Gru: an Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures

Luca Florio, Elisabetta Di Nitto
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Email: {luca.florio, elisabetta.dinitto}@polimi.it

***Abstract*—Microservices architectures support the development of complex applications by composing simple basic blocks, the microservices. This enables runtime adaptation and dynamic reconfiguration of the applications. The approaches that have been developed so far, however, focus on offering only basic self-healing capabilities (e.g., restarting a failed microservice), while do not fully tackle the problem of offering advanced and general mechanisms for building complex and decentralized self-adaptation strategies able to deal with applications built of a large number of nodes. In this work we focus on how to add autonomic capabilities to microservices without changing the way they are implemented but exploiting their containers. Our approach is called Gru and creates a new level of abstraction, containing autonomic intelligence, on top of Docker containers. Gru supports a variety of decentralized autonomic operations and does not make any assumption nor constraint on the structure of the microservices system to be controlled. This paper presents Gru together with a preliminary evaluation of its features.**

***Keywords*-containers; decentralized; Docker; microservices; self-adaptation**

I. Introduction

The development of modern cloud applications has moved from a monolithic approach, where the application is developed as a single unit, to a distributed approach, where the application is composed of small and decoupled elements, each providing a single functionality, that communicate with each other using synchronous or asynchronous techniques. *Microservices architectures* [1] embody this distributed approach. Microservices are highly decoupled software components that take care of a specific application functionality in a complete way and communicate with each other through lightweight REST APIs or asynchronous message queues. They are designed to be easily replicated to cope with high workload as needed and, typically, execute within the context of a *container* that isolates them from any other component and simplifies their replication on different computation nodes.

Containers such as Docker [2] and supporting environments such as Kubernetes [3], simplify the management of microservices and enable their self-healing, for instance, by supporting re-execution of a failed service. We believe that microservices plus containers have the potential of enabling even other more sophisticated autonomic strategies, such as self-adaptation to changing workloads. Moreover, thanks to the decoupling introduced by microservices, such autonomic strategies can be selected and executed in a fully decentralized way, thus avoiding bottlenecks and single points of failure in the case of large-scale applications composed of hundreds of elements, as in the case of Netflix[1].

Following these principles, we are developing **Gru**[2]. It implements a decentralized MAPE loop (Monitor-Analyze-Plan-Execute) [4]. Each Gru agent is controlling a set of Docker containers, running on some host (either physical or virtual) that we call *node*, and is making decisions for them. Examples of decisions are, for instance, the replication or the migration of a microservice, based on the workload that it is receiving. Each Gru agent works on its own, but receives information from the other neighbours. This allows it to base each decision not only on the state of the node it controls, but also on the situation of the neighbours. Decisions made by agents are completely transparent to microservices and applications, which should simply experience a good level of QoS regardless the problems that may occur in the underlying infrastructure.

Gru is still a work in progress, however in this paper we want to present its main features and show an initial evaluation of its capabilities.

II. Background

A. Microservices

Microservices are a fast growing trend in cloud-based application development [1]. The traditional monolithic application is divided into small pieces that provide a single service: the full capabilities of the application emerge from the interaction of this small pieces. Microservices are independent from each other and organized around capabilities, e.g. user interface, front-end, etc. Their decoupling allows developers to use the best technology for their implementation according to the task they have to accomplish: the application becomes polyglot, involving different programming languages and technologies. An application composed of microservices is inherently distributed, being divided into hundreds of different microservices, deployed in a large network infrastructure, that communicate both in a synchronous or asynchronous way, using REST or a message-based system respectively. Through the use of microservices, the application can scale efficiently as it is possible to scale only the microservices that are under heavy

[1] https://www.netflix.com/
[2] https://github.com/elleFlorio/gru

978-1-5090-1654-9/16 $31.00 © 2016 IEEE
DOI 10.1109/ICAC.2016.25

load, not the entire application. The microservices architecture embodies the principles of the DevOps movement, promoting the automation of deployment and testing and reducing the burden on management and operations. Several companies in the recent years moved to a microservices architecture: Netflix has been among the first to adopt microservices, followed by many others (e.g. SoundCloud[3], Groupon[4], etc.)

B. Docker

Microservices usually run into *virtual containers* like Docker [2], [3]. Docker containers can run an application as an isolated process on a host machine, including only the application and all its dependencies (the kernel of the Operating System is shared among other containers) and providing to it only the resources it requires. Docker containers are different from a fully virtualized system like a virtual machine: a virtual machine contains a full OS that runs in isolation on physical resources that are virtualized by an hypervisor on the basis of the ones available in the host machine; a Docker container uses the resources available in the host (both a physical or a virtualized one) that are assigned to it by the Docker Engine. The consequence is that Docker containers can share physical resources and are lightweight: it is possible to run multiple containers on the same machine starting them in seconds. Docker allows developers to implement their application and their services using the technology or language that is most suitable to them. Services deployed in a Docker container can be scaled or replaced just starting or stopping the container running that specific service. Moreover Docker containers can be deployed in very different settings, from servers in a cloud computing infrastructure to ARM-based IoT devices.

III. Challenge and contribution

Through the design and implementation of Gru we want to study the integration of an autonomic manager into a distributed application that is not specifically designed to be autonomic. We believe that finding a way to easily apply autonomic computing to an existing application in a transparent way can push further the study of autonomic systems, as well as enable their application to industrial-grade software.

The focus of our research is not only on applications and algorithms related to autonomic systems, but we want to answer the following question: **can we bring autonomic capabilities to a distributed application that is not designed to integrate them? To what extent can we make such application autonomic?**

The first step in this direction is to understand how to integrate autonomic capabilities into the distributed application. This integration can happen in two ways: (i) using an internal approach, where the autonomic manager is part of the application to manage; (ii) with an external approach, where the autonomic manager is totally independent from the application to manage and communicates with it from the outside. Despite the integration of the autonomic manager inside the distributed application enables a more direct connection between the two parts, we want to focus on the application of an external autonomic manager that is not part of the application to be managed because this is more flexible and lets the developers to focus on the functional part of the application to develop.

To enable the external approach, we propose the use of an intermediate component: the *autonomic enabler* that can be imagined like a "box" containing the application or its components. The **autonomic enabler** is external from the application or the autonomic manager, but it is the link between the two components. The autonomic enabler has no autonomic capabilities and should wrap the application without having a role in its functional part. The autonomic manager actuates autonomic actions against the autonomic enabler containing the managed application, making it autonomic in an indirect way. The use of the autonomic enabler allows the easy integration of the autonomic manager with the managed application, without requiring complex modifications of the application itself. In this way developers can design the application without worrying about the implementation of autonomic capabilities. Since virtual containers present all the aforementioned features, we propose the use of virtual containers as the autonomic enabler.

The autonomic system is then composed of three distinct parts that interact between them to make the application autonomic: the *application* to manage, the *autonomic manager* and the *autonomic enabler*. Since the focus is on distributed systems composed of thousands of interacting components, we adopted a **decentralized approach** for the autonomic management of the application: this approach has the advantage of reducing the risk of bottlenecks and single points of failure.

Our decentralized approach is based on a multiagent system, where agents act as independent and "intelligent" units able to manage a part of the application. Agents implement the MAPE autonomic loop and actuate autonomic actions on the application on the basis of their internal status and on the information coming from a set of peers. Agents decide the actions to actuate to keep the application up and running, and according to some quality constraints imposed by the user. The autonomic capabilities of the application emerge from the local decision of each agent. The decision is based on a partial knowledge of the whole system: agents exchange information only with a subset of the total number of peers that changes after every autonomic loop. Using this strategy the communication overhead is reduced and every agent has a view of a different part of the system at every iteration.

IV. Gru

Gru implements all the characteristics defined in the previous section and exploits Docker as the autonomic enabler of the system. Figure 1 shows an example of instantiation of the Gru framework to support the execution of an application composed of three microservices, μS1, μS2, and μS3. As shown in the figure, these microservices are deployed on a cluster composed of two nodes. Each node hosts a Docker Daemon and a *Gru-Agent* that control the execution of microservices.

[3]https://soundcloud.com/

[4]https://www.groupon.com/

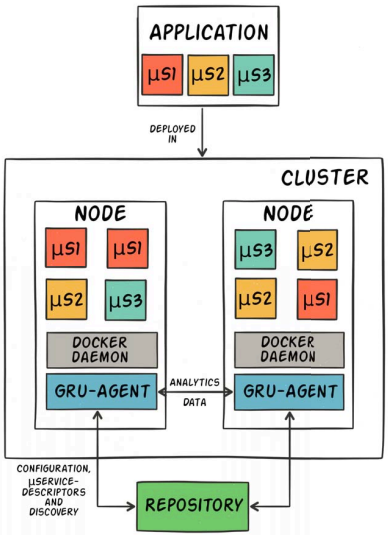

Fig. 1. Gru deployment with two nodes.

Gru-Agents are autonomic managers and communicate with each other to acquire knowledge about other nodes before making an autonomic decision. They also communicate with a Repository that includes configuration information.

The figure shows a snapshot of the application where its microservices have been replicated and distributed on the two nodes to increase availability and the ability to handle the application workload. The decision on whether to replicate is automatically made by each Gru-Agent as we explain in the next subsections. The administrators of our application need only to deploy and configure Docker and the Gru-Agent and to define a descriptor for each microservice. This is called *μService-Descriptor* and is explained below. Gru-Agents are deployed in *nodes* where the microservices of the application are running. The set of nodes where the Gru-Agents are deployed is a *cluster* (Fig 1).

A. Repository

The Repository stores the μService-Descriptors and the configuration of the Gru-Agents; this configuration contains the parameters related to the execution of the autonomic loop (e.g. time interval between each loop), the communication (e.g. number of peers to reach) and the connection to external services (e.g. Docker Daemon, etc.). The configuration and the μService-Descriptors are downloaded from the repository when the Gru-Agent starts. The external repository is used also for the discovery of the Gru-Agents: each agent needs to know the existence of other active peers in the cluster, so it register itself to the external repository and queries it to discover the peers to communicate with. The external repository is currently implemented as an etcd[5] server.

[5]https://github.com/coreos/etcd

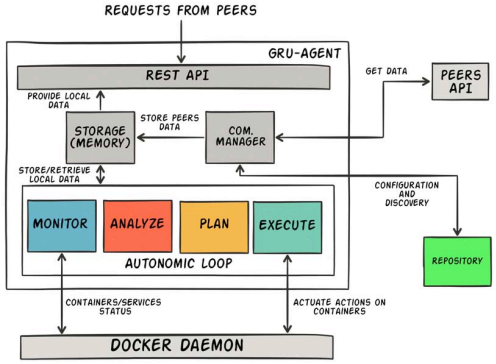

Fig. 2. Gru-Agent components and their interaction.

B. μService-Descriptors

μService-Descriptors are abstractions that model the actual microservices provided by the application. They include the name, type and Docker image of the microservices they represent. Moreover, they define QoS constraints on the execution of the microservice, such as its maximum response time (intended as the maximum time interval between the arrival of a request and its response). Finally, they contain also the list of resources needed by the microservice to operate (i.e. CPUs, RAM), so Gru can provide an autonomic management of the resources assigning to each microservice the correct amount of available resources. In the current implementation this is limited to the assignment of CPUs: differently from Docker, where the user has to specify the CPUs that a container can use, Gru allows the user to set the number of CPUs required by a microservice in the μService-Descriptor, then Gru takes care of assigning to it the free CPUs. This is just a basic feature, but it's a starting point to enable more complex autonomic resource management.

μService-Descriptors have to be created before deploying the application. Gru-Agents use them to properly create the containers running the microservices on the base of the information about the Docker image to use, the resources needed, etc., as well as to manage the microservices according to the constraints defined by the user in the descriptor.

C. Gru-Agents

Gru-Agents represent the autonomic managers of the system and actuate autonomic actions on the Docker containers. Gru-Agents share information about their internal status; they work with partial information, communicating with a subset of the total number of peers. The number of peers in the subset is set in the configuration of the Gru-Agents. With this choice, Gru follows the philosophy behind most of the decentralized approaches available in the literature [5]. The rationale is to avoid global communication that would introduce a significant overhead in the system. The internal components of the Gru-Agents and their interactions are depicted in Figure 2.

The **REST APIs** are used to interact with the Gru-Agent and for inter-agent communications. The **internal storage** is

used to store the local data computed by the agent itself during the autonomic loop and the data gathered from its peers.

The **Communication Manager** interacts with the external repository to obtain the list of active nodes and choses a random subset of peers to communicate with. The random selection of peers is a simple strategy with small computational overhead that allows to have a different and broad view of the system as the number of iterations increases. The Communication Manager exploits the REST API to gather the data from the chosen peers and stores them in the internal storage. The data exchanged between agents are their local data regarding the usage of resources (CPU and memory) and the status of the microservices (e.g. response time of the microservice) running in their node. The communication between Gru-Agents happens before the autonomic loop and is not synchronized: agents can be queried by others in any moment without blocking their execution

Gru-Agents implement the classical MAPE autonomic loop that is triggered periodically, with a time interval set by the user in agents configuration. The **Monitor** component interacts with the Docker daemon to gather the data from the containers (e.g. CPU and memory usage) as well as the data provided by the microservices (e.g. response time). In the current implementation the managed microservices expose their internal metrics writing a line in logs that are accessible through the container. We plan to implement the integration of Gru with external monitoring services.

The gathered data are then sent to the **Analyzer**. This component works on the low level information provided by the Monitor, i.e., CPU and RAM usage, and response time of microservices to estimate the available resources for each microservice and to compute its actual load in terms of response time over maximum response time of the service: $\frac{S_{rt}}{S_{rt,max}}$. Moreover, the Analyzer merges these results with the data gathered from the other peers by the Communication Manager, computing, in the current implementation, the average of all data and obtaining a partial view of the status of the system.

The **Planner** component has to decide the action to actuate according to the partial view of the system produced by the Analyzer. The use of a partial view of the system, compared to the use of local information only, can lead to more effective decisions and requires just a small overhead in terms of communications. The Planner is based on the concepts of *policy* and *strategy*. Policies are rules that involve the execution of one or more actions, while strategies are algorithms that allow the Panner to chose a policy among a set of weighted ones. The policies currently implemented are *scale-out* and *scale-in* of a container running a microservice, that are based, respectively, on the *start* and *stop* actions offered by the Docker containers, and a *no-action* policy.

The Planner creates a weighted scale-* policy for each microservice running in its node. The weight of the policy is computed as $w_{scale-*,ms} = \frac{D_{cpu,ms}+D_{load,ms}}{2}$, where $D_{metric,ms}$ is the normalized difference between the value of the metric of a specific microservice ms and a threshold defined by the user in the configuration of the agent. The value of $D_{metric,ms}$ for the scale-out policy is greater than zero if the value of the metric is above the defined scale-out threshold, while for the scale-in policy it is greater than zero if the value of the metric is below the defined scale-in threshold. The no-action policy is weighted as $w_{no-action} = 1-Max(w_{policies})$, that is the difference between one and the maximum weight between other policies.

Once all the policies are weighted, the decision on the one to actuate is based on a strategy. Strategy takes as input the set of weighted policies and selects one of them according to a specific algorithm. The strategy currently implemented aims at selecting, among the available policies, the one that has a weight higher than a threshold that is randomly selected. It is based on the following algorithm:

1: $policies \longleftarrow Shuffle(policies)$
2: $totalWeight \longleftarrow \sum_{p \in policies} p.weight$
3: $threshold \longleftarrow rand(0, 1)$
4: $delta \longleftarrow 1$
5: $index \longleftarrow 0$
6: **for** $p \in policies$ **do**
7: **if** $\frac{p.weight}{totalWeight} > threshold$ **then**
8: **return** p
9: **else**
10: **if** $(threshold - \frac{p.weight}{totalWeight}) < delta$ **then**
11: $delta \longleftarrow threshold - \frac{p.weight}{totalWeight}$
12: $index \longleftarrow index_p$
13: **end if**
14: **end if**
15: **end for**
16: **return** $policies[index]$

The strategy acquires as input an array of weighted policies $policies$ and shuffles it. It computes the $totalWeight$ as the sum of the weights of all the policies and uses this in the next steps to normalize all policy weights. Moreover, it chooses randomly a $threshold$. It then checks for each policy in $policies$ if its normalized weight is greater than the $threshold$. If this is the case, it then selects that specific policy for execution. Otherwise it looks for the remaining policies in the array. In the search, to address the case in which none of the policy normalized weights passes the threshold, it keeps track of the difference between such weights and the threshold, storing the index of the policy that is closest to the threshold in $index$. Thus, it can select, in the end, the policy with weight closest to the threshold.

The probabilistic selection of the threshold and the shuffling of the policies are adopted to avoid that all the Gru-Agents shared the same behavior by executing always the policy with the highest weight. This would lead to the case in which all agents execute the same action (e.g., scale-in) thus over-reacting to a potential problem. The effectiveness of a similar strategy in a scenario with a high number of nodes has already been proven in a previous work [6].

The use of policy and strategies brings flexibility to the system, allowing the user to define his own policies or strategies.

The **Executor** interacts with the Docker daemon to execute

the actions of the chosen policy on the container running the microservice.

V. Initial evaluation

We tested Gru with a simple experiment to evaluate its capabilities to auto-scale a single microservice under a varying workload. This is not a complete evaluation, but an initial test we use to understand if the system is working properly.

The machine we used for the test is a PC with a *Intel i7* CPU (4 physical cores + 4 virtual cores) and 8GB of RAM. We created 5 virtual machines: one using 2 cores and 2 GB of RAM (*main-node*), while the remaining four using 2 cores and 1GB of RAM (*gru-nodes*). We deployed in the main-node the etcd server, as well as influxdb and Grafana for debugging purposes. We deployed in the main-node also a load-balancer that acts as the endpoint of the application: it receives the requests for the microservice and sends them to an instance of the microservice that is chosen randomly. The main node also runs Apache Jmeter[6] that is used to send requests to the application. Gru-Agents are deployed in each gru-node (gru-nodes are part of the same cluster). Gru-Agents are configured to run the autonomic loop every 2 minutes. The communication is limited to 2 peers randomly chosen at each iteration of the autonomic loop. We created a microservice that runs in a Docker container and simulates a computation with a function that keeps the CPU busy for a time interval that is computed at the arrival of each request. The time for the computation is chosen according to the exponential distribution with $\lambda = 5000ms$. The maximum response time of the service has been set as 3 times the λ, i.e. 15000ms.

The system starts with one active instance of the microservice in a Gru-node and Gru should scale the microservice according to the variation of the workload. The metric considered for the auto-scaling are the CPU utilization of the container and its load (response time over maximum response time). The workload has been generated using Apache Jmeter and goes from 0.1 to 2 RPS (Requests Per Second) and lasts one hour (Fig. 3). The workload has been chosen taking into account the available resources of the nodes. The execution time of the experiment has been set to one hour in order to have a time interval between each autonomic iteration that was not too short, letting Gru-Agents to collect an adequate amount of data. We executed 5 runs of the test obtaining similar results. The results are available online[7], and we report the charts of a test as an example of the performance of Gru (Fig. 3-5).

The results show that Gru is able to scale containers (Fig. 5) in order to keep the response time under the threshold defined in the μService-Descriptor of the microservice, i.e. 15 seconds, with an average response time of 5 seconds (Fig. 4). The number of active instances does not follow exactly the workload, but may present variations (e.g. the peak at time 9.50 in Fig. 5); this is due to the probabilistic approach adopted by Gru. However, since Docker containers can be started and stopped in a time of seconds, this behavior does not represent a relevant problem in the usage of resources.

[6]http://jmeter.apache.org/

[7]https://github.com/elleFlorio/gru-evaluation

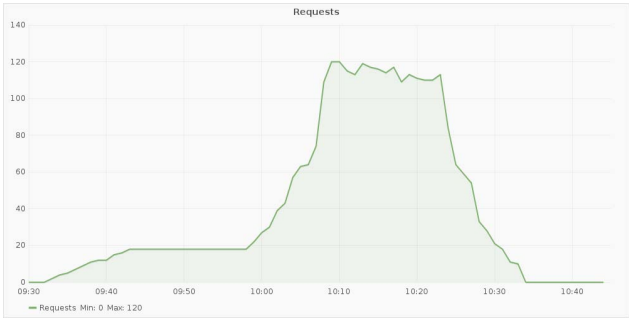

Fig. 3. Requests to the service (mean 1 minute).

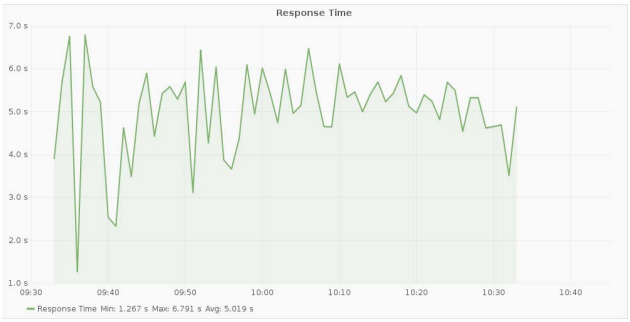

Fig. 4. Response time of service (mean 1 minute).

Fig. 5. Active instances of service during time.

The initial results are promising and suggest that the integration of decentralized self-adaptation in a system composed of microservices is possible using our approach. We are working on the improvement of the system and on its complete and robust validation: we plan to test Gru with a complex application composed of several microservices distributed in tens of nodes.

A. Limitations

Gru is a work in progress and presents some limitations in the current implementation. The autonomic capabilities of Gru are still limited to the auto-scaling of the services. This is enough for an initial validation of our ideas, but we aim

to provide more sophisticated autonomic capabilities. We are studying new autonomic policies, strategies and actions related to the topology of the application and to the optimization of resources. Statistics about the functional part of the application are gathered only through the logs of the application itself. This can be convenient and involves a very simple modification of the managed application. However, we want to integrate our tool with the most used monitoring and data-collector systems in order to make Gru more flexible for the users.

VI. Related work

Docker and microservices are recent technologies, however their management is a problem that has been addressed both in industry and academia. Commercial systems such as **Docker Swarm**[8] and **Google Kubernetes**[9] provide an effective solution to the clustering and management of containers. These systems provide an automatic management of resources and basic self-healing capabilities; however, unlike our solution, they do not provide more advanced self-adaptive capabilities, like the auto-scaling of containers.

The problem of service discovery of microservices is addressed in [7], where the authors present **Serfnode**, a solution based on Docker containers and the Serf project. Serfnode is an agent that acts as a parent container of the services of the application: each service image has its corresponding Serfnode parent that monitors that service and handles the discovery of the service as well as various types of events. Serfnodes communicate using a gossip protocol, providing a fully decentralized system. The problem addressed by Serfnode is limited to the discovery of services, while we want to provide a more comprehensive self-managing solution.

An **architecture for self-managing microservices** has been proposed in order to enable the scalable and resilient self-management of microservices [8]. The authors propose a distributed architecture based on self-managing atomic services and on the election of a cluster leader that takes the decisions and actuates the actions. This solution is totally different from the one we present in this paper: the management logic is inside the microservices and the system is hierarchical, while we apply an external and decentralized approach.

The application of self-adaptive techniques has been widely studied also in the context of cloud services. A **framework for the coordination of multiple autonomic managers** is presented in [9]. Two autonomic managers (based on the MAPE-K control loop) operate at different levels: the application level, where the autonomic manager ensures the best architectural configuration of the application, and the infrastructure level, where the autonomic manager manages the resources in a resource/energy-aware manner. The communication between the two different autonomic manager is based on events and actions. This solution is based on a more hierarchical structure of the autonomic manager, while our solution wants to be a decentralized one.

Decentralized self-adaptation is exploited in [10] to provide an adaptation mechanism for service-based applications in the cloud. This work addresses the problem of services composition, proposing a decentralized solution based on market-based heuristics. This solution is interesting and provides a decentralized approach; however, it addresses specifically the problem of service composition, that is out of the scope of our work: we assume that the composition of services has been defined in the design of the application to manage.

VII. Conclusion

In this paper, we presented a methodology and a tool to apply autonomic computing to applications based on the microservices architecture pattern. We defined the context of our research and the challenge we want to tackle: the transparent integration of an autonomic manager to an application not designed to implement autonomic capabilities. We described Gru, the tool we are developing to reach our goal and to validate our ideas. Gru is able to manage a microservices application deployed in Docker containers and has been validated through an initial test to show its capabilities and its potential. Despite Gru presents some limitations due to the early stage of its development, the results are promising. Our research is still at the beginning, but we believe that the challenge we face is a fundamental step in the study and application of autonomic systems.

Acknowledgment

The research reported in this article is partially supported by the European Commission grant no. FP7-ICT-2011-8-318484 (MODAClouds).

References

[1] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 2015.

[2] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[3] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[4] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[5] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, "On patterns for decentralized control in self-adaptive systems," *Software Engineering for Self-Adaptive Systems II*, pp. 76–107, 2013.

[6] N. M. Calcavecchia, B. A. Caprarescu, E. Di Nitto, D. J. Dubois, and D. Petcu, "Depas: a decentralized probabilistic algorithm for auto-scaling," *Computing*, vol. 94, no. 8-10, pp. 701–730, 2012.

[7] J. Stubbs, W. Moreira, and R. Dooley, "Distributed systems of microservices using docker and serfnode," in *Science Gateways (IWSG)*, 2015.

[8] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds, "An architecture for self-managing microservices," in *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*. ACM, 2015.

[9] F. A. de Oliveira, T. Ledoux, and R. Sharrock, "A framework for the coordination of multiple autonomic managers in cloud environments," in *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, 2013.

[10] V. Nallur and R. Bahsoon, "A decentralized self-adaptation mechanism for service-based applications in the cloud," *Software Engineering, IEEE Transactions on*, vol. 39, no. 5, pp. 591–612, 2013.

[8] https://github.com/docker/swarm/

[9] https://github.com/kubernetes/kubernetes