Politecnico di Milano

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

MycoCloud
Improving QoS by managing elasticity of services in decentralized clouds

Advisors: **Prof. Elisabetta Di Nitto, Prof. Giuseppe Valetto**

Co-Advisor: **Dr. Daniel J. Dubois, Dr. Paul L. Snyder**

Master thesis by: **Donato Lucia**

ID 767372

Academic Year 2012-2013

To my family and
to my friends.

Abstract

Tre le problematiche più attuali nel campo dell'ingegneria del software si annoverano sicuramente quelle legate alla crescente complessità nei sistemi informatici distribuiti. Questo deriva dal constante aumento di utenti con "elevata mobilità", ovvero utenti che accedono e usufruiscono di infrastrutture e applicazioni da diverse posizioni geografiche rendendo imprevedibile il dimensionamento dei sistemi. In tali scenari risulta necessario che l'allocazione di risorse computazionali e di memorizzazione sia dinamica e automatizzata, così da rendere i sistemi capaci di adattarsi a condizioni di carico non completamente prevedibili.

L'emergente campo di ricerca dell'*Autonomic Computing* affronta queste problematiche, analizzando efficienza, scalabilità e in generale elasticità in complessi sistemi decentralizzati. Attraverso questi studi si cerca di formulare e implementare protocolli e meccanismi automatici che siano in grado di sostituire le tradizionali architetture di rete (*topologie*) statiche e con gestione centralizzata. Approcci innovativi in questo senso sono basati su tecniche di "meta-design", ovvero framework concettuali con la quale si cerca di definire e progettare infrastrutture collaborative partendo dall'analisi di comportamenti sociali e/o animali. Tra questi si annoverano sistemi naturali e biologici che includono Amorphous Computing, Swarm intelligence e Cellular Automata. Da questi modelli si osserva come si riesca a dar vita a sistemi che evolvono in maniera autonoma applicando regole in locale senza avere una conoscenza globale.

Lo scopo di questi studi è quello di dare *flessibilità* ai sistemi distributi, ovvero renderli capaci di allocare/deallocare risorse computazionali in modo autonomo e reattivo. Tale flessibilità, che è alla base del modello di *utility-computing*, è anche definita come "elasticità". Essa è evidentemente non ottenibile con sistemi statici e gestione centralizzata in quanto non sono in grado di reagire velocemente ed efficientemente.

Questa flessibilità è in parte abilitata dal modello di risorse *on-demand* delle infrastrutture di *Cloud Computing*. In tali infrastrutture viene garantito bilanciamento del carico (*load-balancing*) tra le risorse e meccanismi di

scale-up (assegnamento di ulteriori risorse) e scale-down (rimozione di risorse inutilizzate) che adattano la capacità computazionale al carico di richieste in input in modo automatico (*auto-scaling*).

Il modello proposto dalle instrastutture di Cloud Computing tuttavia presenta delle limitazioni. Innanzitutto la quantità di risorse è tipicamente limitata per capacità hardware o comunque da contratto dal Cloud provider; inoltre, in base allo stato attuale del Cloud (connettività) e alle condizioni di mercato (costi) un Cloud provider può non essere la scelta più conveniente in un certo momento.

Queste limitazioni stanno spingendo la ricerca alla formulazione di modi efficaci per sfruttare sistemi di Cloud che collaborano per fornire diversi servizi. La capacità di federare Cloud providers differenti diventa particolarmente utile quando piccole società vogliono offrire parte delle loro risorse al mercato Cloud, ad un prezzo competitivo.

A seguito di queste analisi, questa tesi si pone come obbiettivo quello di sperimentare soluzioni che permettano una completa decentralizzazione nella gestione di Cloud decentralizzati e in generale di sistemi distribuiti su larga scala. A tal fine vengono riformulate le soluzioni di bilanciamento del carico e di assegnamento/rimozione di risorse computazionali in modo tale da renderle decentralizzate e non affidate ai singoli Cloud providers. In particolare, queste risorse vengono clusterizzate (raggruppate) in base al servizio offerto e saranno in grado di cambiare il proprio servizio, spostandosi quindi da un cluster ad un altro. Il *cambio servizio*, che sfrutta euristiche di posizionamento, ha lo scopo di garantire auto-adattamento alle condizioni di carico dei servizi, massimizzando così le prestazioni del sistema (QoS).

Table of Contents

List of figures

List of Algorithms

Capitolo 1

Introduction

1.1 Context

One of the today issues in software engineering is to find new effective ways to deal intelligently with the increasing complexity of distributed computing systems. An increasing number of users with greater mobility are constantly requiring more sophisticated functionality from larger applications running on faster architectures. Consequently, computer engineers are gradually leading to rethink the traditional perspective on distributed systems design, replacing systems with fixed topologies, central authorization or supervision with systems highly decentralized and self-managed.

Different paradigms to handle such complex systems and solve large-scale problems have been proposed. The original client/server model led to the evolution of dedicated clusters, followed by grids, and then by large-scale data centers. The last stage of virtualization comes from cloud computing infrastructures that enable on-demand allocation of computational resources.

Cloud Computing model provides a first version of flexibility thanks its auto-scaling mechanisms that adapt the system capacity according to the load generated by service requests. This flexibility in the amount of resources provided is often called *elasticity* and is the basis of the *utility computing model* [35, 9].

Regarding auto-scaling mechanisms, both the research community and the main cloud providers have already developed auto-scaling toolkit [28, 2]. However, most research solutions are centralized and typically bound to the li-

mitations of a specific cloud provider in terms of resource prices, availability, reliability and connectivity. In fact the total amount of resources that can be allocated to a single customer is usually limited by the contract and by the total capacity of the Cloud provider. Moreover auto-scaling services may have an additional cost. Another problem is that, depending on the current state of the cloud infrastructure (i.e., connectivity) and on the market conditions (i.e., costs) a single cloud provider may not be the most convenient at a certain moment.

These limitations are moving current research efforts in Cloud computing to the study of efficient ways to exploit more than a single Cloud system for the deployment of a service. The capability to federate different Cloud providers becomes particularly useful when smaller companies want to offer part of their resources for a competitive price in a cloud market. In a scenario like this we would have several cloud computing systems with less capabilities, guarantees, and stability than a single specialized cloud provider.

The extreme scale and dynamism however, call for robust and efficient protocols capable to self-organize and self-repair the whole system in case of a serious failure in one Cloud provider or in its connectivity. A crucial issue comes also from the need to ensure an high QoS (Quality of service) in the network respect to the requests execution. In this context important is the effort provided by the load-balancer as well as the auto-scaling mechanisms, which are in charge to make the system more flexible against events in the network (e.g. peaks of load, churn).

So far load balancing approaches have been designed for networks with fixed or dynamic topologies having a perfect knowledge of the whole network. They therefore do not address the needs to have a decentralized algorithm where each node is able to delegate exceeding jobs to others, without knowledge of the whole network.

These limitations, whose resolution is one the objective of this work, are in the case of load-balancer based on a not complete extension for every dynamic characteristics of the networks, while in case of auto-scaling solutions are due to the fact that they are still based on centralized algorithms. We want thus design an algorithm that gives mechanisms for a decentralized auto-scaling among the services, giving elasticity to the system.

Our system, denoted with the name MycoCloud, will be fully decentra-

lized, self-organizing and self-adapting, thus reducing the human effort at deployment time in reconfiguring the whole system in case of events in the network (i.e., peaks load, churn). This thanks to the usage of *autonomic self-aggregation techniques* [27] that rewire such highly dynamic systems in clusters (Figure 1.1). Clusters, which are grouped with homogeneous nodes having same characteristics will have a further property of *elasticity*. Nodes "belonging" to a certain cluster will be autonomously able to make decisions whether to ask for capacity to other clusters based on their capacity and on the number of received requests. They will be able to change its own service and "migrate" to another cluster, making the system more elastic and with an improved QoS against load or churn events in the network.

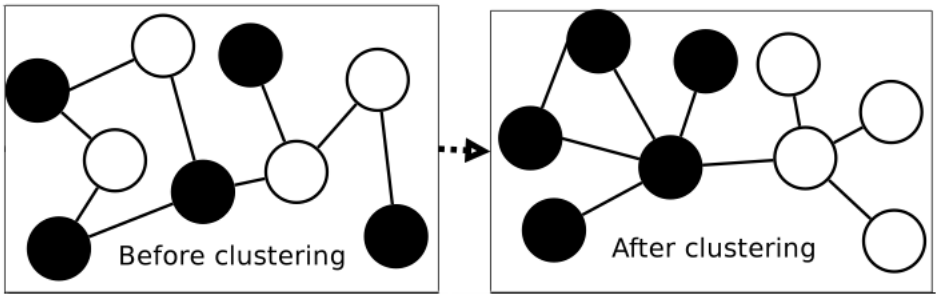


Figura 1.1: Clustering example.

1.2 Background

The emerging field of *Autonomic Computing* is addressing the issues of efficiency, scalability and in general of elasticity in complex decentralized systems by formulating and deploying adaptive, self-configuring, and self-managing protocols and mechanisms that relieve the persistent need for human-driven management. While policies are defined by humans, their deployment, enforcement, and dependency conflict resolution should all be handled by automatic mechanisms to optimize system performances.

However, achieving this vision requires conceptual, physical and logistical modifications to existing systems and protocols. A key issue that is significantly impacting emerging networks and applications, which can be potentially addressed by autonomic communication, is the absence of accurate knowledge and control over topologies of large networks. Network topologies define the

link relationships between the nodes in the network, and have a direct impact [16] on scalability, routing, fault tolerance, security and performance of a distributed system.

The management of such complex systems cannot be handled neither by a central authority or a fixed communication topology. For this reason, the research has post attention in the creation of highly-decentralized large-scale distributed systems, which can be extremely large, scaling to millions of resources (also called Peers). Innovative approaches come from "meta-designing" models that provide to the system self-assembly, self-regulation and evolution. These models come from nature and biological systems [13] including Amorphous Computing [5], Swarm Intelligence [30, 8], and Cellular Automata [40]. Amorphous Computing applies concepts from biology and evolution to develop computer languages and decentralized evolving systems [26]. Swarm Intelligence is inspired by social and behavioral theories in the animal and human kingdom. For example, foraging, nest building, and burial activities of social insects (ants, wasps, termites) follow local rules applied by each entity with no knowledge of the whole. Similarly flocks of birds, schools of fish, herds of mammals, follow local rules that lead to structures with emergent properties.

Biologically-inspired metaphors and models have been lately the subject of active research [20], as they hold promise to enable properties, such as resilience, emergent adaptation, and self-organization, that are desirable for large-scale, distributed systems.

An interesting work called Mycoload [37] investigated an unstructured overlay network, in which nodes that host instances of many different service types are self-organized into virtual clusters. Nodes "belonging" to a virtual cluster do not need to be physically close, rather, they are logically interconnected. They are able to efficiently balance the load generated by incoming requests among themselves, while the overlay as a whole is responsible for routing those requests to the right virtual cluster. This work use a bio-inspired super-peer-based overlay substrate called Myconet [34], which can automatically re-configure its topology.

Mycoload thanks the properties provided by Myconet, deals effectively with a number of dynamic dimensions, including the churn of nodes participating in the decentralized service network, the variation in the number of service types hosted by those nodes, the fluctuations of the traffic entering the network, and

self-healing following highly disruptive network events.

However Mycoload does not deal with any auto-scaling mechanism, paying in elasticity with respect to a sudden increment of load. This instead is considered in a recent approach called DEPAS [10], in which the authors dealt effectively with the problem of decentralized auto-scaling to make the system elastic with respect the load. They designed an algorithm able to perform scaling actions selecting a random cloud provider in a cloud federation, do not using any placement heuristic. In this way however, the selection may not be the best one, since the total amount of resources that can be allocated to a single customer is usually limited by the contract and by the total capacity of the cloud provider.

1.3 Objectives

Given the limitations defined above, this thesis aims to design a decentralized self-adaptive network with the main objective to improve the elasticity among different services provided by Cloud infrastructures. We want reach a complete decentralized adaptation giving to the system the self-* properties needed. In particular we will on the services adaptation enabling a self-scaling property in the system, thanks to which the services grouped in clusters will be able to require/release resources (nodes) from/to other clusters. The algorithm will use a *probabilistic placement heuristic* thanks to which the selection will take into account at the cost of selecting one node.

To achieve this goal we propose MycoCloud approach: a three algorithm layers system that tries to improve QoS in decentralized clouds by managing elasticity of services. The architecture is composed by:

Bottom layer A bio-inspired super-peer-based overlay substrate, which can automatically re-configure its topology.

Middle layer A Self-organized load-balancing algorithm for overlay-based decentralized service networks, through which nodes are able to make their queue lengths proportional to their computational capacity.

Top layer An algorithm that gives to the system an adaptive behavior against increasing/decreasing incoming requests, using a precise placement heu-

ristic. The nodes organized in clusters, will have the ability to ask for further resources sending a *service change request* in the network.

Finally to validate MycoCloud we want use simulations and real experiments deploying it on Cloud networks. To do this, we found the need to build a deployment tool allowing us to deploy MycoCloud in real/virtual machines (e.g., cloud VMs), since the state of the art does not offer something able to do that. Moreover we want design and implement this tool in order to deploy general P2P systems, in which there is the need to execute large-scale experiments.

1.4 Achieved Results

In this thesis we have designed and built MycoCloud reaching a fully decentralized self-adaptive network. This system has shown how is able to react and improve the performances against every type of event in the network. This implementing a three algorithm layers system composed by an overlay manager, that build and self-repair the topology; a decentralized load-balancer and auto-scaling mechanism giving to the system a further elasticity, in particular during workloads peaks. We have seen:

(i) A good convergence rate (i.e., the amount of time that is needed to obtain a satisfactory configuration) in the overlay construction and self-repairing actions.

(ii) Good performances provided by the collaboration of the topology and the load-balancing algorithms looking at metrics such as Response time, network traffic (message exchanged) and system load. They will be discussed in the Chapter 5.

(iii) remarkable improvements provided by the service elasticity with respect a system without adaptation. This has shown the validity of our solution.

To validate our approach we have executed several tests by running large scale simulations (reaching 10 thousands of nodes). First we performed different tests using a *non-adaptive* version on MycoCloud excluding the *Top*

layer. This gave us an evaluation of the performances without the adaptive algorithm. Then we integrated the adaptive algorithm showing improved performances with respect the non-adaptive version. . Finally we reached a good outcome in the development of the deployment tool for live distributed experiments, denoted with the acronym LDE, which has proved to be useful in every type of such large-scale experiments and for MycoCloud, as well. We deployed MycoCloud on Amazon Cloud proving the validity of the tool, but for time constraints we do not collected metrics comparable with the simulations.

1.5 Outline of the Thesis

This thesis is organized as follows:

- Chapter 1 gives an overview of the context of the thesis.
- Chapter 2 discusses the state of the art, that is, what the research in this field has reached until now.
- Chapter 3 shows design and tests of our approach to the problem of elasticity in decentralized cloud infrastructures.
- In Chapter 4 there is the description of a new tool for deployment on large-scale P2P real networks.
- Chapter 5 reports and discusses the final results.

Finally, Chapter 6, concludes this thesis and shows some possible future extensions.

Capitolo 2

State of the art

In this chapter there is a discussion about some works provided by research activities that are close or interesting for this work. In particular in the first section there is an overview about valuable super-peer overlay networks that try to solve the problem of self-organize a complex distributed system. In the second are reported some approaches that try to improve the response time in unstructured networks through decentralized load-balancing and/or auto-scaling algorithms in highly-decentralized network, discussing the best qualities and the weaknesses with respect our approach. The third section discuss how has been addressed until now by other researchers the problem of how to test such large-scale systems in real environments. Finally, in the last section there is a comparison of different frameworks that help to test large networks like that ones we are considering in this work.

2.1 Bio inspired super-peer overlays

Recently the research has post attention in P2P networks, that is in the design of highly-decentralized, large-scale distributed systems, which can be extremely large, scaling to millions of peers, also having members highly unstable.

Most of the P2P applications deployed on the Internet were characterized by the absence of a specific mechanism for enforcing a particular overlay topology. Recently, however, even popular file-sharing applications [15] have started to consider more structured topologies, by introducing the concept of super-peer. Topologies, in this way are organized through a two-level hierar-

chy: nodes that are faster and/or more reliable than "normal" nodes take on server-like responsibilities and provide services to a set of clients. The super-peer paradigm allows decentralized networks to run more efficiently improving the performance of the entire network [25, 23]. Super-peers may take on service roles for other peers, such as indexing files, routing data, or forwarding searches. Designing effective super-peer-based overlay topologies for large-scale P2P networks is not simple, as no global view of the network exists. Further, such networks can be extremely dynamic since peers frequently join and leave (whether by failure or deliberate disconnection).

In this context new ways to think about unstructured overlay networks are emerging. They are inspired by nature and biological systems [13] include Amorphous Computing [5], Swarm Intelligence [30, 8], and Cellular Automata [40]. Amorphous Computing applies concepts from biology and evolution to develop computer languages and decentralized evolving systems [26]. In cellular automata each computing cell is viewed as equivalent to an organic cell that is guided by its environment in order to determine its next state. The cells follow virtual chemical gradient and density trails to identify their position and direction of evolution. Swarm Intelligence is inspired by social and behavioral theories in the animal and human kingdom. For example, foraging, nest building, and burial activities of social insects (ants, wasps, termites) follow local rules applied by each entity with no knowledge of the whole. Similarly flocks of birds, schools of fish, herds of mammals, follow local rules that lead to structures with emergent properties.

Interesting models come from a so called "morphogenetic engineering" approach. A new discipline that works toward new computing principles using neurons, ants and genes as models. Morphogenetic engineering's purpose is to abstracts the behavior of cells, termites and other natural agents into new principles of heterogeneous, controllable self-assembly. Research in morphogenetic engineering is positioned at the interface between the science and engineering of complex systems. It covers the computational modelling of self-organization phenomena and the design of decentralized and adaptive artificial systems inspired by these phenomena.

In [29], the authors discuss how to reproduce complex morphogenesis by investigating and combining its fundamental ingredients: self-assembly and pattern formation under genetic regulation. These comes from the idea that

outside biological and social systems, natural pattern formation is essentially "simple" and random, whereas complicated structures are the product of human design. So far, the only self-organized (un-designed) and complex morphologies that we know are biological organisms and some agent societies. The authors argue that understanding natural emergence should help to design a new generation of artificial complex systems by importing into our machines highly desirable properties that are still largely absent from traditional engineering: decentralization, autonomy (self-organization, homeostasis) and adaptation (learning, evolution). Future engineers would "step back" from their creation and only set generic conditions for systems to self-assemble and evolve, instead of building them directly.

This is the purpose of his work, that is to show how genetic-like regulation at the agent level can be used to control an artificial process of complex self-organization. The model he proposes is viewed from two different vantage points:

(a) *Pattern formation* on moving cellular automata, in which the cells spatially rearrange under the influence of their activity pattern,

(b) *Collective motion* in a heterogeneous swarm, in which the agents gradually differentiate and modify their interactions according to their positions and the regions they form by gradient propagation.

Through a precise interplay of genetic switches and chemical gradients in time and space, an elaborate form is created without an explicit architectural plan or design intervention. On an abstract level, this phenomenon can be described as a combination of self-assembly and pattern formation under the control of non-random genetic regulation stored inside each agent. Local morphogen gradients generated by a fully decentralized, peer-to-peer signalling system (a form of "gossiping"), provide positional information in input. This information can be used in different way to build any type of structure. For example the Fig. 2.1 shown how nodes that carrying various pairs of attachment ports (X, X') and corresponding gradient values (x, x') are driven to specific attachment locations through links that are dynamically created and removed based on "ports" and "gradients". Node ports can be "free" (not linked to other ports from other nodes) or "occupied" (linked), while free ports can be "open" (available for a

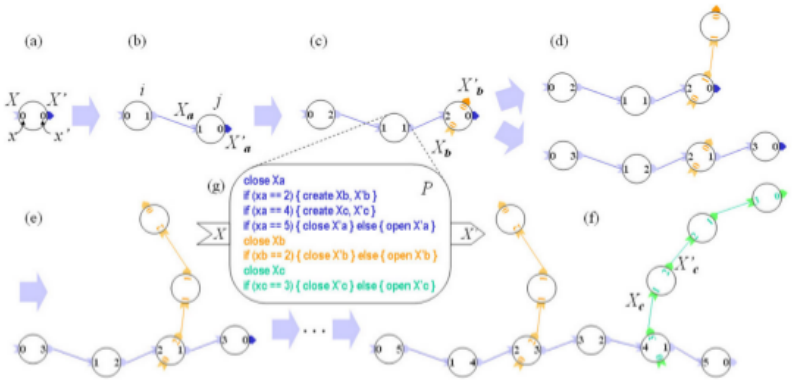


Figura 2.1: *Illustration of the self-assembly mechanisms of complex but precise network topologies by "programmed attachment". (Copyright ©2004-2011 Inderscience Enterprises Limited.)*

connection) or "closed" (disabled). New nodes that just arrived in the system's space, or nodes that are not yet connected, have both ports open and gradients set to 0.

These studies are very interesting since they are giving new challenges in the design of completely self-adapting networks (see Figure 2.2), able to maintain a certain "shape" in front of different event in the network.

Biologically-inspired metaphors and models have been lately the subject of active research, as they hold promise to enable properties, such as resilience, emergent adaptation, and self-organization, that are desirable for large-scale, distributed systems. A valuable example is an approach, that is realized not only as a model, is a bio-inspired super-peer based overlay network that can automatically re-configure its topology, called Myconet [34]. In Myconet, the robustness and sophistication of natural hyphal structures inspires interconnection strategies between peers and super-peers, the promotion of regular peers to super-peers, and the incremental aggregation of regular peers around super-peers.

It is based on models of fungal growth, and uses the concept of *hyphae* (the root-like structures of some species of fungi. Figure 2.3) to guide the self-organization of the network topology. Myconet uses a set of local rules and a hierarchy of peer states that are loosely modeled on hyphal growth dynamics. Peers may either be *hyphal peers* (i.e., super-peers that provide additional services to other peers) or *biomass peers* (less-powerful peers that rely on hyphal peers and their services when available). Hyphal peers switch

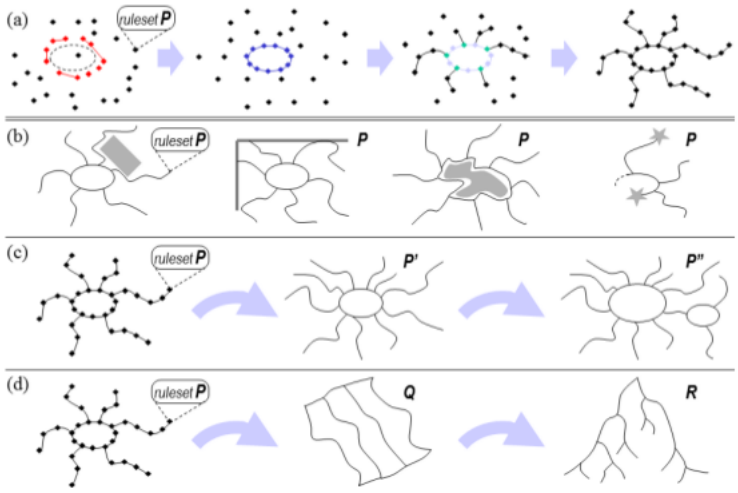


Figura 2.2: Illustration of various types of phenotypic adaptation in a programmable network growth model. (a) Stereotyped development: a certain genotype (port routine P) gives nodes a strong bias toward self-assembling into a certain shape, here a spider-like formation made of one ring and six legs. (b) Developmental "polyphenism": similar to a plant, the same P could give rise to variants of the above shape modified by external conditions from the environment, such as obstacles or attractors. (c) "Polymorphism": slight parametric variants of P may produce other structural variants, such as size of ring, number of legs, or ring location. (d) "Speciation": drastically different genomes create drastically different structures although there is no real qualitative difference with c: it is only a matter of degree and time scale of evolution.

between three further hierarchical protocol states: *extending*, *branching*, and *immobile*.

Each of the states above has a different role 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:

- *Biomass peers* perform only minimal roles in management of the topology.
- *Extending hyphae* act as attachment points, and continuously explore the network looking for new or isolated biomass peers to connect to.
- *Branching hyphae* grow new cross-connections with other hyphal nodes; they also regulate the number of extending hyphae in the network, at times selecting large-capacity biomass peers for promotion to the extending state, and at other times targeting excess extending hyphae for demotion.

- *Immobile hyphae* 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).

Figura 2.3: *Mycelium growing*

It 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 hypal peers that can best service them, while, on the other hand, ensure that hyphal peers that cannot efficiently reach their biomass peers number or other super-peers number targets will become candidate for demotion. It is important to notice that all Myconet rules operate locally. Peers make decisions based on their individual neighborhood, that is, the set of peers to which they maintain links.

The Myconet protocol effectively constructs and maintains a strongly interconnected, decentralized super-peer overlay unlike other unstructured approaches, dynamically adjusting the interconnections between super-peers, increasing efficiency and resiliency to the loss of peers. The Myconet overlay is designed to dynamically maintain a configurable number of links between super-peers to facilitate network tasks. These links exchange information used for topology adjustment, maintain strong interconnections to increase robustness in case of super-peer failure, and act as the communication substrate for application-level protocols running on the overlay. For several protocol

operations (bootstrapping and growing new inter-hyphal connections) random non-neighbor nodes are selected, through the use of a gossip-based mechanism that maintains a cache of known peers and the protocol states of those peers, that is a simplified version of the Newscast protocol [18].

Myconet demonstrates that the fungal metaphor is powerful when applied to peer-to-peer overlays and that it can be used to create a self-organizing peer-to-peer overlay that provides advantages over other approaches in terms of its robustness to failure.

All these self-organizing properties, as described in the next section, have been found useful in order to achieve a self-organized load-balancing among nodes.

2.2 Self-organized load-balancing

The literature concerning the problem of load-balancing is quite extensive and includes solutions for a large variety of scenarios ranging from multiprocessor computers, telecommunication networks, content delivery networks, grid environments and in general centralized and decentralized systems. Recently the problem of load-balancing in decentralized service networks has gained attention: in [32] Ranjan et al. propose a new software fabric to balance service provisioning requests among a set of VMs deployed in a cloud environment using a DHT based self-organizing routing structure. The services build upon the DHT routing structure (shown in Fig. 2.4) extends (both algorithmically and programmatically) the fundamental properties related to DHTs including deterministic lookup, scalable routing, and decentralized network management. Other works such as [14], provide some guarantees on the convergence time of the network to a stable state. However, they only deal with a fixed initial assignment of requests to each peer and no network dynamism concern is taken into account (i.e., the topology is stable).

Currently, the increasing diffusion of ubiquitous and pervasive frameworks living in extremely dynamic environments and experiment variations with disruptive event has escalated the need for robust and adaptive load-balancing techniques. In [31] the authors achieve a load-balancing behavior by modifying the degree of each node proportionally to the available computational resources at the node itself. The execution of a job within a node causes one of its links

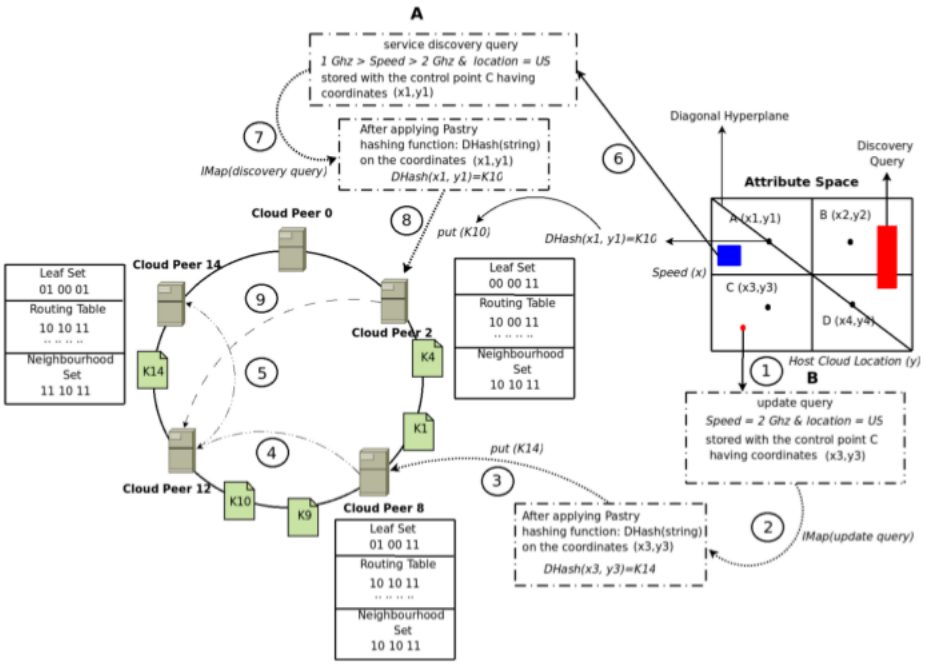


Figura 2.4: *A pictorial representation of Pastry (DHT) overlay construction, multi-dimensional data indexing, and routing: (1) a service hosted within a VM publishes a update query; (2) Cloud peer 8 computes the index cell, C(x3,y3), to which the update query maps by using mapping function IMap(query); (3) Next, distributed hashing function, DHash(x3, y3), is applied on the cell's coordinate values, which yields a overlay key, K14; (4) Cloud peer 8 based on its routing table entry forwards the request to peer 12; (5) Similarly, peer 12 on the overlay forwards the request to Cloud peer 14; (6) a provisioning service submits a service discovery query; (7) Cloud peer 2 computes the index cell, C(x1, y1), to which the service discovery query maps; (8) DHash(x1, y1) is applied that yields an overlay key, K10; (9) Cloud peer 2 based on its routing table entry forwards the mapping request to peer 12.*

to be removed while its completion adds a new link toward a new node. Biased random walks are used to find the best node that is going to execute the job, but they do not consider the case of nodes with different processing capabilities and the effects of churn, as we do in this work.

Another approach has been proposed by Li and Kameda [11]. They consider a system with heterogeneous nodes and jobs, meaning that each node is able to process all the jobs, but with a different rate for each job class. In this case the load-balancing problem is described as an optimization problem decomposed into simple rules to be run at node level in a decentralized way. The problem of load-balancing has also been considered in other domains. Examples are structured peer-to-peer networks to manage distributed hash

tables. In this context, the objective is to balance the association of keys to nodes according to nodes capacity. Even in this case, as we do, random probing of candidate nodes for load-balancing is used as a robust technique against churn.

All these approaches, however, rely on a structured peer-to-peer network model, and do not address the possibility of change the topology. They do not consider in fact churn scalability issues, which is, instead the aim of a recent work called Mycoload [37]. In this work in which has been investigated an extension of Myconet [34] in order to make it suitable to serve as the overlay management system for their load balancing approach.

With the help of Myconet protocol, nodes that host instances of each of many different types of services are able to self-organize into virtual clusters of same-type nodes. Nodes in a virtual cluster do not need to be physically close, rather, they are logically interconnected. They are able to efficiently load-balance incoming requests among themselves, while the overlay as a whole is responsible for routing those requests to the right virtual cluster. On top of that efficient topology, a self-organized load-balancing is applied to the queued requests for the various services residing in a decentralized service network. Moreover, as Myconet had originally been developed to manage peers irrespective of service type, another extension to the protocol was to make it aware of peer types. A peer may have at any given time both same-type and different-type neighbors, but it can only performs load-balancing operations with same-type ones.

Therefore different types of rules are applied when that peer manages its links to same-type vs. different-type neighbors, to ensure the incremental construction of neighborhoods in the overlay, in which one or more hyphal peers aggregate and serve a number of biomass peers of the same type. Load-balancing is performed between neighbors, however, in a previous work [18], since all peers were considered homogeneous in their computing capabilities, the goal of the algorithm was to balance the queues of service requests for neighboring peers as uniformly as possible. In Mycoload, because of the heterogeneous capacity of peers, the goal of load-balancing becomes making queue lengths proportional to the capacity of each peer. Based on this principle, when a load-balancing operation is performed, a peer p_α selects a random same-type hyphal neighbor p_β . If p_α is a biomass peer, it will have only a single neighbor,

a same-type hypha. If p_α is a hyphal peer, it will only try to balance its queue with other same-type hyphal peers; as biomass peers have only a single neighbor, they will be the ones to initiate the load-balancing operations with their parent. In this way, load-balancing operations tend to occur preferentially between the more capable super-peers, which helps ensuring that jobs will be balanced throughout the cluster. So p_α and p_β compare their queue lengths and "ideal" queue lengths are calculated by determining the total number of jobs in both queues and dividing the jobs proportionally based on p_α and p_β's capacities. If the queues are unbalanced, jobs are transferred from the queue that is over its ideal length to the queue that is under that length.

In order to achieve this goal in an efficient way, the network (Myconet) have to guarantee that nodes have in their neighborhood a good number of other nodes offering the same kind of service, also in the presence of network churn, which can occur due either to failures in the system or to nodes entering and leaving the network as part of their normal operation. Mycoload dealt effectively with a number of dynamic dimensions, including the churn of nodes participating in the decentralized service network, the variation in the number of service types hosted by those nodes, the fluctuations of the traffic entering the network, and self-healing following highly disruptive network events.

In Mycoload work however we can find some limitations. The first one is, even if it has a lot of good properties since is built on top a fully decentralized re-configurable network able to self-distribute load among nodes (of same type), it does not allow any scaling mechanism. In fact, if the load directed to a certain service (cluster) goes over the total capacity offered, the system will not be able to satisfy further requests. Another limitation, or better something left, is that it has been evaluated using simulations implemented on top of the Java-based PeerSim platform [19], that is cycle-based and lacks completely the notion of time.

An interesting approach slightly different, but for some aspects comparable with Mycoload comes from DEPAS [10], a decentralized probabilistic algorithm for scaling services in a cloud computing environment. As Mycoload it is built on top of an unstructured P2P network but it is focused in cloud environments. In fact, it discusses about the importance of auto-scaling, so that a system is able to scale when the (balanced) load increase. It points out that even if the main cloud providers have already developed auto-scaling solutions they are

centralized solutions and not suitable for managing large-scale systems, and moreover they are bound to the limitations of a specific provider in terms of availability, reliability and connectivity.

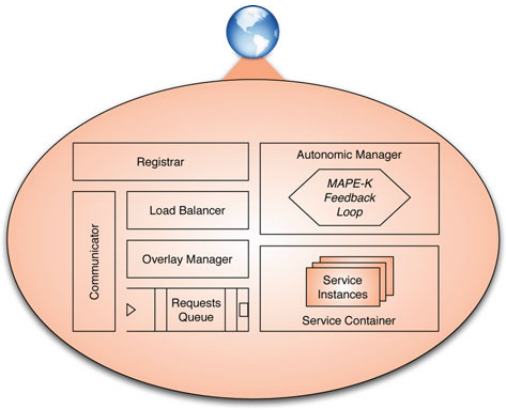


Figura 2.5: *Autonomic service architecture*

DEPAS tries to take advantage of a *cloud federation* proposing a cloud independent solution that allows the auto-scaling over multiple cloud infrastructures. The solution proposed by the authors (see Figure 2.6) is composed of many autonomic services, each one is responsible for processing requests and, at the same time, for running a process that makes decisions in a probabilistic way, whether to self-destroy or replicate the autonomic service by performing local monitoring activities.

All these tasks are enabled thanks to the sub-components that runs in each autonomic service, whose architecture is shown in Fig. 2.5:

- The *Communicator* allows the autonomic service to communicate with other autonomic services using overlay links.
- The *Service Container* instantiates the actual business service that is able to process the requests coming from the clients.
- A *Requests Queue* is employed to dispatch the incoming requests to the *Service Container*.
- The *Load Balancer* implements a decentralized load balancing algorithm that optimizes the size of the queue.
- A gossip-based algorithm is run by the *Overlay Manager* to maintain the links to the neighbors of the autonomic service.

- The *Registar* redister or un-register autonomic services to an external DNS.

- The *Autonomic Manager*, which is the component in charge of executing the auto-scaling algorithm.

The authors tested two different load-balancing algorithms to support the auto-scaling mechanism. The first is based on a DNS round-robin entry point combined with a decentralized approach at autonomic service level. The *Registrar* probabilistically decides to register to or un-register to an external DNS. So that a certain number of autonomic services is register in the DNS, and when an autonomic service receives a request, an admission function is used to decide whether the request will be processed by the current autonomic service, forwarded to a remote autonomic service, or rejected. The second one, which has been also used in Mycoload, balance the loads of two autonomic services exchanging their requests.

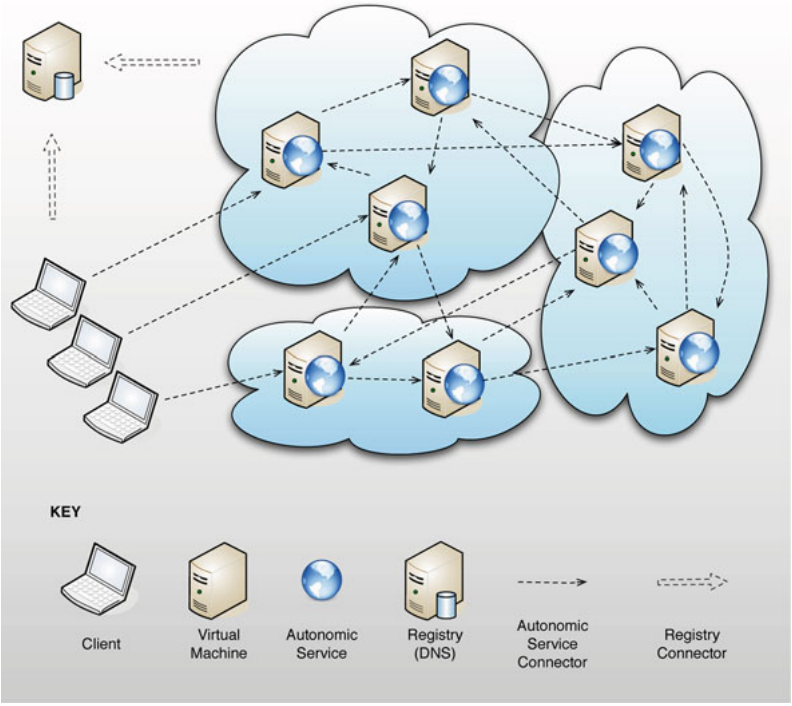


Figura 2.6: *DEPAS solution architecture*

Since the solution needs to operate also in presence of dynamism the system overlay is maintained connected and with a target degree (defined as the average number of neighbors per autonomic service). Even in situations

in which the appearance/disappearance of autonomic services is frequent and unpredictable the autonomic services are organized into an overlay network in which each autonomic service knows a fixed number of other autonomic services, called neighbors. The authors decided to support the overlay with an overlay management protocol, which is an adapted version of the gossip protocol developed by Jelasity et al. [24], because it was proven to be highly scalable, highly reliable with respect to autonomic service failures, and the operations of neighbor addition and removal required by the auto-scaling algorithm can be done with minimum effort.

Finally, the *Autonomic Manager* periodically retrieves the neighborhood load (average over the last time-frame) and if the load is less than the minimum load threshold then the possibility to remove the current autonomic service is considered, otherwise if it is higher than the maximum load threshold then the autonomic service tries to add new autonomic services. The Autonomic Manager contains the list of the underlying cloud provider(s) and the logic that is needed to choose among them. However the selection of the destination provider for autonomic services replica is based on a random policy not considering any placement heuristics.

2.3 Large-scale deployment

Simulation is nowadays most-widely used, since it leads to reproducible results. However, the significance of the results is limited, because simulators rely on a simplified model of reality. More complex validation approaches based on emulation and execution on "real" large-scale testbeds (e.g., grids) do not have this drawback. However, they leave the deployment at the user's charge, which is a major obstacle in practice.

Considering the large scale, involving thousands of autonomic services, in order to evaluate DEPAS, the authors used a system developed on top of the Protopeer toolkit [38], as well as done in Mycoload. However, the results are limited to simulations and are not compared with large scale experiments on real networks. Their experience is limited with 200 instances deployed on the Amazon cloud, basically because the complexity of deploy an high number of machines.

Validating P2P systems at a large scale is currently a major challenge, since it is extremely difficult. We can classify three different ways to test a P2P system:

- *Simulation:* Are often executed on a single sequential machine and allow one to define a model for a P2P system, and then study its behavior through tests with different parameters. The main advantage of simulation is the reproducibility of the results. However, the technical design of the simulated prototype may be influenced by the functionality provided by the simulator to be used, which may result in deviations from reality. Further validation by alternative techniques such as emulation or experiments in real environments is still necessary.

- *Emulation:* Allows one to configure a distributed system in order to reproduce the behavior of another distributed system. Tools like dummynet [22], NIST Net [6], ModelNet [36] or Emulab [39] allow to configure various characteristics of a network, such as the latency, the loss rate, the number of hops between physical nodes, and sometimes the number of physical nodes. This way, networks with various sizes and topologies can be emulated. However, the heterogeneity of a real environment (in terms of physical architecture, but also of software resources) can not be faithfully reproduced. More importantly, deployment of P2P prototypes is essentially left to the user: it is often overlooked, but it actually remains a major limiting factor.

- *Experiment:* Is supposed to run on real networks, on real machines distributed across the world, without any supporting infrastructure layer. It can be seen as last deployment step, which stage the real value of the systems. Even though experiments are not reproducible in general on such platforms, this is a mandatory step in order to validate a prototype. This is even more difficult than in the case of emulation, because of the much larger physical scale.

To sum up, actually deploying and controlling a P2P system over large-scale platforms arises as a central challenge in conducting realistic P2P experiments. Considering for instance also popular P2P software, like Gnutella or KaZaA: workloads of these systems are not fully analyzed and modeled because the

behavior of such systems cannot be precisely reproduced and tested [21]. Recently, different P2P systems (like CFS [12], PAST [4], Ivy [3] and OceanStore [17]) based on smarter localization and routing schemes have been developed. However, most of the experiments published for these systems exhibit results obtained either by simulation, or by actual deployment on small testbeds, typically consisting of less than a few tens of physical nodes [33]. Anyway every approach is linked to the system/prototype, thus there is a crucial need for a tool providing the ability to deploy P2P systems at a large scale, which is independent from the framework.

2.4 P2P Simulators

As introduced, one of our goals is to deploy the application in real networks, in order to achieve valuable results. To do this we need to run a live experiment with the concept of time. So that we had the necessity to change the framework previously used in Mycoload. This framework implemented on top of the Java-based PeerSim platform [19] can be used according to two simulation models: cycle-based, in which peers get the control after a fixed constant time step in a sequential fashion, and event-based in which the cycle-based assumption is removed and simulator components can be active at any time, exploiting full concurrency. The authors in Mycoload decided to adopt the cycle-based model in order to ensure the experiments could scale easily to large number of peers, even if at the cost of loss of precision.

Through an analysis of the various possibilities, we found two frameworks time-based that could fit our requirements for building and evaluating P2P systems in simulation and distributed deployment on real network.

One is Kompics [7] whose model is shown in figure 2.7. It is a neat system, but would have imposed too much friction to try to work with it, since it is poorly documented. Furthermore is heavily over-engineered, and it requires a lot of overhead. We went into the selection process thinking that Kompics is the one that we would end up using, but it just was not a good fit.

So we looked at Protopeer [38], that is a simpler and efficient framework which allows to run simulations of P2P networks over a time domain, using an event-based paradigm ,thanks to the possibility to schedule events. It further-

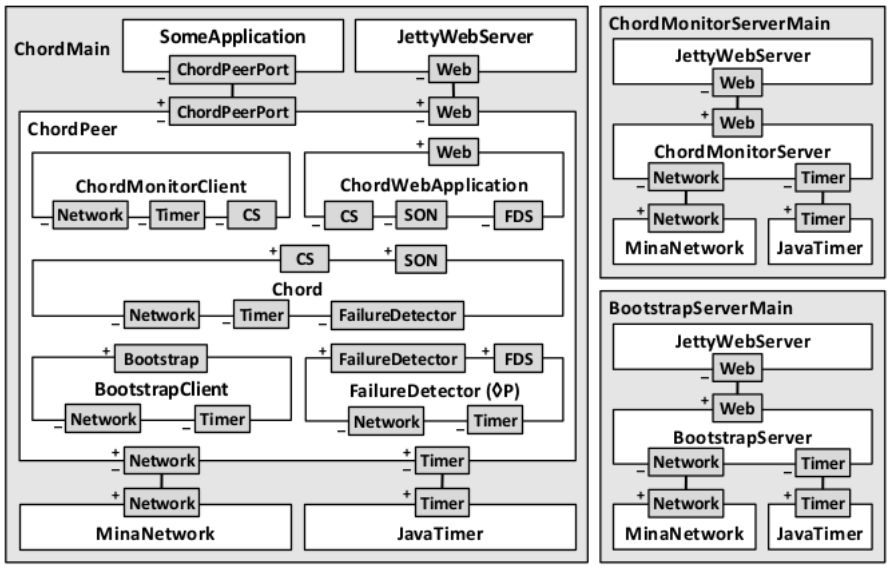


Figura 2.7: *The Kompics architecture. The left figure shows the architecture of a Chord process. The Chord protocol is implemented by the Chord component using Network, Timer, and FailureDetector abstractions. The Network and Timer abstractions are provided by the MinaNetwork [1] (which handles connection management and message serialization) and JavaTimer components. The ChordMonitorClient periodically inspects the Chord status (CS port) and sends it through the network to the ChordMonitorServer (top right). The ChordWebApplication renders this status on a web page upon request from a web server (which provides web browser access). On the right we have the component architectures of the monitoring and bootstrap server.*

more allows to run the system without changing the code in a real computer network where each peer is instantiated in a different machine.

The system shown in Fiure 2.8 is composed of a set of peers that communicate with one another by passing messages. Each application defines its set of messages and message handlers. Messages can be sent either over UDP or TCP, and during the live run messages are serialized using a custom optimized protocol that is considerably less verbose than the standard Java serialization. An application typically also defines a set of timers and handlers for the timer expiration events, and the message passing logic and state of each of the protocols executed after received a message is encapsulated in components called *peerlets*. Peers are constructed by putting several peerlets together. The peerlets, which can be removed or added at run-time, just as the applets or servlets, have the familiar init-start-stop life-cycle. The peer provides the execution context for all of the peerlet instances it contains and the peerlets

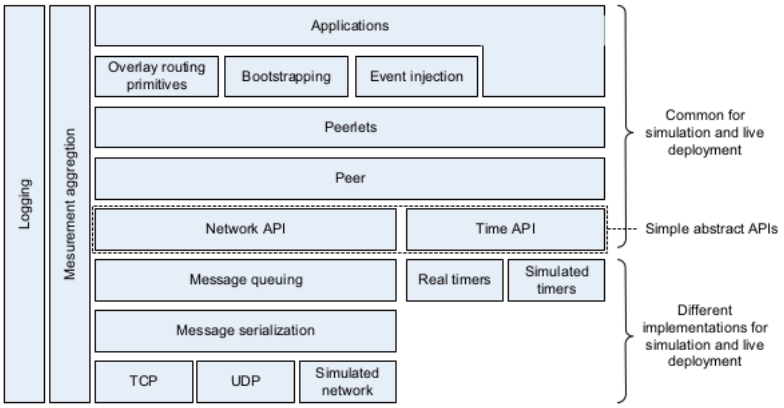


Figura 2.8: *The ProtoPeer architecture. The simplified time and networking APIs form the waist of the ProtoPeer's architecture. The application complexity grows from these APIs up, while the simulation and live networking complexity grows from the APIs down.*

can discover one another within that context and use one functionality of the other.

Finally ProtoPeer provides a good measurement infrastructure helps to collect all the metrics providing basic statistics on-the-fly: average, sum, variance etc. Instrumentation is done by doing calls to the measurement API in the appropriate places in the application code.

For all these considerations we found ProtoPeer extremely effective and versatile respect to our needs.

2.5 Considerations

All the works exploited, which tries to improve the QoS of such complex systems have pros and cons. The approach with a DHT based self-organizing routing structure [14], deals just with a fixed initial assignment of requests to each peer and no network dynamism concern is taken into account. In [31], where is achieved a load-balancing behavior by modifying the degree of each node proportionally to the available computational resources, they are considering a structured peer-to-peer network model, and do not address the possibility of change the topology. The same limitation is showed in [11], where the authors do not considered churn scalability issues and furthermore they do not consider the case of nodes with different processing capabilities and the effects of churn. DEPAS [10] instead, has not the same limitations of

others, but in the auto-scaling algorithm is used a random policy for choose the destination provider for autonomic services replica not considering any placement heuristics, which is instead the aim of this work.

Capitolo 3

MycoCloud Approach

3.1 Overview

This Chapter presents MycoCloud, a decentralized self-adaptive network that aims at improving the QoS in Cloud infrastructures implementing different algorithm layers. In MycoCloud we implemented an augmented version of Myconet with rules to support the clustering of same-type nodes in order to make suitable to serve as the overlay management system for our adaptive algorithm. Furthermore on top of that efficient self-organized topology we used a decentralized load-balancing algorithm used in previous MycoLoad and DEPAS approaches.

In MycoLoad the authors have proposed a self-organized load-balancing algorithm and a topology reconfiguration mechanism for decentralized service networks but without providing any scaling mechanism. Furthermore in their work has not considered a real scenario in real networks (e.g., networks of Clouds). Part of these limitations have been handled in DEPAS, in which through a decentralized probabilistic auto-scaling algorithm integrated into a P2P architecture allowed auto-scaling of services over multiple cloud infrastructures. The problem of this solution in our advice is the selection of the resource for the scale-up action, since it uses a random placement heuristic.

In MycoCloud we handled these limitations trying to reach a complete decentralized adaptation giving to the system the self-* properties needed. In particular we focused on the services adaptation enabling a self-scaling property in the system, thanks to which the services grouped in clusters will be

able to require/release resources (nodes) from/to other clusters. The algorithm will use a *probabilistic placement heuristic* thanks to which the selection will take into account at the cost of selecting one node. This cost, which complete explanation can be found in the subsection 3.2, refers to the load of over the last time-frames. If this load is high the cost of stealing resources would be higher compared one with lower load. This because the load over the history represents an information saying that the load in the next time-frames will be "probably" closer to the load over the last times-frames. In this way by selecting a cluster with a low load we avoid the side effect of stealing resources to the cluster that would be loaded in the next instants.

3.2 Model

Each node in the system will run an algorithm for deciding to change its service type according to the requests that are being received. In particular the algorithm will trigger a ***service change request*** if the local load goes over a certain threshold. We exploited two threshold types. The first one gives to the system a *proactive* behavior, since it tries to maintain the load balanced not only among nodes of same service type, but also between different services. The threshold is not constant but, given two nodes N_1 and N_2 and their respectively load L_1 and L_2, it is calculated as:

$$L_t = \frac{|L_2 - L_1|}{2}$$

The second *threshold* is a fixed value L_t between 0% and 100% making the system *reactive*. Each Node responds (reacts) to external events, that is, for every request received compares the threshold with its load, and takes an action (asking for service change) if the latter is over the threshold.

It is important to underline that the local load of the Nodes is maintained balanced with the load of other Nodes of the same service type by the load-balancer. Using the proactive version the Nodes need to exchange several "balancing" messages to calculate L_t and then, to not exceed it there would be several service changes increasing network traffic. In fact, in addition to the load-balancer algorithm, which maintains balanced the load of same type Nodes, with such proactive behavior the system will try to balance also Nodes of different services.

Given these reasons we decided to use the reactive version. So when the load L_a is over L_t the algorithm running in the Node (also called Asking Node N_a) sends a message directed to a neighbor Node (a node with which it has a direct connection to) of different service N_b asking for *capacity* units:

$$\breve{C}_a = (L_a - L_t) * C_a$$

A Node's capacity (i.e., the number of service requests it can process within a single time unit) is set according to a power law distribution of a given n such that the probability to have a capacity C_n is:

$$P\left[C_n = x\right] = x^{-\alpha}$$

where $1 \leq x \leq C_{max}$. N_b will have C_b capacity units and if its remaining capacity, calculated as:

$$\hat{C}_b = L_t - L_b$$

is $\hat{C}_b \geq 0$ (so it is not overloaded) then:

1. If $C_b > \breve{C}_a$ then N_b gives C_b units to N_a. If $\hat{C}_b \geq \breve{C}_a$ then with probability $P = \frac{C_b}{\breve{C}_a}$ N_b gives C_b capacity units.
2. If $C_b < \breve{C}_a$ then N_b gives C_b capacity units and generate a new service change request asking for the residue capacity $\breve{C}_{new} = \breve{C}_a - C_b$.

Otherwise N_b forwards the request to another Node. In this way the service change message continues to run until the request of capacity is satisfied, with the possibility to be fragmented in more requests. To avoid too much message traffic and that the request continues to be sent without being satisfied, the request message has a *Time To Live* (TTL). A time to live is a value set when the request starts and decremented at each hop in the network, when is 0, the message is discarded. We considered also, the possibility to fragment the service change requests from the start. In this way the N_a generates many messages, each one requesting $\breve{C}_i$, such that $\breve{C} = \sum_i \breve{C}_i$. However it had two limitations: the first one due to the fact that an excessive fragmentation may lead to a significant increase of the network traffic. The second because the algorithm may result slower, since the requests may be satisfied by a single Node at the first message and the fragmentation may be unnecessary.

Using the service elasticity, however, has a cost. We defined this cost as a time for service change:

$$T_{sc}=T_o+T_u$$

where T_o is a time cost paid to reorganize the overlay network and T_u is a time cost of unavailability of the processing capabilities of the Node for a certain amount of time. This simulates the possibility to shutdown the old service (e.g., the virtual machine) and start the new one.

Since the solution aims to improve the QoS of the system, the algorithm needs to react quickly adapting the services capacities according at the increase of the load. Chapter 5 will show how we reach this goal with significant improvements considering different metrics. The principal ones are:

(i) The *System Load*, defined as average load among the different services. It will show how the system is adapting itself reacting on increasing load among different services.

(ii) The *Response Time Optimality* (RTO), defined as

$$RTO = \frac{OptimalResponseTime}{MeasuredResponseTime}$$

where *ExperimentalResponseTime* is the measured response time and *OptimalResponseTime* is the response time of an ideal system composed of a single C capacity Node. RTO gives an idea of how the measured response time is far from being equal to the optimal one of the ideal system.

(iii) The number of *Exchanged Messages*, which is important to understand the network cost T_o.

3.3 Design

MycoCloud implements a three layers architecture:

1. At lowest layer there is superpeer-based overlay substrate that automatically build and re-configure the topology. In subsection 3.3.1 is explained how we exploit this functionality to build a different cluster for each service type as well as the derived benefit for the adaptive algorithm.

2. The middle layer, explained in subsection 3.3.2, implements a load-balancing algorithm for heterogeneous and dynamic networks [27] used in both in MycoLoad [37] and DEPAS [10] approaches.

3. The last layer consists in an adaptive algorithm, whose design is illustrated in subsection 3.3.3.

In MycoCloud is implemented on top of Protopeer framework [38] since it allows to build and simulate such large-scale P2P network. ProtoPeer provides different features useful to build distributed networks, in which every resource is a *Peer*, and every Peer provides the execution context for different components it contains, called *Peerlets*. Peerlets can discover one another within that context and use one functionality of the other. Every message passing logic and state of each of the protocols executed after receiving a message is encapsulated in a *peerlet*. The Peerlets, which can also be removed or added at run-time, just as the applets or servlets, have the familiar init-start-stop life-cycle.

3.3.1 Cluster Construction Rules

We exploit the Myconet's overlay management rules that regulate the interactions of same-service type vs. different-service type peers. These in brief, are a set of promotion/demotion rules, which self-regulate the superpeer population in comparison with the amount and capacity of biomass peers in the network. Promotion and demotion are predicated on a few parameters: a target number of connections each hyphal peer should have to biomass peers (B_S), which is proportional to the capacity of that hyphal peer; and a target number of hyphal links it should "grow" towards other super-peers (C_S), to ensure the robustness of the overlay. C_S is a customizable parameter of the protocol itself. 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 hypal peers that can best service them, while, on the other hand, ensure that hyphal peers that cannot efficiently reach their B_S or C_S targets will become candidate for demotion.

- *Biomass Peers*: All peers begin protocol execution as biomass. A disconnected biomass peer b will attempt to connect to an extending superpeer

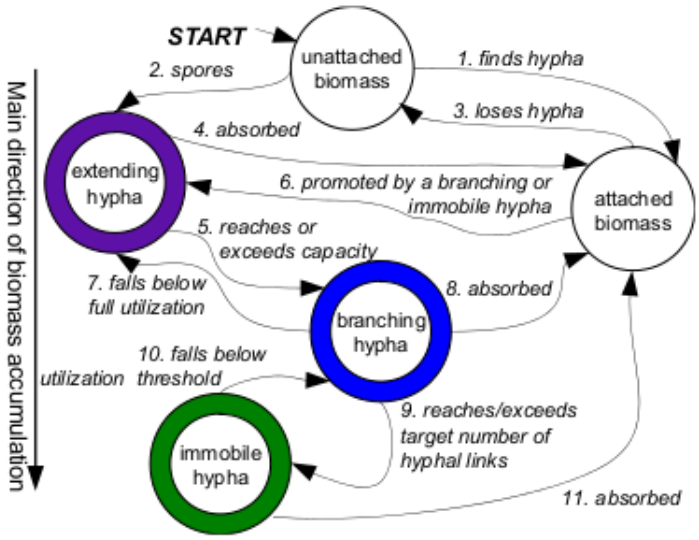


Figura 3.1: Myconet protocol state transitions. Peers adaptively transition between regular peers (biomass) and three different varieties of hyphal peers depending on changing conditions.

it can find through the lower-level gossip protocol. However, it will only select one of its same service type. If no suitable extending superpeer can be located, b will promote itself to extending status. This rule ensures that there are always extending hyphae of each type in the network, since these self-selected extending peers must act as aggregation points for isolated peers of their service type. If a biomass peer ever becomes disconnected, it searches for a new extending hypha to connect to, as during bootstrapping.

- *Hyphal Peers - all states*: Hyphal peers aggregate biomass peers of their same service type up to a target level B_S (proportional to their capacity); to this purpose, hyphal peers in a higher protocol state will always try to "pull" biomass from other neighboring hyphal peers of their type in lower states, or in the same state but of lower capacity, through the execution of absorption rules. Hyphal peers attempt to create and maintain a target number of links C_S to other hyphae of their service type, which is a parameter of the protocol. Another parameter is C_O, that is the number of links hyphal peers try to maintain to hyphae of different service types. C_O ensures that clusters do not become disconnected from one another. Increasing the C_S or C_O parameters increases the number of

cross-connections (and the size of each peer's neighborhood views), while adding overhead to the protocol. This will be discussed in Chapter 5.

- *Extending Peers*: Super-peers remain in the extending state until they have reached their target number of biomass neighbors of the same type B_S , at which point they promote to branching status. Extending peers also anchor themselves to the overlay with a single link to either a branching or an immobile peer of any type. In the event two extending peers of the same type become neighbors, the lower-capacity peer will transfer all of its biomass peers to the higher-capacity and demote itself to biomass status. This is an absorption rule.
- *Branching Peers*: One primary function of these super-peers is to grow new links to other super-peers, thus building robust cross-connections for the overlay. Branching peers are those that have aggregated their target number of same-type biomass peer (B_S), but have not yet reached their targets of C_S and C_O hyphal links to other super-peers of the same type and different types, respectively. With respect to same-type hyphal links, if a branching peer *hb* is under the target number C_S , it will search its neighborhood to determine if any of its neighbors are attached to a suitable same-type hyphal peer that is not already its own neighbor. With respect to different-type hyphal links, if *hb* is under its target C_O , it will randomly select a peer from the gossip cache and grow a neighbor link to it. The resulting peer might be of any type (including the same). A branching peer also attempts to always have one extending peer among its neighbors; if no such peer exists, it will pick its largest capacity biomass child and promote it, obtaining an extending peer of its same type. If a branching peer ever gets over its biomass capacity, it will push excess biomass children to an attached same-type superpeer. Once it has grown C_S and C_O hyphal links, a branching peer will promote to immobile state.
- *Immobile Peers*: Once a peer has achieved immobile status, it will attempt to maintain it through absorption, that is, by pulling biomass from lower-state hyphae of its same type (possibly resulting

in their demotion). Similarly, it will try to grow new hyphal links if some are lost, in order to maintain the target C_S and C_O . If the immobile peer, instead, happens to have more than C_S hyphal links to same-type super-peers, or more than C_O hyphal links to different-type super-peers, it will randomly drop the excess links. An immobile peer should be connected to either zero or one same-type extending peers. In case it has multiple extending neighbors hyphae, it will connect them together, thus triggering the absorption rule. If it has both branching and extending neighbors, it will transfer the extending peers to become children of the branching peers.

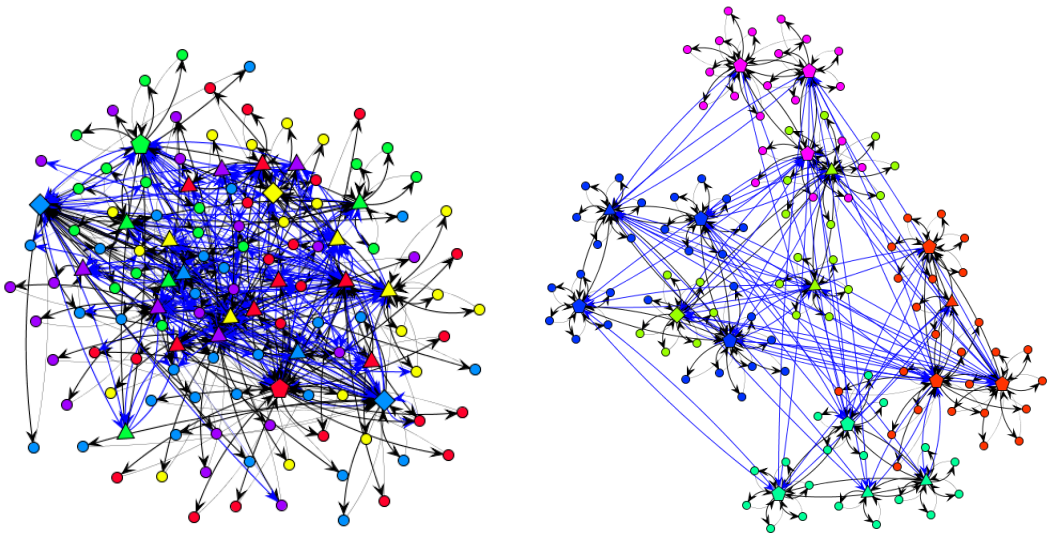


Figura 3.2: Example of Clusterization. (a) Left figure shows network during bootstrapping, after 5 seconds. Clusters are beginning to form. (b) Right figure shows overlay stabilized.

3.3.2 Load-balacing

Load-balancing is - as in MycoCloud and DEPAS - performed between neighbors in a decentralized service network, an example is shown in Figure 3.3. The goal of load-balancing is to make queue lengths proportional to the capacity of each peer. Based on this principle, when a load-balancing operation is performed, a peer p_a selects a random same-type hyphal neighbor p_b. If p_a is a biomass peer, it will have only a single neighbor, a same-type

hypha. If p_a is a hyphal peer, it will only try to balance its queue with other same-type hyphal peers; as biomass peers have only a single neighbor, they will be the ones to initiate the load-balancing operations with their parent. This way, load-balancing operations tend to occur preferentially between the more capable super-peers, which help ensuring that jobs will be balanced throughout the cluster. p_a and p_b compare their queue lengths; "Ideal" queue lengths are calculated by determining the total number of jobs in both queues and dividing the jobs proportionally based on p_a and p_b's capacities. If the queues are unbalanced, jobs are transferred from the queue that is over its ideal length to the queue that is under that length. Therefore the number of requests moved from A to B (B to A) is calculated by T (A, B) (T (B, A)) using the following formula:

$$T(A,B) = -T(B,A) = \frac{|queue_A| * C_B - |queue_B| * C_A}{C_A + C_B}$$

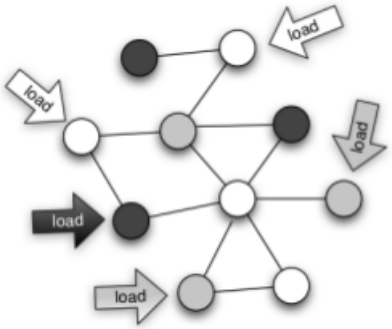


Figura 3.3: Decentralized service network. Nodes, shown in the figure as circles, offer different services (characterized in the figure by different colors), and are organized in an overlay network highlighted in the figure by the arcs connecting the nodes. For an efficient execution of service requests, nodes must redirect the requests exceeding their processing capabilities to other nodes able to handle them.

In this way, each cluster keeps the load among its nodes balanced. This means that a service change request will be issued when the service is really overloaded, not just because a single node. Another benefit exploited by the elasticity algorithm is that when a service change request is issued in the optimal case, that is a perfect balanced cluster, it will explore with one message the load of all the nodes of one cluster. Ideally explore the whole network with n messages where n is the number of clusters (services). In this way the choice for the less loaded service in the network is fast and effective.

3.3.3 Service Elasticity

Since the Myconet rules, described in subsection 3.3.1, one of the first concerning in the design of the service elasticity algorithm was whether enable or not the service change even for super-nodes. This consideration was due to the fact that super-peers are important to maintain the overlay and thus the robustness of the network.

This process led us to consider to makes "changeable" just the *Biomass* Nodes, since they act a marginal role in the overlay construction. Thus nodes in state *Immobile, Branching* or *Extending* would be able to change service only after the demotion rules bringing them to a Biomass state. If this way however, since *Biomass* nodes are that ones with lower capacity, a cluster may became deprived of a lot of *Biomass* keeping just nodes with high capacity. While a cluster that is receiving just *Biomass* will have few "big" nodes and a lot of nodes with low capacity. This situation bring to have two main problems:

1. The overlay lose in convergence time and effective in both the clusters since each cluster needs at the same time: nodes with high capacity promoting to super-nodes states and low capacity nodes becoming Biomass.

2. Realising always nodes with less capacity would means to give more nodes to satisfy a service change request. While giving nodes with higher capacity reduces the number of changes as well as the convergence time.

These considerations led us to exclude just nodes in Immobile state from the service changing, since they are the most stable. This is shown as the most efficient solution, since at the same time, keeps a strong overlay and improve the algorithm effectiveness since nodes with different capacity are able to change service.

Another important consideration is the introduction of a constraint on the resources that a cluster can releases. This constraint consists into not depriving a cluster of all its nodes, that is holding at least one. This is needed because otherwise the service would be eliminated from the network. An example of the constraint application, code-named *cluster safe condition*, is shown in Fig. 3.4. The check is made on the number of neighbors, that is: if a node has no neighbors of the same type it cannot change, since this means that it is the last node of the cluster.

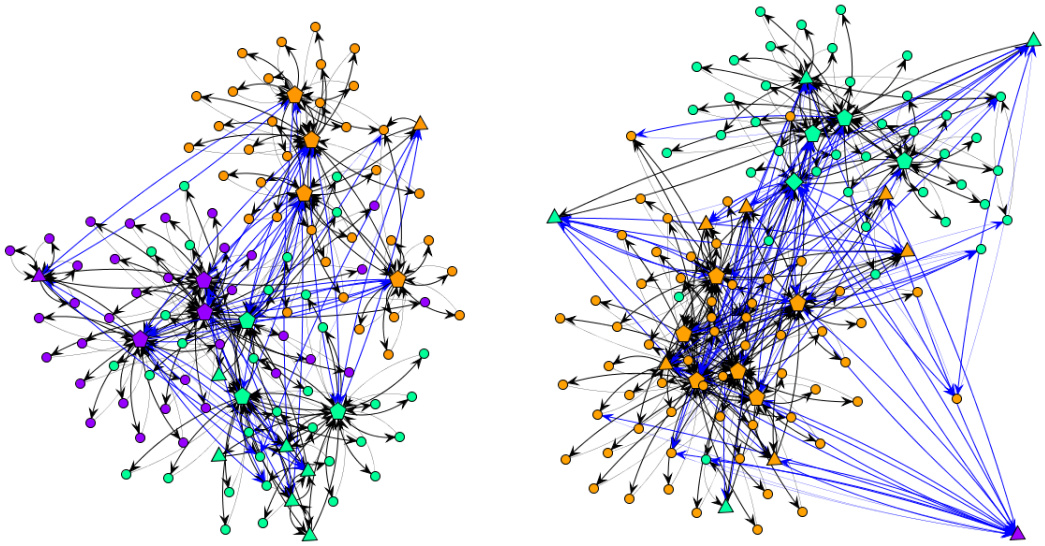


Figura 3.4: Example of Cluster Safe Condition. (a) Left figure shows Clusters before service changes. (b) Right figure shows how two services take all resources of a third service, which maintains the last node.

After these analysis we defined our algorithm as composed of different Components implementing different behaviors. Every Component is a *Peerlet* and can act as *Generator*, *Forwarder* or *Executor*. (Note: SCR is the acronym of Service Change Request)

SCR Generator Running in every node (see algorithm 1). This component checks the local load for every incoming request and decides whether issue a *service change request*.

SCR Dispatcher Running in every node (see algorithm 2). It is in charge to receive the service change requests taking one of the following actions:

(i) To forward the request to another node;

(ii) To handle the request locally.

SCR Immobile handler Running in nodes in *Immobile* state (see algorithm 4). Act as a smart Forwarder, in fact a node in this state cannot change its service.

SCR Basic handler Running in nodes in the other states, that is *Branching*, *Extending* or *Biomass* (see algorithm 5). Is the component that decides if execute the change service or not.

An important assumption is that nodes of a given type are aware of the remaining capacity $\hat{C}$ of neighbor nodes of different types (this information is regularly diffused in the node's neighborhood). We will say that one Node is less loaded with respect to others when its remaining capacity $\hat{C} \geq \hat{C}_i$.

The figure. 3.5 at page 39 shows the class diagram model, in which there is:

The *SCRGenerator* that checks the local load for every incoming requests. If the load is over the threshold L_t it asks to the *NeighborServiceManager* for a lower loaded neighbor with different service. The *NeighborServiceManager* is the peerlet in charge to keep the information about all the neighbors. The algorithm 1 shows the possible scenario in this situation:

- If the node has in its neighborhood one or more nodes with different service having $\hat{C} > 0$ (not overloaded), the *SCR* is sent to a random one of them.

- In case the node has no neighbor of different service or every of them is overloaded ($\hat{C} \leq 0$), the *SCR* is sent to a random super-node. This situation is typical for Biomass nodes since they have just one link with a super-node of the same service. In this way thus, the *SCR* can reach other clusters. In fact, as explained in the previous section 3.3.1 super-nodes are the nodes in charge to maintain overlay links connecting all the clusters.

Algorithm 1 Service change request: Generation

```
1: if L>Lt then
2:     Cr ← (L − Lt) ∗ C
3:     Nl ← getLessLoadedNeighborDifferentServiceType()
4:     if Nl == null then
5:         Nl ← getRandomSuperNode()
6:     end if
7:     sendServiceChangeMessage(Nl, Cr, serviceType)
8: end if
```

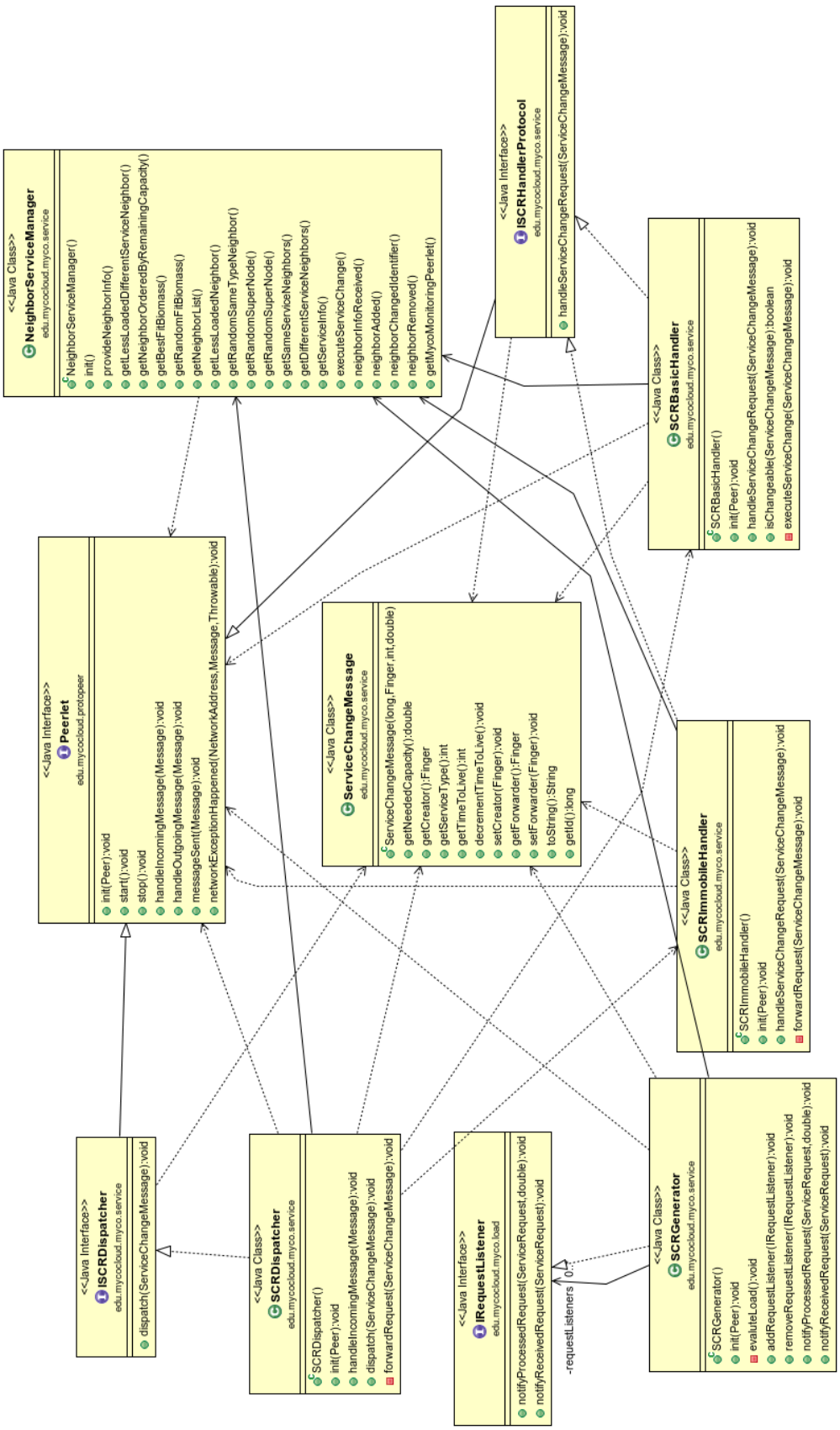


Figura 3.5: Service change design: Class Diagram

The *SCRDispatcher* component serves like a proxy (see Algorithm 2): for every incoming *SCR* it checks if the node's service is different with respect to that one to change. This check is needed because can occur that a *SCR* has not been yet able to go outside its own cluster.

- If the *SCR* is in a different cluster, *SCRDispatcher* delegates the handling of the request to either the *SCRBasicHandler* or the *SCRImmobileHandler* according to the actual state of the node.
- If the *SCR* is still in its own cluster, the *SCRDispatcher* forwards the *SCR* (See algorithm 3).

Algorithm 2 Service change request: Dispatcher

```
1: message is the Service Change Request
2: if message.serviceTypeToChange! = this.serviceType then
3:     if this.state == Immobile then
4:         //See SCRImmobile Algorithm 4
5:         immobileProtocolPeerlet.handleServiceChangeRequest(message)
6:     else
7:         //See SCRBasic Algorithm 5
8:         basicProtocolPeerlet.handleServiceChangeRequest(message)
9:     end if
10: else
11:     //See Forward protocol Algorithm 3
12:     forwardRequest(message)
13: end if
```

The forwarding function (algorithm 3) checks first for a less loaded neighbor with different service with respect to the *SCR*. If the node has no neighbors with different service, the *SCR* is sent to a random super-node.

Algorithm 3 Service change request: Forward protocol

```
1: message is the Service Change Request passed as argument
2: if message.isNotExpiredTTL() then
3:     N_l ← getLessLoadedNeighborOfDifferentType(message.serviceType)
4:     if N_l == null then
5:         N_l ← getRandomSuperNode()
6:     end if
7:     sendMessage(N_l, message)
8: end if
```

The *SCRImmobileHandler*, whose algorithm is shown in 4, implements the most important role for the *placement heuristic* since it has to decide whether to solve the *SCR* in the actual cluster or to forward it to another one.

- The Node, or better the super-node, first explores its overlay links looking for a less loaded node with different service with respect to the *SCR*. In this case forward the *SCR*.

- If the condition is unsatisfied, it sends the *SCR* to one of its Biomass. The selected will be a random *Well-Fitting Biomass*. A Well-fitting means picking up a Node in a set of Biomass not overloaded (with remaining capacity $\hat{C} > 0$) having capacity units C enough to satisfy the request ($C \geq \hat{C}$). If this set is empty, a random one not overloaded ($\hat{C} > 0$) will be selected.

We exploited also a *Best-Fitting Biomass* selection. This would be a Biomass (not overloaded) having capacity C as close as possible to the requested capacity $\breve{C}$. The thought led us to consider this selection was to try to reach a further optimization on the placement heuristic, that is deprive the service only for the requested capacity.

We decided to not adopt this version we noticed that especially in smaller network the selection in most cases falls often in the same Biomass having lower capacity. This causes to have *SCR* sent to Biomass having recently changed their service, not giving them enough time to settle in the new cluster. Furthermore when a service asks for capacity very likely it needs a little more capacity then the one it is requesting since the load would be the same in the entire cluster because the load-balancer. This would lead to have more service change requests and delay in the adaptation.

Algorithm 4 Service change request: Immobile handler

```
1: message is the Service Change Request message
2: N_l ← getLessLoadedNeighbor(message.service)
3: if N_l.load<this.load then
4:     sendMessage(N_l, message)
5: else
6:     N_fit ← getRandomWellFittingBiomass()
7:     if N_fit == null then
8:         N_wfit ← getRandomBiomass()
9:     end if
10:    sendMessage(N_fit, message)
11: end if
```

Finally the *SCRBasicHandler*, whose logic is shown in Algorithm 5, is executed when a node is in state *Branching*, *Extending* or *Biomass*. It is in

charge to perform the service change, if the cluster safe condition is satisfied. Furthermore it checks if its capacity is enough to satisfy the *SCR*. If this is not the case it sends a new *SCR* to a random neighbor having its same service type, asking for $\breve{C}_{new}$ capacity units.

Algorithm 5 Service change request: Basic handler

```
1: C_r ← (L − L_t) * C
2: if clusterSafeCondition then
3:     changeService(newService)
4:     if C_r<C_a then
5:         // Send residue capacity request to a neighbor of same service type
6:         C_anew ← C_a − C
7:         N_r = getRandomSameTypeNeighbor()
8:         sendServiceChangeRequest(N_r, serviceType, C_anew)
9:     end if
10: else
11:     // forward message to a random neighbor
12:     N_r = getRandomSameTypeNeighbor()
13:     sendMessage(N_r, message)
14: end if
```

3.4 Adaptation examples

In this section are shown few simple examples of possible scenarios of the adaptive protocol. They are considered indicative to give a better understand of the algorithm.

Figure 3.6 shows a scenario in which a sequence of actions conclude in a service change. The actions are in order:

1: Biomass node (1) that is over-threshold ($L_1 > L_t$) and its *SCRGenerator* peerlet issue a *SCR* asking for capacity $\breve{C}_1 = (L_1 - L_t) * C_1$. Node 1 has no neighbor of different type, so the *SCR* is sent to its super-node (2).

2: The *SCRDispatcher* in Node 2 realizes that the *SCR* comes from its cluster and looks for the less loaded neighbor having different services with respect to *SCR*. This node is 3, which is in this simple case the only one neighbor with different service. Node 3 has $\hat{C} > 0$ so it is not overloaded and receive the *SCR*.

3: The *SCRDispatcher* of the node 3 realizes that the *SCR* comes from a different service so delegate the handle of it to the *SCRImmobileHandler*.

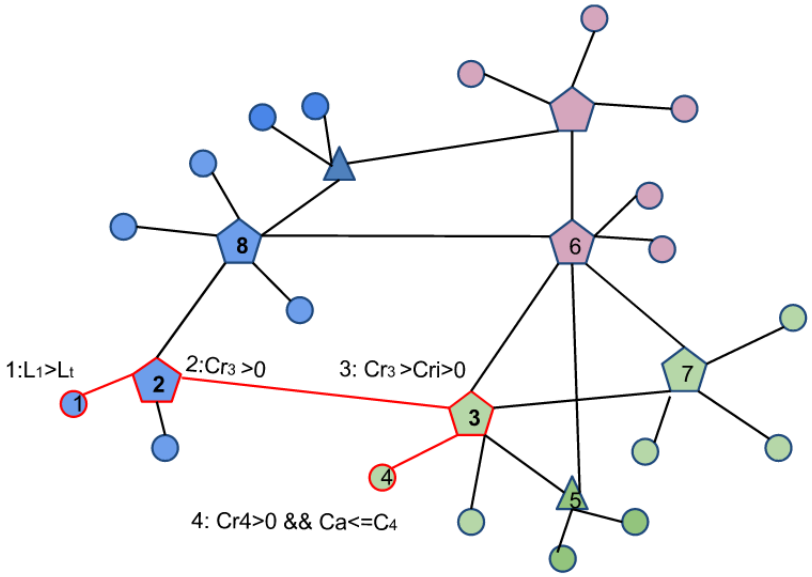


Figura 3.6: Service change example: Basic scenario

The *SCRImmobileHandler* looks for less loaded neighbors, but in this case it is the less loaded with respect to its neighbors 5,6 and 7 ($\hat{C}_3 > \hat{C}_i$). Thus *SCRImmobileHandler* in the node 3 looks for a random well-fitting Biomass, which in this case is the node 4, and sends the *SCR* to it.

4: The *SCRDispatcher* of the node 4 check the service type and realizes that the *SCR* comes from a different service so delegate the handle of it to the *SCRBasicHandler* Node 4 execute the service change and moves from the green cluster to the blue one. Before this, it checks if the capacity request is satisfied. In this case it is $C \geq \breve{C}_1$, no further action are performed.

Figure 3.7 shows a second scenario in which more actions are needed to conclude in a service change. The actions are in order:

1: Biomass node (1) that is over-threshold ($L_1 > L_t$) and its *SCRGenerator* peerlet issue a *SCR* asking for capacity $\breve{C}_1 = (L_1 - L_t) * C_1$. Node 1 has no neighbor of different type, so the *SCR* is sent to its super-node (2).

2: The *SCRDispatcher* in Node 2 realizes that the *SCR* comes from its cluster and looks for the less loaded neighbor having different services with respect to *SCR*. The node 3, however has $\hat{C} \leq 0$ so it is overloaded too. The *SCR* is sent to a random super-node, in this case 8.

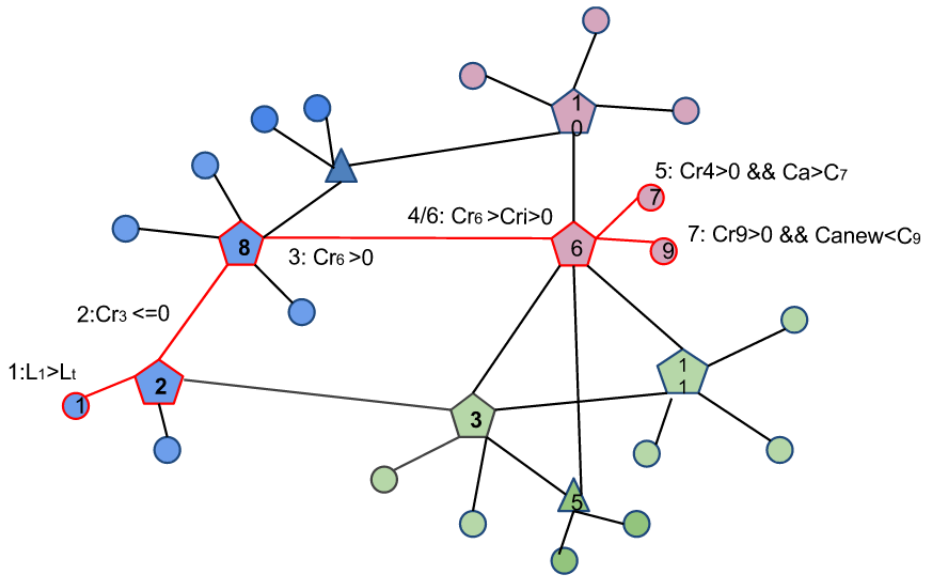


Figura 3.7: Service change example: Overlay scenario

3: The *SCRDispatcher* in the node 8 realizes that the *SCR* comes from the same service and looks for its less loaded neighbor with different service. In this case is the node 6, which receives the *SCR*.

4: The *SCRDispatcher* of the node 6 check the service type and realizes that the *SCR* comes from a different service so delegate the handle of it to the *SCRImmobileHandler*. The *SCRImmobileHandler* looks for less loaded neighbors, but in this case it is the less loaded with respect to its neighbors 10,11,5 and 3 ($\hat{C}_6 > \hat{C}_i$). Thus *SCRImmobileHandler* in the node 6 looks for a random well-fitting Biomass, which in this case is the node 7, and send the *SCR* to it.

5: The *SCRDispatcher* of the node 7 check the service type and realizes that the *SCR* comes from a different service so delegate the handle of it to the *SCRBasicHandler*. Node 7 execute the service change and moves from the violet cluster to the blue one. Before this, it checks if the capacity request is satisfied. In this case it is not ($C < \breve{C}_1$), so the *SCRBasicHandler* generate a SCR_{new} asking for the remaining capacity $\breve{C}_{new} = C_7 - \breve{C}_1$. This new request is sent to a random same service type neighbor, which is in this case node 6.

6: Node 6 in the meanwhile has lost a Biomass (7) and it has gone through

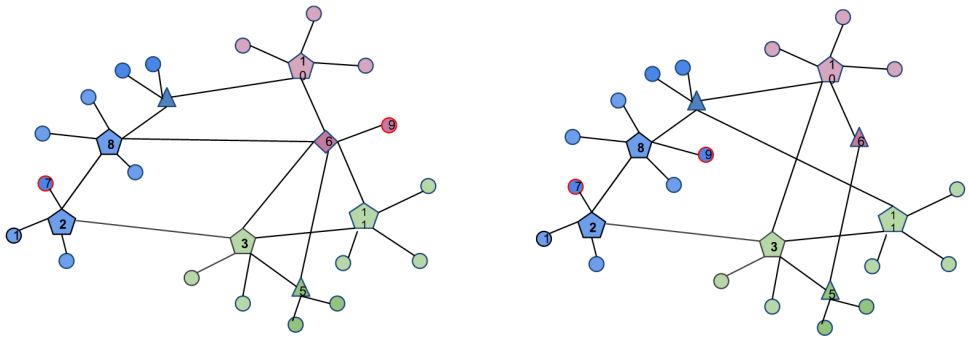


Figura 3.8: Service change example: (i) Left figure shows the changing after one service change. The node 7 goes to the blue cluster and the Immobile node 6 downgrade to the Branching state. (ii) Left figure shows the changing after one service change. The node 9 goes to the blue cluster and the Branching node 6 downgrade to Extending state.

the Myconet demotion rules, leading it to the lower state of *Branching*. Figure 3.8 shows the topology changing. Node 6 thus, will be able to change now. The *SCRDispatcher* of the node 6 check the service type and realizes that the *SCR* comes from a different service so delegate the handle of it to the *SCRBasicHandler*. The *SCRBasicHandler* looks for less loaded neighbors.That one, in this case is node 9. Thus *SCRBasicHandler* in the node 6 send the *SCR* to node 9.

7: The *SCRDispatcher* of the node 9 check the service type and realizes that the *SCR* comes from a different service so delegate the handle of it to the *SCRBasicHandler*. Node 7 execute the service change and moves from the violet cluster to the blue one. Before this, it checks if the capacity request is satisfied. In this case it is $C \geq \breve{C}_1$, no further action are performed. Note that after this change node 6 loses another Biomass, so it downgrade again to *Extending*. Figure 3.8 shows the topology changing.

Finally figure 3.9 shows a third scenario in which a *SCR* is not able to go outside the cluster and it is discarded because TTL. The actions are in order:

1: Biomass node (1) that is over-threshold ($L_1 > L_t$) and its *SCRGenerator* peerlet issue a *SCR*. Node 1 has no neighbor of different type, so the *SCR* is sent to its super-node (2).

2: The *SCRDispatcher* in Node 2 realizes that the *SCR* comes from its cluster and looks for the less loaded neighbor having different services with respect

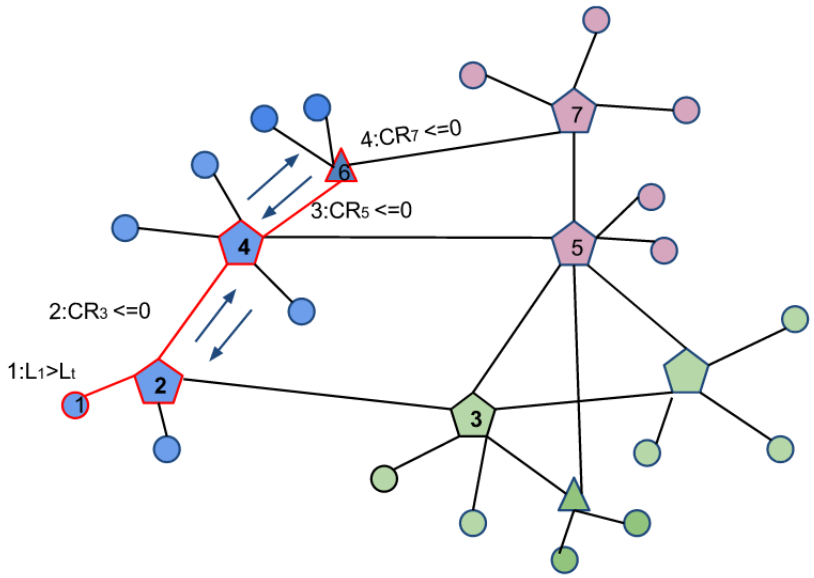


Figura 3.9: Service change example: Discarded request scenario

to *SCR*. The node 3, however has $\hat{C} \leq 0$ so it is overloaded and the *SCR* is sent to a random super-node (4).

3: The *SCRDispatcher* in Node 4 realizes that the *SCR* comes from its cluster and looks for the less loaded neighbor having different services with respect to *SCR*. The node 5, however has $\hat{C} \leq 0$ so it is overloaded and the *SCR* is sent to a random super-node that can be 2 or 6. Let's assume the chosen one is 6.

4: The *SCRDispatcher* in Node 6 realizes that the *SCR* comes from its cluster and looks for the less loaded neighbor having different services with respect to *SCR*. The node 7, however has $\hat{C} \leq 0$ so it is overloaded and the *SCR* is sent to a random super-node (4).

5: The *SCR* continues to be bounced between the super-nodes 2-4-6 and finally discarded after the *time to live* (TTL) is expired.

Capitolo 4

Live Distributed Experiments

Simulation is nowadays most-widely used, since it leads to reproducible results. However, the significance of these results is limited, because simulators rely on a simplified model of reality. More complex validation approaches based on emulation and execution on "real" large-scale testbeds (e.g., grids) do not have this drawback. However, they leave the deployment at the user's charge, which is a major obstacle in practice.

4.1 New Deployment Tool

We intend to develop this tool (named *LDE* from the acronym of Live Distributed Experiments), in order to use it for the validation of MycoCloud, but trying to make it as general as possible so that it can be easily utilized in other P2P environments.

We state here four macro requirements which should be observed by a deployment and control tool to successfully support general large-scale P2P experiments. The final goal of this tool will be to deploy, configure, control and analyze large-scale distributed experiments on a federation of clusters on different cloud providers from a single control point.

The tool shall:

- *Requirement 1*: Provide the ability to easily deploy on cloud machines as well as normal ones in clusters. Therefore, the tool should support various type of access (password, access key, etc..).

- *Requirement 2*: Allow designers of P2P prototypes to trace the behavior of their system. Therefore, the tool should allow to retrieve the outputs of each peer, such as log files as well as result files, for off-line analysis.
- *Requirement 3*: Provide the ability to efficiently deploy peers on a large number of physical nodes. For instance, a multi-deployment on single physical node strategy may be useful when testing with a huge number of nodes in clouds providers, since the have a utilization cost.
- *Requirement 4*: Provide the ability to synchronize peers for the run. Indeed, a peer should have the possibility to wait for other peers that maybe, are reaching a specific state before going through the next step of a test.

4.2 Tool Description

With the idea to develop a tool that gives us the possibility to deploy our prototype, we wanted to design something that could be usable also for other P2P experiments as well as ours. Thus, we wanted to make the tool as general as possible, and not dependent on specific prototype. In this way the tool is not aware of the framework, the simulator, and the protocols used. The purpose is to deploy and run every node in all the available machines, and to help the retrieval of possible outputs.

Having this in mind we designed a multi-thread command-line tool that execute a parallel deployment among every node.

4.2.1 Model

To reach the goal of generality the proposed tool implements a model containing different *components* (see Figure 4.1). Each component is designed as an abstraction of the possible entities involved in every deployment:

- An ***Host*** is an abstraction of the machine in which to deploy the application. The machine can be either a Virtual machine or real machine. Each Host maintains all the information needed to communicate with the machine. *Address, Port* and *Authentication* information (e.g., username, password, public key)

- A ***Node*** indicates a runnable instance to upload and execute in every *Host*. The relation *Node-Host* is many-to-one. This means that in one *Host* can be deployed more than one *Node*.

- The ***HostManager*** is the Component in charge to keep all the information about the *Hosts* and the *Nodes*. This information is specified in a configuration file. An example is shown in (add Fig. or section)

- An ***Executor*** is independent and executed in concurrency with respect to the others. It is actually a single thread and is able to perform every task needed during the deployment phases.

- The ***Controller*** of the experiment is in charge to perform all the actions thanks to different *Executors*. (The relation is one-to-many).

- A ***Connection***, in particular an SSH connection, is an opened channel between the controller machine (where LDE is executed) and the *Host*. This way, there will be several connections, one per *Host*.

- An ***LDEFile*** represents any file involved in the deployment phase. These files, whose meaning will be explained ahead, can be (*i*) configurations files of the *Nodes*, (*ii*) output files generated by the *Nodes*, or (*iii*) files handled by LDE.

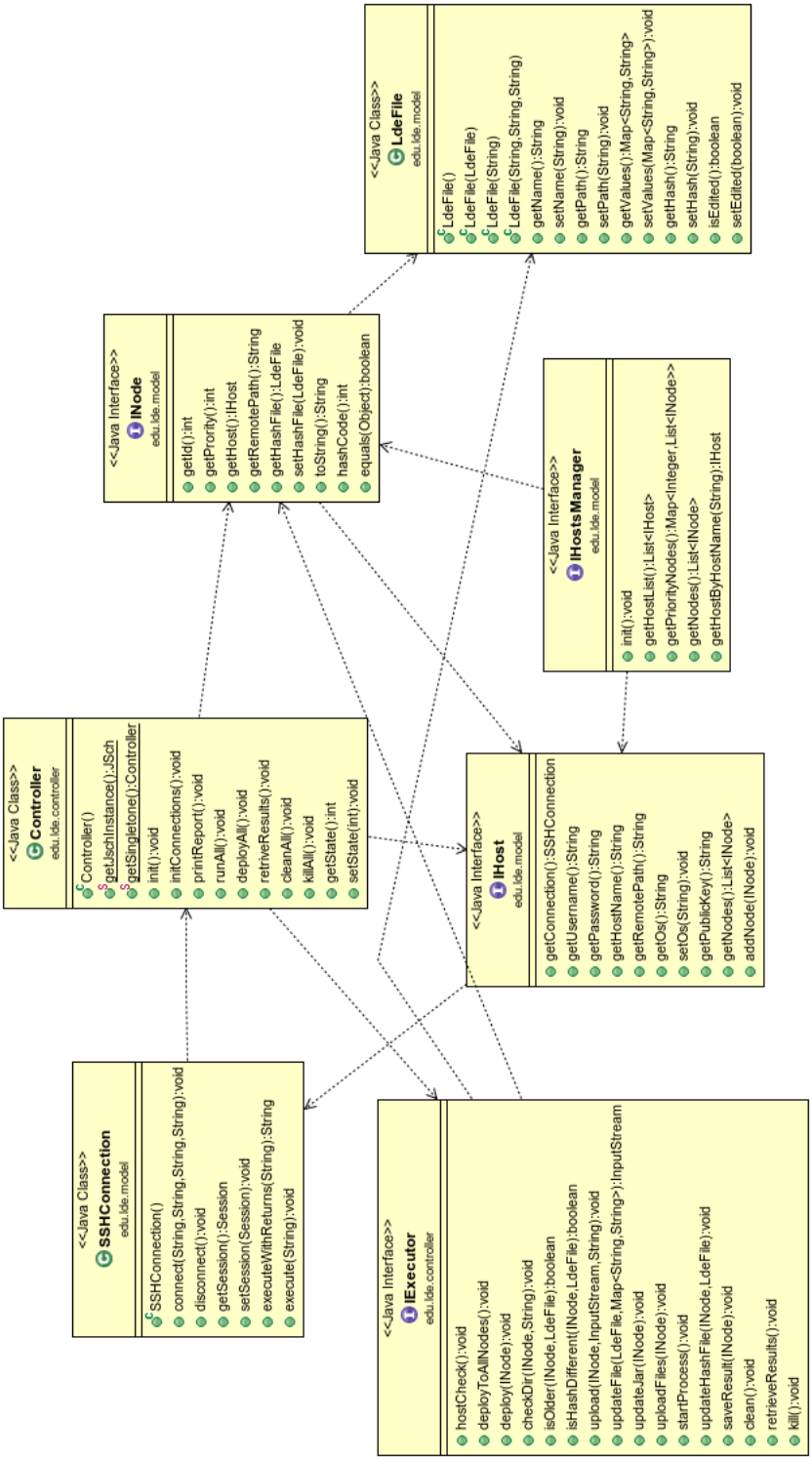


Figura 4.1: *Deployment tool design. LDE Model Class Diagram*

4.2.2 How it works

To perform a deployment with LDE, first of all is needed provide to it all the information for the deployment, that is the *Host* and *Nodes* information and properties for LDE. This information as said are located in the two main configuration files. After this is possible run LDE choosing among two modality: NORMAL and INTERACTIVE. The former is the standard one, it makes sequentially all the deployment steps described below, to achieve a complete experiment. The latter, instead, allows to make every one manually. This is enabled for debugging purposes, since we think it can be useful at the beginning phases have a way to make all the steps manually.

Here are listed all the steps sequentially executed by the *Executors*:

1. ***Authentication:*** LDE needs to open ssh connections and, furthermore, it needs to deploy to several types of *Host*, which can be both a real machine or a Virtual machine, running on local or remote private cluster rather then on public clouds. So it provides an authentication mechanism, which provides the possibility to open a *Connection* through one *Host* using user-name and/or password and/or public-key.

2. ***Deploy:*** Perform the upload of the software component (*Node*). Before that step are executed two checks:

   (a) ***Path check:*** As said, every *Node* needs a directory path on the remote *Host*'s file system. So in this step the existence of the path is checked. If it is not present the *Executor* will creates it. Note that if is not specified any path will be considered the remote OS default path ($HOME).

   (b) ***Hash check:*** if the software is an updated version it will be uploaded, otherwise not. This is most important for a performance point of view. In fact, avoiding not necessary uploads there is an huge save of time. This check is performed comparing the hash of the Component (*Node, LDEFile*) already uploaded and the new Component to upload.

3. ***Run:*** Once the deployment is completed, a different *Executor* for each *Node* will send a run command. The run command can be sent at the

same time to every node (since by different threads) or with a given priority. Priority means that the *Nodes* if needed can be started before of others. The priority is a configurable value from 0 to 2, where 0 is the maximum priority, 2 the lowest. This has proven to be useful in situations in which a given protocol needs that some *Nodes* start before others.

4. ***Retrieve results:*** At the end of the experiment, whose duration should be indicated in a configuration file. LDE checks if some output files are generated. If so they will be retrieved. *Note:* Since the output files have different names and extensions in different application, and also because it can be useful download more than one file per *Node* there is a configurable parameter in which it is possible specify the pattern (a regular expression).

The steps listed above describes the basic behaviour, which could be seen in our advice, as a common way to deploy a P2P experiment, composed by software Components that need to be uploaded, edited and executed on local or remote distributed machines.

LDE provides other (optional) features, which can be useful in some other situations. An important one is the possibility to merge some configuration files (*LDEFile*) in the remote Component (e.g., jar) in the Host. This means that if the Component (*Node*) is already uploaded by previous experiments, and you want execute other experiments changing just some parameter of the Nodes, LDE provides an upload of just these files merging them in the Component in the Host. This happens at the *deploy* step through the hash check action, in which the *Executor* will notice that are changed the configuration files but not the *Node*. This can be very effective in situation like these, in which the experiment involves hundreds or even thousands of *Nodes*.

4.3 Deployment example with MycoCloud

The design and the use of LDE is to allow deployment for any type of large-scale experiments, with the possibility to customize, that is, add or edit the default features. However the first objective was to allow us to deploy

MycoCloud in real networks (Cloud networks, in particular) and to achieve this objective we added to LDE one function useful for this scope.

This feature enables dynamically edits of Nodes configuration files at deployment time, that is between the steps 1 and 2 (see subsection 4.2.2). This function helped us to achieve three important facilities in the deployment of MycoCloud.

- The first one is an automatic port number assignment to the nodes. These port numbers are randomly generated given a port range specified in LDE. In this way given the Cloud firewalls rules we could give to LDE a specific range - previously enabled on the firewalls - from which the ports are picked up and assigned to every node.

- The second instead use the Nodes id generated by LDE, assigning them to every (ProtoPeer) Node. These id, which will correspond to a *ProtoPeer index*, are placed into a specific Node configuration file. ProtoPeer in fact use a concept of *index* that is basically an id, but every node needs to know in advance which is its own one.

- The last is to "say" to every ProtoPeer node its own assigned *Address*. Within this action is also indicate at every Node which is the Address of the Node with (assigned) index 0 (also called *Node Zero*). The need to indicate which is the *Node Zero* to the other nodes is due to the fact that the Node Zero is the entry point for the network bootstrapping. In ProtoPeer in fact, a small number of Nodes is in charge to contact the Node Zero to start the experiment.

Node zero furthermore needs to start before the other nodes. For this reason we implemented the feature, described in the subsection 4.2.2, which gives *run priorities* to the Nodes.

With the "basic" LDE described before and a "smart" use of functionality already provided we have been able to deploy and execute MycoCloud in cloud providers.

Capitolo 5

MycoCloud Evaluation

This chapter gives a detailed description of the experiments developed to evaluate the proposed approach. In the first section are described the experimental parameters. The second section discusses how the experiments have been set up and the metrics we extracted. Finally, we describe the scenarios chosen to evaluate MycoCloud and the results obtained.

5.1 Experimental setting

The parameter space of systems involving thousand of peers can be very large, and careful selection is of utmost importance in order to create representative experiments for the most common scenarios. Table 5.1 lists some parameters that are specific to the MycoCloud protocol and that have been fixed after some experimental trials and sensitivity analyses. For the experiments discussed has been used values of $C_S = 5$ for the target number of

Symbol	Description	Value
C_S	Target number of peers of the same type	5
C_O	Target number of peers of different type	3
MN_{period}	Activation period Myconet protocol	0.5
MN_{period}	Activation period load-balancing protocol	0.3
C_{max}	Maximum capacity per node	60
μ	Average service execution time	1

Tabella 5.1: Fixed experiment parameters

same-type hyphal links (the Myconet default) and $C_O = 3$ for the target number of different-type hyphal links, although we also tested the protocol with values of $2 \leq C_S \leq 6$ and $2 \leq C_O \leq 6$. The sensitivity of the network to values of C_S and C_O lower than the default is limited to possible churn of the network increase somewhat, but the performance of the service network in processing requests remains similar. Higher values did not result in improved performance, but showed increased messaging overhead.

Other parameters used in the simulations depend on the specific scenario to be tested. In particular, we vary the network size composing the system from 200 to 5000 peers. The number of distinct services in the network is important, as it puts under pressure the ability of MycoCloud to form many interconnected clusters of same-type peers. In our experiments we vary the number of services up to 25.

In the experiments we fixed $C_{max} = 60$ with a power-law distribution. The simulation time is from 200 to 400 seconds, depending on the time needed to show the characteristics we want prove.

5.2 Experiments

We decided to test MycoCloud in a stable scenario. In this scenario the network is static, in the sense that no peers join or leave the existing network for the course of the experiment. This because a churn scenario was not needed to prove our considerations. Then we designed three different type of load scenarios which are representative for the adaptive algorithm. In general every of them makes the whole system in hard conditions, and are designed to compare MycoCloud in adaptive version and in no adaptive version. Basically without the third layer (see section Design in Chapter 3 for reference). In particular:

Test Peaks described in the subsection 5.2.1, is executed with 200 and 1000 nodes and with 5 services for both the cases. This test is performed for two reasons. The first is to give a graphic example of how the clusters evolve over the time, in which every service takes one at time almost the whole resources of the system at sudden load peaks. The second reason is to makes in trouble the algorithm, generating load peaks one close to the other. This gives an idea of the convergence speed of the

algorithm since one service at time will stole all the resources, and when other peaks comes to the next clusters, they will be completely deprived of their resources.

Test Easy is a trivial success case for the adaptation version, where a service receives much more load then its own capacity. This test is performed with 1000 nodes and 5 services, since at large scale the situation would be the same. The results of this experiment are illustrated in the subsection 5.2.2 at page 63.

Test Constant The results of this test, executed with 1000 and 10.000 nodes are reported and described in the subsection 5.2.3 at page 65. This test provides an example of situation in which in all the services the load has a shape with some steps growing until become constant. The main objective of this test is to find a lower bound for the improvement provided by the elasticity algorithm.

In every experiment will be evaluated the following different metrics:

(i) *Response Time* relevant for the clients, is a way to compare much closer the benefits of the elasticity.

(ii) *Response Time Optimality* gives an idea on how far is the measured response time is far from being equal to the optimal one of the ideal system.

(iii) *Load Per Service* to see how the algorithm balances the load of all the services.

(iv) *Services Capacity* useful to understand how the capacity varies according the load events among the services.

(v) *Network Messages* to evaluate the increase of messages given by the algorithm.

5.2.1 Test Peaks

Here are discussed the results of the experiment code-named Test Peaks introduced above. This test executed with 1000 nodes and 5 services has load peak coming at turn to every service as showed in figure 5.1.

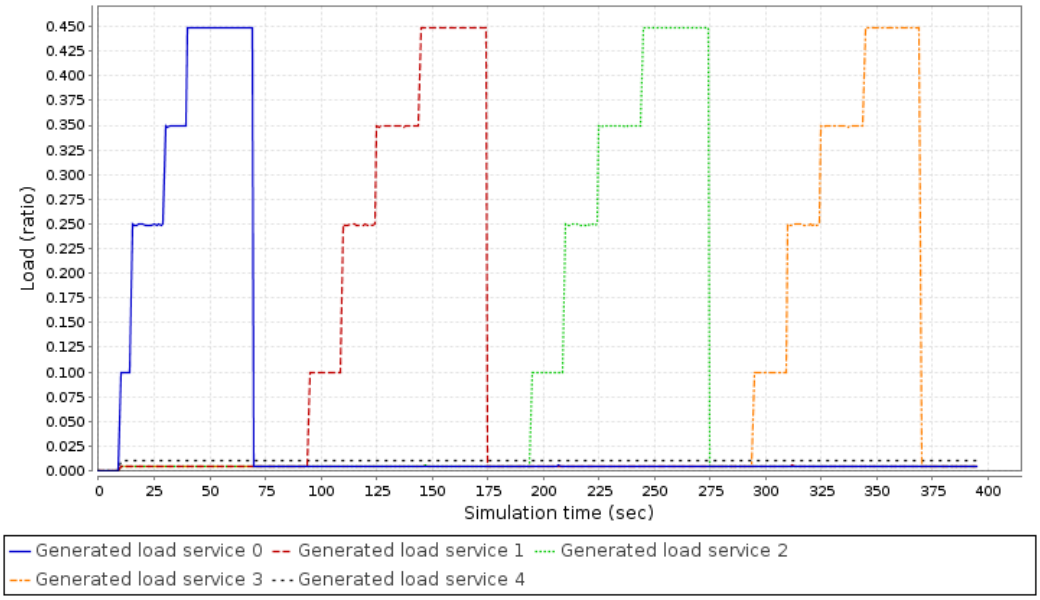


Figura 5.1: Test Peaks - Load shape

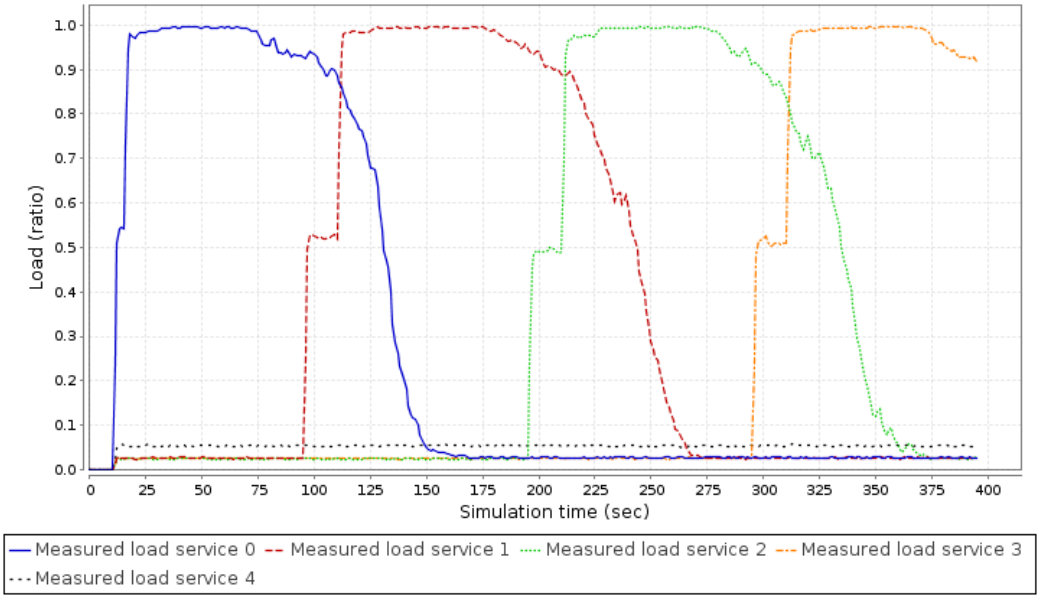


Figura 5.2: Test Peaks - No Adaption: Load per service

The load injected in each peak is equal about to the 45% of the whole system capacity, thus a given capacity for each services is not enough to handle it. This can be seen in figure 5.2, where is shown how the load reach the 100% above the services, which are not able to satisfy the amount of requests. Figure 5.5 shows how the queues grow making drastically worse the performances of whole system (See Response Time in Figure 5.6 and Response Time Optimality in Figure 5.8).

Figura 5.3: Test Peaks - Adaption: Load per service measured

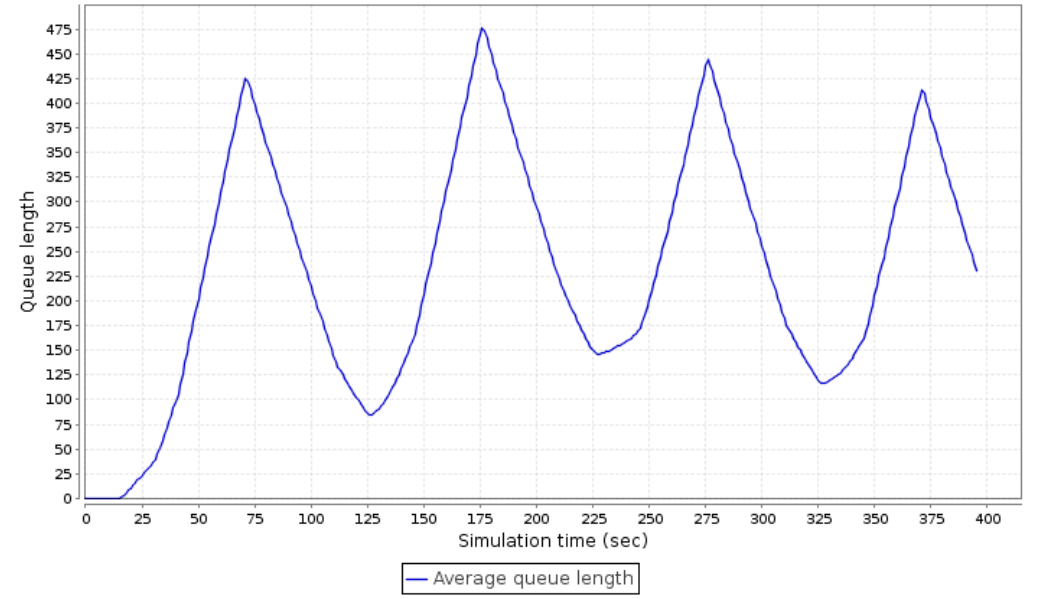


Figura 5.4: Test Peaks - No Adaption: Queue length

At contrary enabling the adaptive algorithm the system is able to counteract to the sudden load peaks moving in turn the needed resources to service in trouble. The figure 5.10 shows how each service takes the capacity from other services (that are less loaded) during the peak instants and the load measured among the services tends to be equalized by the algorithm moving resources where necessary. This is underlined by the queue length in this case (figure **??**) respect to the precedent. For a comparison, the constant capacity

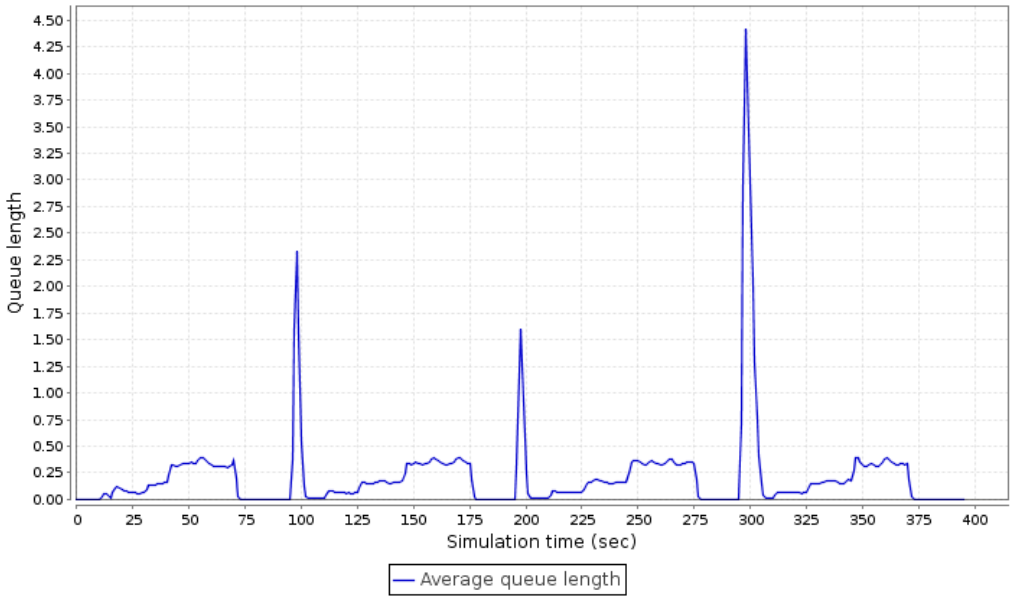


Figura 5.5: Test Peaks - Adaption: Queue length

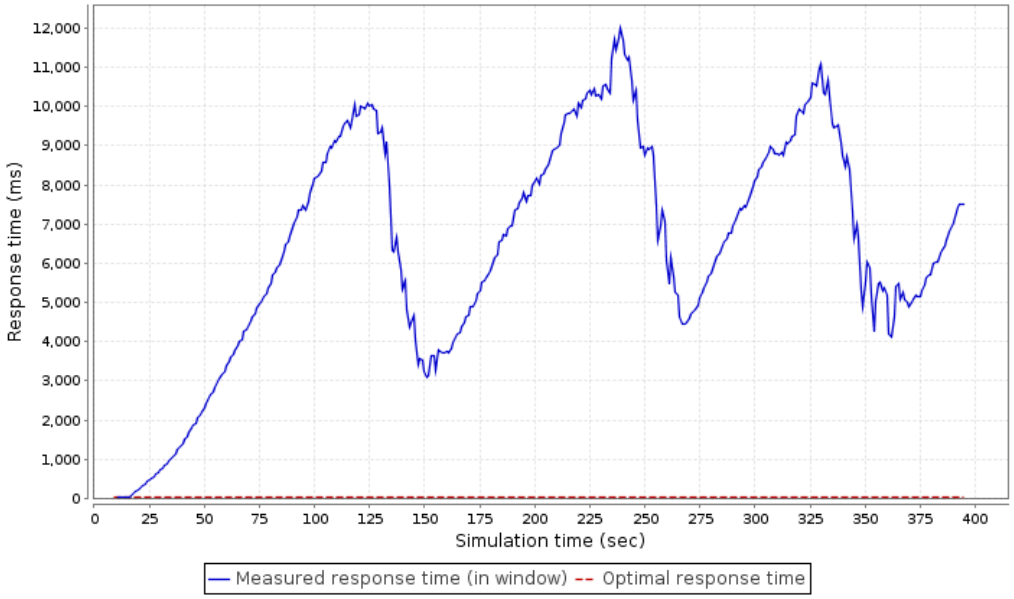


Figura 5.6: Test Peaks - No Adaption: Response Time

in non adaptive version in shown in the figure 5.11.

The Response Time and the Response Time Optimality for the adaptive case are showed respectively in Figure 5.7 and Figure 5.9. These graph shows also the side effect of the deprivation of all the resources from a service. In fact when the peaks arrive to the other services (1, 2 and 3) they will be with very few nodes (Capacity) and they will affect much more than the service 0. This characteristic is showed by the instantaneous peaks in the RT and RTO

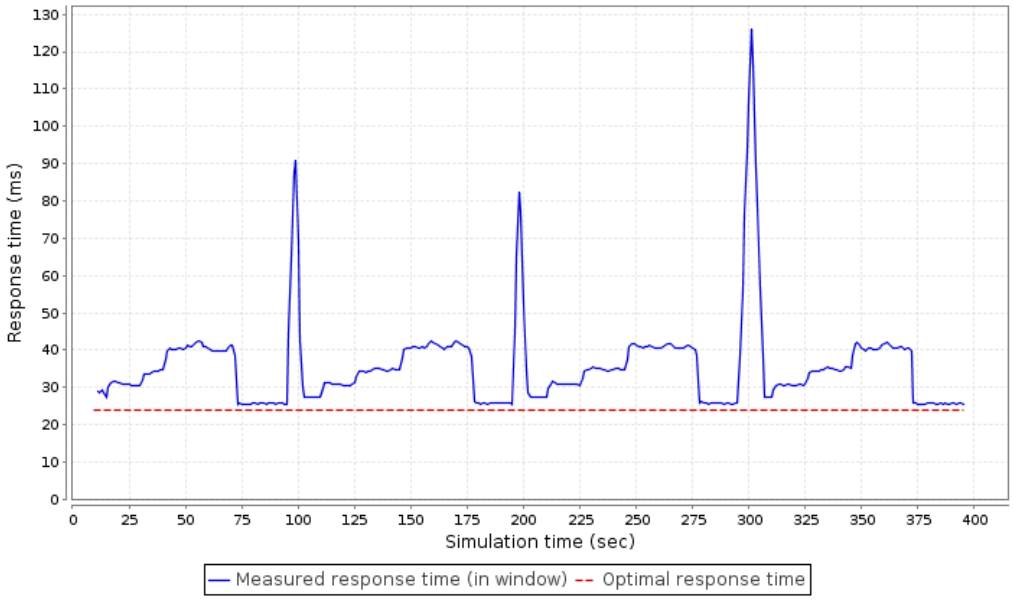


Figura 5.7: Test Peaks - Adaption: Response Time

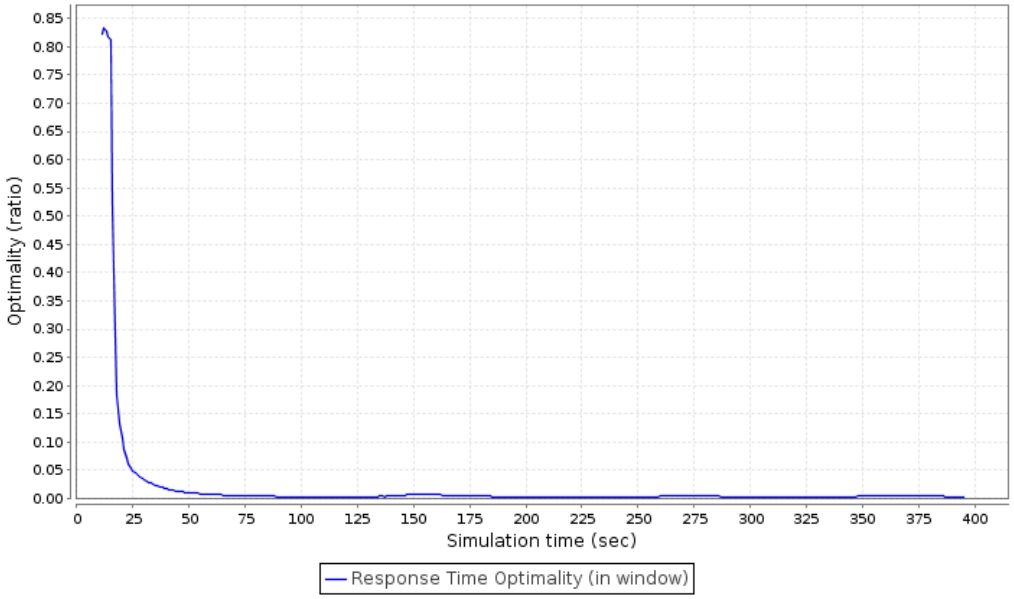


Figura 5.8: Test Peaks - No Adaption: Response Time optimality

at times 100, 200 and 300. This "side effect" is very limited, rather instead shows the very fast convergence into readapt the whole system, and as seen the benefit are remarkable for this edge case.

Figure 5.12 and Figure 5.13 show how the network traffic increases in corrispondence of the increase of service change. In particular this increase is due to the fact that the topology have to rebuild itself to repair for "unexpected" events.

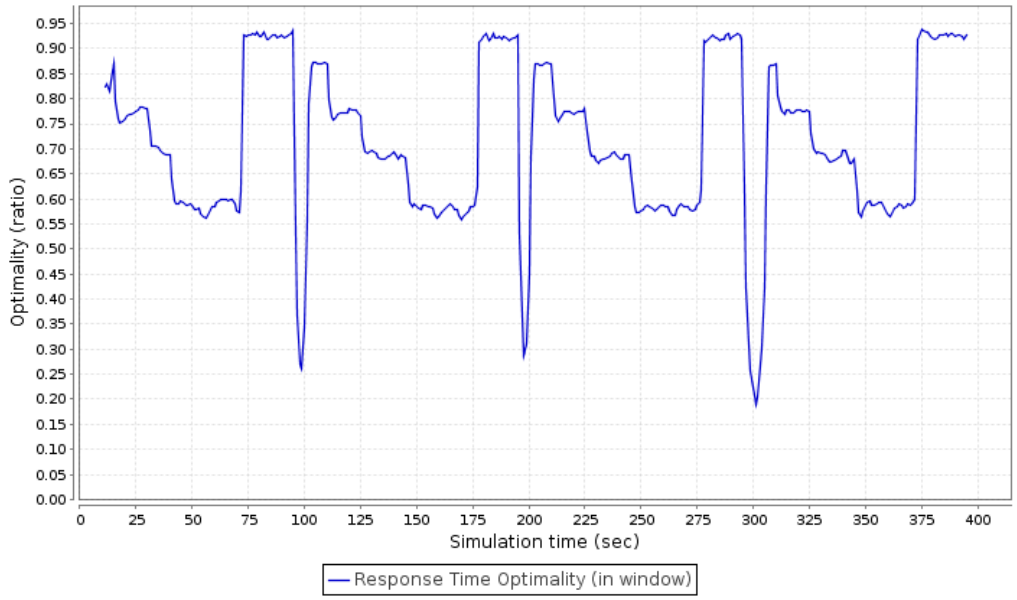


Figura 5.9: Test Peaks - Adaption: Response Time Optimality

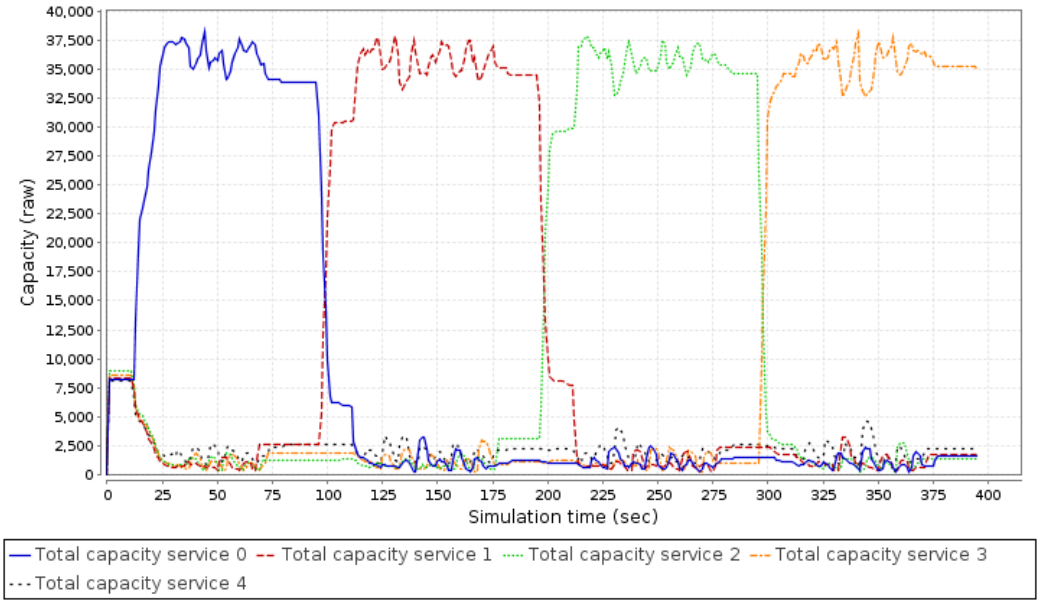


Figura 5.10: Test Peaks - Adaption: Capacity for each service

Finally the same test has been executed with 200 nodes in order to show a graphic overview of the system evolution during the peaks. Figure 5.14 gives an image of the system before the peaks when is perfectly clusterized. Figure 5.15 instead show how the clusters at turn steal all the resources from the other ones.

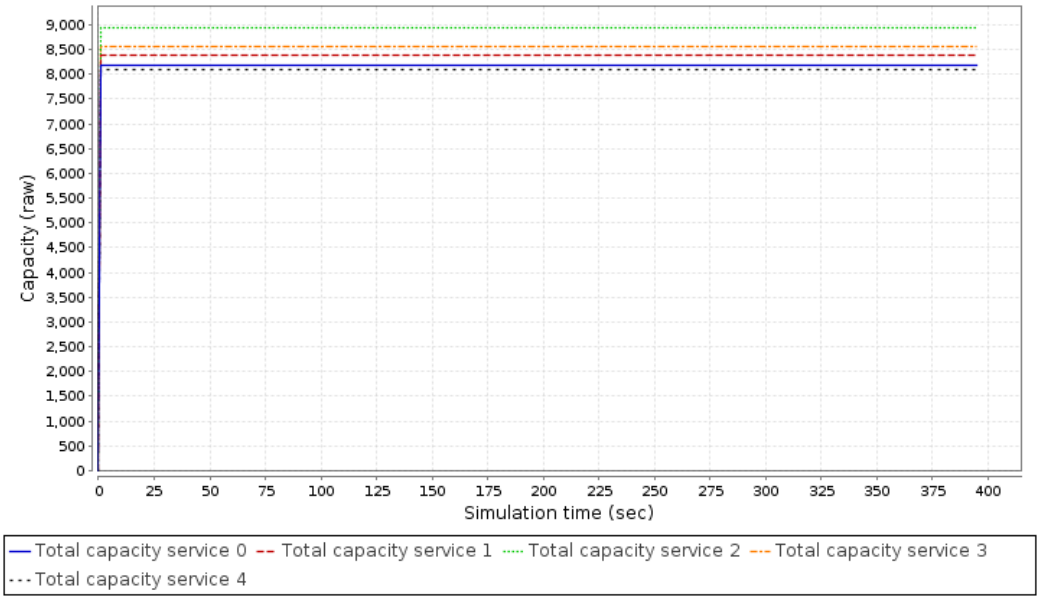


Figura 5.11: Test Peaks - No Adaption: Capacity for each service

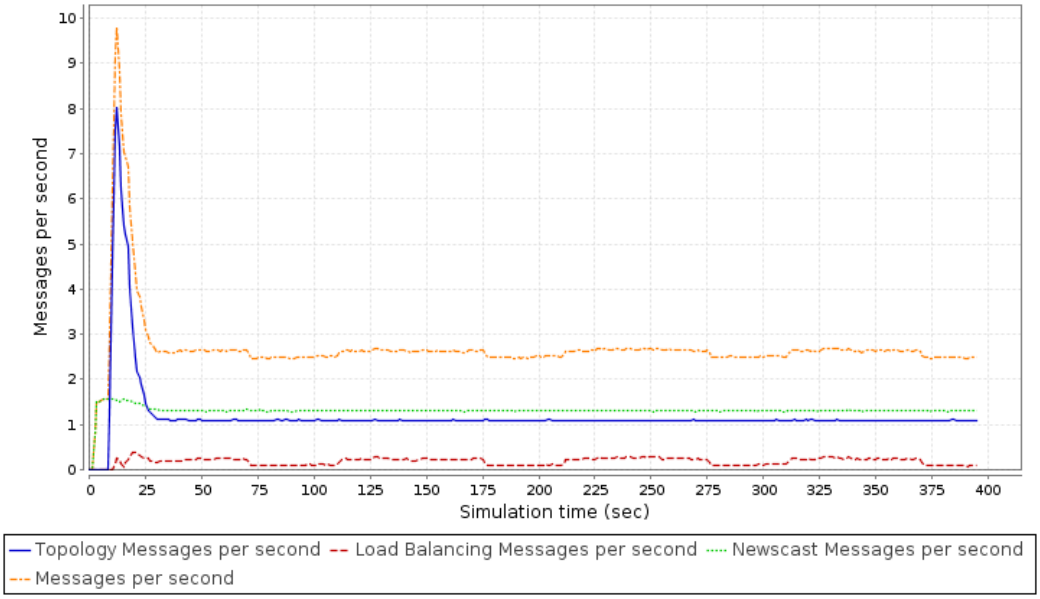


Figura 5.12: Test Peaks - No Adaption: Network Messages

5.2.2 Test Easy

This last test is more trivial then the previous ones. In this scenario in fact there is a service that receives a load a little higher then its capacity. In this case with a static system the service cannot asks for other resources and is not able to handle the requests. Figure 5.17 shows the load in each services and in particular shows the load on service 0 growing to 100%.

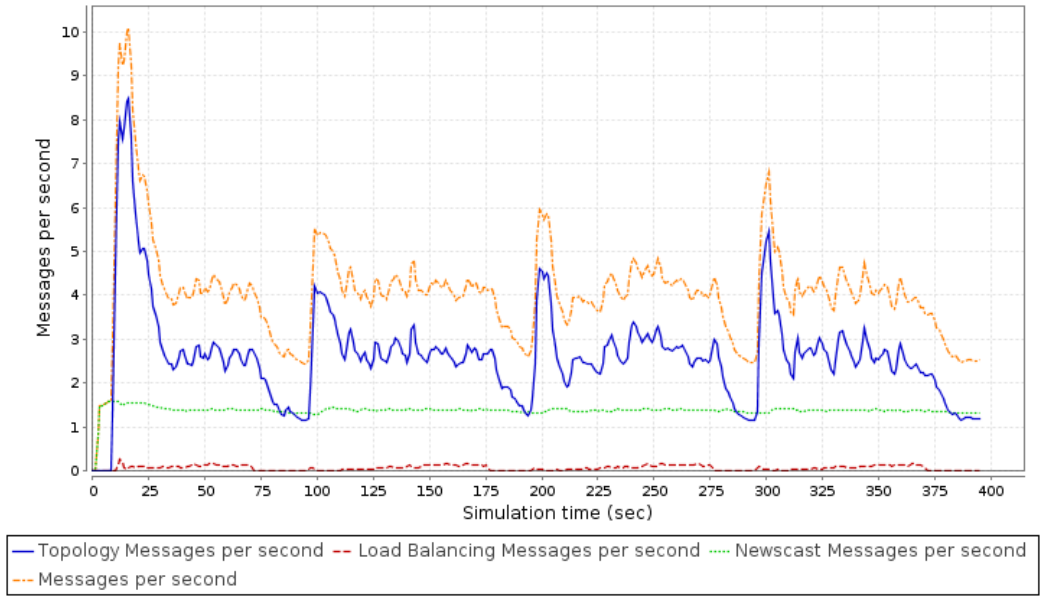


Figura 5.13: Test Peaks - Adaption: Network Messages

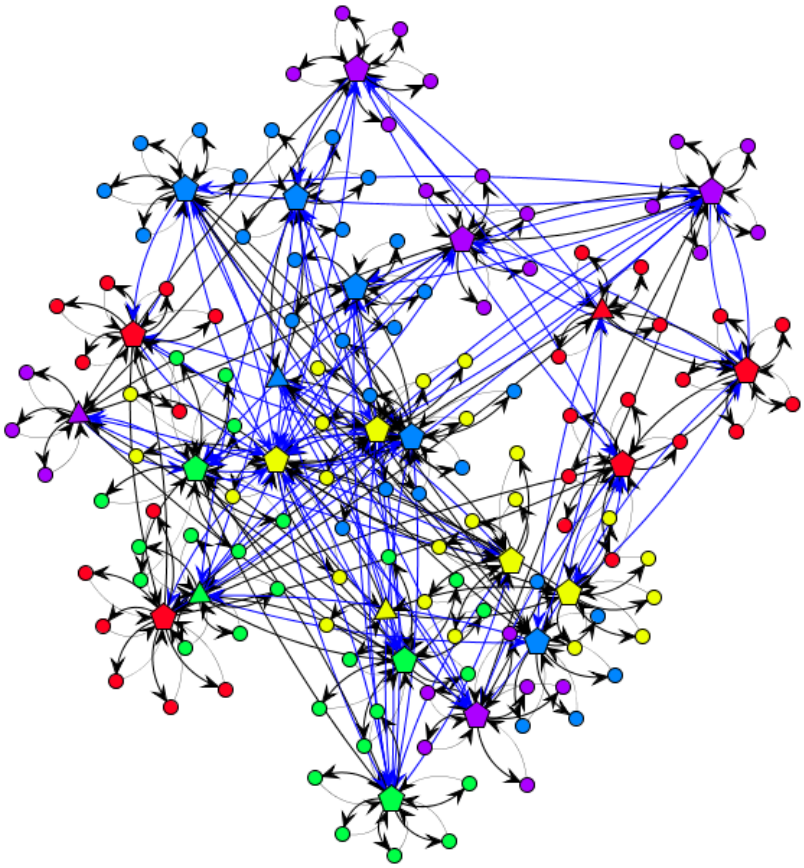
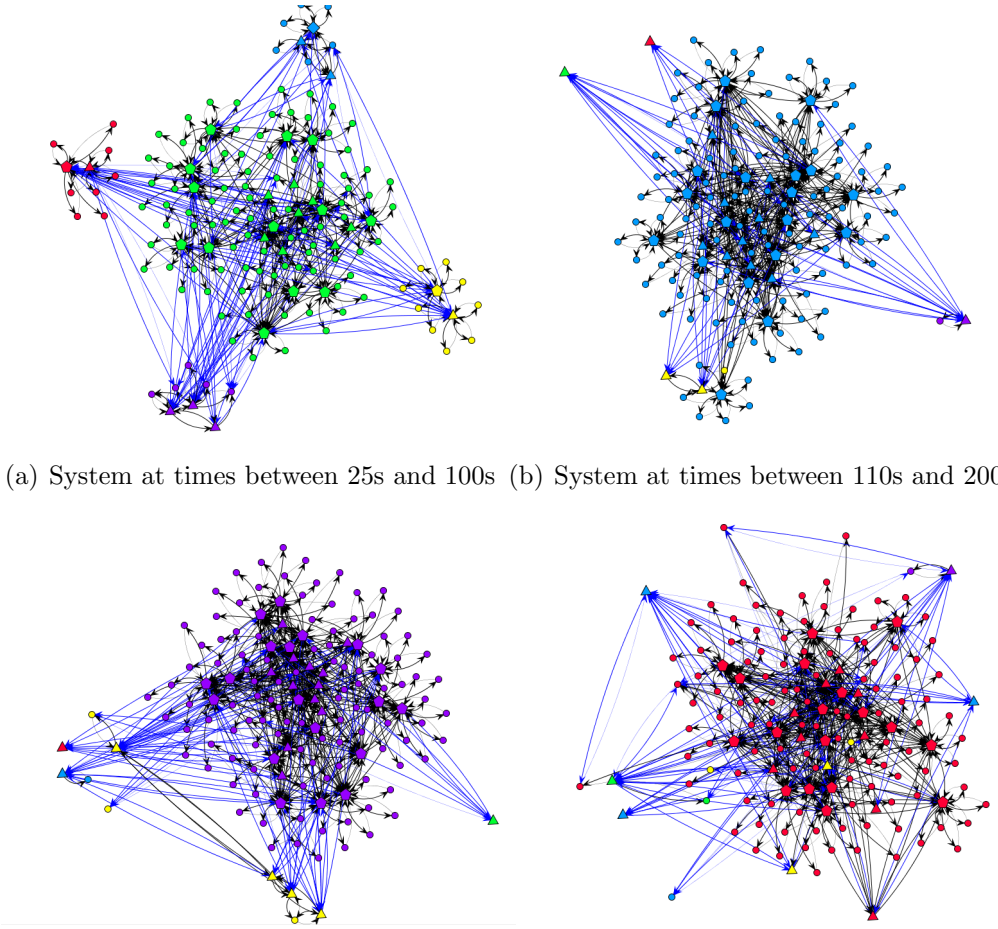


Figura 5.14: Test Peaks - System at start



(a) System at times between 25s and 100s (b) System at times between 110s and 200s

(c) System at times between 210s and 300s (d) System at times between 310s and 400s

Figura 5.15: Test Peaks - System evolution over the time

The performances in case of no elasticity are dramatically worst then the adaptive version. The Response Time and the Respose Time optimality for both the cases are shown respectively in Figure 5.18 and Figure 5.20 for the no Adaptive and in Figure 5.19 and Figure 5.21 for the Adaptive case.

A wiev of the elasticity is given also by Capacity graphs in Figure 5.22 and Figure 5.25.

5.2.3 Test Constant

In this test, called test constant, is shown a scenario in which every services has a growing load that at certain point becomes constant. Every service is loaded from about 30% to about 85% as shown in figure 5.27, thus no one is overloaded. The idea is to design a scenario in which the System is able to

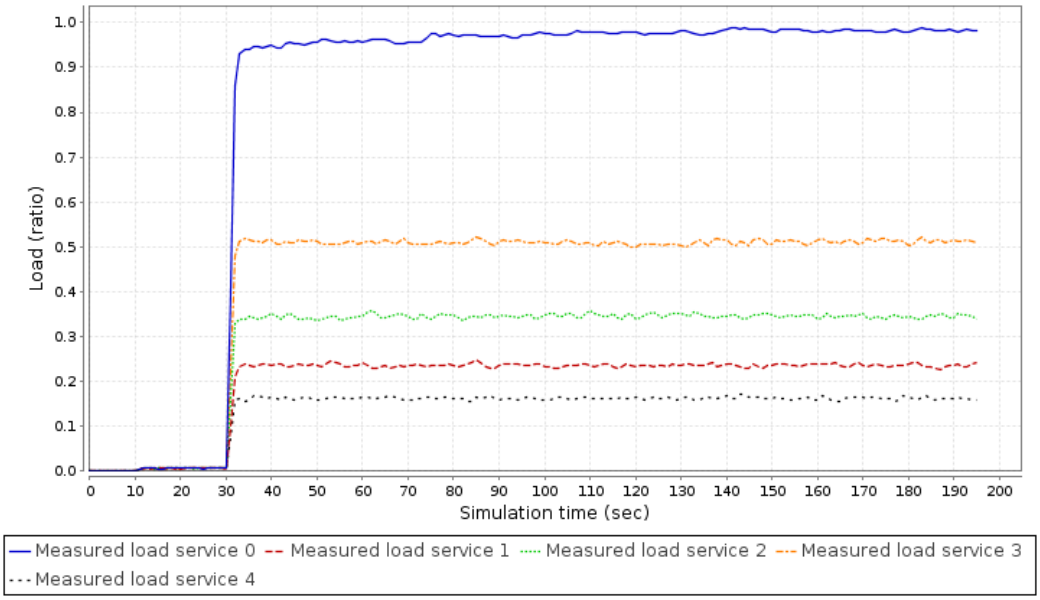


Figura 5.16: Test Easy - No Adaption: Load per service

Figura 5.17: Test Easy - Adaption: Load per service

execute its own requests without necessity of further capacity. Furthermore we want define a random grow fo the load among the service, which in case of no adaptation makes no difference, but in case of adaptation would be an hard situation. This because almost at the same time even in coincident instants some services will try to aquire resources.

The first test is with 1000 nodes and 5 services, and as said the system without adaptation is able to handle the incoming load. In Figure 5.28 and Figu-

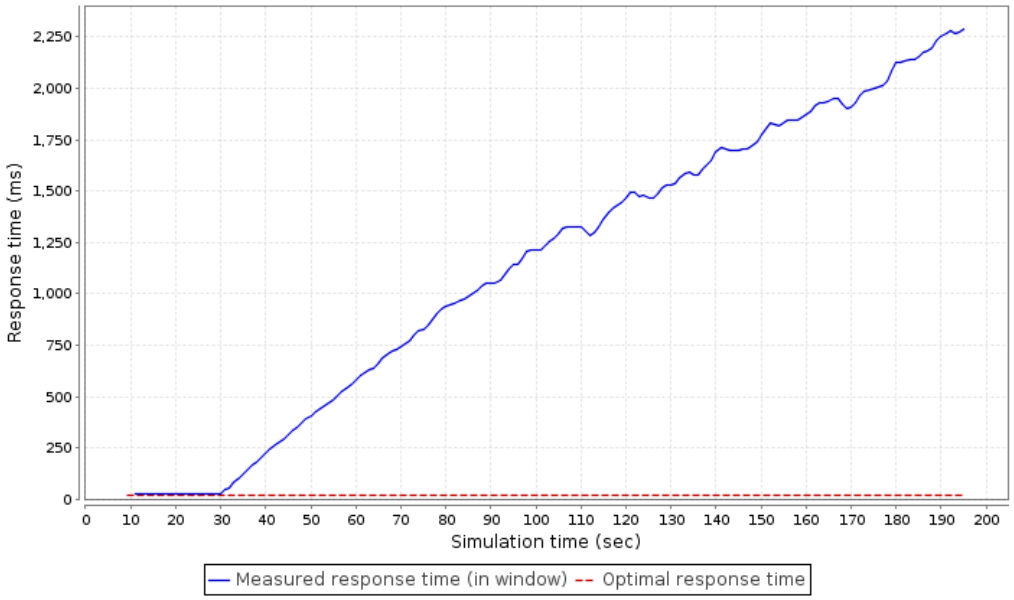


Figura 5.18: Test Easy - No Adaption: Response Time

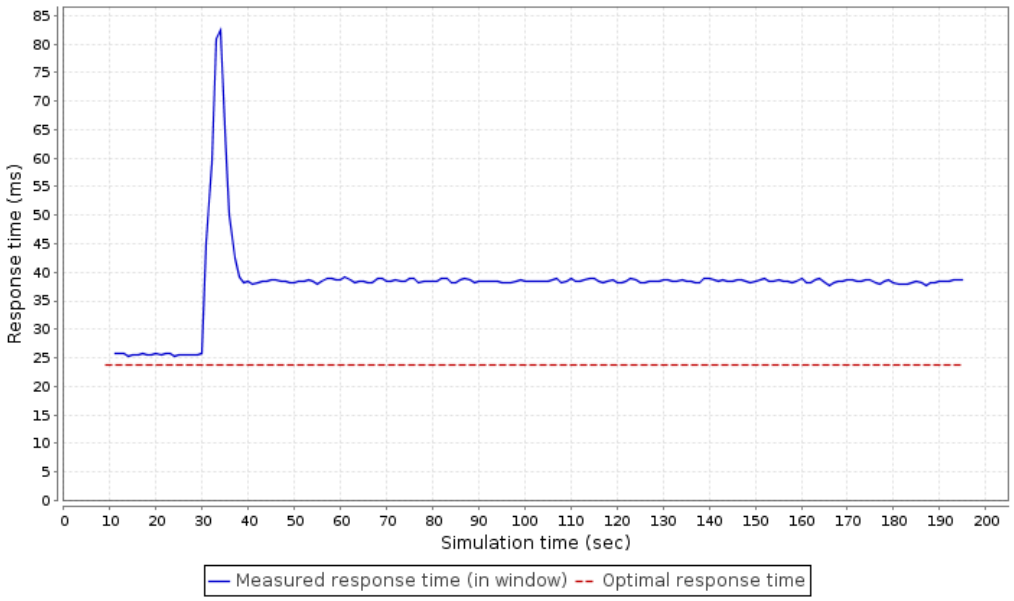


Figura 5.19: Test Easy - Adaption: Response Time

re 5.30 are showed both the Response Time and the Response Time Optimality for this case.

Since the high load in service 1 and 4 (figure 5.27) the system is not able to reach an ideal Response Time because the queues are not equal to zero (see Figure 5.32).

The same experiment with the adaptive shows how the system in at the beginning tries to reach a balance of the capacity among all the services. In

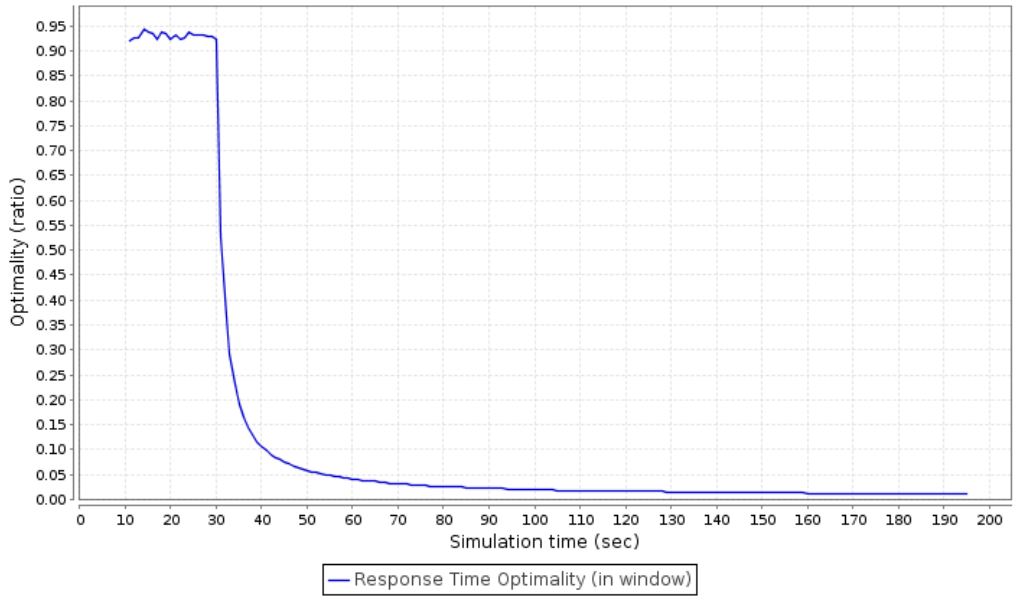


Figura 5.20: Test Easy - No Adaption: Response Time Optimality

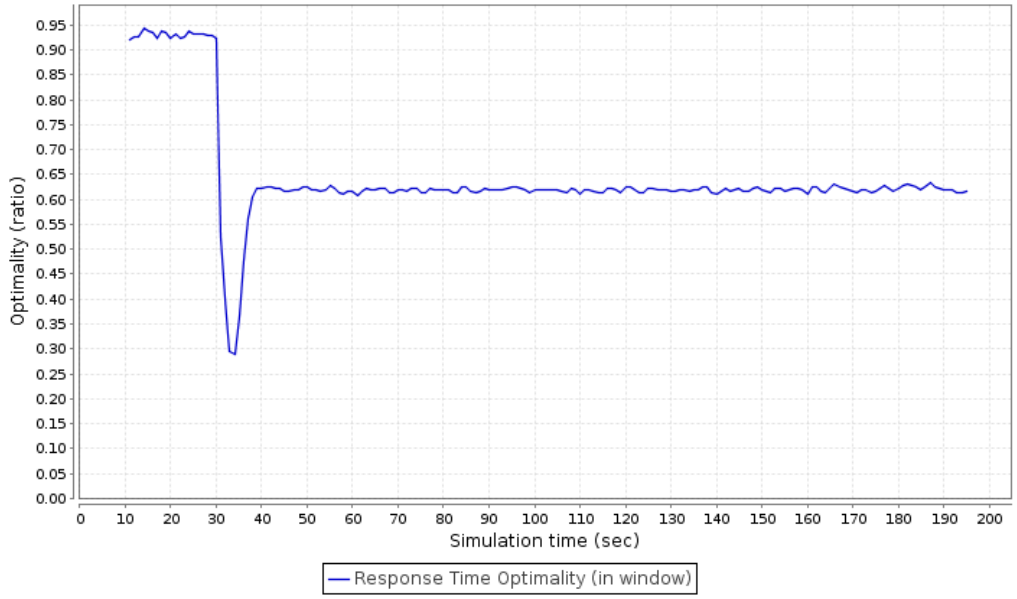


Figura 5.21: Test Easy - Adaption: Response Time Optimality

this phase the services try to take the resources each other a reach a final "agreement" bringing the Response Time of about 15% better then the test with no adaption.

As anticipated this was a design on purpose, to make in trouble the algorithm and the results show that it has some peaks in instants in which some services comes up asking for capacity previously given. These are little instants showing the fast react of the system, and in general even if this is an hard cases

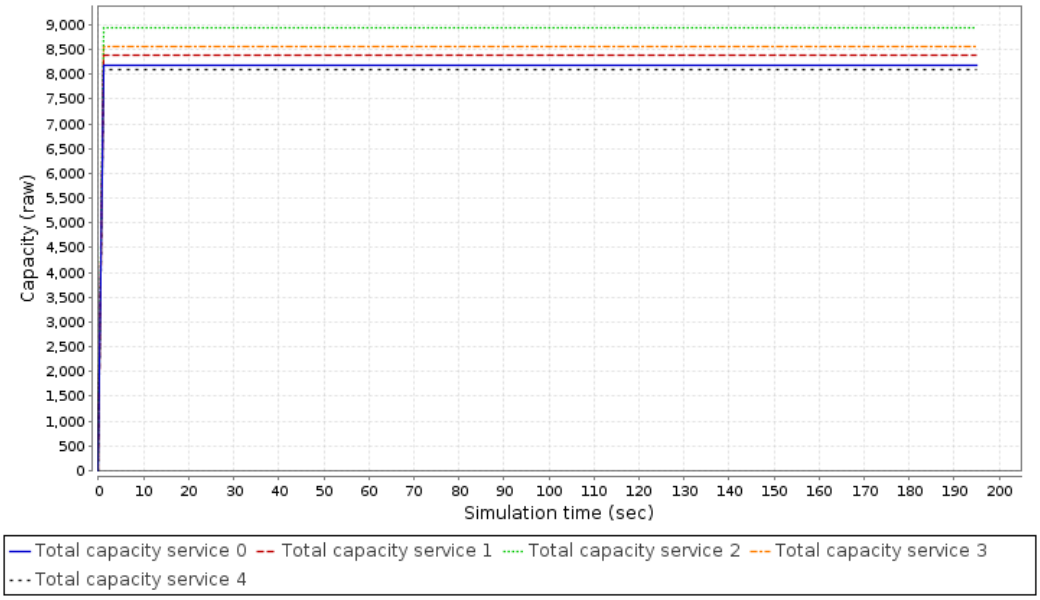


Figura 5.22: Test Easy - No Adaption: Capacity

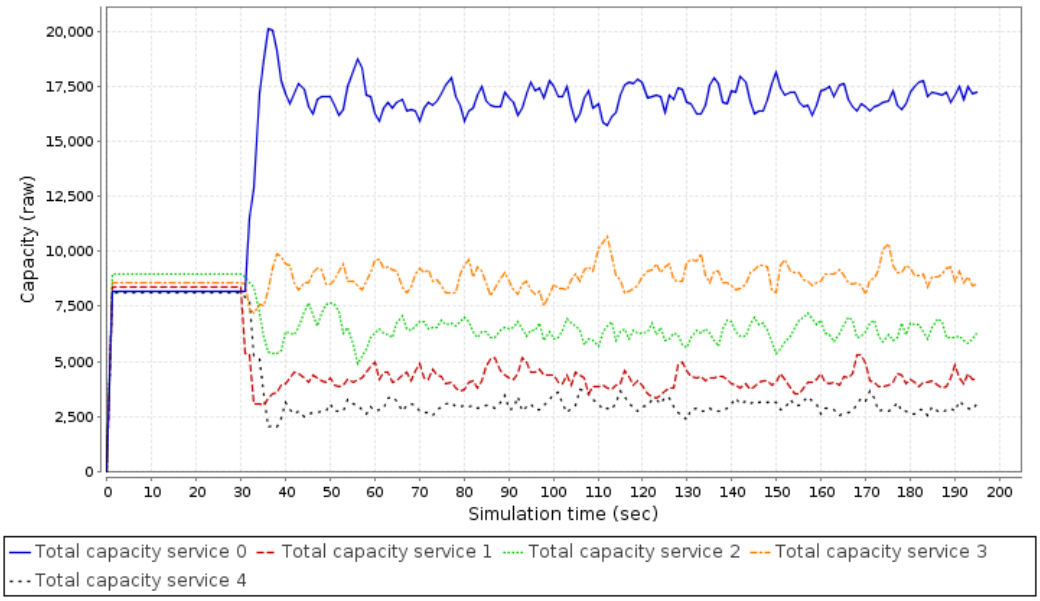


Figura 5.23: Test Easy - Adaption: Capacity

expressly designed, after this critical instants the performance are better with respect the no adaptation case. For a view of the Response Time and the Response Time Optimality see Figure 5.29 and Figure 5.31.

Another interesting graph is the capacity per services (Figure 5.35) in which is shown how the services finally converges to the capacity each one need.

Finally, in this test we made a comparison choosing two levels for the threshold. The results for the adaptive test described above (Figure 5.31 and

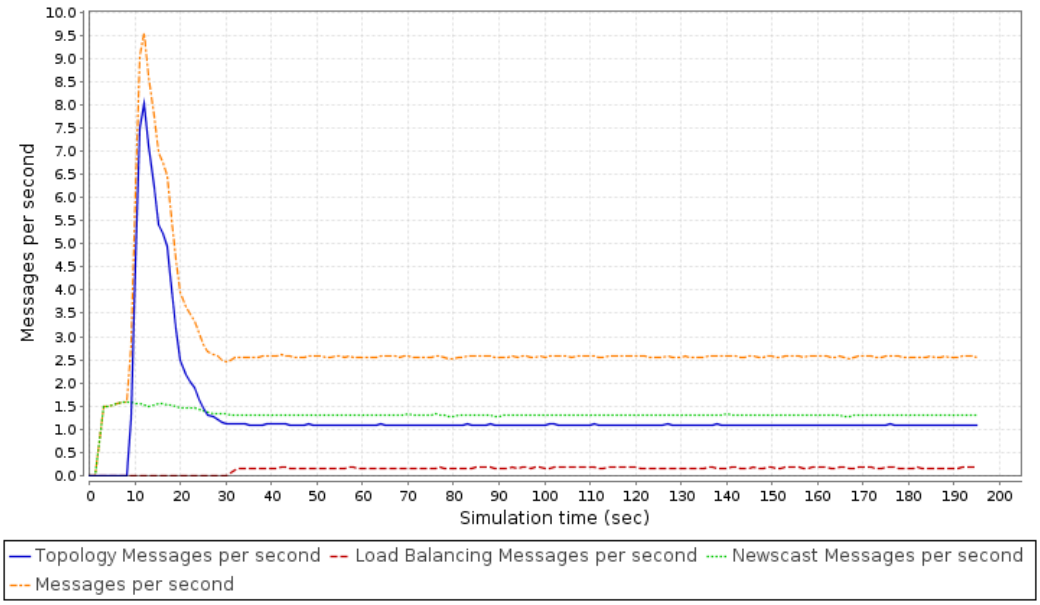


Figura 5.24: Test Easy - No Adaption: Network messages

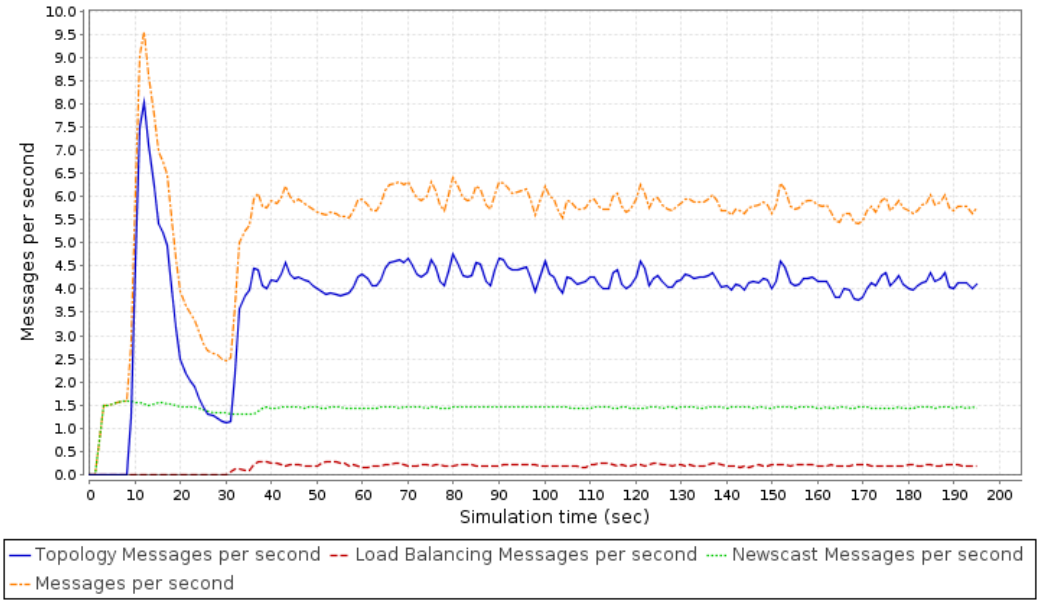


Figura 5.25: Test Easy - Adaption: Network messages

Figure 5.35) were with a threshold set to 50%. While the test executed with a threshold at 75% showed more "noise", how can be seen in the Capacity graph in Figure 5.38. This is due to the fact that the higher threshold set in the second case is more or less at the same value of the load among services, which means that they will continue to steal resources each other.

Figure 5.36 and Figure 5.37 show how the performance are slightly worse with respect to the previous case with a lower threshold (50%). However even

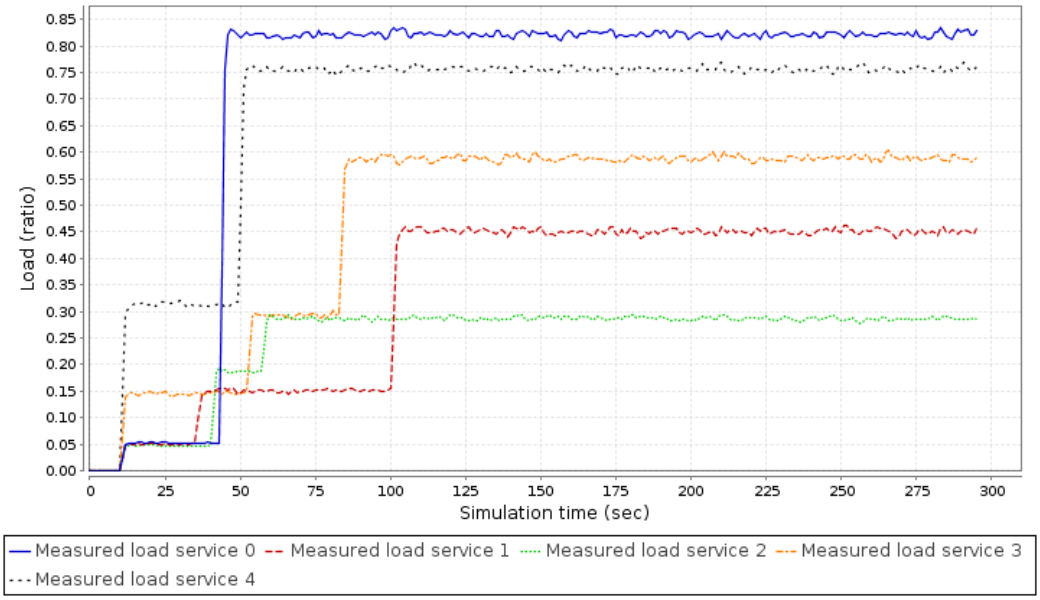


Figura 5.26: Test Constant 1k No Adaptation - Load per service

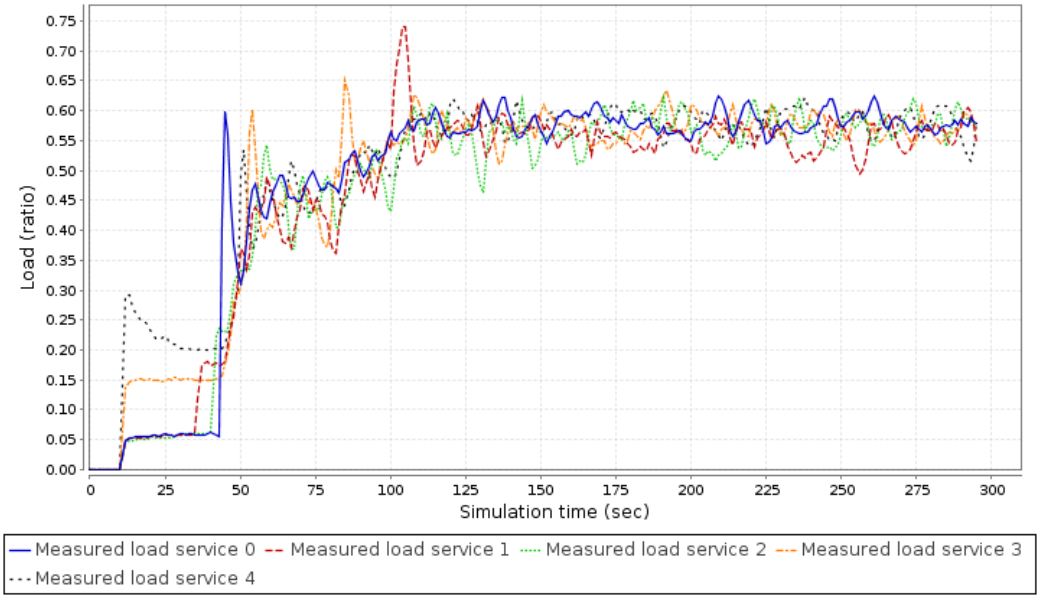


Figura 5.27: Test Constant 1k Adaptation - Load per service

in this case the Response Time and the Response Time Optimality are better than the case without adaptation. This gives an idea of when and how can be useful have a right threshold. In the other previous test (Section 5.2.1) this evaluation was not necessary since the services were either overloaded or underloaded.

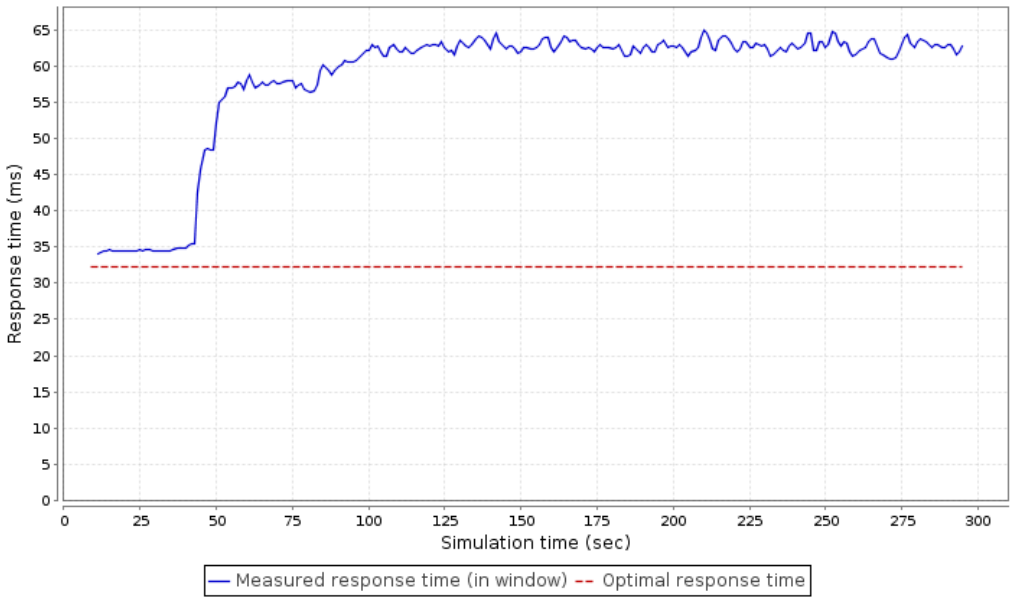


Figura 5.28: Test Constant 1k - No Adaption: Response Time

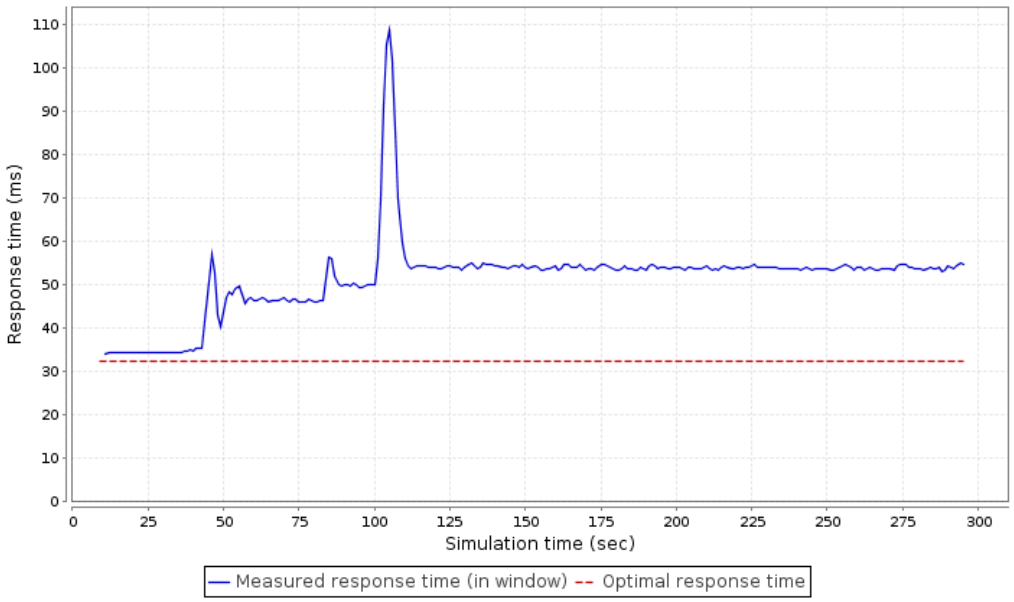


Figura 5.29: Test Constant 1k - Adaptation: Response Time

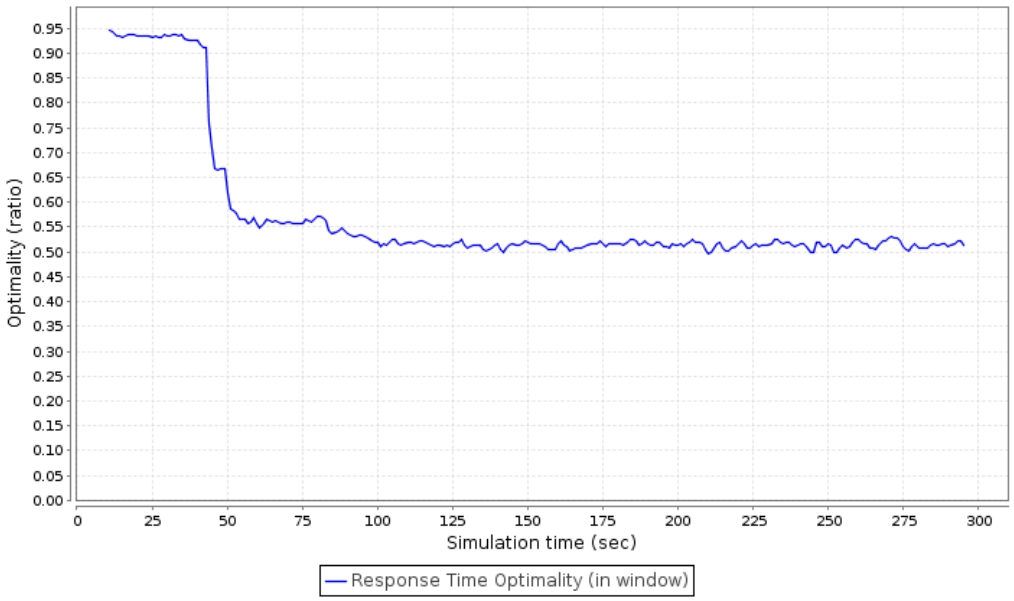


Figura 5.30: Test Constant 1k - No Adaption: Response Time Optimality

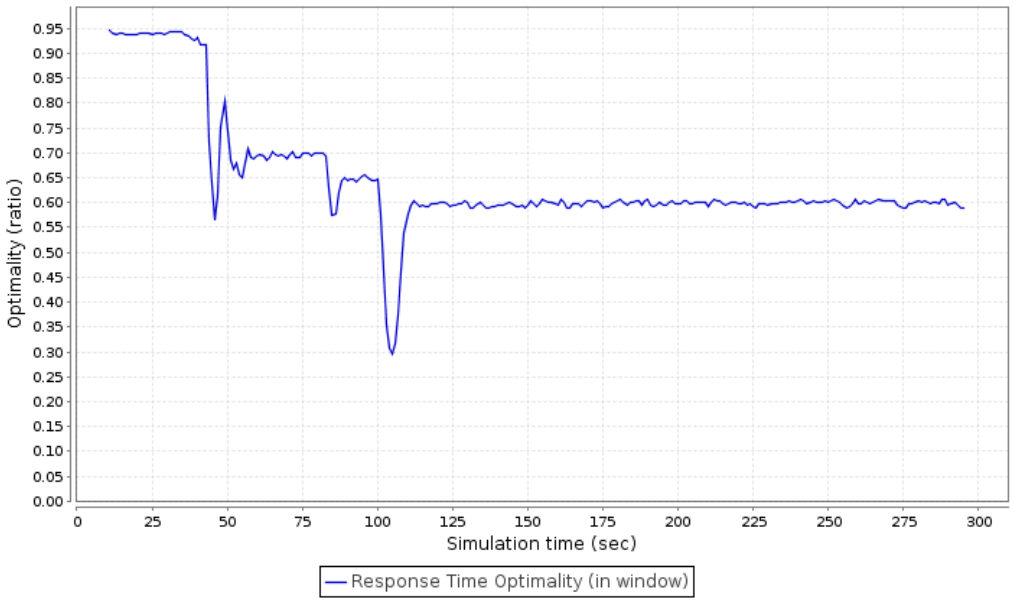


Figura 5.31: Test Constant 1k - Adaptation: Response Time Optimality

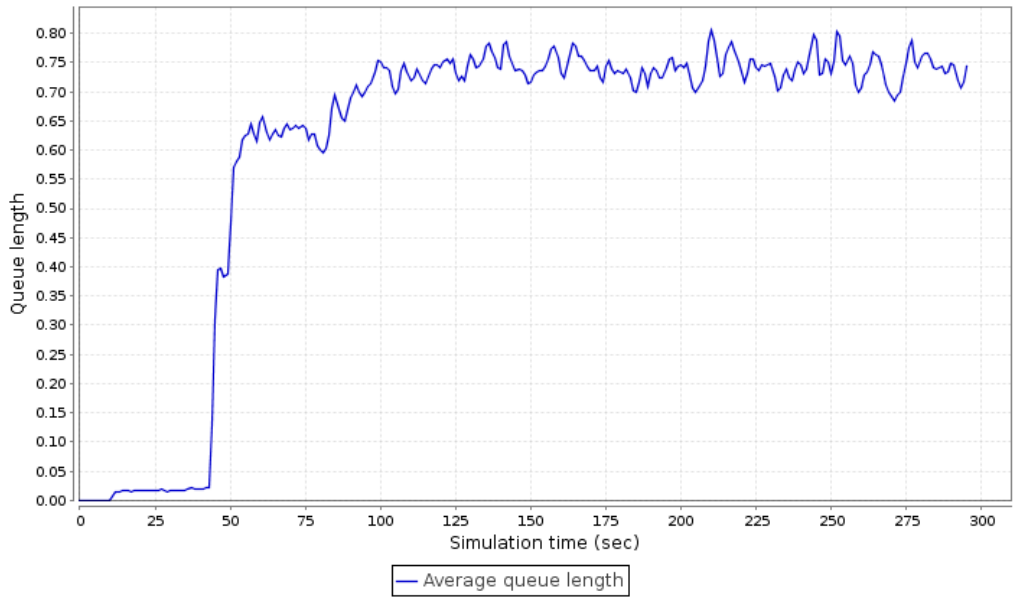


Figura 5.32: Test Constant 1k - No Adaptation: Queue

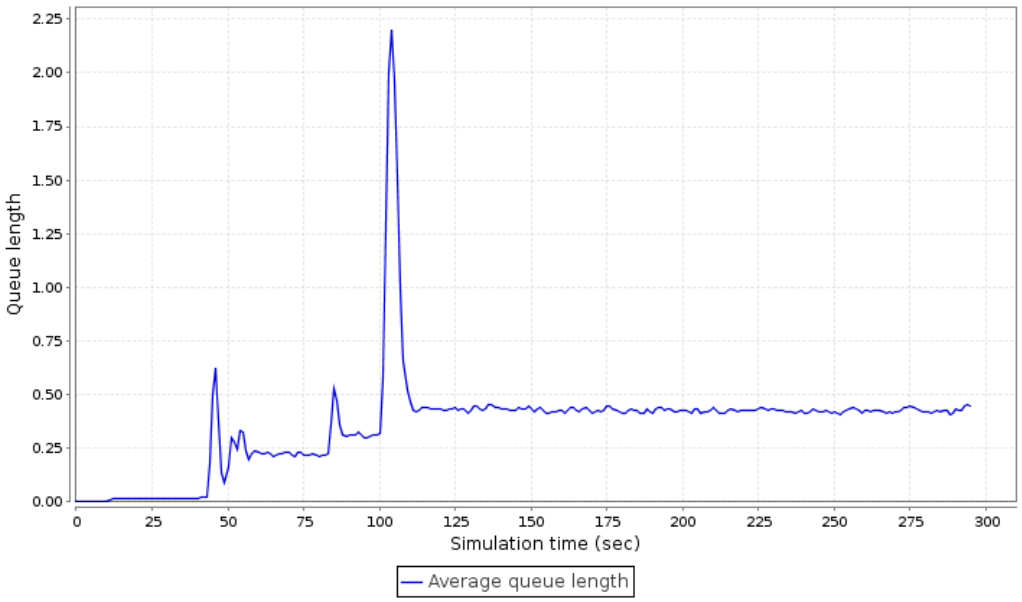


Figura 5.33: Test Constant 1k - Adaptation: Queue

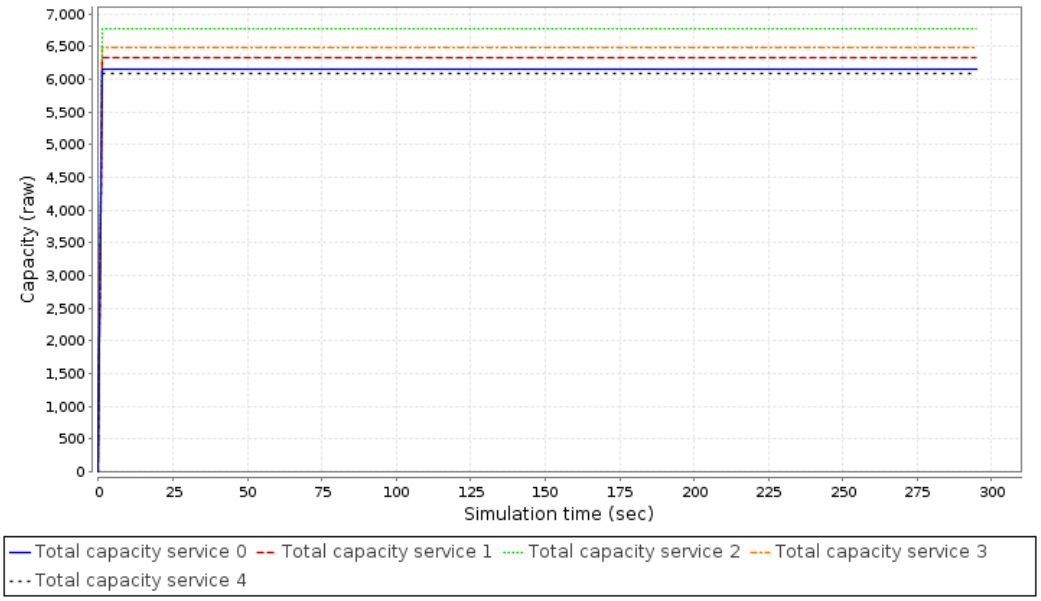


Figura 5.34: Test Constant 1k- Adaptation: Capacity per service

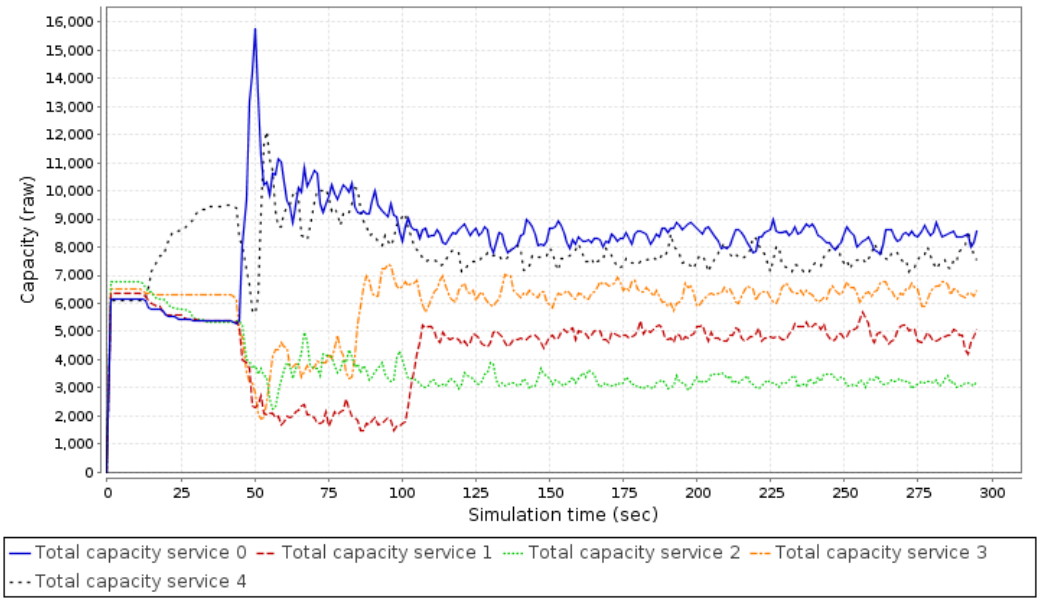


Figura 5.35: Test Constant 1k- Adaptation: Capacity per service

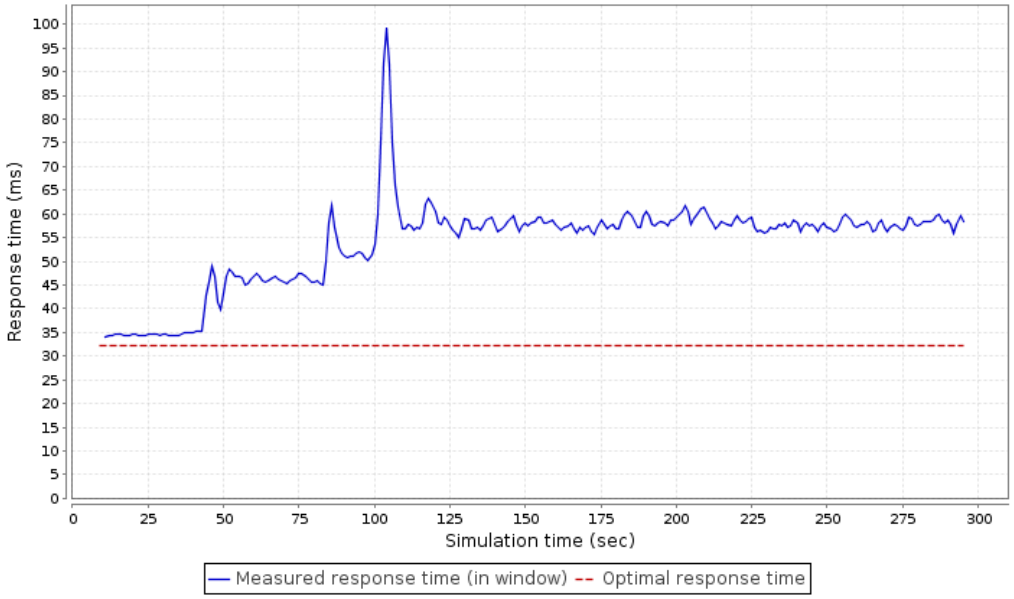


Figura 5.36: Test Constant 1k - Adaptation: Response Time

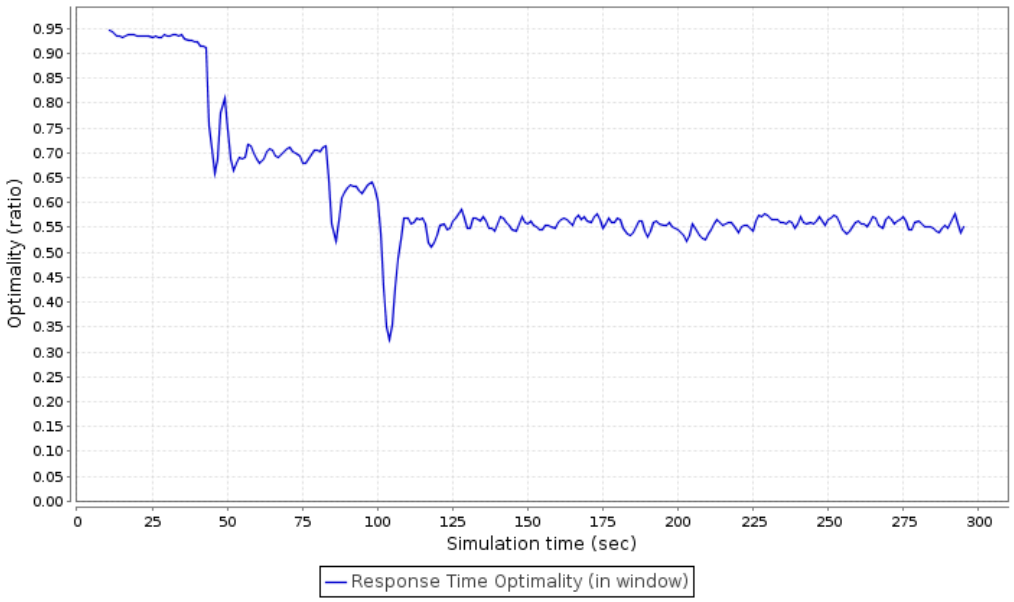


Figura 5.37: Test Constant 1k - Adaptation: Response Time Optimality

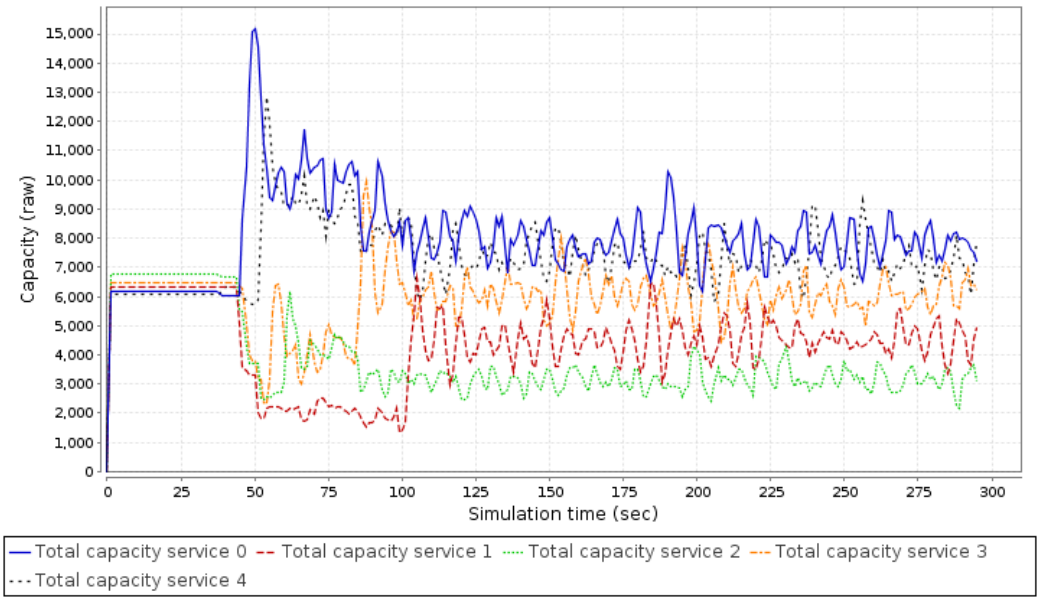


Figura 5.38: Test Constant 1k - Adaptation: Capacity per service

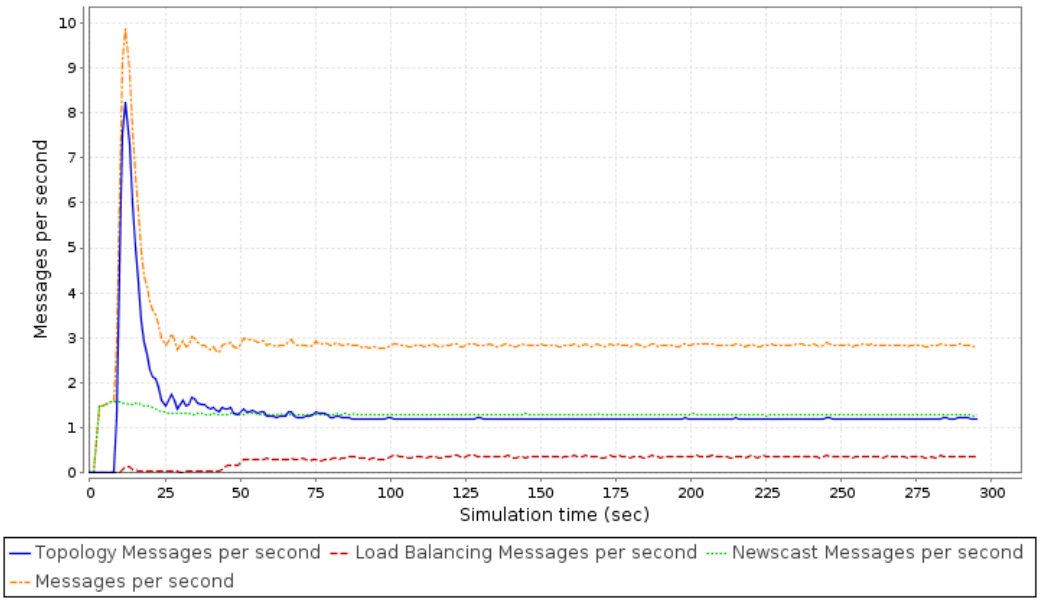


Figura 5.39: Test Constant 1k - No Adaptation: Network messages

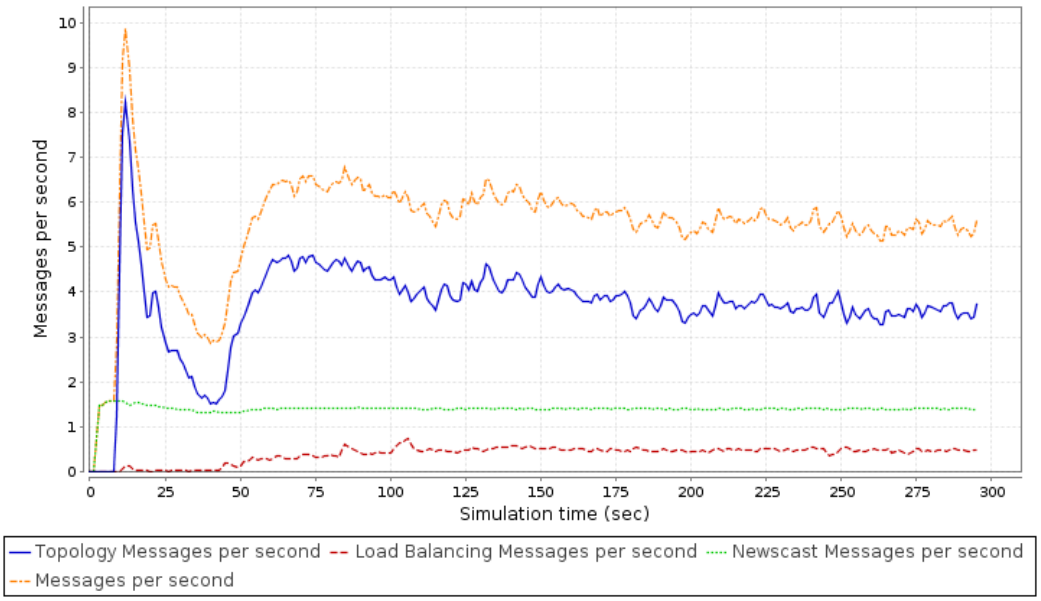


Figura 5.40: Test Constant 1k - Adaptation: Network messages

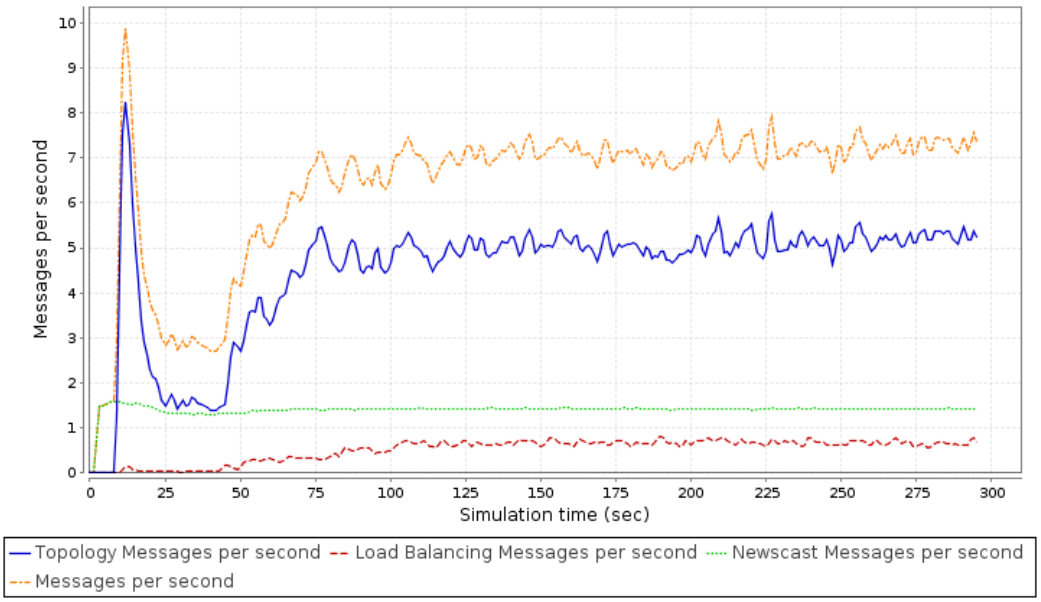


Figura 5.41: Test Constant 1k - Adaptation: Network messages, higher threshold

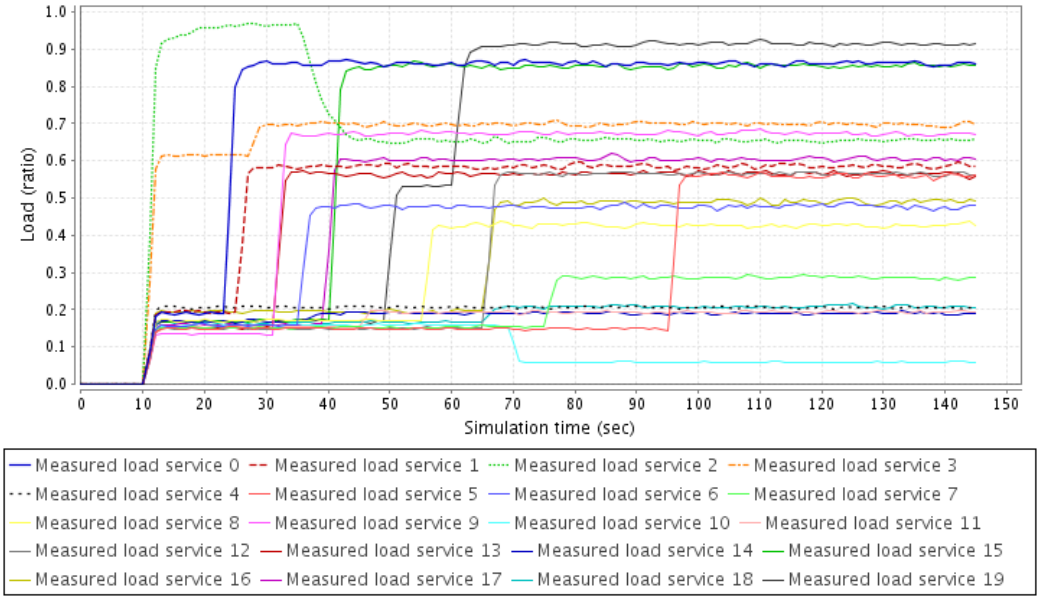


Figura 5.42: Test Constant 10k - Load per service

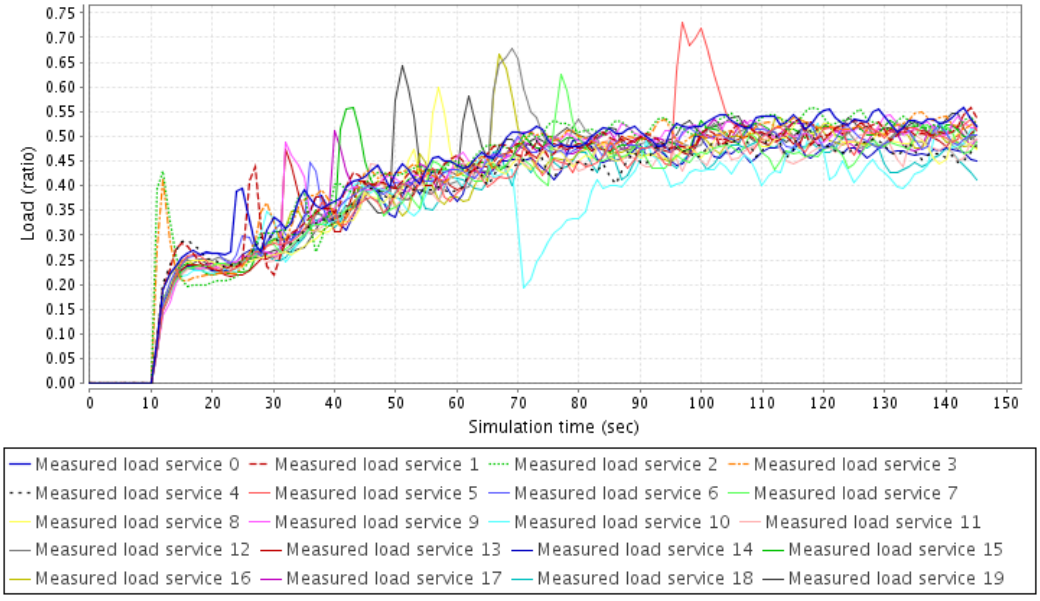


Figura 5.43: Test Constant 10k- Adaptation: Load per service

Finally Figure 5.39, Figure 5.40 and Figure 5.41 show the increasing in message traffic respectively for the cases with no adaptation, adaptation with 50% threshold and adaptation with 75%. The results show how the topology messages have an increase in adaptation test especially in case of higher threshold. This, as said before, is due to the topology algorithm that needs to continuously rebuild the network, resulting higher in the test with a wrong threshold.

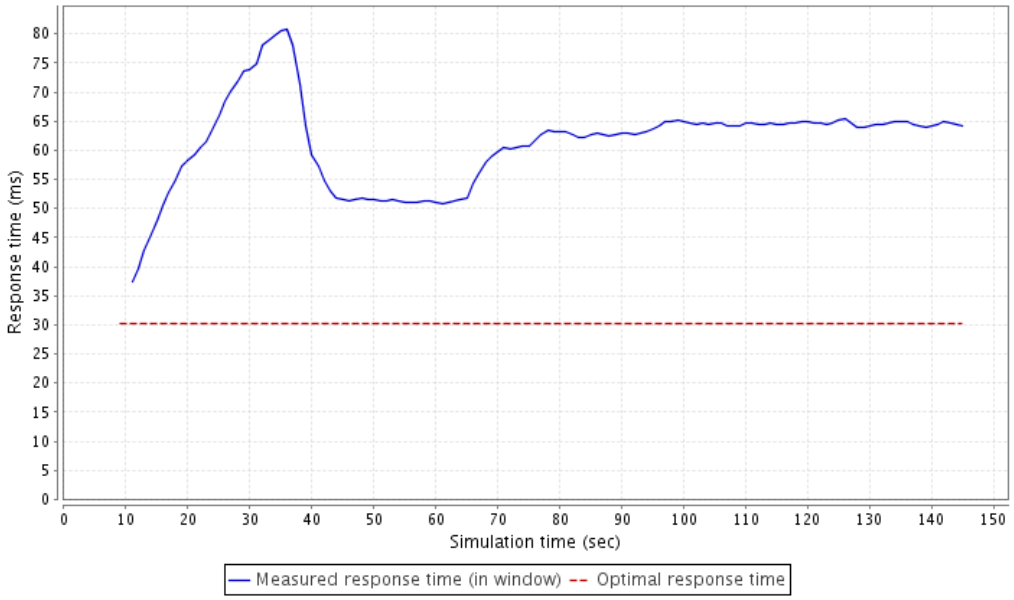


Figura 5.44: Test Constant 10k - No Adaptation: Response Time

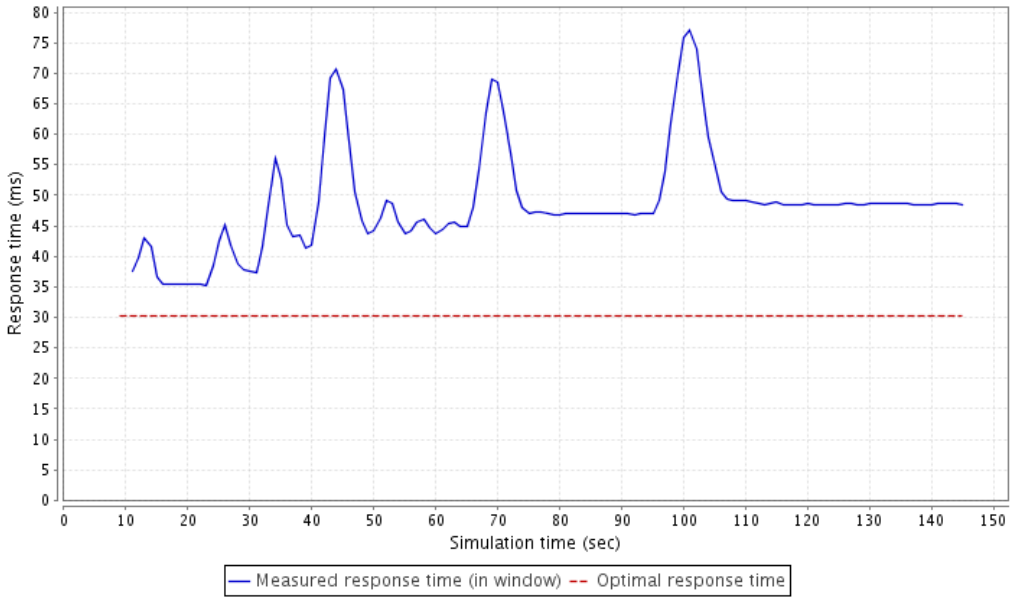


Figura 5.45: Test Constant 10k- Adaptation: Response Time

On this load pattern has been performed also a larger-scale test, involving 10.000 nodes and 20 services. This test has two main scopes: the first is to evaluate the general performances of the system at very large-scale. The second is to prove that in both the cases analyzed (adaptation and no adaptation) the trend is the same.

This test is also a little more challenging for the adaptive algorithm since much more services "fight" asking for resources each others in very close time

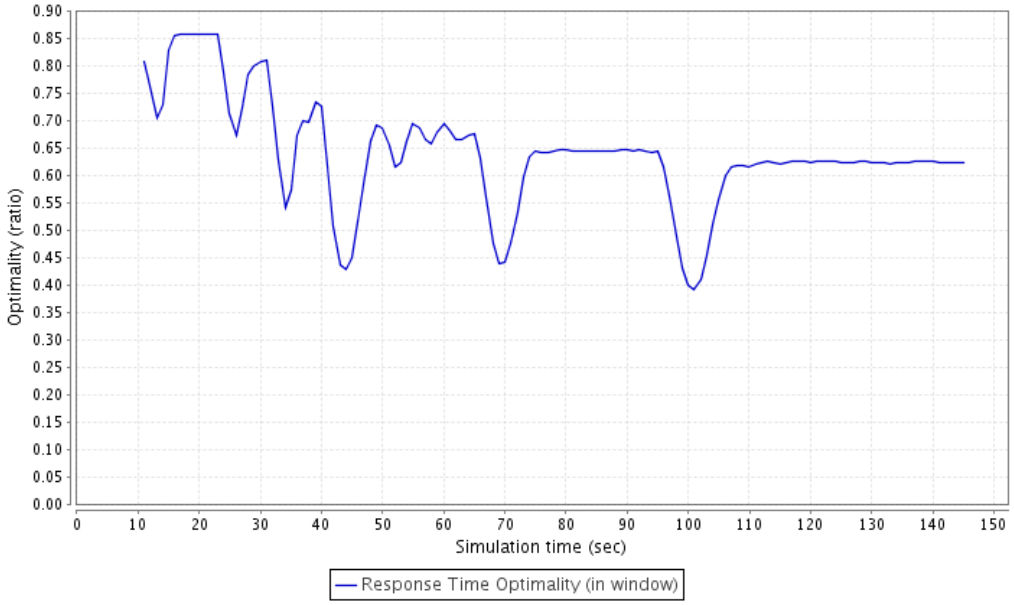


Figura 5.46: Test Constant 10k - Adaptation: Response Time Optimality

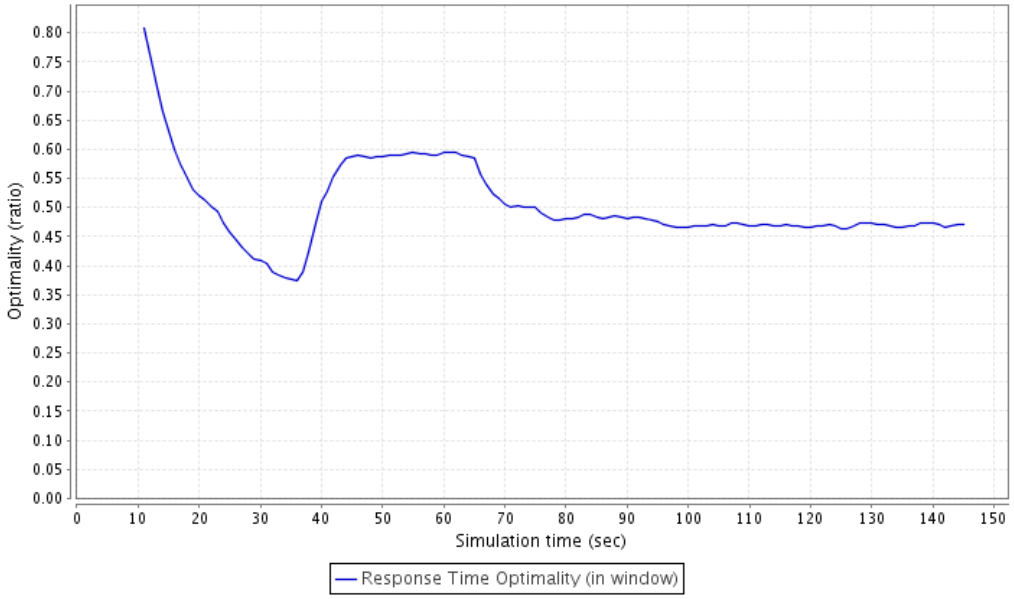


Figura 5.47: Test Constant 10k- No Adaptation: Response Time Optimality

instants. In fact along our experiments this kind of behavior is shown as the most challenging for the algorithm. The load among all the services is shown in Figure 5.42. Despite this the performances improvement is higher than the no Adaptive version.

Figure 5.45 and Figure 5.46 have shown an improved Response Time, thus Response Time Optimality, with respect the no adaptive case (Figure 5.44 Figure 5.47).

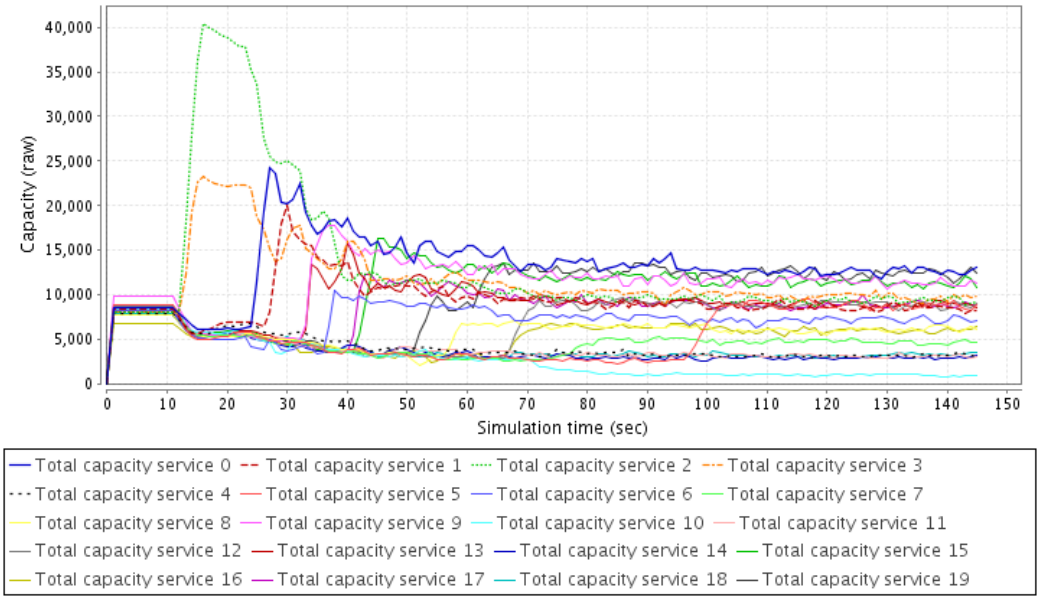


Figura 5.48: Test Constant 10k- Adaptation: Capacity per service

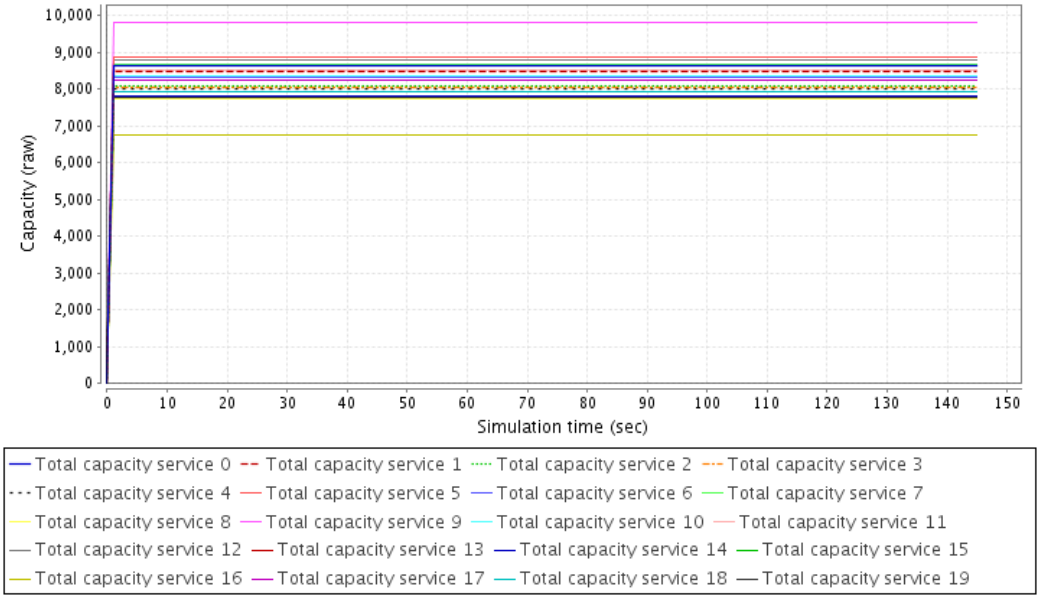


Figura 5.49: Test Constant 10k- No Adaptation: Capacity per service

It is useful also to see how the capacity per service shown in Figure 5.48 converges to a stable assignment of capacity for each service after a first "according phase". For a comparison see Figure 5.49, in which is shown the case in which the capacity remains fixed for each service in case of no adaptation.

Another idea of the service adaptation is given by the load measured among all the services, which converge to the same one with adaptation (Figure 5.43), while is fixed in the static case (Figure 5.42).

5.3 Comparisons and discussion

In this section we discuss the achieved results for MycoCloud. The main aspect is in the improvement given by the elasticity algorithm in every condition. The tests described in the previous Chapter showed how even in cases designed to put in trouble the elasticity algorithm, it showed better performances compared to a static system. In other cases, and in general on almost every test executed the benefits were net. Moreover the elasticity repair for some inefficiency coming from the load-balancer algorithm. As seen in the results provided above the *Response Time* do not reach a desired optimality in a static system, while enabling elasticity among the services the performances become better.

Another important property evaluated in the elasticity algorithm is the level of reactivity (i.e., the ability to quickly re-adapt the assigned capacity to every service with sudden changes in the workload). In this regard, the tests described above were designed on purpose to stress the algorithm. These tests have shown how this property is satisfied even in hard-edge cases, ensuring a fast reaction in every condition.

The main cost to pay with such elastic system is a little increase in the network traffic. This is shown to be higher especially for the topology messages, since every time a node changes its service the topology needs to rebuild itself. This situation can be seen as churn events, in which nodes leave and join the network and the topology needs to reorganize itself.

A relevant factor is that all the reported tests do not consider churn events (i.e., faults). This because the effectiveness of the overlay and load-balancer algorithms against this potential adversarial environment conditions due to dynamism in the network were already tested showing good results in [34, 37]. Regarding the elasticity algorithm we believe that global performances compared a static system in churn scenario would be just a strong remark of this approach. In fact, a system that lose nodes in a static service cannot replace them, while enabling elasticity it can ask capacity from other services readjusting the whole system.

We finally observed that MycoCloud is able to deal also with a relevant number of nodes (10.000 nodes) while maintaining unaltered the quality of achieved results, thus showing a good level of scalability for the technique

itself. The key to the higher scalability of MycoCloud is that a node does not need to know neither the total number of nodes nor the total capacity of the system.

Capitolo 6

Conclusions and Future Works

In this thesis we have presented MycoCloud, a decentralized algorithm for scaling services in a cloud computing context. In particular, we propose a novel algorithm based on a heuristic placement, which, in combination with a robust overlay network among services and a decentralized load balancing technique, provides an effective decentralized solution for improving the performances in the cloud. The solution we have proposed will be suitable in all the situations in which a centralized solution is not feasible, like, for example, when dealing with multiple cloud providers in a federated cloud scenario. We implemented our technique using Protopeer toolkit and validated it with an extensive set of simulations.

The results attested the validity of our approach showing a net improvement with respect to no adaptive systems. We have also seen that the approach also inherits the typical benefits of bio-inspired self-organization, such as the scalability with respect to the number of peers, and to dynamism introduced by service elasticity. Experiments in fact show that MycoCloud scales to a large number of nodes (10.000) while maintaining the same performances.

The main extensions we consider for this system are: (i) consider other parameters in the heuristic placement, (ii) consider the possibility to have additional resources, and (iii) consider the possibility to have node "multi-services" nodes. The former direction would takes in account other characteristics that are not currently used in the selection. Example of additional characteristics to add to the heuristic function may be topological/physical locations of the different peers or commercial costs in case of Cloud providers. The second possible extension focuses on the possibility to have "transparent"resources,

that is resources in idle state which can switch-on a service on request. In this way we would evaluate situations in which the heuristic placement may have other choices for the selection if the algorithm, looking for a possible improvement for the performances. The last extension is to have nodes able to run more then on services at same time and instead of perform an atomic service change, rather would provide a certain amount of capacity. Given the results this looks as a way to reduce the network traffic in case of an high number of service changes, case in which the topology needs to rebuild itself.

Ringraziamenti

Il primo immenso ringraziamento va ai miei genitori. Li ringrazio per aver sempre creduto in me e per avermi sostenuto affichè raggiungessi i miei obbiettivi. Ringrazio poi le mie due sorelle Marilena ed Antonella, che con il loro affetto mi sono sempre state vicine nonostante la notevole distanza.

Un ringraziamento va a tutta la mia famiglia, e in particolare a due dei miei zii. Zii, ma allo stesso tempo padri e amici, Domenico e Donato. Sono da sempre e sono tutt'ora i miei modelli di vita, ed è principalmente grazie a loro che sono quello che sono.

Un doveroso grazie va ai miei amici di una vita: Angelo, Valerio, Giulio, Davide, Marco, Raffaele, Gerardo, Fabio, Domenico e Giancanio, meglio conosciuti come *La ditta*, e Serena, Rossella e Simona. Li ringrazio per avermi regalato i migliori anni della mia vita, e soprattuto per continuare a farlo.

Ringrazio poi la mia "seconda famiglia", ovvero i miei coinquilini ma soprattutto amici: Mario, Ludovica e Noemi, più l'adottivo "zio" Davide. Mario un grandissimo amico con cui condivido praticamente tutto; Ludovica, un'amica sulla quale si può sempre contare per serate sconclusionate (vedi gite notturne in taxi); Noemi, un'amica sempre pronta a farsi ascoltare; e Davide un amico su cui si può far sempre affidamento, soprattutto se si tratta di una birra.

Infine, non posso esimermi dal ringraziare Elisabetta. La ringrazio per avermi dato in più occasioni modo di lavorare su progetti importanti e formativi durante questi due anni al Politecnico. Tra questi in particolare quello svolto negli Stati Uniti, dove ho avuto la fortuna di lavorare con Peppo, Paul e Daniel che ringrazio, che è stata per me un'importante opportunità di crestita professionale e personale.

Donato Lucia

Bibliografia

[1] Apache mina. http://mina.apache.org/, 2004-2009.

[2] Nimbus project. http://www.nimbusproject.org/, September 2011.

[3] Muthitacharoen A., Morris R., Gil T.M., and Chen B. Ivy a read/write peer-to-peer file system. pages 31–44, Boston, MA, 2002. 5th Symposium on Operating Systems Design and Implementation (OSDI '02).

[4] Rowstron A. and Druschel P. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. pages 188–201, Banff, Alberta, Canada, 2001. 18th ACM Symposium on Operating Systems Principles (SOSP '01), Chateau Lake Louise.

[5] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. F. Knight Jr., R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss. Amorphous computing. *Commun. ACM*, 43:74–82, 2000.

[6] Carson ans M. and D. Santay. Nist net - a linux-based network emulation tool. Computer Communication Review, 2004.

[7] Cosmin Arad, Jim Dowling, and Seif Haridi. Building and evaluating p2p systems using the kompics component framework. In *Peer-to-Peer Computing*. IEEE Ninth International Conference, 2009.

[8] E. Bonabeau, M. Dorigo, and G. Theraula. Swarm intelligence from natural to artifical systems. Oxford Press, 1999.

[9] Adam C and Stadler R. A middleware design for large-scale clusters offering multiple services. *IEEE Trans Netw Service Manag*, 3:1–12, 2006.

[10] Nicolò M. Calcavecchia, Bogdan A. Caprarescu, Elisabetta Di Nitto, Daniel J. Dubois, and Dana Petcu. Depas: a decentralized probabilistic algorithm for auto-scaling. *Computing*, 94:701–730, September 2012.

[11] L. De Castro. *Fundamentals of Natural Computing: Basic Concepts, Algorithms, And Applications*. Chapman & Hall/CRC, 2006.

[12] Dabek F., Kaashoek F., D. Karger, R. Morris, and Stoica I. Wide-area cooperative storage with cfs. pages 202–215, Banff, Alberta, Canada, 2001. 18th ACM Symposium on Operating Systems Principles (SOSP '01), Chateau Lake Louise.

[13] N. Forbes. Imitation of life: How biology is inspiring computing. MIT Press, 2004.

[14] M. Franceschelli, A. Giua, and C. Seatzu. Load balancing over heterogeneous networks with gossip-based algorithms. ACC'09, pages 1987–1993. IEEE Press, 2009, Piscataway, NJ, USA, 2009.

[15] G. Kan. Gnutella. *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*. O'Reilly & Associates, March 2001.

[16] K. P. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity. *ACM SIGCOMM 2003*, 2003.

[17] Kubiatowicz J., Bindel D., Chen Y., Eaton P., Geels D., Gummadi R., Rhea S., Weatherspoon H., Weimer W., C. Wells, and Zhao B. Oceanstore: An architecture for global-scale persistent storage. In *Computer Science*, number 2218, pages 190–201, Cambridge, MA, 2000. 9th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS 2000), Springer-Verlag.

[18] M. Jelasity, W. Kowalczyk, and M. Van Steen. Newscast computing. Technical Report IR-CS-006, Vrije Universiteit Amsterdam Department of Computer Science Internal, 2003.

[19] M. Jelasity, A. Montresor, G. Jesi, and S. Voulgaris. Peersim: A peer-to-peer simulator.

[20] L. Jie and H. Kameda. Load balancing problems for multiclass jobs in distributed/parallel computer systems. volume 47, pages 322–332. IEEE Transactions on Computers, March 1998.

[21] Gummadi K.P., Dunn R.J., Saroiu S., Gribble S.D., Levy H.M., and Zahorjan J. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 314–329, Bolton Landing, NY, 2003. ACM Press.

[22] Rizzo L. Dummynet and forward error correction. pages 129–137, New Orleans, LA, 1998. 1998 USENIX Annual Technical Conference.

[23] W. Liu, J. Yu, J. Song, X. Lan, and B. Cao. Erasp: An efficient and robust adaptive superpeer overlay network. 4976:468, 2008. Lecture notes in Computer science.

[24] Jelasity M., Voulgaris S., Guerraoui R., Kermarrec A-M, and van Steen M. Gossip-based peer sampling. *ACM Transactions on Computer System*, 35:1–36, 2007.

[25] A. Montresor. A robust protocol for building superpeer overlay topologies. pages 202–209, Zurich, Switzerland, August 2004. 4th International Conference on Peer-to-Peer Computing.

[26] R. Nagpal. Programmable self-assembly using biologically-inspired multiagent control. *Autonomous Agents and Multiagent Systems*, 2002.

[27] E. Di Nitto, D. J. Dubois, R. Mirandola, F. Saffre, and R. Tateson. Applying self-aggregation to load balancing: experimental results. volume 3rd International Conference on Bio-Inspired Models of Network, Information and Computing Sytems, pages 14:1–14:8, Brussels, Belgium, 2008.

[28] Leads O. Opennebula: the open source toolkit for cloud computing. The 3rd Conference about open source in the data center (OSDC 2011), April 2001.

[29] Doursat R. Programmable architectures that are complex and self-organized: From morphogenesis to engineering. University of Southampton, Winchester, UK, 2008. 1th International Conference on the Simulation and Synthesis of Living Systems (ALIFE XI).

[30] J. Kennedy R. C. Eberhart, Y. Shi. *Swarm Intelligence*. Morgan Kauffmann, 2001.

[31] M. Randles, O. Abu-Rahmeh, P. Johnson, and A. Taleb-Bendiab. Biased random walks on resource network graphs for load balancing. *The Journal of Supercomputing*, 53:138–162, 2010.

[32] R. Ranjan, L. Zhao, X. Wu, A. Liu, A. Quiroz, and M. Parashar. Peer-to-peer cloud provisioning: Service discovery and load-balancing. In *Cloud Computing*, volume 0 of *Computer Communications and Networks*, pages 195–217. Springer London, 2010.

[33] Rhea S., Eaton P., Geels D., Weatherspoon H., Zhao B., and Kubiatowicz J. Pond: the oceanstore prototype. Number 14, pages 202–215, San Francisco, CA, 2003. 2nd USENIX Conference on File and Storage Technologies (FAST '03).

[34] Paul L. Snyder, Rachel Greenstadt, and Giuseppe Valetto. Myconet: A fungi-inspired model for superpeer-based peer-to-peer overlay topologies. San Francisco, California, USA, September 2009. Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2009.

[35] Sharma U, Shenoy P, Sahu S, and Shaikh A. A cost-aware elasticity provisioning system for the cloud. In *IEEE Computer Society*, pages 559–570, Washington, DC, USA, 2011. 31st international conference on distributed computing systems, ICDCS '11.

[36] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. 5th OSDI, December 2002.

[37] Giuseppe Valetto, Paul Snyder, Daniel J. Dubois, Elisabetta Di Nitto, and Nicolò M. Calcavecchia. A self-organized load-balancing algorithm for overlay-based decentralized service networks. pages 168–177. Self-Adaptive and Self-Organizing Systems (SASO), Fifth IEEE International Conference, October 2011.

[38] Galuba W, Aberer K, Despotovic Z, and Kellerer W. Protopeer: a p2p toolkit bridging the gap between simulation and live deployment. pages 60:1–60:9, ICST, Brussels, 2009. 2nd international conference on simulation tools and techniques, Simutools '09.

[39] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. Emulab. In *5th Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, USA, December 9-11 2002.

[40] Stephen Wolfram. *A new kind of science*. Wolfram Media Inc, Champaign, Ilinois, US, United States, 2002.