2015 IEEE 8th International Conference on Cloud Computing

Mycocloud: Elasticity through Self-organized Service Placement in Decentralized Clouds

Daniel J. Dubois[1]. Giuseppe Valetto[2], Donato Lucia[3], Elisabetta Di Nitto[3]
[1]*Imperial College London, Department of Computing, London, United Kingdom*
[2]*Fondazione Bruno Kessler, Distributed Adaptive Systems Unit, Trento, Italy*
[3]*Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milan, Italy*

***Abstract*—We present Mycocloud, a fully self-organized approach to service placement. Mycocloud supports service elasticity within a network of hosts with heterogeneous computational capacity. Mycocloud proposes a completely decentralized algorithm that continuously calculates the dynamic placement of different services on the host nodes, in response to the varying demand for each service, the churn and dynamism of the node set contributing resources to the system, and the changes in the overlay topology. Our simulation results show how Mycocloud provides good performance in terms of convergence rate, response time, system load and network traffic overhead.**

I. Introduction

In cloud computing, one of the most important problems from both the provider's and customer's points of view is to efficiently and dynamically commit cloud resources, based on incoming service requests. Recent research in this area has focused on topics such as *load balancing* [14, 22, 25, 28], that is, how to spread the request load for resident cloud services among available suitable resources; and *auto-scaling* [5, 11], that is, how to recruit and release resources on the fly to satisfy variable service loads, while minimizing both over– and under–provisioning. Load balancing and auto-scaling functionality is furthered by *multi–cloud frameworks* [15, 21], which enable multiple cloud providers to lend and borrow resources to one another as seamlessly as possible.

Load balancing and auto-scaling work synergistically toward elastic service provisioning; they try to commit the right volume of resources to satisfy the incoming load for each service, while maintaining the average level of utilization of committed resources within a desirable range.

Another element that can enable elastic service provisioning is *service placement*, that is, how to automate decisions about what services are hosted on what cloud nodes, based on some optimization criteria. Service placement has been used, for instance, to consolidate multiple services on the same host, to reduce power consumption [3, 4], or to arrange the geographic location of hosted services, to reduce network latency and improve response time [18, 31]. These service placement algorithms can be seen as complementary with respect to load balancing and auto-scaling, since they help to optimize non-functional system properties without actually moving the load or adding/removing resources.

In many prevalent commercial cloud offerings, service placement is in practice conflated with auto-scaling. In fact, service placement is a computationally hard problem [8]; current approaches approximate its solution by using a centralized manager that requires complete and up-to-date knowledge of the state of the system [4, 13]. Those assumptions become quite demanding as the number of services, the size, and dynamism of the cloud hosting facilities increase.

In this paper we present Mycocloud, a fully decentralized approach to handle service placement in support to elastic service provisioning. We have conceived Mycocloud in the context of our continuing work on a bio-inspired approach for the self-organized construction, maintenance and management of a service provisioning network made of heterogeneous nodes that are connected via a peer-to-peer overlay [27, 28]. Our work goes in the direction of recent trends that advocate, to various extents, to move cloud capabilities out of data centers and diffuse them towards the edge of the network, such as fog computing [29], cloud 2.0 [17], crowd computing [10, 19], etc.

Our approach is most closely related to volunteer clouds [9, 24] and peer-to-peer clouds [2]. Those are systems in which multi-tenancy tends towards pulverized ownership, where the pool of participating computing resources is very dynamic, and the system may not be able to exert direct control on those dynamics, that is, it may not rely on auto-scaling to recruit or release resources on demand. In such a context, the Mycocloud service placement mechanism provides elasticity by adapting the portion of the overall resources that are committed to each running service, based on the incoming loads for all hosted services.

We show across a range of simulation scenarios that the fully self-organized service placement algorithm offered by Mycocloud can provide substantial benefits in terms of elastic provisioning. Since Mycocloud does away with any centralization of information, processing or decision-making, it is an intrinsically fault-tolerant and scalable solution. Furthermore, since Mycocloud is agnostic to how computing resources enter and exit the cloud environment, it remains completely orthogonal to the issue of auto-scaling; thus, it can, in principle, also be combined with existing auto-scaling approaches.

This paper is organized as follows. Section II describes

2159-6190/15 $31.00 © 2015 IEEE
DOI 10.1109/CLOUD.2015.89

the context and the problem we are addressing. Section III describes our background work. Section IV describes the proposed self-organization algorithm. Section V shows some experimental results. Section VI discusses related work, and, finally, Section VII concludes the paper.

II. Context and Problem

We consider a cloud computing system in which resources can appear and disappear in an unpredictable fashion and in which applications run best effort, without strict Service Level Agreements. This is a common assumption in peer-to-peer/voluntary/edge cloud systems [2, 7, 16, 32].

A. System Model

The state of the system at a certain time t is defined by the following elements:

- N: set of resources, which we also call nodes interchangeably;
- L: set of bidirectional links between nodes, which represent open communication channels;
- $G = (N, L)$: topology of the system, which must be a connected graph;
- K: set of possible services, which we also call classes interchangeably; given a node $i \in N$, $k_i \in K$ defines its corresponding class;
- $C_i \in \mathbb{R}^+, i \in N$: node computational capacity (i.e., the number of requests the node can process within a single time unit);
- $U_i \in \mathbb{R}[0,1], i \in N$: last known utilization of a node;
- $\lambda_j, \in \mathbb{R}^+, j \in K$: flow of incoming requests of class $j \in K$ that are sent to a random node i such that $j = k_i$ (e.g., a request of a given class is always assigned to a node of the same class).

The elements above are all subject to change overtime (e.g., nodes are subject to churn, the number of available services can increase, etc.), causing unpredictable involuntary system transitions.

There are also two types of system transitions that are used to effect voluntary changes: (*i*) *service change*, which modifies K; (*ii*) *topology change*, which modifies L, as long as G is still a connected graph.

B. Problem Statement

Our goal is to dynamically effect service placement, in order to minimize the response time R of the system, defined as the average time that a request spends in the system (transferring time plus queuing time plus processing time).

$$\text{Goal: } \textit{minimize}(R)$$

An equivalent formulation of the goal is the maximization of system utilization, which can be achieved by replacing the services to best match the flow of incoming requests.

$$\text{Equivalent Goal: } \textit{maximize}(\sum_{i \in N} U_i)$$

A graphical representation of a system in its initial state, and in its desired state is depicted in Figure 1. The nodes

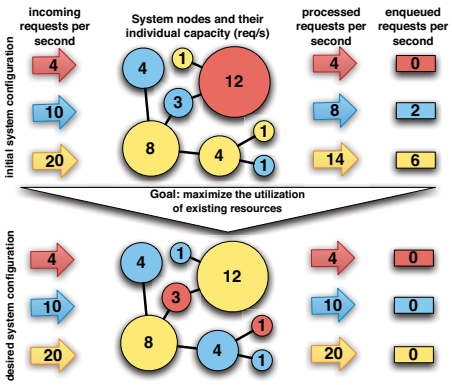


Figure 1. Inefficient service placement vs efficient service placement. Different services are depicted with different colors.

are the system resources, colors are service classes, incoming/outgoing arrows the incoming/outgoing requests. The top part of the figure shows a sub-optimal placement of services over the resources that results in an inefficient utilization of the system, which cannot process all the incoming requests. The second figure is a situation that maximizes our goal: it has the best placement of the services over the resources.

Our formulation of the optimization problem is compatible with the GAP problem, which is NP-hard and APX-hard [8]. Mycocloud is a decentralized algorithm that searches for a service placement configuration that satisfies the goal above, based on the partial knowledge held by each node. This knowledge includes the state of that node, plus information on the states of its immediate current neighbors.

III. Background

Mycocloud service placement is built on top of two previous results on self-organized service provisioning: Myconet [27], a bio-inspired approach to topology management for the P2P overlay underlying the decentralized cloud, and Mycoload [28], which provides service load balancing functionality on top of the Myconet topology. We hereby provide a brief overview of both, for the sake of clarity and to facilitate the description of the Mycocloud algorithm in Section IV.

A. Topology management: Myconet

Myconet is a self-organized capacity-aware topology management algorithm for P2P overlays with super-peers [27]. One Myconet distinguishing trait is its multi-level hierarchy of peer states, which are loosely modeled on the growth dynamics of *hyphae* (the root-like structures of some species of fungi). Nodes are peers, and they can either be *biomass peers* (regular nodes with limited capacity) or *hyphal peers* (i.e., nodes that are promoted to super-peers based on their capacity, which provide additional services to biomass peers). Furthermore, hyphal peers switch between three hierarchical protocol states: *extending*, *branching*, and *immobile*, with distinct roles in managing the topology of

the overlay. All peers begin as biomass, and follow a multi-step promotion/demotion process to and from the various super-peer states:

- *Extending peers* act as attachment points, and continuously explore the network looking for new or isolated biomass peers to connect to.
- *Branching peers* grow new cross-connections with other hyphal peers; they also regulate the number of extending peers in the network, at times selecting large-capacity biomass peers for promotion to the extending state, and at other times targeting excess extending peers for demotion.
- *Immobile peers* similarly control the number of branching peers in the network, and ensure the chosen level of cross-connectivity between hyphal peers remains stable; immobile peers are selected by the protocol dynamics as being of the highest capacity (and, implicitly, reliability, as those peers have remained in the network long enough to be promoted to this state).

Myconet also has rules that regulate the interactions between neighboring hyphal peers; these rules, on the one hand, aim at transferring biomass peers towards the higher-capacity hyphal peers that can best service them, while, on the other hand, ensure that hyphal peers that cannot efficiently reach their target for serviced biomass peers, or inter-connection with other super-peers, become candidate for demotion.

Evaluation results show that Myconet constructs and maintains in a fully decentralized way a strongly inter-connected super-peer overlay, which is particularly efficient in optimizing the super-peer selection and the allocation of biomass peers to super-peers based on their capacity. It is also strongly robust and resilient vs. changes in the network, either due to regular peer churn, large-scale peer disconnections, or even targeted attacks against super-peers.

B. Load-balancing: Mycoload

Mycoload is a fully decentralized load balancing algorithm we have layered on top of Myconet [28]. In Mycoload, the nodes host instances of many different classes of services, and the algorithm operates in a twofold way: (*i*) it self-organizes nodes into virtual clusters of the same service class that are inter-connected via an extended Myconet algorithm that is aware of service classes; and (*ii*) re-routes incoming service requests among the virtual clusters, based on their class, and assigns those requests proportionally to the capacity of each node participating in the same cluster.

Mycoload uses to its advantage the super-peer layout of the topology constructed by Myconet, since incoming service requests are quickly routed through the existing interconnections between hyphal peers to the clusters of the right service class, where most of the load is absorbed by the highest-capacity peers, i.e., hyphal peers in immobile or branching state, and the rest is propagated to their lower-capacity immediate neighbors.

Evaluation results show that the Mycoload approach inherits the desirable topology properties of Myconet, notably, its resilience against perturbations of the overlay. Moreover, it is able to spread service load quickly and in a close-to-optimal way, using only local rules and local interactions between neighboring nodes. However, Mycoload – like other pure load balancing mechanisms, as we discussed in Section I – is limited in the sense that it does not include any form of scaling; that is, the total capacity per service type remains static (modulo appearance and disappearance of nodes from the system for exogenous reasons). If, for instance, the load becomes higher than the total capacity available for a given service type, the system will not be able to intervene and service response time will increasingly suffer.

Mycocloud is designed to complement load balancing capabilities like those of Mycoload with self-organized service placement dynamics. Mycocloud also remains complementary to auto-scaling approaches (such as DEPAS [5]) that can be added, to influence the recruiting of resources with additional capacity on demand, as well as their release.

IV. Mycocloud Algorithm

While Mycoload is in charge of balancing the load among nodes belonging to the same service class, Mycocloud comes into place when the load continues to be unbalanced among different service classes, and addresses the situation by changing the distribution of service classes through the nodes used for a certain application. This maximizes the overall system utilization and minimizes the average response time. The algorithm we propose automates the decision of changing service class in a decentralized way, at the level of single nodes and their neighborhoods. As in our previous work, we assume stateless services.

A. Algorithm parameters and actions

Mycocloud uses the following input parameters, which are known at each node i by either direct measurement, or by periodic diffusion from the neighbor nodes:

- Ne_i: set of nodes x such that $(x,i) \in L$, i.e., the set of all the neighbors of i.
- U_i: utilization of node i.
- U_{max}: utilization threshold for considering a node overloaded. This is a Mycocloud constant.
- C_i: capacity of node i.
- C_i^{avail}: capacity available in the node i, defined as $(U_{max} - U_i) \times C_i$. This value becomes negative when i is overloaded.
- k_i: service class of node i.
- $S_i \in \{\texttt{Biomass}, \texttt{Extending}, \texttt{Branching}, \texttt{Immobile}\}$: Myconet status of node i, as explained in Section III-A. The set of Myconet states is considered ordered in the following way: $\texttt{Biomass} < \texttt{Extending} < \texttt{Branching} < \texttt{Immobile}$.
- H_{max}: maximum number of times a service change request can be forwarded. This is a Mycocloud constant.

A node i knows also U_x, C_x, C_x^{avail}, k_x, and S_x of all $x \in Ne_i$. It can perform in its lifetime the following actions:

- `ChangeServiceAction (k)`, which causes node i to change its own class to k, provided that a node $x \in Ne$ exists such that $k_i = k_x$. This is to ensure that after i changes its class, this class does not disappear from the neighborhood.
- `ServiceChangeRequest (x, k, C, H)`, which causes node i to forward to neighbor x a request to change x class to k. C is the needed amount of computational capacity for class k and H is the number of times the message has been forwarded.

The possible events that trigger the execution of Mycocloud actions are: (*i*) *Overload event*, which is triggered when an incoming request arrives and $U_i > U_{max}$; (*ii*) *Service change request event*, which is triggered when a `ServiceChangeRequest` is received from another node. These are presented in detail below.

B. Overload Event Actions

When an Overload Event is triggered on the local node i, this node tries to select a random neighbor node x such that $k_x \neq k_i$ and $C_x^{avail} > 0$. If it succeeds, i issues a `ServiceChangeRequest(x, k_i, C^needed, 0)`, where C^{needed} represents the amount of computational capacity for class k_i needed to bring U_i under the U_{max} threshold, that is, $-C_i^{avail}$.

If no node satisfies the criterion above, then a neighbor node y is chosen randomly such that $S_y \geq S_i$. If node y selection was successful, a `ServiceChangeRequest(y, k_i, C^needed, 0)` is issued.

The idea behind these actions is that an overloaded node first tries to share its load with a node that belongs to a different underloaded class. If this is not possible, it tries to send the request to a supernode (a node that has a relevant role within Myconet topology[1]), regardless its load or class, to increase the chance that the `ServiceChangeRequest` reaches a node that can process it (see next section).

C. Service Change Request Event Actions

Once the local node i receives from a neighbor node j a `ServiceChangeRequest` message, which contains the properties k, C, and H, it goes through the following cases, evaluated in the given order:

Case 1 (Node with the same service class)

This is the case in which a node i receives a `ServiceChangeRequest` to a class $k = k_i$. Assuming that the underlying load balancing mechanism works properly, this means that i is overloaded (or close to be overloaded) as the node that has originated the request. Thus, i tries to select a neighbor $x \in Ne$ such that $k_x \neq k$ and x has the minimum available capacity C_x^{avail} in the neighborhood sufficient to fulfill the needs of the service class. If it finds x, then it generates a `ServiceChangeRequest(x, k, C, H+1)`. Otherwise, the node tries to find a random node y such that $S_y \geq S_i$ (i.e., a supernode) and, if found, generates a `ServiceChangeRequest(y, k, C, H+1)`.

Case 2 (Immobile node)

This is the case in which the request is received by a node that has $S_i =$ `Immobile`. In this case the node does not try to change its service type, regardless its utilization level, because, according to the topology reconfiguration algorithm we use (see Section III-A), it has the highest degree in the network; therefore, changing its type may destabilize the network and increase the probability of partitioning its clusters of services. Thus, the immobile node reacts to this request by selecting a neighbor $x \in Ne$ that has the maximum available capacity $C_x^{avail} \geq C$. If it finds x, then it generates a `ServiceChangeRequest(x, k, C, H+1)`. Otherwise, a random Biomass neighbor y is selected, such that $C_y^{avail} > 0$. If a valid y has been found, a `ServiceChangeRequest(y, k, C, H+1)` is generated.

Case 3 (Node with available capacity)

This is the case in which the request is received by a node i, with $k_i \neq k$, that can actually contribute to satisfy it, at least partially, because $C_i^{avail} \geq 0$. In this situation the node first executes a `ChangeServiceAction(k)` to fulfill the request. In the case $C_i^{avail} < C$, the request is only partially fulfilled, and therefore the remaining capacity should be obtained from another node y such that $k_y = k_i \wedge C_y^{avail} > 0$ by generating a `ServiceChangeRequest(y, k, C − C_i^avail, H+1)`.

If the `ChangeServiceAction` cannot be executed, then the node performs the same actions as in Case 1.

Case 4 (Node overloaded)

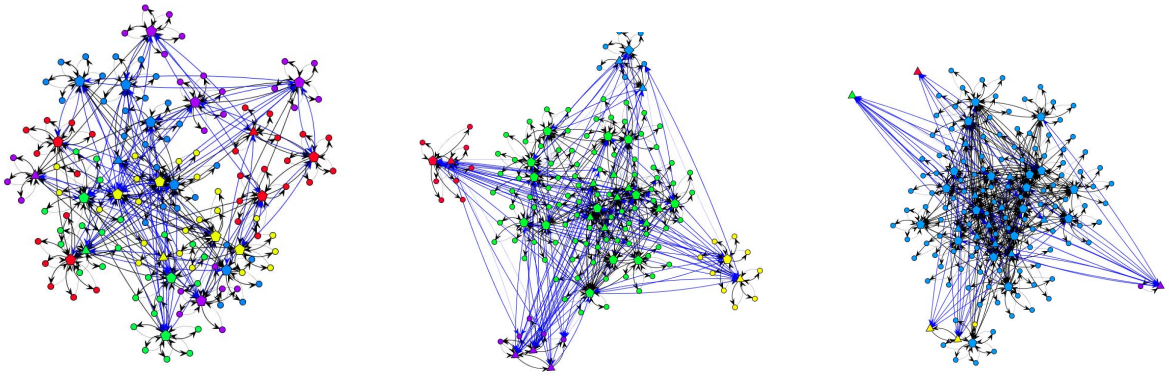
This is the case in which the request is received by an overloaded node i such that $C_i^{avail} < 0$. The action of the node in this case is the selection of a neighbor $x \in Ne$ such that $k_x \neq k \wedge C_x^{avail} > 0$ and, if it exists, it generates a `ServiceChangeRequest(x,k, C, H+1)`.

D. Implementation

The Mycocloud approach has been divided into components written in Java, which support the *Peerlet* model of the *ProtoPeer* toolkit [12]. This choice facilitates testing and reusing of the code both in a real system and a simulation environment. Protopeer help decoupling components from all the parts that would be dependent on a live networked system such as event dispatching, network communication, utilization monitoring, and service class change. The components we have implemented, available for download in [20], are described as follows.

- *Topology Management Components*: responsible for managing the links among the nodes. The links are decentrally modified according to the NewsCast protocol [30], to discover and connect new nodes, and to the

[1]High Myconet role is correlated to a high capacity and a high number of neighbor nodes



(a) Initial topology (b) Peak of green class (c) Peak of blue class

Figure 2. Topology representation of the evolution of a network of 200 nodes and 5 services when running Mycoload. Node colors represent service class, node shapes represent Myconet state (circle is Biomass, triangle is extending, square is Branching, pentagon is Immobile), node sizes represent the capacity. The network first receives a peak of green requests, then a peak of blue requests.

Myconet protocol to reconfigure the overlay structure of existing nodes (see Section III-A).

- *Load Balancing Components*: responsible for balancing the queues of directly connected nodes, which use nodes capacities as weights (see Section III-B).
- *Changing Service Components*: responsible for monitoring incoming requests and incoming messages. Based on the monitored information, overload events and service change request events are generated locally. These components also provide the logic to handle them, as explained in the previous subsections. This component may interact to the rest of the system by asking the system to change its service class, or to send a `ServiceChangeRequest` to another node directly connected to it.

Figure 2 shows the topology of a network of 200 nodes and 5 classes, which is generated by our implementation, when the system is started, and after it evolves in response to large amounts of requests of the same service class.

V. Evaluation

We present hereby experiments that show the behavior of Mycocloud with respect to its goal, as stated in Section II. In particular, we evaluate its capability to self-adapt the total capacity for each class, in order to match incoming load and service utilization, and thus reduce the system response time.

A. Setting up the Experiments

To evaluate Mycocloud we used ProtoPeer [12], a well-known Java-based simulator for P2P systems. Simulation remains an indispensable experimental approach for validating and evaluating P2P protocols at scale, as pointed out for example in [26] [2]. We have instrumented the simulator with the components we have described in Section IV-D together with additional components we have implemented for generating the load and measuring the results. In particular, the *load generation component* injects load into the system in terms of requests to be executed, which are sent to nodes that belong to the same class of request. Traffic is injected in a round-robin way, which intentionally leads to an unbalanced workload since nodes may have different capacities. For what concerns the *monitoring component*, it measures the average response time, the utilization, the current active class, and the overhead traffic generated in terms of messages exchanged. The network is configured with a fixed delay of 1ms to deliver each message. All the developed software is available at [20].

The constants of our experiments are:

- $C_{min} = 15$, $C_{max} = 60$, and $C_\alpha = 2$, which describe how the capacity is distributed among the nodes. In particular, the capacity is exponentially distributed with parameter C_α, such that $P[C_i = a] = a^{-C_\alpha}$, $i \in N$, and then capped between C_{min} and C_{max}.
- $\mu = 1$ request/second, which is the request throughput per capacity unit.
- $U_{max} = 50\%$, which is the utilization threshold.
- $H_{max} = 8$, which is the maximum amount of times a request can be forwarded.

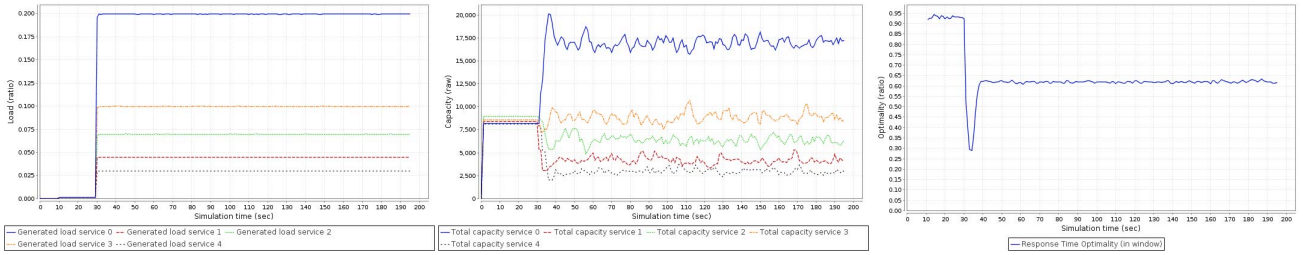
The variable input parameters are:

- $|N|$: total number of nodes in the system.
- $|K|$: total number of service classes in the system.
- $\lambda_j(t)$: incoming requests per class $j \in K$ at time t.

The output parameters of our experiments are:

- *RTO (response time optimality)*, defined as ORT/R. R is the measured response time, while ORT is the optimal response time, i.e., the minimum response time predicted by a centralized oracle that assumes that all the capacity is aggregated on a single node that can process all the classes of requests, with no network latency. This is the main performance metric of our evaluation.

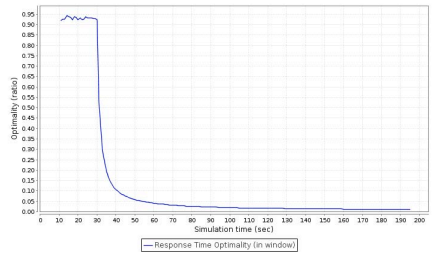
[2] *"A fundamental problem [...] is the evaluation of new protocols. So the technology of P2P network simulation has become a main method for understanding, researching and evaluating the P2P network algorithms and protocols."* [26]



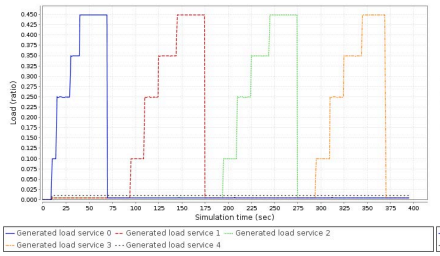
(a) Exp 1: Incoming load per service

(b) Exp 1: Capacity distribution

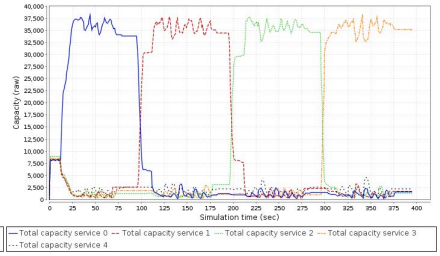
(c) Exp 1: RTO (with MycoCloud)



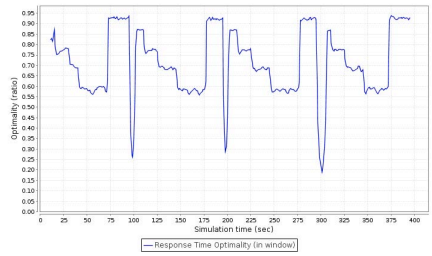
(d) Exp 1: RTO (without MycoCloud)



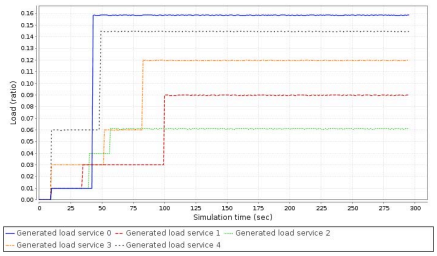
(e) Exp 2: Incoming load per service



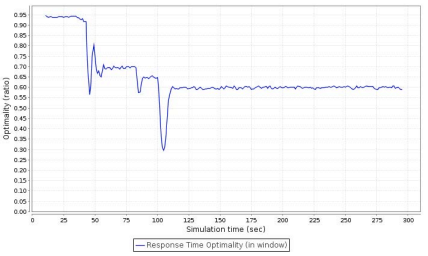
(f) Exp 2: Capacity distribution



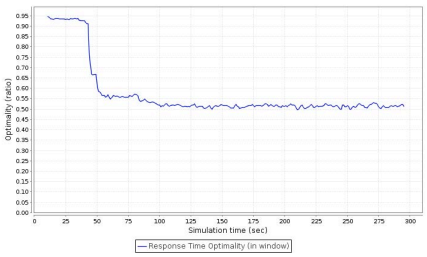
(g) Exp 2: RTO (with MycoCloud)



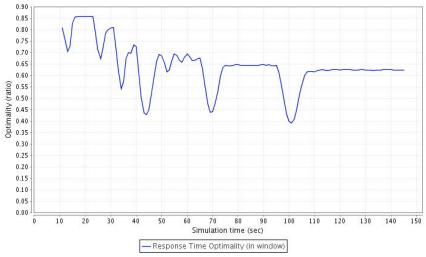
(h) Exp 3: Incoming load per service



(i) Exp 3: RTO (with MycoCloud)



(j) Exp 3: RTO (without MycoCloud)



(k) Exp 4: RTO (with MycoCloud)

Exp	A: with MycoCloud	B: without MycoCloud	A-B: Cost
1	5.8	2.6	3.2
2	6.9	2.7	4.2
3	5.5	2.9	2.6
4	7.8	2.9	4.9

(l) #MSGS (node messages per second)

Figure 3. Results of Experiment 1 (single service overload), Experiment 2 (multiple peaks), Experiment 3 (no overload), and Experiment 4 (scalability).

- *#MSGS* (number of network messages per node per second), which is defined as the number of messages exchanged by the combined Mycocloud approach for each node in each second, defined as the sum of the messages used to reconfigure the topology, to balance the load, and to exchange service change requests. This is the main cost metric of our evaluation.

- *Capacity Distribution of a class* $k \in K$, which is calculated as $\sum_{i \in N \wedge k_i = k} C_i$. It represents the total amount of capacity of the system for each class k. This parameter is important to verify whether the capacity distribution follows the same changes of the incoming requests.

B. Experiment 1: single service overload

Variable input parameters: $|N| = 1000$, $|K| = 5$, $\lambda_j(t)$ as in Figure 3a.

In this experiment we show a situation in which the amount of the incoming requests for some services (services 0 and 3 in Figure 3a) is larger than the available capacity for such services (see Figure 3b). After the requests start to arrive, we can see that Mycocloud modifies the capacity

of the service in such a way to adapt it to the current demand. The performance, measured by the RTO, goes in 8s from 28% to 63%. If we run the same load on the same system in which we only use topology reconfiguration and load balancing, without the Mycocloud approach for changing services, we obtain a bad divergent situation, in which queues of the overloaded service diverge, quickly leading to the RTO crashing, as shown in Figure 3d.

C. Experiment 2: multiple peaks

Variable input parameters: $|N| = 1000$, $|K| = 5$, $\lambda_j(t)$ as in Figure 3e.

This experiment is similar to Experiment 1, but with the load distributed in a way that some of the services have a peak at different times (see Figure 3e). Also in this case we can see an improvement in the distribution of service capacities, which adapts to incoming traffic (see Figure 3f). The RTO drops to 20-25% after each peak, but the algorithm brings it up over 60% in about 5-10s, as shown in Figure 3g, that is, well before the peak culminates, which shows a good capability to adapt in situations that are demanding in terms of capacity but also very dynamic in terms of incoming load.

D. Experiment 3: no overload

Variable input parameters: $|N| = 1000$, $|K| = 5$, $\lambda_j(t)$ as in Figure 3h.

In this experiment we test a different scenario, in which the incoming load does not overload any of the service classes, but puts them at times above the service utilization threshold U_{max}. The load for each service presents step-wise increases of different magnitude, which occur at random moments. An example is shown in Figure 3h.

Mycocloud is challenged in this case to find a better service placement configuration, since the system is near an equilibrium, but multiple classes are under stress at the same time, and request additional resources to one another, in an attempt to return to a utilization level below U_{max}. Even in such a case, Mycocloud is able to efficiently self-organize to a stable configuration that achieves an RTO of about 60% (see Figure 3i). If we de-activate Mycocloud and run the same load profile, the only adaptation comes from Mycoload, which – since no service capacity is ever saturated – is able to balance the load, but achieves a RTO level of only about 55% (see Figure 3j). These results show that Mycocloud is able to improve the service placement configuration also when there are no overloaded services.

E. Experiment 4: scalability

Variable input parameters: $|N| = 10000$, $|K| = 20$, $\lambda_j(t)$ has the same proportions of Experiment 3, but with 20 classes of services.

This experiment is analogous to Experiment 3, but with a much larger network, in order to test its scalability. We can see from Figure 3k that the system stabilizes also in this case to a RTO of about 62.5%, which is comparable to the results obtained in Experiment 3.

F. Discussion

All the experiments show how Mycocloud is able to closely follow the dynamics of the incoming load, and efficiently adapt the service placement configuration accordingly. Even in Experiment 3, which was designed to induce resource contention among the service classes, Mycocloud converges quickly, and improves the response time, with a small but appreciable improvement of RTO of about +5%.

In terms of optimality, we consider reaching an RTO level of 60% or more a positive result. In fact, since we could not draw from the literature a congruent benchmark on service placement for elasticity to compete with, to evaluate Mycocloud we used an oracle, whose assumptions, as discussed in Section V-A, are much less realistic than those of Mycocloud, and which is very demanding and impossible to emulate in practice.

The cost to reduce service response time is paid in terms of exchanged messages between nodes. In Figure 3l we show the measured value for #MSGS in the different scenarios, and compare it to the same scenarios, but with only topology reconfiguration (Myconet) and load balancing (Mycoload) activated. We can see that in all cases the extra cost of Mycocloud is under 5 messages per second per node, which is still in the same order of magnitude of the cost we have with only topology reconfiguration and load balancing.

VI. Related Work

As already mentioned in the introduction, there is extensive research on the topics of elasticity [6, 11], load-balancing [22, 25], multiple cloud [2, 15, 21], and service placement [1, 4]. In this section, for the sake of space, we focus exclusively on service placement as it is the piece of work that is closest to Mycocloud.

The problem of service placement is studied in literature as the optimal allocation of services to cloud resources according to a given criterion. Moens *et al.* [18] address the placement problem to optimize the latency and the cost of a cloud service-oriented application. They solve it in a centralized way using evolutionary optimization heuristics. They experimented with up to 15 nodes and 4 services. Ooi *et al.* [23] focus on a hierarchical control over placement of services: each service is called a "team", which receives advertisement from available resources, and decides which one to use or discard, with the purpose of obtaining an optimal level of performance and replication. They experimented with up to 30 nodes and 5 services. Mycocloud differs from these approaches because of its decentralized nature (vs. hierarchical) and its ability to handle system of a much higher scale. Another class of placement work focuses on VM migration. Some centralized approaches are proposed by Ghanbari *et al.* [13] (which focuses on resource costs) and Breitgand *et al.* [4] (which focuses on energy saving and load-balancing). Some decentralized approaches are proposed by Barbagallo *et al.* [3], which

propose a gossip-based algorithm to reduce the total energy consumption in a data center; and Sedaghat *et al.*, which offer a highly scalable decentralized agent-based peer-to-peer approach for optimizing the placement of VM in a cloud computing system, in particular they focus on the double optimization problem of maximizing the utilization of the machines while reducing their cost through resources consolidation, which are problems related, but different from the specific ones we address with Mycocloud.

VII. Conclusion

We have presented Mycocloud, a self-organized algorithm that calculates service placement for elasticity in a decentralized cloud computing environment. Mycocloud is based on a heuristic placement, which requires only local knowledge and is continuously executed on each individual node hosting a service. We have implemented Mycocloud using the ProtoPeer toolkit, and validated it with a set of diverse simulation scenarios.

Mycocloud is effective, as a complement or an alternative to recruiting additional resources via auto-scaling, whenever some services are overloaded vis-a-vis their capacity, while others are under-utilized. Evaluation results show the validity of our approach in supporting elasticity, and indicate a net improvement with respect to the same scenarios without service placement. Mycocloud also brings about the typical benefits of self-organized approaches, such as scalability with respect to the number of peers (up to 10,000 nodes in our experiments) without performance degradation, and resilience to dynamism.

Future work directions include an evaluation carried out in a distributed and networking environment, an analysis of the effects of delays for changing or migrating services, and an extension in which nodes host multiple services at once, and split their total computational capacity among them.

VIII. Acknowledgements

Daniel J. Dubois has been partially funded by the People Programme (Marie Curie Actions) SPANDO FP7 project n. 629982. Elisabetta Di Nitto has been partially funded by the SeaClouds FP7 project n. 610531.

IX. References

[1] A. Andrzejak, S. Graupner, V. Kotov, and H. Trinks. Algorithms for self-organization and adaptive service placement in dynamic distributed systems, 2002.

[2] O. Babaoglu, M. Marzolla, and M. Tamburini. Design and implementation of a p2p cloud system. In *SAC'12*, pages 412–417, 2012.

[3] D. Barbagallo, E. Di Nitto, D. J. Dubois, and R. Mirandola. A bio-inspired algorithm for energy optimization in a self-organizing data center. In *Self-Organizing Architectures*, pages 127–151. 2010.

[4] D. Breitgand, A. Marashini, and J. Tordsson. Policy-driven service placement optimization in federated clouds, 2011.

[5] N. M. Calcavecchia, B. A. Caprarescu, E. Di Nitto, D. J. Dubois, and D. Petcu. Depas: a decentralized probabilistic algorithm for auto-scaling. *Computing*, 94(8-10):701–730, 2012.

[6] E. Caron, L. Rodero-Merino, F. Desprez, and A. Muresan. Auto-Scaling, Load Balancing and Monitoring in Commercial and Open-Source Clouds. Research Report RR-7857, 2012.

[7] A. Chandra, J. Weissman, and B. Heintz. Decentralized edge clouds. *Internet Comp.*, 17(5):70–73, 2013.

[8] C. Chekuri and S. Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *J. on Comp.*, 35(3):713–728, 2005.

[9] V. D. Cunsolo, S. Distefano, A. Puliafito, and M. Scarpa. Volunteer computing and desktop cloud: The cloud@ home paradigm. In *NCA'09*, pages 134–139, 2009.

[10] D. J. Dubois, Y. Bando, K. Watanabe, and H. Holtzman. Shair: Extensible middleware for mobile peer-to-peer resource sharing. In *ESEC/FSE'13*, pages 687–690, 2013.

[11] G. Galante and L. C. E. d. Bona. A survey on cloud computing elasticity. In *UCC'12*, pages 263–270, 2012.

[12] W. Galuba, K. Aberer, Z. Despotovic, and W. Kellerer. Protopeer: A p2p toolkit bridging the gap between simulation and live deployement. In *Simutools'09*, pages 60:1–60:9, 2009.

[13] H. Ghanbari, M. Litoiu, P. Pawluk, and C. Barna. Replica placement in cloud through simple stochastic model predictive control. In *IEEE Cloud'14*, pages 80–87, June 2014.

[14] C.-C. Lin, H.-H. Chin, and D.-J. Deng. Dynamic multiservice load balancing in cloud-based multimedia system. *IEEE Syst. Jour.*, 8(1):225–234, 2014.

[15] L. Mashayekhy, M. Nejad, and D. Grosu. Cloud federations in the sky: Formation game and mechanism. *Trans. on Cloud Computing*, PP(99):1–1, 2014.

[16] P. Mayer, A. Klarl, R. Hennicker, M. Puviani, F. Tiezzi, R. Pugliese, et al. The autonomic cloud: a vision of voluntary, peer-2-peer cloud computing. In *SASO Workshops '13*, pages 89–94, 2013.

[17] E. Miluzzo. I'm cloud 2.0, and i'm not just a data center. *Internet Comp.*, 18(3):73–77, 2014.

[18] H. Moens, B. Hanssens, B. Dhoedt, and F. De Turck. Hierarchical network-aware placement of service oriented applications in clouds. In *IEEE NOMS'14*, pages 1–8, May 2014.

[19] D. G. Murray, E. Yoneki, J. Crowcroft, and S. Hand. The case for crowd computing. In *MobiHeld '10*, pages 39–44, 2010.

[20] Mycocloud Test Implementation. http://www.doc.ic.ac.uk/~djdubois/mycocloud.tar.bz2.

[21] E. D. Nitto, M. A. A. d. Silva, D. Ardagna, G. Casale, et al. Supporting the development and operation of multi-cloud applications: The modaclouds approach. In *SYNASC'13*, pages 417–423, 2013.

[22] K. A. Nuaimi, N. Mohamed, M. A. Nuaimi, and J. Al-Jaroodi. A survey of load balancing in cloud computing: Challenges and algorithms. In *NCCA'12*, pages 137–142, 2012.

[23] B.-Y. Ooi, H.-Y. Chan, and Y.-N. Cheah. Dynamic service placement and replication framework to enhance service availability using team formation algorithm. *J. Syst. Softw.*, 85(9):2048–2062, 2012.

[24] M. Ryden, K. Oh, A. Chandra, et al. Nebula: Distributed edge cloud for data intensive computing. In *IC2E'14*, pages 57–66, 2014.

[25] S. Shaw and A. Singh. A survey on scheduling and load balancing techniques in cloud computing environment. In *ICCCT'14*, pages 87–95, 2014.

[26] G. Shi, Y. Long, H. Gong, C. Wan, C. Yu, X. Yang, and H. Zhang. A high scalability p2p simulation framework with measured realistic network layer support. In *IPCCC 2008. IEEE International*, pages 311–318. IEEE, 2008.

[27] P. L. Snyder, R. Greenstadt, and G. Valetto. Myconet: A fungi-inspired model for superpeer-based peer-to-peer overlay topologies. SASO'09.

[28] G. Valetto, P. L. Snyder, D. J. Dubois, E. Di Nitto, and N. M. Calcavecchia. A self-organized load-balancing algorithm for overlay-based decentralized service networks. In *SASO'11*, pages 168–177.

[29] L. M. Vaquero and L. Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *SIGCOMM Comput. Commun. Rev.*, 44(5):27–32, 2014.

[30] S. Voulgaris, M. Jelasity, and M. Van Steen. A robust and scalable peer-to-peer gossiping protocol. In *Agents and Peer-to-Peer Comp.*, pages 47–58. 2005.

[31] Q. Zhang, Q. Zhu, M. F. Zhani, et al. Dynamic service placement in geographically distributed clouds. *J. on Selected Areas in Comm.*, 31(12):762–772, 2013.

[32] H. Zhuang, R. Rahman, and K. Aberer. Decentralizing the cloud: How can small data centers cooperate? In *P2P'14*, pages 1–10, 2014.