions allowed such a trim to be applied, the minimum size obtained was in case of a custom Fedora installation, which still had over 100 MB without any of our packages. Therefore we have turned towards specialized distributions like Slitaz which allowed a fully functional image of about 30 MB that booted almost instantly. Another solution could have been TinyCore, AlpineLinux or even a customized Gentoo version.

But before deploying it on various cloud providers we hit yet another problem, as a large number of providers, especially the youngest ones, only allowed a set of predetermined virtual images to be started.

Therefore currently the mOSAIC team is trying to move back to a mainstream distribution, but applying what we have learned in our experiment. It seems that with its latest `LTS` version, 12.04, Ubuntu is providing a minimal installation, named Ubuntu Core. ◇

LXC As detailed in §4.6.4 "Container controller", `LXC` is enabling us to isolate the running components and other services.

Until so far no large issues have been raised against this solution, except maybe the lack of advanced tools integrating it with other sub-systems of the Linux kernel, like the firewall or networking; and especially the fact that for guaranteed security it must be used in conjunction with one of the `LSM` solutions like SELinux or AppArmor. ◇

5.2.3 Load balancers, proxies

None of these solutions has currently been put into use as part of the mOSAIC platform, but because they are necessary for a final version of the solution, we have investigated their usefulness and potential issues. Moreover they are the most mature solutions, or at least from those available as open source.

HAProxy HAProxy is a transport layer load balancer, supporting both `HTTP` or any other `TCP` based protocol, focusing on throughput and low latency. It is the de facto solution in such cases, being recommended by RightScale in their documentation, and even by the makers of Riak and RabbitMQ as an intermediary layer between the clients and the server behind enabling seamless failover.

The only problem we have identified is related with dynamic configuration of load balanced endpoints, as these need to be specified in a static configuration file. This leads to the cumbersome solution of writing such a configuration file formatter and then delivering signals to the daemon to reload its configuration. ◇

Nginx Powering in January 2012 over 12% of the Internet, according to Netcraft's report, Nginx is one of the most popular open source `HTTP` servers, second to Apache's `HTTPD`. In our case this could prove useful for implementing a `HTTP` gateway, as described in §4.6.7 "Protocol gateways".

As in the case of HAProxy the main issue is related with dynamic configuration, but a similar solution could be employed. ◇

Mongrel2 A very young yet promising project, Mongrel2 is a language agnostic `HTTP` server whose main purpose is that of proxying `HTTP` request towards various backends, thus matching very closely what we have described in §4.6.7 "Protocol gateways". Therefore we spend some time describing its functions and how they could be used in our advantage.

The most peculiar feature is the way in which it communicates with the background servers, not through an already existing standards such as FastCGI a late descendant of the `CGI` era, and neither through reverse proxying as is the norm in most other web servers. But instead once requests have been parsed they are forwarded with the help of a ZeroMQ socket, and when the backend is ready it sends back the response through another such socket. Because it relies on ZeroMQ, a technology we shall take a closer look to in a later section, it allows for interesting behaviours, like for example a backend can chose to become available or unavailable at any time by disconnecting itself from the ZeroMQ socket, but still being able to send the replies back for the pending requests; or a backend could register to multiple servers at the same time ensuring thus that if the server dies it can still fulfill requests from clients via other servers. Even more interesting is the fact that the backend giving the response could be a totally different than the one receiving the request; or the fact that the same response be used as response for multiple outstanding requests.

And on the `OaM` side Mongrel2 is the only server, to our knowledge, that is configured through a database, specifically SQLite, allowing thus easier automation of management tasks. ◇

5.2.4 Distributed databases

This section is about RDBMS alternatives, informally named NoSQL solutions, but only for a selection of the possible choices, namely those that are quite easy to embed in our platform and provide a distributed storage solution. Thus although there are many alternatives to any of these we have chosen them mainly because constructing a cluster of them implies no special roles, all started instances being equal.

Riak and Cassandra Implementing a key value store, Riak is one of the first resources provided as components as part of the mOSAIC platform. Cassandra, although not yet embedded in our platform is a good candidate for providing a columnar database to our developers.

Both of them are based on the paper [14] that stands at the base of Amazon's S3, which implies that scaling such a distributed resource implies nothing more than creating a new component and joining it with its ring peers.

The main issue encountered for both of them is that joining peers must be manually done by the operator through a CLI tool. Thus our task was to automatize the joining procedure, which in case of Riak was almost trivial before the 1.0 version, starting from there the procedure involves a more complex state tracking. In case of Cassandra we haven't yet investigated exactly the implications but we expect to be similar with Riak.

As a final remark, one of the core parts of Riak, namely riak-core is also the central part of our component controller and component hub, as we have reused their code with great success. ◇

CouchDB According to today's NoSQL glossary, CouchDB is a document oriented database, allowing users to store JSON documents and attachments, and querying them either by the document key, or through map-reduce aggregations.

It is a distributed database not in the sense of scalability, but of reliability, providing a very useful master-master replication procedure.

This feature has made us start using it to keep all the data our platform needs, data that either is of static nature, or which doesn't change too often. Thus we intend on each of our nodes to run such a CouchDB instance and to replicate it with one, or multiple, central ones for reliability.

Unfortunately we don't expect a lot of applications to need CouchDB as a database, as such it is not a priority for us to provide it as a component. ◇

5.2.5 Messaging systems

The following two technologies are the communication heart of our own platform or hosted applications, and the same for countless other `PaaS` solutions.

RabbitMQ Implementing the 0.9 version of the `AMQP` protocol and model, RabbitMQ is an invaluable message broker especially for the hosted applications, allowing them to communicate asynchronously in various patterns.

Although it supports a distributed environment, running a couple brokers part of the same cluster, we currently support inside the platform only the single broker version. The reason is that because RabbitMQ uses Erlang's Mnesia distributed database, automatically scaling up, and especially down, the cluster is not an easy task involving complex state tracking. ◇

ZeroMQ Even though it is presented as a messaging middleware, ZeroMQ is nothing more than an advanced socket library, with high ambitions of getting embedded into the Linux's kernel.

We use it mainly because it abstracts away the network both in what regards transport and issues like reconnecting and timeouts, but also because it allows various communication patterns to be built, like described in [55].

The only issues we have with ZeroMQ is related to its heavy ties with an UNIX environment, making it unsuitable for developers using Windows on their workstations, thus giving us a hard time in providing a local testing environment not involving virtual machines.

The second issue is related with its stability, as even though with the recent releases it has improved in this respect, there are still corner cases which if hit trigger assertion failures taking down usually the entire application, a very unpleasant experience for us as developers. ◇

5.2.6 Data formats

Sometime this aspect is greatly underestimated, especially in distributed environments involving multiple programming languages. Data formats are maybe the main enabling, or on the contrary impediment, in connecting sub-systems.

In general they fall under two categories: • **self describable, textual formats** — suitable for low frequency, and small payload communication, with the advantage of being human readable, thus easing debugging and testing; the most cited example in this category is `XML`, or in recent years its contenders `JSON` or `YAML`; • **compact, efficient, binary formats** — suitable for high frequency, low latency, but still small payload communication, being recommended by their compactness and efficient parsers; usually they are called serialization formats and there are countless solutions, at least one for each programming language out there.

JSON Although it is nothing more than the textual representation of JavaScript data structures, it is our interoperability data format of choice, way in front of `XML`.

Our reasons are described below, and contrasting it with `XML`, we add that in any single respect `XML` does not fulfill them: • **simplicity** — the full specification of `JSON` is only a couple of pages long, contrasted with the tens of pages of all the disparate `XML` related specifications, that are needed to have a complete and self contained description of the standard; • **pervasiveness** — as it is simple, for any programming language, there are at least a few parsers and formatters available, and if not writing one is trivial; this compared with `XML` for which in some programming languages there isn't an implementation supporting the entire specification; • **low overhead** — because we are speaking of communication, it proves suitable as a medium sized message encoded in `JSON` is a couple times smaller than `XML`, and quite close to an efficient binary representation; • **matches perfectly with**

data structures — because it allows only primitive values, and lists or dictionaries, it can be converted perfectly to any programming language as native data structures, and even for example in Erlang where dictionaries are not available it integrates with the environment as lists of pairs; contrasted with `XML` which has its document model implying complex object hierarchies suitable mostly for object oriented languages.

Moreover because our chosen database to host the platform's data is CouchDB, `JSON` provides a perfect match even for configuration values. ◇

ProtocolBuffers Developed by Google and being almost exclusively used inside their systems, ProtocolBuffers is a perfect choice for binary communication.

Its main advantage, as in the case of `JSON` is pervasiveness, Google providing support for parser generation in Java, Python, C++ and their Go language, but others have already ported it to most other programming languages. This is in contrast with other solutions that are either programming language specific, or has a narrower spread. ◇

5.3 Outcomes

As said, the implementation described herein is part of the work done in the context of the European FP7 research project mOSAIC, thus in the current section we intend to cover a short description of the software artifacts actually developed, and available as free open source code, the repositories being given in §A "mOSAIC platform repositories".

We mention that all the following modules were either written or adapted by the author.

5.3.1 mOSAIC component controller and component hub

This module is a prototype of what has been described in §4.7 "Component hub and controller", in short a distributed message bus used by components to discover and communicate with one another. Also at the current state, this module also embeds a simplified component harness, and has its own node discovery system based on either `UDP` multicast or `DNS`, allowing it to be started on most Linux based systems without many dependencies of its own, except the language run-time.

The code as written fulfills the following roles, not necessarily in line with the design previously presented as the design is an extension and cleanup of what has been prototyped: • automatically discovers other nodes part of the same platform's cluster; • allows components to be created, destroyed or queried; • as said, component discovery and interoperation; • collects logs from all components and services running on the same node; • manages the `TCP` or `UDP` port allocation for components; • exports most of these functionalities through a `HTTP` based service, although not quite `REST`-ful; • and at the same time it provides a nice `WUI` interface for the developer.

The implementation was done in Erlang by reusing the libraries riak-core, Mochiweb, and Webmachine, among many others. The `WUI` was implemented in NodeJS and relies on the exposed web-services.

5.3.2 mOSAIC resource and components

The following are the initial resources provided as part of the mOSAIC platform, together with accompanying components called "*drivers*" inside the project, which have the role of abstracting the semantics for each resource type to a set of common operations among multiple implementation. This allows then the developer to create resource independent code, provided that he does not need functionalities specific to only a particular implementation.

The identified resource types are: • key-value stores, as implemented by Riak, Membase, Voldemort, and a few other open source solutions; • distributed file systems, as implemented by Hadoop, GridFS, to some extent `S3`, and others; • message queuing systems, as exposed by `AMQP` implementers such as RabbitMQ, or `JMS` providers; • columnar databases, as implemented by Cassandra or Hadoop, although to some extent the common denominator is not very wide.

mOSAIC HTTP gateway The first of these components is not actually a resource, but a gateway prototyping what has been described in §4.6.7 "Protocol gateways".

Its purpose is very simple, when a `HTTP` request gets in, it parses and forwards the request in a custom format comprised of a `JSON` envelope with all the request's meta data and a binary payload representing the request's body. The forwarding is towards an `AMQP` topic exchange, provided by our RabbitMQ component, with a correlation identifier, exchange and routing key to send the response back to. The handlers — which currently are either an adapted Jetty servlet which implements this protocol or any component compliant with these rules — create a shared queue registered to that particular topic exchange catching all requests; the plan was to use the routing key derived from the `URL`, thus allowing different types of handlers. Once the request was handled it sends the response to the designated target, and acknowledges the request.

As a result of the above implementation, more exactly of the `AMQP` characteristics, a very interesting feature appeared, that is if a handler has crashed, the `AMQP` broker will forward the unacknowledged request to another handler, thus having a "*poor man's*" fault tolerant `HTTP` stack.

The implementation was done in Erlang by reusing the Misultin embedded web-server among other libraries. ◇

mOSAIC RabbitMQ component This is nothing more than a customized version of the `AMQP` message broker RabbitMQ. The customization consisted in preparing a specialized build system for it, and injecting a small amount of code to control the startup and respond to the component controller's messages.

Although RabbitMQ provides a clustered working mode it has not been used. Unfortunately because of our customization and lack of spare development time our version is already a few generations old lagging behind the upstream stable one. ◇

mOSAIC Riak component In a similar way with the RabbitMQ component, this one adapted the distributed key-value store Riak to be used inside the platform.

Unlike the previous one, we have relied on Riak's distributed ability, as using it was quite simple, at least in version 0.14. But as above, our customized version lags behind the stable one. ◇

5.3.3 mOSAIC "*Real time feeds*" application

One of the first applications, better said show case application or demo, built on top of the prototyped platform was a small `ATOM` or `RSS` feed alerter.

It had a simple idea behind it which went like this, there are a lot of interesting news sites that continuously publish new stories on various topics, but the only way to get updates in real time is through a poll mechanism. The poll mechanism implies that either manually from the browser, or automatically through some `cron`-like job, the user would constantly have to fetch the content and make a comparison with the previous version to see what has appeared in the meantime. We mention that all this applies also to feed streams like `ATOM` or `RSS`.

Therefore we wanted to provide users with a push method which somehow, detailed below, would identify what has been changed and immediately announce the users via some communication channels like mail, thus `SMTP`, chat, thus `XMPP`, or even through a Web 2.0 interface.

The code for this sample application is pointed out in §A "mOSAIC "*Real time fees*" application repositories".

Solution The solution was not too complicated, as in general having no other chance, we have decided to go with the poll mechanism ourselves, periodically checking for new content, and if we discovered any new items to alert the user.

But then what is the catch? How did we do it better? The answer lies not in the technique, but in the fact that if we expose this as a public service, where many users would come in and subscribe their interests, we would obtain a clustering effect, that is there would be a great overlap between the user choices. As such we could do the polling only once per source and alert all the interested parties. Moreover we can apply an adaptive polling solution, adjusting the intervals based on both the rate at which new items are posted, but also based on the popularity of a certain source.

On the other hand most of the popular sources, usually social media like Twitter, open other possibilities by streaming the updates to subscribed applications, thus eliminating the need of polling, but requiring computing power to process the stream, and sometimes even charging for the data. ◇

Architecture We roughly describe the architecture depicted in §5.1 "Real time feeds architecture" starting from the user's request submitted through the `WUI`:

- the user triggers via the browser a request for updates specifying the interested source `URL` and the sequence number of the latest seen item; this happens continuously at an acceptable interval;
- this request reaches our `HTTP` gateway, which parses and forwards it via a queue to one of our frontend instances;
- the frontend retrieves the `ATOM` or `RSS` source `URL` from the request, and submits an asynchronous message to the fetcher to notify it that currently there is someone interested in that particular source; then without waiting for feedback it checks the content already parsed and stored in the timeline store, retrieving only those items which were created since the specified sequence number; we are aware of the fact that we give back to the user stale data, but for

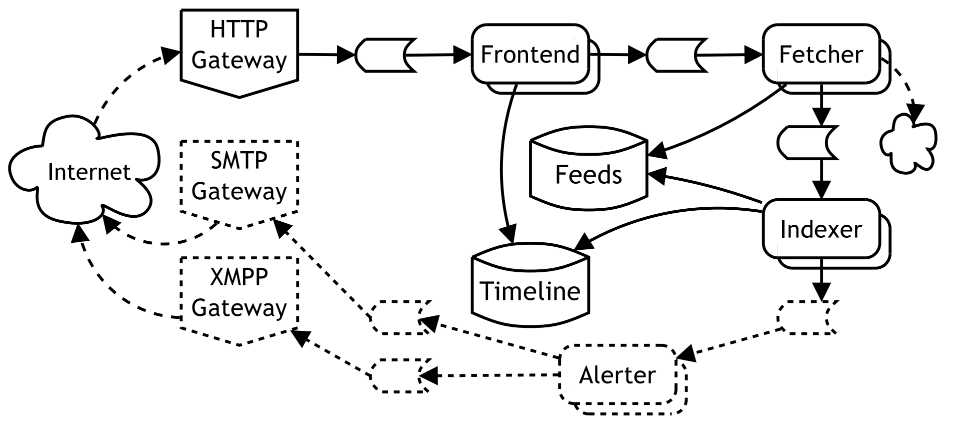

Figure 5.1: Real time feeds architecture
The connector objects are queues. The dashed lines are direct communications outside the cluster. The dotted shapes are not yet implemented.

a frequently changing feed this would induce only a small lag, while on more stable ones it would have no visible effect;

- the fetcher first checks inside a cache, currently in memory visible only for that particular fetcher, and then inside the feeds store, to see if it has already tried recently to download the same `URL`, and if not it starts fetching it; it does this process in parallel up to a certain number of feeds to limit both the load on itself and not to trigger `DoS` rules on content provider's side; once the raw contents is fetched, and if it differs to the one previously had, it writes it into a feed store and forwards a notification to the indexer specifying the key to the new content;
- the indexer fetches the new feed content, parses it, and compares the obtained timeline with the previously stored one; if new items are found they are stored and forwarded to a queue for other interested components;
- the various alerters consume the same message and push alerts via various gateways, like `SMTP` or `XMPP`;
- although not depicted in the figure, there is another component that periodically checks all the known subscriptions, for those users that want to receive alerts through email for example and not use the `WUI`, and based on how frequently the source changes and which was the last time it was fetched, pushes notifications to the fetcher.

This simple architecture allows a wide flexibility in the modeled workflow, and at the same time it is highly scalable, as we can easily start or stop instances of components in every layer without causing service disruptions. ◇

Observations Unfortunately although the previously described architecture is quite sound, while implementing it we hit a lot of bumps which we summarize in what follows, most of them pertaining to be considered as future work:

- `HTTP` **polling** — although we have eliminated the need to poll the same source for each client, we have moved this to our frontend, but this is related with the limitations of Web 2.0 technologies, as there are two main solutions, one to use long polling, and the other one to use WebSockets; although both are valid, the first one although is supported by our `HTTP` gateway, it would have raised either scalability or development issues on the Jetty based servlet container; meanwhile the second solution is not supported by our gateway, and is still emerging even in the browser's world; thus we conclude that this problem, although not an easy nut to crack, is not inherent to the chosen architecture but to the implementation techniques of our gateway and frontend;

- `HTTP` **generic fetcher** — as we have implemented our fetcher we have come to the conclusion that this would have been best approached by implementing a generic `HTTP` fetching component as it has nothing explicitly related to `ATOM` or `RSS`; this would help the developer as this is not an easy task, having to take into account both the `HTTP` protocol peculiarities like temporary failures, redirects, conditional fetching, transfer encoding, and others, but as mentioned before also throttling and other common limits as imposed by most content providers;

- `AMQP` **delivery consistence** — because each of our frontends sends an unconditional request to the fetcher to retrieve the source if stale, and because these requests are forwarded through a queue we have an important synchronization issue; as if there are many users interested in the same source, most of the time we would have in our queue multiple fetch requests for the same `URL`; but because RabbitMQ routes these messages in a round robin fashion, different fetchers would receive a similar request almost in parallel, which means that fetchers would need to synchronize themselves in order to not duplicate the work, but this synchronization raises other scalability issues; on the other hand there is a RabbitMQ custom exchange type that by using consistent hashing over the routing key, would always route the same type of request to the same consumer, except for a small window when new ones register or unregister, thus allowing the fetcher to ignore duplicate requests; an even better solution would have been another customized RabbitMQ exchange which would track similar pending messages and ignoring duplicates, but this would be harder to implement especially in the distributed context;
- **lightweight atomic store** — another solution for the previous problem would have been an atomic data store, not necessarily distributed, that would allow fetchers to take ownership of a particular `URL` download job; also a timeout option for records would have proven useful for automatic cleanup of the accumulating data; examples of such implementation would be Memcached or even Redis.

In conclusion although the current prototyped platform allows us to build scalable applications, it still lacks some resources or customized features to ease the development and efficientize the execution. ◇

Chapter 6

Conclusions

6.1 Summary

In retrospect the current work has tried, and hopefully succeeded, to present a few aspects related to building an open source `PaaS` solution.

We have looked from an economical, business and developer perspectives, and tried to better understand the factors that push such a solution in one direction or another, sometimes even in opposite directions. Then we have presented what are the most pertaining applications, their generic architecture, and evolution over time to the current form, starting thus to gather requirements for our design.

Then to have a more concrete image of what is indeed needed, we have tried to gather architecture and implementation details about some real world applications, but popular ones, thus certainly demanding. Although we have found some of these, unfortunately we did not find enough detailed documentation, but at least we deduced their high level design from the scarce sources.

Before starting to think about our own architecture, we have turned once more to the real world for examples, this time to existing platforms, but unfortunately as in the case of applications, we did not found enough details about their internals. At least we have found some interesting ideas.

Then before presenting the design we have given definitions to the main concepts, in order to have a clear idea about what we are speaking of. Now having a common vocabulary, and after seeing what is needed, we have put together a set of requirements that would guide our decisions, but also validate the final implementation.

Based on those requirements, we have proceeded to give a high level architecture of the proposed platform, followed by a design of the most important sub-systems. Afterwards we have given more attention to the security issues, and detailed the design of two sub-systems completely handled by the author.

Next we have presented the technologies and encountered issues while implementing a prototype of the proposed platform, but not before highlighting the guidelines that together with the requirements directed our selection of technologies and the development.

In conclusion we once more underline that the current work was written while the author was involved in the European FP7 research project mOSAIC, thus part of the concepts and design presented herein was conceived together with members of the project.

Still the author has tried in the current work to present an extended design than of the project's, and in some respects even diverging and taking a different path than the original one. But the implementation details are mostly as experienced by the author while working on the project.

Unfortunately the endeavour is not over yet, because the implementation taken as a whole is still mostly a prototype, many sub-systems needed to be finalized.

Thus we conclude the work with two extra sections, one about what are the future plans, and one giving credit where credit is due.

6.2 Open problems and future work

As said there is still enough work to be done. Starting with the design, we have treated superficially some of the sub-systems like for example monitoring, logging, application controller and others, although they are quite essential for a useful platform. On the other hand the security analysis and proposals must be more thoroughly checked and preferably their properties proven with formal logic.

Then on the implementation side a lot remains to be done, finalizing those sub-systems still in development, starting others from scratch, hopefully replacing brittle scripts with ones in more robust languages, and many other issues.

But maybe the most difficult task still remaining is to convince possible developers to first trust us, and then give our platform a spin followed by their invaluable feedback.

6.3 Acknowledgements

Although the last section of the work, this position does not imply its importance, on the contrary as stated at least a couple of times by now, the current work would not have existed without the help or insight of the people mentioned herein.

First of all as the current work has been written under the umbrella of the European FP7 research project mOSAIC, its contents could not have been written if the project would not have existed. Therefore the first person on our list is prof. dr. Dana Petcu, the scientific coordinator of the project, and the director of IeAT, which has tirelessly worked during the inception and early days of the project ensuring we were on the right track, both from a scientific and organizational point of view, and continuing in a similar manner after the project has bootstrapped. Then, still related with the mOSAIC project, it couldn't have been possible without the involvement of the entire IeAT team that sparked the project's idea after a few initial brainstorming sessions. Namely a great impact was made by the following persons: Marian Neagul which gave insight on the `OS` and development issues and at the same time a counter balance for the times we forgot that cloud computing is nothing magical; then Silviu Panica which provided the infrastructure and `OS` distributions related information, and prepared the customized `OS` and a local testing infrastructure; Georgiana Macariu which working on the `API`s and some resource components always reminded us of the projects scope, thus keeping us grounded in our goal. But all four, other IeAT members, myself, and other project partners have together shaped the project's direction. I would definitely want to add here Massimiliano Rak from Seconda Universita di Napoli, that also played an important role in the project's design and especially the platform's, which at the same time proved to be a perfect match for interminable technical discussions, which made the other members mad.

And in the end strictly related with the current work, I mention once more prof. dr. Dana Petcu which played the role of coordinator, pressing me to finish it, and giving feedback on the way. At the same time I would like to add two persons that played the role of "*guinea pigs*" as I have forced them to proofread the work, my IeAT colleague Adrian Craciun, and my girlfriend Andreea Pitici. Plus the indirect contribution of a former colleague and friend Radu Cugut, that made me aware of most economical and business aspects presented in the first chapter, always challenging me to find a non-technical reason in every decision, especially those involving the developer.

Thus once more a big warm thanks to all those that helped me, directly or indirectly, in writing this work, from current to former colleagues, close friends to partners, and even those people I have exchanged emails with discussing about encountered issues.

Bibliography

References

[1] Dana Petcu et al. 2012.
"Portable Cloud Applications — From theory to practice" (cited in pg. 32).

[2] Marc E. Frincu and Ciprian Craciun. 2011.
"Multi-objective Meta-heuristics for Scheduling Applications with High Availability Requirements and Cost Constraints in Multi-Cloud Environments".
http://dx.doi.org/10.1109/UCC.2011.43 (cited in pg. 61, 70).

[3] S. Panica et al. 2011.
"Serving legacy distributed applications by a self-configuring cloud processing platform".
http://dx.doi.org/10.1109/IDAACS.2011.6072727 (cited in pg. 61).

[4] D. Petcu et al. 2011.
"Building an interoperability API for Sky computing".
http://dx.doi.org/10.1109/HPCSim.2011.5999853 (cited in pg. 32).

[5] Dana Petcu, Ciprian Craciun, and Massimiliano Rak. 2011.
"Towards a Cross Platform Cloud API — Components for Cloud Federation".
http://dblp.uni-trier.de/rec/bibtex/conf/closer/PetcuCR11 (cited in pg. 32, 35).

[6] Dana Petcu et al. 2011.
"Architecturing a Sky Computing Platform".
http://dx.doi.org/10.1007/978-3-642-22760-8_1 (cited in pg. 31, 32).

[7] Marc Shapiro et al. 2011.
"Conflict-free Replicated Data Types" (cited in pg. 72).

[8] Robert N. M. Watson et al. 2010.
"Capsicum: practical capabilities for UNIX".
http://research.google.com/pubs/archive/36736.pdf (cited in pg. 57).

[9] Gianni Fenu and Simone Surcis. 2009.
"A Cloud Computing Based Real Time Financial System".
http://dx.doi.org/10.1109/ICN.2009.71 (cited in pg. 6).

[10] Ian T. Foster et al. 2009.
"Cloud Computing and Grid Computing 360-Degree Compared".
http://arxiv.org/abs/0901.0131 (cited in pg. 6).

[11] Peter Mell and Tim Grance. 2009.
The NIST Definition of Cloud Computing.
http://www.csrc.nist.gov/groups/SNS/cloud-computing (cited in pg. 6, 7).

[12] Luis M. Vaquero et al. 2008.
"A break in the clouds: towards a cloud definition".
http://dx.doi.org/10.1145/1496091.1496100 (cited in pg. 6).

[13] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007.
"Paxos Made Live: An Engineering Perspective".
http://dx.doi.org/10.1145/1281100.1281103 (cited in pg. 40, 65).

[14] Giuseppe DeCandia et al. 2007.
"Dynamo: Amazon's highly available key-value store".
http://dx.doi.org/10.1145/1323293.1294281 (cited in pg. 46, 64, 89).

[15] Ulf T. Wiger. 2007.
"Extended Process Registry for Erlang".
http://dx.doi.org/10.1145/1292520.1292522 (cited in pg. 66).

[16] Joe Armstrong. 2003.
"Making Reliable Distributed Systems in the Presence of Software Errors".
http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf (cited in pg. 17, 31, 85).

[17] Eric S. Raymond. 2003.
The Art of UNIX Programming(cited in pg. 52, 85).

[18] Russ Cox et al. 2002.
"Security in Plan 9".
http://static.usenix.org/events/sec02/cox.html (cited in pg. 60, 72).

[19] Joe Armstrong. 2000.
"Increasing the realibility of email services".
http://dx.doi.org/10.1145/338407.338532 (cited in pg. 63).

[20] Colin J. Fidge. 1988.
"Timestamps in Message-Passing Systems that Preserve the Partial Ordering".
http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf (cited in pg. 40).

[21] Leslie Lamport. 1978.
"Time, Clocks, and the Ordering of Events in a Distributed System".
http://dx.doi.org/10.1145/359545.359563 (cited in pg. 40).

Miscellaneous

[22] *Amazon Web Services Glossary – Signature Version 4 Signing Process.*
http://docs.amazonwebservices.com/general/latest/gr/signature-version-4.html (cited in pg. 73).

[23] *Applying the Unix Process Model to Web Apps.*
http://adam.heroku.com/past/2011/5/9/applying_the_unix_process_model_to_web_apps (cited in pg. 34).

[24] *Architectural Design of mOSAIC's API and platform.*
http://www.mosaic-cloud.eu/dissemination/deliverables/FP7-256910-D1.1-1.0.pdf (cited in pg. 31, 32, 51).

[25] *Cloud Computing Use Cases.*
http://cloudusecases.org/ (cited in pg. 7).

[26] *Cloud Foundry – a Developer's Perspective.*
http://www.slideshare.net/mccrory-me/cloud-foundry-a-developers-perspective (cited in pg. 29).

[27] *Diaspora Goes Zero Ops with Heroku.*
http://success.heroku.com/diaspora (cited in pg. 7).

[28] *Downwardly Scalable Systems.*
http://www.welton.it/articles/scalable_systems (cited in pg. 42).

[29] *Fallacies of Distributed Computing.*
http://rgoarchitects.bit.ly/4z9L3w (cited in pg. 17).

[30] *Get started with the Linux key retention service.*
http://www.ibm.com/developerworks/linux/library/l-key-retention/index.html (cited in pg. 81).

[31] *Goodbye Google App Engine.*
http://www.carlosble.com/2010/11/goodbye-google-app-engine-gae (cited in pg. 7).

[32] *Heroku Blog — Polyglot Platform.*
http://blog.heroku.com/archives/2011/8/3/polyglot_platform (cited in pg. 26).

[33] *Heroku Blog — The Big Kickoff.*
http://blog.heroku.com/post/list?page=32 (cited in pg. 23, 32).

[34] *Heroku Dev Center — Stacks.*
https://devcenter.heroku.com/articles/stack (cited in pg. 26).

[35] *High Scalability — 7 Lessons Learned While Building Reddit to 270 Million Page Views a Month.*
http://highscalability.com/blog/2010/5/17/7-lessons-learned-while-building-reddit-to-270-million-page.html
(cited in pg. 19, 20).

[36] *Introduction to D-BUS.*
http://www.freedesktop.org/wiki/IntroductionToDBus (cited in pg. 66).

[37] *Life of an App Engine Request.*
http://www.youtube.com/watch?v=oAMMBP_SacA (cited in pg. 30).

[38] *LinkedIn — A Professional Network built with Java Technologies and Agile Practices.*
http://www.slideshare.net/linkedin/linkedins-communication-architecture (cited in pg. 21, 22).

[39] *LinkedIn — Communication Architecture.*
http://www.slideshare.net/linkedin/linked-in-javaone-2008-tech-session-comm (cited in pg. 21, 22).

[40] *Linux Capabilities: making them work.*
http://ols.fedoraproject.org/OLS/Reprints-2008/hallyn-reprint.pdf (cited in pg. 56).

[41] *Logs Are Streams, Not Files.*
http://adam.heroku.com/past/2011/4/1/logs_are_streams_not_files (cited in pg. 40, 53).

[42] *Open Cloud Manifesto.*
http://www.opencloudmanifesto.org/ (cited in pg. 6).

[43] *Open Telecom Platform.*
http://ericssonhistory.com/Global/Ericsson%20review/Ericsson%20Review.%201997.%20V.74/Ericsson_Review_Vol_74_1997_1.pdf
(cited in pg. 32, 85).

[44] *RackSpace Manuals – Authentication.*
http://docs.rackspace.com/servers/api/v1.0/cs-devguide/content/Authentication-d1e506.html
(cited in pg. 73).

[45] *Recommendation for Key Management.*
http://csrc.nist.gov/groups/ST/toolkit/key_management.html (cited in pg. 77).

[46] *Reddit — And a fun weekend was had by all. . .*
http://blog.reddit.com/2010/03/and-fun-weekend-was-had-by-all.html (cited in pg. 20).

[47] *Reddit — January 2012 / State of the Servers.*
http://blog.reddit.com/2012/01/january-2012-state-of-servers.html (cited in pg. 19–21).

[48] *RFC: On "inet_tcp_dist" and "erl_epmd" interaction.*
http://erlang.org/pipermail/erlang-questions/2011-October/062004.html (cited in pg. 86).

[49] *RFC: Using "user" keys for userspace tools.*
http://marc.info/?t=134022095900003&r=1&w=2 (cited in pg. 81).

[50] *RightScale — EC2 Site Architecture Diagrams.*
http://support.rightscale.com/12-Guides/EC2_Best_Practices/EC2_Site_Architecture_Diagrams
(cited in pg. 23, 24).

[51] *The Java EE 6 Tutorial — Development Roles.*
http://docs.oracle.com/javaee/6/tutorial/doc/bnaca.html (cited in pg. 11).

[52] *The Twelve Factor App.*
http://www.12factor.net/ (cited in pg. 23, 25).

[53] *Why We Moved Off The Cloud.*
http://code.mixpanel.com/2011/10/27/why-we-moved-off-the-cloud (cited in pg. 7, 8).

[54] *Your Coding Philosophies are Irrelevant.*
http://prog21.dadgum.com/142.html (cited in pg. 43).

[55] *ZeroMQ — The Guide.*
http://zguide.zeromq.org/ (cited in pg. 40, 90).

Appendix A

Repositories

The current work's repository The source code of the current work itself is available under the GNU Free Documentation License version 1.3, at `http://github.com/cipriancraciun/masters-thesis`. ◇

mOSAIC platform repositories The source code is provided as open source under the Apache License version 2.0, and the author's forks can be found in the following locations:

- `http://github.com/cipriancraciun/mosaic-distribution`
- `http://github.com/cipriancraciun/mosaic-node`
- `http://github.com/cipriancraciun/mosaic-node-wui`
- `http://github.com/cipriancraciun/mosaic-credentials-service`
- `http://github.com/cipriancraciun/mosaic-components-httpg`
- `http://github.com/cipriancraciun/mosaic-components-riak-kv`
- `http://github.com/cipriancraciun/mosaic-components-rabbitmq`
- `http://github.com/cipriancraciun/mosaic-erlang-tools`
- `http://github.com/cipriancraciun/mosaic-blueprints`
- `http://github.com/cipriancraciun/riak_core`

The official mOSAIC project developer's page is `http://developers.mosaic-cloud.eu/`. ◇

mOSAIC "*Real time fees*" application repositories Similar with the code for the entire platform, it is available at `http://github.com/cipriancraciun/mosaic-examples-realtime-feeds`. ◇

Component hub and controller payloads Because the proposed solution is still a work in progress we point the interested persons to the following `URL` where the technical details of these two sub-systems are roughly described: `http://wiki.volution.ro/Mosaic/Notes/Hub`. ◇

Appendix B

Acronyms

AMQP	*Advanced Message Queuing Protocol*	FQDN	*Fully Qualified Domain Name*
API	*Application Programmable Interface*	FSM	*Finite State Machine*
ASP	*Active Server Pages*	GCE	*Google Compute Engine*
ATOM	*Atom Syndication Format*	GUI	*Graphical User Interface*
AWS	*Amazon Web Services*	GWT	*Google Web Toolkit*
BIOS	*Basic Input Output System*	HMAC	*Hash-based Message Authentication Code*
BPEL	*Business Process Execution Language*	HTML	*Hyper-Text Markup Language*
BPMN	*Business Process Model and Notation*	HTTPD	*(Apache) HTTP Daemon*
CA	*Certificate Authority*	HTTP	*Hyper-Text Transfer Protocol*
CAPEX	*Capital Expenditures*	HTTPS	*HTTP Secure*
CDN	*Content Delivery Network*	IaaS	*Infrastructure as a Service*
CEE	*Common Event Expression*	IDE	*Integrated Development Environment*
CGI	*Common Gateway Interface*	IMAP	*Internet Message Access Protocol*
CHAP	*Challenge Handshake Authentication Protocol*	IO	*Input / Output*
CI	*Continuous Integration*	IPC	*Inter Process Communication*
CLI	*Command Line Interface*	IP	*Internet Protocol*
CMS	*Cryptographic Message Syntax*	IPS	*Intrusion Prevention System*
COW	*Copy on Write*	IT	*Information Technology*
CPU	*Central Processing Unit*	J2EE	*Java 2 Enterprise Edition*
CRDT	*Conflict-free Replicated Data Type*	JMS	*Java Message Service*
CRL	*Certificate Revocation List*	JSF	*Java Server Faces*
CSP	*Communicating Sequential Processes*	JSON	*JavaScript Object Notation*
CSR	*Certificate Signing Request*	JSP	*Java Server Pages*
DDL	*Data Definition Language*	JVM	*Java Virtual Machine*
DHCP	*Dynamic Host Configuration Protocol*	LAMP	*Linux + Apache + PHP + MySQL*
DML	*Data Manipulation Language*	LDAP	*Lightweight Directory Access Protocol*
DN	*Distinguished Name*	LSB	*Linux Standard Base*
DNS	*Domain Name System*	LSM	*Linux Security Module*
DNSSEC	*DNS secure*	LTS	*Long Term Support*
DoS	*Denial of Service*	LVM	*Logical Volume Manager*
EBS	*(Amazon) Elastic Block Service*	LXC	*Linux Containers*
EC2	*(Amazon) Elastic Compute Cloud*	MD5	*Message-Digest Algorithm v5 (RFC 1312)*
EPMD	*Erlang Port Mapper Daemon*	MDA	*Mail Delivery Agent*
ERB	*Embedded Ruby*	MDC	*Mapped Diagnostic Context*
ESB	*Enterprise Service Bus*	MIME	*Multipurpose Internet Mail Extensions*

MMS	*Microsoft Media Server*
MTA	*Mail Transfer Agent*
NAS	*Network Attached Storage*
NAT	*Network Address Translation*
NIST	*National Institute of Standards and Technology*
NoSQL	*No SQL*
OaM	*Operations and Maintenance*
OCSP	*Online Certificate Status Protocol*
OPEX	*Operational Expenditures*
ORM	*Object Relational Mapper*
OS	*Operating System*
OTP	*Open Telecom Platform*
PaaS	*Platform as a Service*
PEM	*Privacy Enhanced Mail*
PKI	*Public Key Infrastructure*
PNAT	*Port Network Address Translation*
POP3	*Post Office Protocol v3*
POSIX	*Portable Operating System Interface*
PXE	*Preboot Execution Environment*
QA	*Quality Assurance*
QoS	*Quality of Service*
RAM	*Random Access Memory*
RDBMS	*Relational Database Management System*
REST	*Representational State Transfer*
RFC	*Request For Comments*
RIA	*Rich Internet Application*
ROR	*Ruby on Rails*
RPC	*Remote Procedure Call*
RSS	*Really Simple Syndication*
RTSP	*Real Time Streaming Protocol*
S3	*(Amazon) Simple Storage Service*
SaaS	*Software as a Service*
SAN	*Storage Area Network*
SCTP	*Stream Control Transmission Protocol*
SES	*(Amazon) Simple Email Service*
SGE	*Sun Grid Engine*
SHA1	*Secure Hash Algorithm v1*
SLA	*Service Level Agreement*
SMIME	*Secure / Multipurpose Internet Mail Extensions*
SMTP	*Simple Mail Transfer Protocol*
SNMP	*Simple Network Management Protocol*
SOAP	*Simple Object Access Protocol*
SOA	*Service Oriented Architecture*
SQL	*Structured Query Language*
SRP	*Single Responsibility Principle*
SSH	*Secure Shell*
SSL	*Secure Socket Layer*
TCP	*Transfer Control Protocol*
TDD	*Test Driven Design*
UDDI	*Universal Description Discovery and Integration*
UDP	*User Datagram Protocol*
UI	*User Interface*
UML	*Unified Modeling Language*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
UUID	*Universally Unique Identifier*
VLAN	*Virtual LAN*
VM	*Virtual Machine*
VPN	*Virtual Private Network*
WSDL	*Web Service Definition Language*
WSGI	*Web Server Gateway Interface*
WS	*Web Service*
WUI	*Web User Interface*
XML	*Extensible Markup Language*
XMPP	*Extensible Messaging and Presence Protocol*
YAML	*Yet Another Markup Language*
ZFS	*Zetta Byte File System*

Appendix C

Glossary

9P	http://en.wikipedia.org/wiki/9P
AlpineLinux	http://alpinelinux.org/
Amazon	http://www.amazon.com/
Android	http://www.android.com/
Apache	http://apache.org/
AppArmor	http://apparmor.net/
Apple	http://www.apple.com/
aufs	http://aufs.sourceforge.net/
Azure	http://www.windowsazure.com/
BaseCamp	http://basecamp.com/
Bash	http://en.wikipedia.org/wiki/Bash_%28Unix_shell%29
Basho	http://basho.com/
BigTable	http://research.google.com/archive/bigtable.html
BOOM	http://boom.cs.berkeley.edu/
BSD	http://en.wikipedia.org/wiki/Berkeley_Software_Distribution
Btrfs	http://btrfs.wiki.kernel.org/
Capsicum	http://www.cl.cam.ac.uk/research/security/capsicum
Cassandra	http://cassandra.apache.org/
C	http://en.wikipedia.org/wiki/C_%28programming_language%29
C++	http://en.wikipedia.org/wiki/C%2B%2B
CentOS	http://centos.org/
Chord	http://pdos.csail.mit.edu/chord
Clojure	http://clojure.org/
Cloud Foundry	http://cloudfoundry.org/
CloudMailin	http://cloudmailin.com/
Amazon CloudWatch	http://aws.amazon.com/cloudwatch
collectd	http://collectd.org/
Common Lisp	http://en.wikipedia.org/wiki/Common_Lisp
Condor	http://research.cs.wisc.edu/condor
CouchDB	http://couchdb.apache.org/
C#	http://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29
curl	http://curl.haxx.se/
Databus	http://www.linkedin.com/
DataNucleus	http://www.datanucleus.org/
D-BUS	http://www.freedesktop.org/wiki/Software/dbus
Debian	http://www.debian.org/
DevOps	http://en.wikipedia.org/wiki/DevOps
Diaspora	http://joindiaspora.com/
Django	http://www.djangoproject.com/
Doozer	http://github.com/ha/doozerd/
DotCloud	http://www.dotcloud.com/
DotCom bubble	http://en.wikipedia.org/wiki/Dot-com_bubble
.Net	http://www.microsoft.com/net
Dynamo	http://en.wikipedia.org/wiki/Dynamo_%28storage_system%29
EBay	http://www.ebay.com/
Eclipse	http://eclipse.org/
Erlang	http://www.erlang.org/
Facebook	http://www.facebook.com/
Factotum	http://plan9.bell-labs.com/magic/man2html/4/factotum
FastCGI	http://www.fastcgi.com/
Fedora	http://fedoraproject.org/
Flume	http://cwiki.apache.org/FLUME
FP7	http://cordis.europa.eu/fp7
FreeBSD	http://www.freebsd.org/
App Engine	http://developers.google.com/appengine
GDocs	http://docs.google.com/
Gentoo	http://www.gentoo.org/
Git	http://git-scm.com/
GitHub	http://github.com/
GMail	http://gmail.com/
Go	http://golang.org/
GoGrid	http://www.gogrid.com/
Google	http://www.google.com/
gproc	http://github.com/uwiger/gproc
Grails	http://grails.org/
Graylog2	http://graylog2.org/
GridFS	http://www.mongodb.org/display/DOCS/GridFS+Specification
grsecurity	http://grsecurity.net/
Hadoop	http://hadoop.apache.org/
HAProxy	http://haproxy.1wt.eu/

Heroku	http://www.heroku.com/
Hibernate	http://www.hibernate.org/
High Scalability	http://highscalability.com/
ZeroC Ice	http://zeroc.com/ice.html
iCloud	http://www.icloud.com/
IeAT	http://www.ieat.ro/
InfoQ	http://www.infoq.com/
Internet	http://en.wikipedia.org/wiki/Internet
iOS	http://www.apple.com/ios
Java	http://www.java.com/
JavaScript	http://en.wikipedia.org/wiki/JavaScript
Jetty	http://www.eclipse.org/jetty
Joyent	http://www.joyent.com/
JSON-RPC	http://json-rpc.org/
Kanban	http://en.wikipedia.org/wiki/Kanban
Kerberos	http://web.mit.edu/kerberos
Lego	http://www.lego.com/
LinkedIn	http://www.linkedin.com/
Linux	http://kernel.org/
Log4j	http://logging.apache.org/log4j
Loggly	http://loggly.com/
Logplex	http://github.com/heroku/logplex
logrotate	http://linux.die.net/man/8/logrotate
Lua	http://www.lua.org/
Maven	http://maven.apache.org/
Membase	http://www.couchbase.com/membase
MemcacheDb	http://memcachedb.org/
Memcached	http://memcached.org/
Memcache	http://memcached.org/
Microsoft	http://www.microsoft.com/
Misultin	http://github.com/ostinelli/misultin
Mnesia	http://www.erlang.org/doc/apps/mnesia/
Mochiweb	http://github.com/mochi/mochiweb
MongoDB	http://www.mongodb.org/
Mongrel2	http://mongrel2.org/
Monit	http://mmonit.com/
mOSAIC	http://developers.mosaic-cloud.eu/
MyProxy	http://grid.ncsa.illinois.edu/myproxy
MySQL	http://www.mysql.com/
Netcraft	http://netcraft.com/
NewRelic	http://newrelic.com/
Nginx	http://nginx.org/
NodeJS	http://nodejs.org/
NoOps	http://www.infoq.com/news/2012/03/NoOps
Office 365	http://office365.microsoft.com/
OpenPG	http://www.openpgp.org/
OpenShift	http://openshift.redhat.com/
Oracle	http://www.oracle.com/
Paxos	http://en.wikipedia.org/wiki/Paxos_%28computer_science%29
PHP	http://www.php.net/
Plan9	http://plan9.bell-labs.com/plan9
Postfix	http://www.postfix.org/
PostgreSQL	http://www.postgresql.org/
ProtocolBuffers	http://code.google.com/p/protobuf
Python	http://www.python.org/
RabbitMQ	http://www.rabbitmq.com/
Rack	http://rack.github.com/
Rackspace	http://www.rackspace.com/
Reddit	http://www.reddit.com/
RedHat	http://www.redhat.com/
Redis	http://www.redis.ro/
riak-core	http://github.com/basho/riak_core
Riak	http://wiki.basho.com/Riak.html
RightScale	http://www.rightscale.com/
RSA	http://en.wikipedia.org/wiki/RSA_%28algorithm%29
Ruby	http://www.ruby-lang.org/
runit	http://smarden.org/runit
Scala	http://www.scala-lang.org/
Scribe	http://github.com/facebook/scribe
Scrum	http://en.wikipedia.org/wiki/Scrum_%28development%29
SELinux	http://selinuxproject.org/
SendGrid	http://sendgrid.com/
Sinatra	http://www.sinatrarb.com/
Slf4j	http://www.slf4j.org/
Slitaz	http://www.slitaz.org/
SmartOS	http://smartos.org/
Solaris	http://en.wikipedia.org/wiki/Solaris_%28operating_system%29
Sony	http://www.sony.com/
SQLite	http://www.sqlite.org/
syslog	http://en.wikipedia.org/wiki/Syslog
systemd	http://freedesktop.org/wiki/Software/systemd
TinyCore	http://distro.ibiblio.org/tinycorelinux
(Apache) Tomcat	http://tomcat.apache.org/
Twitter	http://twitter.com/
Ubuntu Core	http://wiki.ubuntu.com/Core
Ubuntu	http://www.ubuntu.com/
UNIX	http://en.wikipedia.org/wiki/Unix
Upstart	http://upstart.ubuntu.com/
V8	http://code.google.com/p/v8
Vision	http://www.project-vision.eu/
VMware	http://www.vmware.com/
Voldemort	http://project-voldemort.com/
Vserver	http://linux-vserver.org/
Web 2.0	http://ro.wikipedia.org/wiki/Web_2.0
Webmachine	http://wiki.basho.com/Webmachine.html
WebSocket	http://en.wikipedia.org/wiki/WebSocket
WikiDot	http://www.wikidot.com/
WikiMedia	http://www.wikimedia.org/

Wikipedia	http://wikipedia.org/
Windows	http://windows.microsoft.com/
WS-Security	http://en.wikipedia.org/wiki/WS-Security
X.509	http://en.wikipedia.org/wiki/X.509
Xen	http://xen.org/
YouTube	http://www.youtube.com/
ZeroMQ	http://www.zeromq.org/