Decentralized learning for self-adaptive QoS-aware service assembly

Article *in* Future Generation Computer Systems · February 2020

DOI: 10.1016/j.future.2020.02.027

Mirko D'Angelo
Ericsson

Mauro Caporuscio
Linnaeus University

Raffaela Mirandola
Politecnico di Milano

Decentralized Learning for Self-Adaptive QoS-Aware Service Assembly

Mirko D'Angelo[a], Mauro Caporuscio[a], Vincenzo Grassi[b], Raffaela Mirandola[c]

[a] *Linnaeus University, Växjö, Sweden*
[b] *Università di Roma Tor Vergata, Roma, Italy*
[c] *Politecnico di Milano, Milano, Italy*

Abstract

The highly dynamic nature of future computing systems, where applications dynamically emerge as opportunistic aggregation of autonomous and independent resources available at any given time, requires a radical shift in the adopted computing paradigms. Indeed, they should fully reflect the decentralized perspective of the execution environment and consider QoS, scalability and resilience as key objectives. In this context, the everything-as-a-service (XaaS) paradigm, which envisions the creation of new services as an assembly of independent services available within the environment, can greatly help in tackling the challenges of developing future applications. However, in order to be effective, XaaS paradigm requires self-adaptive service assembly solutions able to cope with the unpredictable variability and scalability of the execution environment, the lack of global knowledge, and the QoS requirements of services to be built. We contribute in this direction by designing a fully decentralized and collective self-adaptive service assembly framework whose main features are: (i) self-assembly, i.e., the ability to operate autonomously, (ii) online-learning, i.e., the ability to dynamically learn from experience, (iii) QoS-awareness, i.e., the inclusion of QoS requirements as driving forces for self-assembly, (iv) scalability, i.e., the ability to cope with a large number of services, and (v) resilience, i.e., the ability to maintain the persistence of service delivery when facing unexpected changes (e.g. in the number and/or QoS of services). Simulation experiments show that our solution makes the system able to quickly converge to viable assemblies that improve and maintain over time the social welfare of the system, despite the local perspective of each participating service.

Key words: Service Assembly, Quality of Service, Decentralized Learning, Self-Adaptive Systems

1. Introduction

Future computing environments are envisioned to be populated by myriads of pervasive and connected real world things, which collaborate with each other to offer boundless opportunities to industry and society – e.g., smart home, smart city, intelligent traffic system, and smart power grid.

In this setting, an application can be considered as a network-based system where a large open-ended collection of autonomous and heterogeneous pervasive resources dynamically interact with each other, to provide users with rich functionalities. Indeed, applications dynamically emerge [20] as opportunistic assembly of resources of interest available at any given time. To achieve this vision, key objectives are (*i*) to facilitate the design, implementation, and assembly of resources irrespectively of their specific nature, and (*ii*) to facilitate the discovery and opportunistic assembly of resources of interest.

To this end, software engineering best practices suggest the exploitation of the so called *everything-as-a-service* (XaaS) paradigm, which allow for uniformly representing physical things, hardware resources and software applications as independent and isolated entities offering services. Besides, the XaaS facilitates the creation of new complex services as an assembly of independent services available within the environment [3].

However, to be actually adopted in a future computing environment, the XaaS paradigm requires effective solutions for problems that include: *interoperability* (how to make heterogeneous services

Email addresses: mirko.dangelo@lnu.se (Mirko D'Angelo), mauro.caporuscio@lnu.se (Mauro Caporuscio), vincenzo.grassi@uniroma2.it (Vincenzo Grassi), raffaela.mirandola@polimi.it (Raffaela Mirandola)

able to connect and interact with each other) and *management* (how to build and maintain over time a suitable assembly of services that are collectively able to fulfill a given task). Both problems are not new per se in an XaaS context [3], but are highly exacerbated by the peculiar characteristics of future computing environments: the former by the unprecedented heterogeneity and variety of services to be interconnected, and the latter by the extreme openness, variability and unpredictability of this environment.

In this paper, we focus on *management* aspects, assuming that suitable solutions for interoperability exist. That is, we consider the following problem:

> *how to dynamically build a service assembly and maintain it over time despite the openness, variability, unpredictability and scalability of the execution environment.*

Actually devising an effective solution for this problem in future computing environments requires to tackle several challenges. Indeed, the intrinsic dynamism of this environment, with services joining/leaving the system or changing their behavior, makes each assembly potentially "unstable", which is then required to self-adapt – i.e., to autonomously modify its structure or behavior over time – according to the changing environment. However, the lack of global knowledge, which is difficult to achieve and maintain in a large distributed system of autonomous services, makes hardly usable centralized policies for service assembly and adaptation. Moreover, several applications for future computing environments (e.g., augmented reality, intelligent transportation systems) are likely characterized by quality of service (QoS) requirements concerning attributes such as timeliness or availability, which should thus be considered in the construction of the assembly.

In this respect, the presence of functionally equivalent services, with different values of their QoS attributes, could make nontrivial determining the "best" selection of offered services to be bound to other services requiring them. Indeed, the possibly load-dependent nature of some QoS attributes (e.g., those concerning performance), might rule out simple greedy service selection policies that always select what currently appears as the best service, as they could easily lead to service overloading, and consequent worsening of the system's social welfare.

Finally, dealing with myriads of heterogeneous and autonomous services, as in future computing environments, requires to devise service selection and assembly procedures that scale with increasing system size. Indeed, services should coordinate with each another to decentralize the control [54]. That is, assembly and adaptation must be collective: services shall self-assemble and self-adapt in a way that addresses critical runtime uncertainty while preserving the benefits of the collaborative interdependencies.

1.1. Paper Contribution

To tackle the challenges outlined in the previous section, we claim that the supporting infrastructure for future generation software should fully embrace the inherently distributed and decentralized nature of its environment by: (*i*) departing from centralized solutions, in favor of decentralized and collective mechanisms, and (*ii*) adopting intelligent strategies to enable services to self-assemble according to both their functional and QoS requirements, and to self-adapt to changing operating conditions.

To this end, we propose a fully decentralized and collective self-adaptive service assembly framework, characterized by the following features:

- *Self-assembly*: the ability to operate autonomously, thus relieving for the most part from the need of direct human intervention;
- *Online learning:* the use of a reinforcement learning (RL) approach to make each participating service able to dynamically learn from its experience the most effective assembly rule to be followed, thus overcoming the lack of global knowledge;
- *QoS awareness*: the inclusion of both functional and QoS requirements as driving forces for both the dynamic self-assembly of services and the self-adaptation of a previously defined assembly (if changes occurred in the operating environment make the assembly no longer adequate);
- *Load-dependent QoS*: the explicit consideration of load-dependent QoS attributes, which leads to the definition of a service assembly rule that takes into account the potentially negative impact of service overloading conditions;
- *Scalability*: the ability to deal with a large number of services in different scenarios;

- *Resilience*: the ability to maintain the persistence of service delivery when facing unexpected changes in the execution environment (e.g. in the number and/or QoS of services).

To the best of our knowledge, the proposed framework is the first one able to provide all these features within the general area of works addressing the service selection problem, as discussed in Section 3.

This paper extends preliminary results already presented in [6]. With respect to that paper, main improvements concern: (*i*) a well defined method to take into account load-dependent attributes in service selection rules, (*ii*) a refined definition of the system architecture, (*iii*) a more articulated presentation of the approach and a deeper comparison with related works, and (*iv*) an extensive evaluation of the approach's *efficacy*, *efficiency*, *resilience* and *scalability* in different settings.

In order to evaluate the proposed approach we run a large number of controlled experiments simulating the different scenarios that characterize future computing environments: *openness*, *variability*, *unpredictability* and *scalability*. Simulation experiments show that our solution makes the system able to quickly converge to viable assemblies that improve and maintain over time the quality of the system, despite the local perspective of each participating service. To this end, we formally define two *Social Welfare* indexes used to measure the effectiveness of our approach. In particular, we adopt a two-fold perspective: (*i*) we measure the average quality globally delivered by all services and (*ii*) we measure whether there is an unbalanced distribution among services of this global quality. The replication package is publicly available to researchers interested in replicating and independently verifying the results presented in this paper[1]. Indeed, it includes both simulation models and collected raw data used to evaluate the approach.

The paper is organized as follows. In Section 2 we present a motivating scenario illustrating the type of problem we intend to tackle and the main characteristics required to deal with it. In Section 3 we survey the related works classifying them according to the previously devised requirements. The overview of the proposed approach together with the goals we want to achieve are presented in Section 4. In Section 5 we define the system model, whereas in Section 6 we detail the core elements of our approach. In Section 8 we present experimental results obtained through simulation, while in Section 9 we discuss some threats to validity. Section 10 presents conclusions and hints for future work.

[1] https://github.com/mi-da/Decentralized-Learning-Service-Assembly

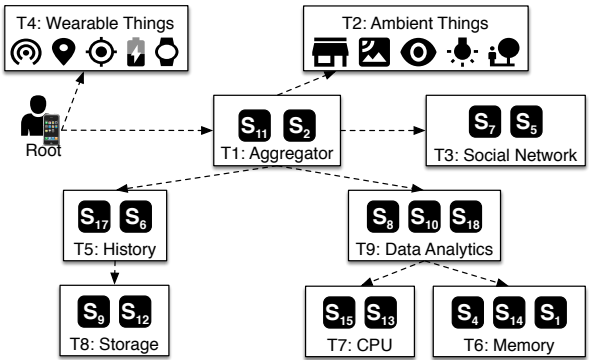


Figure 1: Motivating Scenario – Social Sensing Application

2. Motivating Scenario

As a motivating scenario, let us consider the social sensing application (e.g., inspired by [35]) shown in Figure 1. Its goal is to infer the social situation of a person (attending a meeting, biking, dancing in a dancefloor, etc.) from data coming from different sensors, and to make it available to her friends through social networks. This application can be built from the aggregation of some logically distinct building blocks that likely include: a "*root*" module implementing the application's orchestration logic; a set of different sensors providing data about the person's context (i.e., *Wearable Things* and *Ambient Things*); one or more compute intensive modules that aggregate data from both these sensors to extract relevant features and draw conclusions about the current situation (i.e., *Aggregator*); a *Social Network* module to share the current social situation; a *History* module to save history data for late (offline) processing. Besides, these modules require basic *Computing*, *Storage* and *Memory* resources to fulfill their processing, storage and memory needs, respectively. In an XaaS perspective, each of these software modules and the underlying infrastructure resources can be seen as entities that offer services, and that could require to this end services offered by others.

We can easily figure out a deployment scenario for this application where all the dependencies ex-

pressed by its modules are resolved by services hosted at a single node (typically, some kind of personal device like a smartphone). However, in the envisioned network-based system with autonomous and pervasive resources, several alternatives are likely available. For example, sensor data dependencies can be, at least partially, fulfilled by sensors hosted by other personal devices, or dispersed in the environment.

Performance or energy consumption considerations could suggest to resolve the computing power dependencies of the data analytics modules by processing services offered by surrounding nodes of an edge computing infrastructure [21]. Different offerings could be available to this end, with different characteristics in terms of performance, cost, security. These characteristics could not be stable: for example, a computing service that presently qualifies itself as "good" could actually not reveal to be so, if it is contemporarily targeted by requests coming from other applications operating in the same environment that overload it. Moreover, the availability of services could change (e.g., because of mobility, or independent decisions of their owner); this will require to restructure a previously defined assembly.

If we now imagine a crowd of persons using their own instance of the social sensing application we are considering, each of these instances will ultimately result in a set of services that will need other services to resolve their dependencies. Each such service will be likely interested in experiencing a "good" quality from the services that will be selected to this end. However, the interest of each service should be coordinated with the one of others, to avoid globally unbalanced selections that could actually worsen the delivered quality. Moreover, the selection made should be maintained over time, adapting it to changing conditions, to guarantee the delivery of the services and to fulfill their quality expectations.

To deal efficiently with this type of scenarios we need approaches that are able to fulfill the following requirements:

R1: Decentralization – the approach should be able to deal with large distributed systems of autonomous devices/services where a global centralized knowledge does not exist.

R2: QoS-awareness – the approach should take into account QoS attributes (e.g., availability, reliability and performance).

R2.1: Load-dependency – the approach should take into account that some QoS attributes (e.g., performance) depend on actual load during operation.

R3: Resilience – the approach should be able to deal with dynamic scenarios and maintain the persistence of service delivery when facing unexpected changes (e.g., devices/services join or leave the system, and services change their QoS attributes).

R4: Scalability – the approach should be able to deal with a large number of devices/services.

R5: Online learning – the approach should be able to learn from past experience and adapt the system behavior to fulfill its goals.

3. Related Work

Our decentralized service assembly framework lies within the general area of works addressing the *service selection-composition* problem in a distributed environment [9, 28, 36, 51]. In this respect, we share with most of these works the general assumption that the set of dependencies of a service is known when it is released into its operating environment. This is reasonably true for dependencies concerning infrastructure services like computing power or storage. It holds also for higher level dependencies that have been defined at service design time, or as a result of a previous planning activity that has decomposed some global goal into a set of subgoals delegated to different services that must integrate each other.

Different approaches have been proposed to address the service selection-composition problem. In particular, several papers have formulated it as an optimization problem centrally solved by a dedicated broker, assuming a known in advance set of candidate services (e.g., see [9] and references therein). In this section however, we do not cover the vast amount of literature on QoS-aware service selection and assembly that is not directly related with our work. The interested reader can look at the work of Moghaddam et. al. [36] for a review of optimization-based approaches for service selection, [51] for a survey on service trust and reputation models taking into account multiple QoS metrics, and [28] for a review on QoS-aware service composition approaches using computational intelligence mechanisms.

Hereafter, we review works closer to ours, that is assuming an open and unknown operating environment, and decentralized service discovery and

	[22]	[48]	[20]	[38, 52]	[32]	[8]	[45]	[39, 53]	[18]
R1: Decentralization	✓	✓		✓	✓	✓	✓	✓	✓
R2: QoS-awareness			✓	✓		✓	✓	✓	✓
R2.1: Load-dependency							✓		
R3: Resilience			✓	✓		✓	✓		
R4: Scalability		✓		✓		✓		✓	
R5: Online Learning			✓	✓	✓		✓	✓	

Table 1: Related work classification

selection operations (apart from [20]). In Table 1 we classify some of these works according to the requirements identified in Section 2.

In Georgiadis et al. [22] a dynamic set of agents cooperate to preserve some architectural constraints. Differently from our work, adaptation actions are not taken autonomously, but globally coordinated by means of a totally ordered broadcast that implements a distributed locking scheme. This global coordination mechanism requires explicit interaction among all agents. The resulting overhead thus limits the scalability of the architecture and its possible adoption in pervasive environments. FlashMob [48] overcomes some of the limits of [22]. The work adopts a gossip-based adaptive decentralized self-assembly procedure. However, FlashMob requires that each peer maintains and disseminates global state information consisting of the whole assembly of offered and required services. FlashMob also does not explicitly deal with global QoS goals and load dependent QoS attributes. Referring to Table 1, only **R1** and **R4** are satisfied.

Filho et al. [20] propose an approach for the autonomous assembly of software components. Similarly to us, the authors aim at tackling environment dynamics through the use of online reinforcement learning, but differently from us assume a centralized assembly and learning module (i.e., **R1** is not satisfied). Their work experiments through a software prototype the response time of a particular application, blindly exploring to this end different assemblies. On the contrary, we perform an extensive set of simulations to take into account scalability and mobility aspects. Moreover, we assume a known correlation between the QoS of the components to assemble and the result of the assembly, based on general load-depended QoS model, which drives the selection of services to be assembled.

Two works adopt decentralized learning techniques to satisfy QoS requirements and test the scalability of the approach, but do not deal with load-depended attributes (i.e., **R2.1)** [38, 52]. In Moustafa et. al. [38] the proposed technique employs reinforcement learning in order to address large-scale dynamic service environments. However, load-dependent QoS attributes are not explicitly taken into consideration and the template of the workflow is known a-priori at design-time by the services. Moreover, while the authors of [38] investigate three specific QoS attributes (availability, response time, and reliability), we propose a general load-depended QoS model and show its possible instantiation with different QoS attributes. In [52], Wang et al. present a framework for large-scale adaptive service composition. Their model integrates multi-agent reinforcement learning to achieve adaptation, and game theory to enable agents to work for a common composition task. Similarly to us, their approach aims at adjusting the selection process by using a reinforcement learning technique to improve QoS. However, differently from our approach, their agents are just service executor in the system that should maximize a common goal by learning different collaboration schemes. Actually, our work addresses several scenarios highlighted in their future work. Specifically, in [52] it is stated that the behavior of the designed approaches should be explored when agents may seem competitive if they hog a resource to the detriments of other agents. In this respect, we show how it is possible to improve the global QoS in case of load-dependent QoS attributes. Moreover, in addition to the QoS constraints, [52] envisions to take trust into consideration in view of the existence of junk services and, to this end, to incorporate reputation mechanism into their framework. In this respect, our work already embeds a reputation mechanism, which balances through a trust model the advertised QoS of a service with the actually experienced QoS.

Ben Mahfoudh et. al. [32] propose a decentral-

ized composition model for IoT scenarios based on RL and a bio-inspired coordination model. Like in our framework, the service selection process and the resulting coordination is achieved in a fully decentralized way via RL. The authors experiment with a real IoT deployment of five agents, but the investigation of (load dependent) non-functional requirements guarantees, maintenance, scalability and resilience is left for future work. Referring to Table 1, only **R1** and **R5** are satisfied.

In a previous work [8] we proposed a fully decentralized middleware called GoPrime for the adaptive self-assembly of distributed services. It is based on the use of a gossip protocol to achieve decentralized information dissemination and decision making. With GoPrime it is possible to build and maintain an assembly of services that, besides functional requirements, fulfils also global QoS and structural requirements. Despite taking into account QoS aspect, GoPrime does not cover requirements **R2.1** about load-dependent attributes and **R5** on the exploitation of online learning techniques.

Shaerf et al. [45] study the process of multi-agent reinforcement learning in the context of load balancing in a distributed system, without the use of either central coordination or explicit communication. They consider a system consisting of a certain number of agents using a finite set of resources, each having a time-dependent capacity. This work is the most similar to ours, as it employs a load-dependent model for the QoS-aware service selection procedure. The presented experimental study deals with a relatively small system of 100 agents. Hence, referring again to Table 1, **R4** is not satisfied. The paper considers a scenario with a single type of dependency, and where the agents already know the full set of available resources. On the contrary, we assume that each node does not know in advance the other nodes (and the services they offer) in the environment, but discover them dynamically.

Moustafa and Zhang [39] propose an approach based on multi-objective reinforcement learning to facilitate the QoS-aware service composition problem. In this work, two algorithms are devised to handle different composition scenarios based on user preferences. In [53], an adaptive service composition technique using hierarchical reinforcement learning is proposed. Similarly to our approach, the authors consider multiple and possibly conflicting QoS attributes, and weight them to obtain a single reward scalar value. An interesting contribution of the paper is the integration of automatic task decomposition and Hierarchical Reinforcement Learning (HRL) to address the curse of dimensionality problem. In these two works **R2.1** and **R3** are not satisfied [39, 53].

Fagernes and Couch [18] investigate the impact of altruistic and selfish strategies in a decentralized scenario where agents make independent decision on resource consumption (i.e., use and adjustment). Their study shows that selfish approaches wins when no collected information about the state of the system forms the basis of the decision. This result supports our technique, where agents are designed to exploit local information and are designed with a selfish behavior (emerging in an overall/global system cooperation). Differently from the aforementioned work, where theoretical scenarios are simulated for 10 agents, our work applies extensive simulations with static and dynamic scenarios up to 5000 agents. Moreover, we do not assume equal load on all components. Vice-versa, we learn at runtime in a decentralized way how to assemble a given set of agents, exposing load-dependent attributes, to achieve a global goal. In this sense, despite the similarities, our work tackles a more complex scenario. In this work, only **R1** and **R2** are satisfied.

Finally, our decentralized service assembly framework lies within the general area of works addressing the service selection problem and, to the best of our knowledge, is the first satisfying **R1**-**R5**. In the next section we outline the overview of the approach.

4. Approach Overview

Generalizing the example scenario we have outlined in Section 2, thus abstracting from the specific characteristics of a single application, we envision a pervasive computing system consisting of a set of *Services* that can be connected to each other through a suitable set of *assembly bindings*, connecting required services to offered services. These services include both software implemented services (independent and isolated computational units, each owning its data), and virtualized hardware facilities (e.g., storage and computing resources) offered by a set of network connected *Nodes*. Hence, the considered assembly bindings represent an abstracted and unified view of both functional connections among software services and deployment relationships among software services

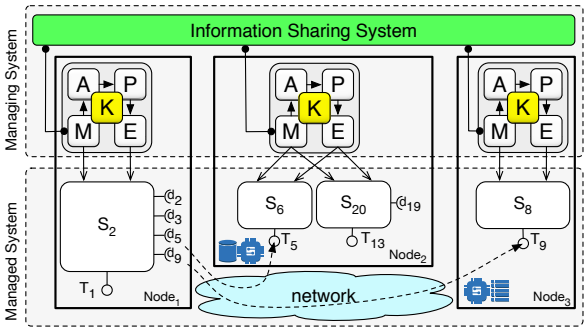


Figure 2: Overview of the Envisioned Scenario Architecture

and the underlying infrastructure virtualized resources. An assembly binding can be established between an offered and a required service only if the former fulfills requirements specified by the latter, concerning in general both the *type* of delivered service (e.g., computing service) and the *context* where it should be delivered (e.g., close to some specific location).

For illustration purposes, the lower part of Figure 2 depicts a possible instance of the the general scenario we are envisioning, consisting of four software implemented services S_2, S_6, S_8 and S_{20}, and three nodes $Node_1$, $Node_2$, and $Node_3$. The resulting assembly shows that S_6, S_{20}, and S_8 are bound to the virtualized resources (depicted as dark blue icons) of their respective hosting nodes to fulfill their basic computing, storage and memory requirements (depicted by the inclusion relationship between nodes and services). Moreover, while S_2 is bound to S_6 and S_8 to resolve some of its functional dependencies (as depicted by dashed arrows), it also has some dependencies not yet resolved.

Our goal is to have systems like this one able to self-manage both the initial assembly construction and its self-adaptation according to possible changes that could occur (e.g., services or nodes being added or removed, or changing their QoS attributes). To this end, each node instantiates a self-adaptation engine architected according to the well-known MAPE-K model, with *Monitor* (M), *Analyze* (A), *Plan* (P) and *Execute* (E) components, plus a *Knowledge* (K) component that maintains relevant information for the other components (e.g., system state, adaptation rules).

The resulting ensemble of distinct MAPE-K loops should thus coordinate its activities to collectively act as a *managing system* for the *managed system* consisting of the nodes and the services deployed on them (as shown in the upper part of Figure 2). In this respect, possible decentralized coordination patterns for multiple MAPE-K loops have been described in [54]. In this paper, we adhere to the *information sharing pattern* presented in [54], where each node self-adapts locally by implementing its own MAPE-K loop, but requires state information from other nodes in the system to drive its self-adaptation activities. Apart from state information sharing, nodes are not required to coordinate other adaptation activities. Hence, this pattern supports autonomous adaptation decisions at each node, and enables scalability thanks to the required loose coordination, limited to state information exchange. For these reasons, it appears well suited for the scenarios we are considering.

Information shared by the *Monitor* components mainly concerns, in our case, the services hosted by each node, and their functional and non-functional properties. According to our decentralization perspective, we assume that the *Monitor* components use to this end some suitable decentralized protocol for service publishing and discovery in pervasive computing environments [44].

In this paper, we focus on the activities of the *Analyze* and *Plan* components. According to the MAPE-K information sharing pattern, their role is to locally select, within a set of candidates built thanks to the monitoring activity, the offered services that can resolve the dependencies of local services, trying to maximize some quality notion expressed in terms of a given set of QoS attributes (see Section 5). However, this egocentric perspective could impair the quality expectations of other nodes, and could ultimately impair the node itself, e.g., if its decisions contribute to overloading a service it is using, thus worsening its performance. Hence, our goal is to devise locally managed Analyze and Plan activities that are anyway able to maintain and improve some kind of *social welfare* for all participating nodes and services.

To tackle this issue in the dynamic scenario we are considering, where services can have multiple dependencies to be resolved, and are characterized by multiple load-dependent QoS attributes, we make nodes learn on their own the service selection rule to be applied, using a *reinforcement learning* approach. According to this approach, the learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them [47]. These features fit well with the sce-

nario we are focusing on, where nodes do not know each other (and the services they offer) in advance, but discover themselves dynamically. In particular, we focus on temporal-difference (TD) learning methods [47], which can learn directly from raw experience without a model of the environment's dynamics, and can be implemented in an on-line, fully incremental fashion. We base the learning on two kinds of knowledge that are incrementally acquired by each node: information about the existence of offered services and their advertised quality, achieved through the monitoring activity based on service discovery, and the direct experience of the services' quality, acquired by each node after actually binding to the selected services. We use this second kind of knowledge to balance, through a *trust model*, the advertised quality with the actually experienced quality, building to this end a two-layer TD-learning model. We detail this model in Section 6.

5. System Model

In this section we introduce the terminology and notation used in the rest of the paper, define the model of the system we are considering, and formally define the performance indexes we will use to measure the effectiveness of our approach. The notation used in this paper is summarized in Appendix A.

5.1. Structural Model

We consider a set $\mathbf{S}$ of distributed services, hosted by nodes of a peer-to-peer (P2P) system (e.g. nodes of an edge cloud architecture), communicating each other through a network. We denote by S, S_i single elements of the set $\mathbf{S}$. A service $S \in \mathbf{S}$ is a tuple $\langle Type, Context, Deps, In_t, Out_t, \mathbf{u} \rangle$, where:

- $S.Type \in \mathbf{T}$ denotes the type of the provided interface (we say that $S.Type$ is the type of S, and denote by T, T_i single elements of the set $\mathbf{T}$). We assume the existence of a function $match_T : \mathbf{T} \times \mathbf{T} \rightarrow [0,1]$ such that $match_T(T_1, T_2) = 0$ if type T_1 does not match type T_2 and $match_T(T_1, T_2) > 0$ if a matching exists according to some suitable matching criterion [42].

- $S.Context \in \mathbf{C}$ denotes the context of the service S. We assume the existence of a function $match_C : \mathbf{C} \times \mathbf{C} \rightarrow [0,1]$ such that $match_C(C_1, C_2) = 0$ if context C_1 does not match context C_2 and $match_C(C_1, C_2) > 0$ if a matching exists according to some suitable matching criterion [7].

- $S.Deps \subseteq \mathbf{D} = \mathbf{T} \times \mathbf{C}$ is the set of required dependencies for S: for each $d = (T, C) \in S.Deps$, we introduce the following function to indicate whether a service S' is able to resolve dependency d:

$$match_S(d, S') = \begin{cases} 1 & \text{if } match_T(T, S'.Type) > 0 \\ & \wedge\ match_C(C, S'.Context) > 0 \\ 0 & otherwise \end{cases}$$

  If $S.Deps = \emptyset$, then S has no dependencies. We assume that $S.Deps$ is fixed for each service and known in advance. Note that the dependency set $S.Deps$ does not contain duplicates, meaning that a service may depend at most once on any specific interface type in a given context.

- $S.In_t \subseteq \mathbf{S}$ is the set of services to which S is bound at time t, to resolve its dependencies.

- $S.Out_t \subseteq \mathbf{S}$ is the set of other services that are bound to S at time t, to resolve one of their dependencies.

- $\vec{\mathbf{q}} \subseteq \mathbb{R}^m$ is a vector of m "internal QoS" attributes (e.g., reliability, cost, response time), which express the quality of the service S, depending only on internal characteristics of S and of the node hosting it. If S has a non-empty set of dependencies, then $\vec{\mathbf{q}}$ gives only a partial view of the QoS of S, which also depends on the QoS of the services used to resolve them. For example, in case of a completion time attribute, the corresponding $\vec{\mathbf{q}}$ entry could represent the execution time in isolation of S internal code on the hosting node, without considering the completion time of called services.

At each time point $t \in \mathbb{N}$ a service is either *fully resolved* or *partially resolved*. A service S is *fully resolved* if either: (i) S has no dependencies ($S.Deps = \emptyset$); or (ii) for all $d \in S.Deps$ there exists a fully resolved service $S' \in S.In_t$ such that $match_S(d, S') = 1$. On the other hand, a *partially resolved* service S has a non-empty list of dependencies, and at least one dependency is either not matched, or is matched by a partially resolved service.

A *service assembly* $\mathbf{A}$ is a directed graph $\mathbf{A} = (\mathbf{S}, \mathbf{E})$, where $\mathbf{E} \subseteq \mathbf{S} \times \mathbf{S}$ is the set of resolved dependencies. Specifically, a directed edge $(S_i, S_j) \in \mathbf{E}$ denotes that S_i is using S_j to resolve one of its dependencies. In general, a given S_i can have multiple simultaneous incoming bindings (i.e., $S_i.In_t$), one for each dependency, and multiple simultaneous outgoing bindings (i.e., $S_i.Out_t$) from other services using S_i to resolve one of their dependencies.

Finally, we point out that our model adheres to the *Service Statelessness* design principle [15], and services do not maintain the interaction state between service invocations – i.e., each request is served in complete isolation, without relying on information from previous requests. Hence, we assume that the state of a given interaction $(S_i, S_j) \in \mathbf{E}$ between S_i (i.e., the consumer) and S_j is kept by S_i, and requests include all information necessary for their processing. Service statelessness enhances (i) decoupling of interacting services, (ii) flexibility of the model, since it allows for easily rearranging the assembly at runtime and, (iii) scalability, by exploiting service caching and replication.

As an example, referring to the social sensing scenario in Figure 1, the Data Analytics services (S_8, S_{10}, and S_{18}) have multiple incoming bindings from the Aggregator services (S_{11} and S_2), and multiple outgoing bindings to the services providing computing (S_{15} and S_{13}) and storage (S_4, S_{14}, and S_1) capabilities.

According to this model, the problem introduced above can be restated as:

> *how to dynamically build and maintain over time a fully resolved assembly of distributed services that are able to collectively fulfill both functional and QoS requirements in case of openness, variability, unpredictability and scalability of the execution environment.*

5.2. Load-dependent QoS Model

As pointed out in the Introduction, we want to explicitly take into account in our QoS-driven assembly construction, the possible load-dependent nature of the QoS attributes we are considering. Indeed, load dependence obviously holds for attributes in the performance domain (e.g., response time), where the load has a negative impact on their value. Besides, it may also hold for other domains like dependability, where increasing load could increase the likelihood of failures [29, 46], or cost, for example in case of cost schemes based on congestion pricing [43].

To this end, we first introduce a load model that allows capturing in a simple yet sufficiently expressive way load dependencies among services in the system. Then, we define a load-dependent QoS model to express in terms of a given set of QoS attributes the QoS delivered by a service S. This QoS depends on both the internal QoS $\vec{\mathbf{q}}$ defined above and the QoS of other services to which S is bound to resolve its dependencies.

For each service $S \in \mathbf{S}$ the associated *load model* is a pair (σ_S, Θ_S), where:

- $\sigma_S \in \mathbb{R}$ ($\sigma_S \geq 0$) models a *traffic source*, represented by the rate of service requests addressed to S from external users;[2]
- $\Theta_S = \{\theta_{S,d} : \mathbb{R} \rightarrow \mathbb{R} | d \in S.Deps\}$ is a set of *transfer functions*, where for each $d \in S.Deps$, $\theta_{S,d}(\lambda)$ is the average rate of service requests sent to dependency $d = (T, C)$ when S is subject to an incoming rate λ of service requests.

We assume that both σ_S and Θ_S are kept updated by a monitoring activity locally performed at the node hosting S.

With these definitions, the load addressed to S is:

$$\lambda_S^{tot} = \sigma_S + \sum_{S' \in S.Out_t} \theta_{S',(S.Type,S.Context)}(\lambda_{S'}^{tot}) \quad (1)$$

As a consequence, for the highest services in the hierarchy of dependencies (root services, i.e., those services with $S.Out_t = \emptyset$) it is $\lambda_S^{tot} = \sigma_S$. Otherwise, for lower services in the hierarchy it is recursively defined by Equation (1).[3]

Let $\vec{\mathbf{Q}}_t \subseteq \mathbb{R}^m$ be a vector of m QoS attributes, which express the QoS of the service S at time t, depending on the internal QoS of S, the QoS of the services it is bound to to resolve its dependencies, and on the load model.

Given these definitions, the QoS for a service S

[2] By external users we mean entities (human users, other services/devices) not included in $\mathbf{S}$. σ_S=0 means that S is possibly accessed only by other services in $\mathbf{S}$.

[3] Lower level services include virtualized hardware resources.

is defined as follows:

$$\vec{\mathbf{Q}}_t(S) = \begin{cases} \mathbf{L}(\vec{\mathbf{q}}(S), \lambda_S^{tot}) & \text{if } S.Deps = \emptyset \\ \perp & \text{if } S \text{ is partially resolved} \\ \mathbf{C}\left(\mathbf{L}(\vec{\mathbf{q}}(S), \lambda_S^{tot}), \vec{\mathbf{Q}}_t(S_1), \ldots, \vec{\mathbf{Q}}_t(S_k)\right) & \\ & \text{if } S \text{ fully resolved,} \\ & \text{with } S.In_t = \{S_1, \ldots, S_k\} \end{cases} \quad (2)$$

In Equation (2) if S has no dependencies ($S.Deps = \emptyset$), then S is by definition fully resolved, and $\vec{\mathbf{Q}}_t(S)$ is calculated by means of a suitable quality function $\mathbf{L} : \mathbb{R}^m \times \mathbb{R} \rightarrow \mathbb{R}^m$, which combines the internal QoS $\vec{\mathbf{q}}(S)$ and the load λ_S^{tot} addressed to S. Instead, if S has a nonempty set of dependencies ($S.Deps \neq \emptyset$) and is not fully resolved, $\vec{\mathbf{Q}}_t(S)$ is set to $\perp$, i.e., the special value that is guaranteed to be "worse" than the QoS of any fully resolved instance of S. Finally, if S has a nonempty set of dependencies and is fully resolved, $\vec{\mathbf{Q}}_t(S)$ is computed using a function $\mathbf{C} : \mathbb{R}^{(1+|S.In_t|)m} \rightarrow \mathbb{R}^m$, which combines the quality function of S (i.e., $\mathbf{L}(\vec{\mathbf{q}}(S), \lambda_S^{tot})$) with the QoS of all S dependencies.

Appendix B provides some examples showing how the general Equation (2) can be instantiated for specific QoS attributes, namely *execution time*, *reliability* and *cost*.

5.3. Social Welfare Indexes

In this section we formally define the indexes, based on the model defined above, that we will use to measure the effectiveness of our approach with respect to its ability in achieving a good local and social welfare. In particular, with regard to the latter, we adopt a two-fold perspective. On the one side, we measure the average QoS delivered by all services. On the other side, we measure whether there is an unbalanced distribution among services of this global QoS. The adopted measures are defined as follows.

For each service $S \in \mathbf{S}$, the vector $\vec{\mathbf{Q}}_t(S)$ details the QoS delivered by S in terms of a set of distinct QoS attributes. To facilitate dealing with multiple and possibly conflicting QoS attributes, we transform $\vec{\mathbf{Q}}_t(S)$ into a single scalar value, using the Simple Additive Weighting (SAW) technique [56]. According to SAW, we define the QoS of a service S as a weighted sum of its normalized QoS, as follows:

$$Q_t(S) = \sum_{i=1}^{m} w_i \frac{Q_{i,t}(S) - V_i^{min}}{V_i^{max} - V_i^{min}} \quad (3)$$

where $Q_{i,t}(S)$ denotes the i-th entry of $\vec{\mathbf{Q}}_t(S)$, V_i^{max} and V_i^{min} denote, respectively, the maximum and minimum value of $Q_{i,t}$, and $w_i \geq 0$, $\sum_{i=1}^{m} w_i = 1$, are weights for the different QoS attributes expressing their relative importance. In our framework, $Q_t(S)$ is the measure that each node egocentrically tries to maximize.

Now, let $\mathbf{S}_t^{full} \subseteq \mathbf{S}$ be the set of fully resolved services at time t. As a first measure of the achieved social welfare, we define the *Average QoS* delivered by all services in $\mathbf{S}_t^{full}$:

$$\xi_t = \frac{1}{|\mathbf{S}_t^{full}|} \sum_{S \in \mathbf{S}_t^{full}} Q_t(S) \quad (4)$$

However, ξ_t as defined in Equation (4) does not allow to capture to what extent all involved services fairly share the average QoS. To this end, as a second measure of the achieved social welfare, we introduce the following *fairness* measure based on the *Jain's fairness index* [27] to measure how uniform is the QoS delivered by all the participating services:

$$\zeta_t = \frac{(\sum_{S \in \mathbf{S}_t^{full}} Q_t(S))^2}{|\mathbf{S}_t^{full}| \sum_{S \in \mathbf{S}_t^{full}} Q_t(S)^2} \quad (5)$$

The value of this fairness index ranges from $\frac{1}{|\mathbf{S}_t^{full}|}$ (worst case) to 1 (best case), and it is maximum when all services deliver the same QoS level $Q_t(S)$. In general, this index penalizes situations where the QoS delivered by different services is highly unbalanced. Hence, by using ζ_t, we intend to reward assemblies that result in a fair share of the overall QoS measured by ξ_t. We point out that, in our load-dependent setting where increasing load tends to worsen the QoS attributes, the more uniform is the global QoS, the more uniform the load distribution tends to be.

6. System Operations

Figure 3 details the main elements of the envisioned approach. As shown in the figure, each service (e.g., S_1) exposes its dependencies through an `invoke` interface and provides a `probe` interface to support the monitoring activities. Besides, the MAPE-K activities are carried out at each node by three different components implementing, respectively, monitoring and information sharing, analysis

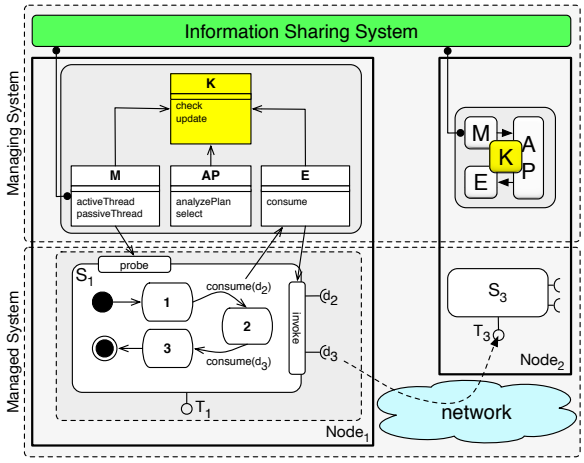


Figure 3: Detailed System Architecture

and planning, and the execute activity (this latter activity through the *consume* function).

The implementation of monitoring and information sharing is briefly sketched in §6.1, while the implementation of RL-based analysis and planning, which constitutes the core of our approach, is described in §6.2.

6.1. Monitoring and Information Sharing

The information sharing pattern, implemented by the monitoring activity (M), is based on periodic *advertisement* activity of offered services (on behalf of nodes hosting them).

Such advertisement activity is implemented according to a classical advertisement schema [34], which includes two concurrent threads that communicate each other (see Figure 3): an *activeThread* advertises information concerning specific service, and a *passiveThread* reacts to messages received from the Information Sharing System. In particular, for each locally hosted service S_i, the *activeThread* reads from the knowledge base (K) monitored information concerning the QoS delivered by S_i, and advertises a message $\mathbf{m} = \langle S_i \rangle$ containing this information, with type equal to $S_i.Type$. This may happen either proactively, e.g., every Δt time units (periodic advertising scheme), or reactively, e.g., when changes in the value of monitored variables are detected, which are outside predefined tolerance windows.

On the other hand, upon receiving a new message $\mathbf{m} = \langle S \rangle$ the *passiveThread* invokes a function *update* provided by the K component, which is in charge of updating the knowledge with the received information.

We implement this collaboration scheme through a Publish/Subscribe (P/S) mechanism [17]. In particular, in Section 8 we experiment with two different P/S architectures: a centralized broker overlay and a P2P gossip-based unstructured overlay.

6.2. TD-learning based Analysis and Planning

The analysis and planning activities (AP in Figure 3) are locally implemented at each node, as described by Algorithm 1. The goal of these activities is to (i) analyze the information kept by the knowledge K (i.e., the set of service candidates $Known$), and (ii) select the services of interest that resolve the dependencies of local services (i.e., $Hosted$).

Algorithm 1 consists of a thread that actively analyzes, every Δ_t time units, the knowledge K (i.e., the set of service candidates $Known$). Whenever the analysis performed by the $\textsc{Check}_K()$ function notices a variation in K (line 4), then a new plan is required by calling the $\textsc{Select}()$ function that implements the P activity (line 7). In this section we focus on the latter, and omit details about $\textsc{Check}_K()$.

$\textsc{Select}()$ implements a selection rule that, among the set of candidate services contained in K, properly chooses the services of interest that resolve the dependencies of local services (i.e., $Hosted$), driven to this end by the (egocentric) goal of maximizing their QoS $Q_t(S)$, as defined by expression 3 in Section 5.3. The definition of the function $H_t()$ used to this end (lines 10 and 12) is discussed in the following (see expression 11 below for its final formal definition).

In our approach, $\textsc{Select}()$ implements a TD-learning method [47] that calculates, based on historical data, a *value function* expressing how good a particular action is in a given situation. TD methods are well suited in our context, since they can learn from raw experience, without relying on any predefined model of the environment. Indeed, the variability is faced on-line, in a fully incremental fashion. The general formulation of a TD method is:

$$E_\eta \leftarrow E_{\eta-1} + \alpha[R_\eta - E_{\eta-1}] \qquad (6)$$

where E_η is the estimated value function at learning-step η, $\alpha \in (0, 1]$ is the learning-rate parameter, R_η is the reward obtained by taking the action, and $E_{\eta-1}$ is the value function calculated at

Algorithm 1 Analysis and planning

```
1: function AnalyzePlan
2:     loop
3:         Wait Δt
4:         b ← Check_K()
5:         if (b = true) then
6:             for all (S_i ∈ Hosted) do
7:                 Select(S_i)

8: function Select(inout S ∈ Hosted)
9:     for all (d ∈ S.Deps) do
10:        m ← arg max_j{H_t(S_j) | S_j ∈ Known ∧ match_S(d, S_j) = 1}
11:        if (∃S_k ∈ S.In_t | match_S(S_k, d) = 1) then
12:            if H_t(S_k) < H_t(S_m) then
13:                S.In_t ← S.In_t \ {S_k} ∪ {S_m}
14:        else
15:            S.In_t ← S.In_t ∪ {S_m}
```

the previous step – i.e., the *historical data*. In simple incremental averaging estimation methods [47], the learning-rate parameter α changes at every learning-step and is calculated as $1/k$, where k is the number of accumulated rewards at learning-step η. Its rationale is to increasingly give more weight to the accumulated experience. However, we will depart from this definition of the learning-rate α to cope with the foreseeable non stationarity of the considered environment. It is worth noticing that every learning-step η_i occurs at specific time t_{η_i} such that $\forall \eta_i, \eta_j : \eta_i \leq \eta_j \iff t_{\eta_i} \leq t_{\eta_j}$ (see Figure 4).

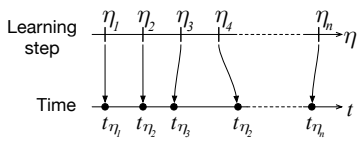


Figure 4: From learning steps to time

The proposed service selection rule implements a *Two-layer* Hierarchical Reinforcement Learning (2HRL) [16] technique, which considers both locally monitored data, and data advertised by the monitoring activities (M) of other nodes.

In the following, we first describe the role of each of the two layers in our approach, and then discuss in detail the Select function implementation (Algorithm 1).

First layer: learning from local data. The first layer aims at learning the QoS of candidate services in *Known* by relying on direct experience only, without considering the information advertised by other nodes. Let $Q^R_\eta(S_j)$ be the QoS observed while interacting with a given S_j at step η. This value is used to predict the QoS $Q^E_{\eta+1}(S_j)$ expected from the same S_j at the next step (i.e., $\eta + 1$). Specifically, the planning activity P periodically checks whether the knowledge K has changed. If there is a variation, then P calculates for all $S_j \in S.In_t$, the expected QoS value $Q^E_{t_\eta}(S_j)$ by instantiating for this specific goal the general Equation (6):

$$Q^E_{t_\eta}(S_j) = Q^E_{t_{\eta-1}}(S_j) + \alpha_j[Q^R_{t_\eta}(S_j) - Q^E_{t_{\eta-1}}(S_j)] \quad (7)$$

where $Q^E_{t_\eta}(S_j)$ is the estimated QoS (i.e., value function) at time t_η, $\alpha_j \in (0, 1]$ is the learning rate, $Q^R_{t_\eta}(S_j)$ is the QoS (i.e., the reward) obtained by directly interacting with S_j at time t_η, and $Q^E_{t_{\eta-1}}(S_j)$ is the QoS estimated at the previous step – i.e., the historical data. $Q^R_{t_\eta}(S_j)$ is calculated based on expression 3, using the directly observed values for the considered QoS attributes.

The learning-rate parameter α_j in Equation (7) is calculated as follows:

$$\alpha_j = \frac{1}{\hat{T}(S_j) \mod z} \quad (8)$$

where $\hat{T}(S_j)$ is the number of times that S_j has been invoked and z is a *learning window*, i.e., a fixed time-window of size z. The rationale for this definition is to avoid giving too much weight to old historical data, which would be inappropriate in a highly dynamic environment, but at the same time retaining to some extent the idea of giving increasingly more weight to the accumulated experience. To this end, Equation (8) splits in practice a single non-stationary problem into a set of smaller stationary problems, where the averaging method suggested in literature is applied. We note that another way proposed in the literature to deal with the problem of properly defining the learning-rate parameter α_j in a non stationary environment, is to use a constant step-size parameter $\alpha_j = \overline{\alpha}$ to be defined at design-time [47]. However, setting this parameter is difficult for the application developer, since it is not correlated with the timing of the learning process.

Second layer: learning from advertised data. The second layer aims at integrating into the learning process of each node the information remotely monitored and advertised by other nodes.

Understanding how much a node can trust the received data is crucial for selecting the services $S_j \in Known$ that best fit the goal of maximizing the expected QoS. To this end, let $\mathbf{m}_{\overline{t}}$ be a message received at time $\overline{t} < t_\eta$, and let $Q_{\overline{t}}(S_j)$ be the advertised QoS for each $S_j \in \mathbf{m}_{\overline{t}}$, i.e., $Q_{\overline{t}}(S_j)$ is the last QoS value known for S_j. $Q_{\overline{t}}(S_j)$ is calculated based on expression 3, using the values of the QoS attributes advertised by each service and received through the service discovery infrastructure.

We estimate the *level of trust* $\tau_{t_\eta}(S_j)$ of the QoS advertised by S_j by instantiating as follows the general Equation (6):

$$\tau_{t_\eta}(S_j) = \tau_{t_{\eta-1}}(S_j) + \alpha_j[F_{t_\eta}(S_j) - \tau_{t_{\eta-1}}(S_j)] \quad (9)$$

where $\tau_{t_\eta}(S_j)$ is the estimated *level of trust* (i.e., value function) at time t_η, $\alpha_j \in (0, 1]$ (see Equation 8) is the learning rate calculated within the *learning-window*, and $F_{t_\eta}(S_j)$ is the reward measuring how much accurate is the data received about S_j. Specifically, the *accuracy* $F_{t_\eta}(S_j)$ of the advertised S_j QoS at time $\overline{t}$ is defined as:

$$F_{t_\eta}(S_j) = 1 - \frac{|Q^R_{t_\eta}(S_j) - Q_{\overline{t}}(S_j)|}{max(Q^R_{t_\eta}(S_j), Q_{\overline{t}}(S_j))} \quad (10)$$

Thus, $F_{t_\eta}(S_j) \in [0, 1]$, and the rationale for its definition is that the closer $Q_{\overline{t}}(S_j))$ is to $Q^R_{t_\eta}(S_j)$, the higher is its accuracy.

Finally, the two learning layers are combined in a new function H that, given a service $S_j \in Known$, expresses its expected QoS at time t_η:

$$H_{t_\eta}(S_j) = \tau_{t_\eta}(S_j) \cdot Q_{\overline{t}}(S_j) + (1 - \tau_{t_\eta}(S_j)) \cdot Q^E_{t_\eta}(S_j) \quad (11)$$

The rationale for the $H_{t_\eta}(S_j)$ definition is that, if trust is high (i.e., $\tau_{t_\eta}(S_j) \approxeq 1$) then advertised data $Q_{\overline{t}}(S_j)$ is considered highly relevant in the evaluation of S_j. Viceversa, whenever the trust in advertised data is low (i.e., $\tau_{t_\eta}(S_j) \approxeq 0$), then local experience $Q^E_{t_\eta}(S_j)$ is considered more relevant than advertised data for evaluating S_j.

As an example, referring to the social sensing scenario of Section 2, a low-trusted edge service providing computing capabilities (i.e., CPU) could lead the surrounding wearable devices to prefer to locally process the information or to look for alternative services. On the other side, services advertising (and providing) a good level of quality, will be most likely selected. This mechanism has the additional benefit to discourage attacks aimed at compromising the decentralized system. In fact, a potential malicious service joining the environment would be immediately untrusted for declaring a potential good level of quality that at the end would not be provided.

Algorithm 2 Execute

```
1: function CONSUME(in d ∈ D)
2:     S_j ← get_S{S_j ∈ S.In_t | match_S(d, S_j) = 1}
3:     ⟨Q^R(S_j), res⟩ = invoke(S_j)
4:     return res
```

The function SELECT (see Algorithm 1) shows how the 2HRL technique is used to build, for a given $S \in Hosted$, the set of bindings $S.In_t$. SELECT determines, for each dependency $d \in S.Deps$, what is the service $S_m \in Known$ that matches the dependency d and achieves the maximum value of H_t (line 10). Then, if dependency d is already bound to a service S_k (i.e. $S_k \in S.In_t$) with lower value of H_t, S_m replaces S_k (line 13). Otherwise, S_m is directly bound to dependency d (i.e. added to $S.In_t$) (line 15). In this way, the function selects the most rewarding service to resolve d, according to what it dynamically and continuously learns thanks to the 2HRL technique. We point out that the decision taken by each node through the SELECT function is based only on local knowledge (set $Known$ and function $H_t()$). In the experiments of Section 8 we will show that, despite this, the resulting assembly guarantees good levels of social welfare as measured by the indexes ξ_t and ζ_t defined in Section 5.3.

Finally, once the set of bindings $S.In_t$ is built, Algorithm 2 shows how the services are consumed. In particular, whenever a given hosted service S needs to invoke one of its dependencies $d \in S.Deps$, it delegates the Execute activity (see Figure 3), which in turn consumes the service $S_j \in S.In_t$ that actually solves the dependence, and returns the obtained result res to S. Further, the QoS value, i.e., $Q^R_{t_\eta}(S_j)$, obtained by the interaction with S_j at time t_η (line 3), is stored into K and used by the planning activity to predict the QoS $Q^E_{t_{\eta+1}}(S_j)$ expected from the same S_j at the next step.

7. Design Rationale

In the following we discuss some of the choices we made that represent the rationale of our approach.

Learning Techniques. Reinforcement learning is a particular machine learning method where agents evaluate their actions with respect to the specific reward obtained when performing them [47]. RL methods can be classified as *model-based* or *model-free*. Model-based RL algorithms (e.g., [30]) make use of a predefined model of the system to enhance agents exploration. Model-based RL methods are well suited in situations where learning is centralized and/or agents have a global view of the system. On the other hand, model-free methods, like TD-learning, are able to learn without relying on any predefined model of the environment. This improves scalability and make model-free methods the most appropriate for dealing with dynamic environments with scattered system knowledge [24, 13]. Indeed, in TD-learning methods agents are able to incrementally learn the policies and the value functions through single actions and the observed experience [47], as they interact with the environment. Further classification can be made between RL algorithms that learn *on-policy* and those that learn *off-policy*. On-policy methods (e.g., Q-learning) learn the optimal actions using the absolute greedy policy and behaves using other policies (e.g., ϵ-greedy). Whereas, off-policy methods (e.g., SARSA) learn the optimal actions and behaves using the same policy. Finally, some RL methods work in a *continuous* action space (e.g., Deep Deterministic Policy Gradients), whereas others (e.g., Q-learning, SARSA) observe and act in a *discrete* observation/action space.

The specific choice of exploiting a TD-reinforcement learning strategy for implementing the analysis and planning has been driven by the set of requirements **R1-R5**, as identified in Section 2. Our 2HRL algorithm can be classified as an off-policy method similar to Q-learning. Indeed, it applies a greedy policy and acts in a discrete space. The rationale behind our choice is that, differently from on-policy methods, we aim at promptly trigger reward while exploring the environment to allow the resulting assembly to quickly react to environmental changes. However, departing from Q-learning we adopt a hierarchical learning structure, which aims at learning both the future action-reward and the trust of a particular service. The benefit of our approach with respect to a single-layer reinforcement strategy is shown in the experimental evaluation (see Section 8).

Furthermore, the proposed service assembly approach belongs to the area of multi-agent reinforcement learning. In multi-agent systems with multiple intelligent decision-makers, learning approaches are categorized as cooperative or non-cooperative [25]. Cooperative methods rely on the external enforcement of cooperative behavior among entities while, in non-cooperative methods, entities make decisions independently and everyone wants to pursue greater benefits even by harming the others. There are two major categories of cooperative learning approaches: *team learning* and *concurrent learning*. In team learning, a single learner searches for behaviors for the entire team of agents. Such approaches may have scalability problems as the number of agents is increased. On the other side, concurrent learning uses multiple concurrent learning processes (e.g., one per agent) to reduce the joint space problem by projecting it into N separate subproblems [41]. According to this taxonomy, our approach lies in the cooperative concurrent learning category. In particular, the framework establishes cooperative behavior among agents without any prior-knowledge or mediations. In fact, similarly to other approaches [13], the emergent behavior of the system is the result of the agents' unaware collaboration.

Exploration vs. Exploitation. One of the challenges that characterize reinforcement learning with respect to other kinds of learning is the trade-off between exploration and exploitation [47]. Our system adopts a QoS-based service discovery mechanism, hence the exploration phase is informed rather than blind as in many cases [45, 52]. We ensure the exploitation of good services by adopting the greedy selection rule implemented in Algorithm 1. In fact, as shown in Equation (11), our selection rule selects the service with the best value of the function H. With respect to exploration, we do not use a randomized strategy (e.g., ϵ-greedy) because we guarantee anyway an exploration quota thanks to Equation (8). Indeed, since the level of trust of a service is calculated by using the learning window strategy, the reliance of our TD-learning method in a service is constantly re-evaluated and the expected QoS of a service adjusted accordingly.

Stability, Convergence, Adaptation and Scalability. Other important issues to address in multi-agent reinforcement learning environments are stability, convergence, adaptation and scalability. Stability means the convergence to a stationary policy, whereas adaptation ensures that performance is

maintained or improved as the other agents are changing their policies [4]. Convergence to equilibria is a basic stability requirement [23, 26]. Reinforcement learning methods for multiple independent agents (as in our case) do not guarantee convergence to the optimal joint action in scenarios where miscoordination is associated with high penalties [31]. In this paper, we do not address a formal investigation of the convergence (or not) of our approach to some optimal or sub-optimal solution. However, the extensive experimentation presented in Section 8 shows that our approach makes the system able to converge to a good and stable solution. Moreover, regarding adaptation, the experiments show that our approach makes the system able to self-manage the initial assembly of services and to dynamically self-adapt to variations.

Finally, departing from centralized approaches, where Analysis and Plan are usually performed by a single entity, our approach spreads such activities through all the participating nodes (see Figure 3). Hence, while in a centralized approach the analysis and planning should take into account all services available in the environment and calculate their expected QoS, in a decentralized approach each service calculates the expected QoS only for those services it is bound to at time t. In this way, our decentralized planning strategy improves the scalability of the system, by delegating to each service in the environment the computation of the expected QoS. It is worth noticing that when the number of dependencies of each service grows, the planning activity could still be required to perform many updates. In this case an interesting direction to tackle this state-explosion problem could be to adopt a clustering technique grouping similar services inside the same cluster (e.g., time-series clustering [1]) and calculate the expected QoS value only for each identified cluster.

8. Experimental Evaluation

In this section we present an extensive set of simulation experiments to assess the effectiveness of our approach. The experiments aim at evaluating the performance and scalability properties of our approach in both static and dynamic scenarios. To this end, we implemented a large-scale simulation model for the PeerSim simulator [37]. PeerSim is a free Java package designed to efficiently simulate peer-to-peer protocols, which provides a cycle-based engine implementing a time-stepped simulation model. The cycle-based engine is well suited to evaluate peer-to-peer protocols, where the most important metric is the convergence speed measured as the number of rounds (message exchanges) that are needed to reach a desired configuration. Such a performance metric (number of interactions) has the advantage of being independent of the details of the underlying hardware and network infrastructure.

8.1. Experiment Settings

The simulation model implements the monitoring and information sharing activities outlined in Section 6.1 by relying on the P/S interaction paradigm [17]. P/S architectures allows scalable and flexible diffusion of information and has been proven to be effective in pervasive computing environments [44].

From an implementation viewpoint, the P/S model offers the routing and matching functionalities through an overlay infrastructure organized as a network of peer brokers. With this organization, participants connect to one broker that acts as their proxy to the P/S middleware [11, 12, 40]. As an extreme case, there could be a single centralized broker that stores and manages all the subscriptions and acts as the unique data dispatching component. On the other side, in fully distributed P2P architectures, events flow directly from publishers to subscribers with no intermediary brokers, and all the matching and routing responsibilities are carried out by participants themselves. Solutions of this type can be based on a structured [33] or unstructured overlay [19]. We consider in our experiments two different P/S event service architectures, one based on a central broker and the other one based on a P2P gossip-based unstructured overlay, as they represent two different extreme solutions for the implementation of the P/S event service.

As an example, central-broker solutions could be adopted when the social sensing application we are considering as motivating example is deployed in environments managed by some institution (e.g., schools, train stations, etc..). In this case, authorities could provide a local fog server managing access control and event service functionalities for the participating entities. On the other side, unstructured solutions could be more appropriate in fully decentralized scenarios characterized by high mobility and variability (e.g., runners during a marathon race).

Without loss of generality, we focus only on functional dependencies of the services and do not take into account their possible context dependencies. That is, we consider a system with N services and NUM_INT different interface types $\mathbf{T} = \{T_1, \ldots, T_{\text{NUM\_INT}}\}$.

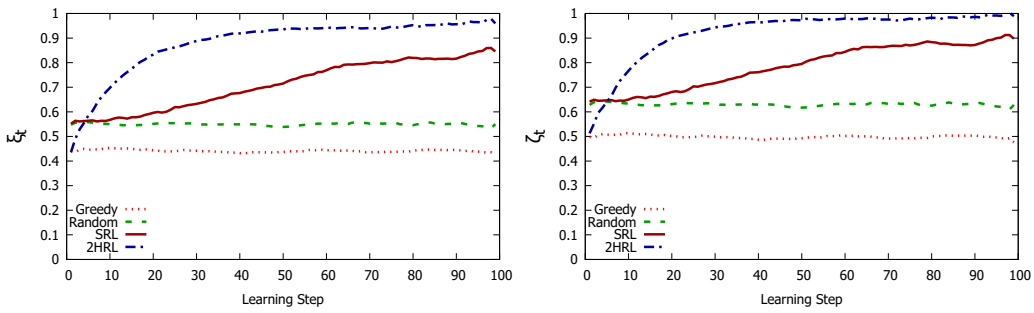
Since our experimentation does not aim at measuring the impact of multiple services on single nodes, without loss of generality we assume that each node hosts a single service. Hence, the number of nodes inside the system is equal to the number of services, i.e., N. We create $\lfloor N/\text{NUM\_INT} \rfloor$ services of each type and, for each service S we randomly set the number of its dependencies. To avoid loops in the dependency graph, we allow a service S to only depend on services of type strictly greater than $S.Type$. Therefore, for each service S we initialize the dependency set $S.Deps$ as a random subset of $\{S.Type+1, S.Type+2, ..., \text{NUM\_INT}\}$. Note that, according to this rule, services of type $T_{\text{NUM\_INT}}$ have no dependencies. In the experimentation, we define a probabilistic attachment for every service in such a subset with probability p=0.7. Finally, we assume that the load-dependent quality function $L(q(S), \lambda_S^{tot})$ (see Sec. 5) of each service S is defined such that it returns values in the range $(0, 1]$. In particular, for each service S, this function is generated as a randomly monotonic decreasing function. The *quality* $Q_t(S)$ is defined such that it returns values in the range $(0, 1]$ (see Equation (3)). Furthermore, other parameters of our 2HRL approach are set as follows: (i) the learning-window parameter z is set to 5, and (ii) for all $S_i \in \mathbf{S}$ the initial trust $\tau_0(S_i)$ is set to 0.95. For the learning window, we have experimentally selected the value $z = 5$ as a good trade-off between the weight to be given to historical data and the high dynamicity of the environment that makes obsolete these data. Ideally, knowing in advance all the dynamics of the system (e.g., nodes leaving or changing their quality attributes) could lead to the optimal selection of this learning parameter as it is related to the degree of non-stationarity of the environment. The rationale for the high initial trust, set to 95%, is to give each service the chance to behave according to the quality it declares. This level is anyway constantly re-estimated at every learning step through Equation (9).

As remarked in Section 5.3 we assess the effectiveness of our approach by evaluating to what extent the 2HRL service selection rule we adopt has a positive impact on the social welfare, notwithstanding its local perspective. The social welfare indexes we use to this end are the *average quality* ξ_t, and the *fairness* ζ_t, defined in Section 5.3. Both ξ_t and ζ_t are higher-is-better metrics whose upper-bound, with our parameters setting, is 1. All experiments, apart from the scalability with respect to the number of nodes, are run by considering 100 learning steps and results are computed by taking the average of 50 independent simulations. Results are plotted using the simple moving average technique to avoid rumors and better show the trend of the curves.

8.2. Simulation Results

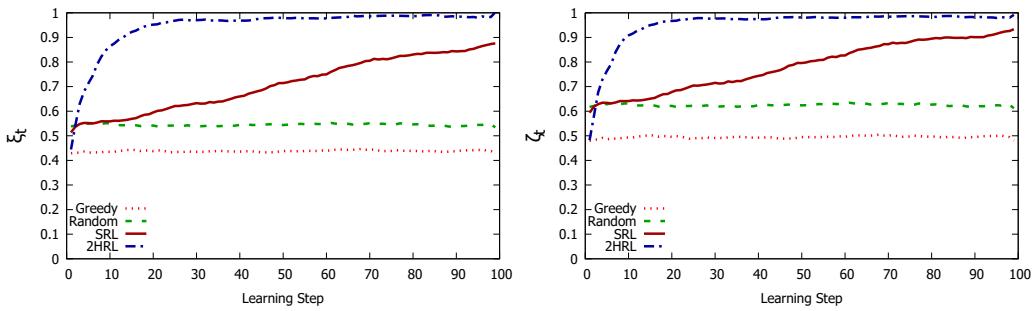
In our experiments we compare our approach with other three techniques: (i) a baseline *Random* algorithm, which does not consider quality values but randomly selects, among the available services, those services that satisfy the required functional dependencies; this algorithm is commonly used as a bottom-line quality indicator in learning problems, as learning heuristics performing no better than zero-knowledge random algorithms are obviously not worth to take into consideration; (ii) a *Greedy* algorithm, which always selects among the available services, those services with maximum quality; the greedy selection strategy proved to achieve good performance results in case of non load-dependent QoS attributes in similar problems [38, 20] (see Section 3 for further discussion); since most of the existing approaches in the literature adopt a greedy selection strategy, we consider the *Greedy* algorithm as reasonably representative of those approaches; (iii) a single-layer reinforcement learning (SRL) algorithm, presented in [45], which exploits past experience to predict the behavior of known services; this SRL algorithm is the most similar in literature to our 2HRL, as it employs a load-dependent model for the QoS-aware service selection procedure.

We conduct experiments for two kinds of scenarios: (i) *Static* scenarios, where the set of available services remains stable over time. In these experiments, we investigate the impact of two P/S architectures (centralized broker and P2P gossip-based unstructured overlay) and the scalability with respect to increasing system size. The goal of these experiments is to assess the effectiveness of the 2HRL approach in establishing a service assembly that fulfills the stated goals. (ii) *Dynamic* scenarios, where the set of available services is subject to variations over time. In particular, we investigate



(a) Average QoS, higher is better (b) Fairness, higher is better

Figure 5: Unstructured gossip-based P/S architecture: Static scenario



(a) Average QoS, higher is better (b) Fairness, higher is better

Figure 6: Centralized broker P/S architecture: Static scenario

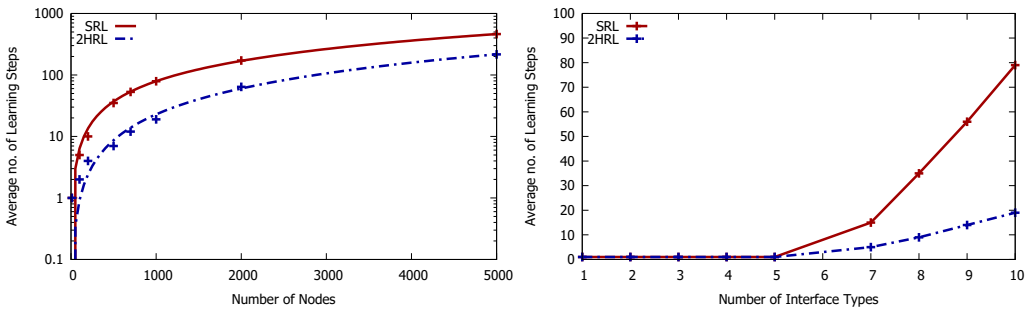
the impact of variations in the set composition (because of services entering/leaving the system) or in the quality of offered services. The goal is to assess the resilience of the 2HRL approach, i.e. its ability in maintaining the optimal operation point, once it has reached it, by self-adapting to changes that might happen in the operating environment. Indeed, open-end collections of distributed P2P nodes are necessarily prone to failures since autonomous nodes might suddenly leave/join the system at any time, as well as change their local quality.

8.2.1. Scalability

Impact of P/S architectures. In this first experiment (see Figures 5a - 5b and 6a - 6b) we consider a static scenario involving $N = 1000$ services with NUM_INT $= 10$ different interface types. We experiment the impact of two P/S architectures on the performance of the system: centralized broker and P2P gossip-based unstructured overlay. Both cases show how ξ_t and ζ_t, calculated on the fully resolved assembly resulting from the application of different selection rules, vary as a function of the number of learning steps η (i.e., over time). In particular, they show that the 2HRL approach outperforms the other selection rules by building a fully-resolved assembly whose ξ_t and ζ_t tend to the upper-bound.

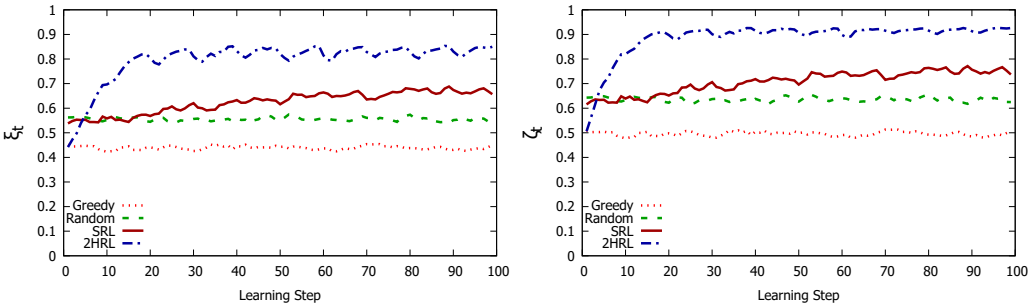
By comparing Figures 5a - 5b and 6a - 6b, we see that an interesting outcome of this experiment is that all the four considered rules, which offer different implementations of the SELECT() function are influenced only to a limited extent by the chosen P/S architecture. In this respect, we note that the adoption of a central broker allows the information to propagate faster and in a more reliable way, thus slightly improving the convergence speed of the curves. Hence, in the next experiments, as a worst case, we will only show the results for an unstructured P/S architecture based on gossiping. Anyway, the trend of the curves (not shown here) obtained using the central-broker P/S architecture is similar.

Scalability with respect to the number of nodes. This experiment (see Figure 7a) aims at assessing



(a) Scalability w.r.t. the number of nodes (b) Scalability w.r.t. the number of interface types

Figure 7: Learning scalability, lower is better

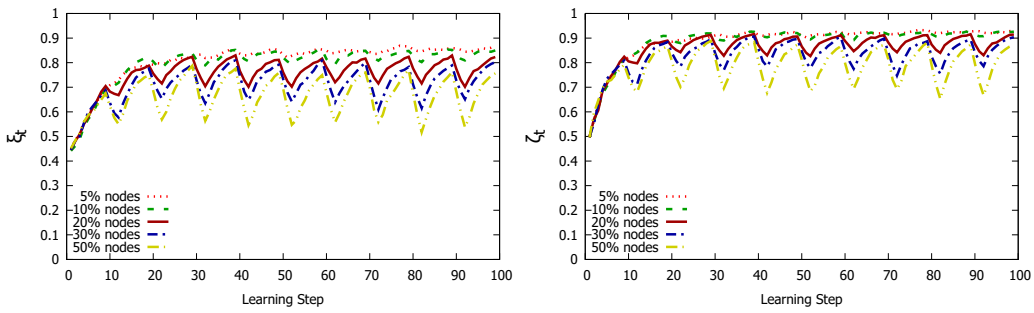


(a) Average QoS, higher is better (b) Fairness, higher is better

Figure 8: Self-adapting to nodes leaving and joining the system: Different Rules

how the learning time changes for the considered selection rules, with increasing system size (number of nodes). In particular, we measure the average number of learning steps needed to reach a given value of the average system QoS - i.e, $\xi_t = 0.8$. This is a lower-is-better metric. The number of different interface types is set to NUM_INT $= 10$ and we experimented with configurations up to 5000 nodes. Figure 7a shows the curves fitted to the experimented configurations. On the ordinate axis, the average number of learning steps is reported in logarithmic scale. The Greedy and Random selection rules are never able to reach the requested value of the average system QoS, and hence they are not shown in the figure. The 2HRL and SRL curves have similar trends, but the 2HRL rule reaches in every case the desired value in less than half of the steps necessary to the SRL rule. In order to reach an acceptable value of the average system QoS, our approach requires a number of learning steps that is linearly proportional to the number of nodes in the system. However, it is worth noticing that this happens only when the whole system is started from scratch (i.e., all entities have an empty knowledge). In fact, as we will show in Section 8.2.2, once the system reaches a good operating point, variations inside the environment are quickly addressed in few learning steps.

Scalability with respect to the number of interface types. This experiment (see Figure 7b) aims at assessing how the learning time changes for the considered selection rules, with increasing number of interface types. In particular, we measure again the average number of learning steps needed to reach a certain value of the average system QoS - i.e, $\xi_t = 0.8$. The number of nodes is set to $N = 1000$. Again, the Greedy and Random selection rules are never able to reach the requested value of the average system QoS, and are not shown in the figure. The 2HRL rule reaches in every case the desired value in less than half of the steps necessary to the SRL rule. However, increasing the number of interface types, the system requires a larger number of steps to reach an acceptable value of the aver-



(a) Average QoS (b) Fairness

Figure 9: Self-adapting to nodes leaving and joining the system: 2HRL

age system QoS. This happens since, in our setting, increasing the number of interface types increases also the number of possible dependencies for each service.

8.2.2. Resilience

Nodes leave and join the system. This experiment considers a dynamic scenario where a number of nodes unexpectedly leave the system and others enter. In particular, starting from the static scenario experimental setting – i.e., $N = 1000$ services with NUM_INT $= 10$ different interface types – we randomly remove every $\eta = 10$ learning steps 10% of nodes and insert the same amount of new nodes inside the system.

Figure 8 reports how the different selection rules react to the environment change. In particular, it shows how the 2HRL selection rule allows services to promptly react and to self-organize into fully-resolved assemblies that improve the average system QoS and fairness when changes occur. In fact, removing known services from the system and inserting new ones causes a performance drop, since the experience of the nodes previously inside the system is lost, and the behavior of the new nodes is not known a priori. However, in a short time, the 2HRL-based system is able to re-establish good values for ξ_t and ζ_t.

We experiment also another scenario, where we examine the behavior of the 2HRL rule only, with respect to the system dynamism degree. In particular, starting again from the static scenario, we evaluate the resilience of the 2HRL system for different numbers of nodes leaving and entering the system, every $\eta = 10$ learning steps. Figure 9 shows how the 2HRL selection rule is able to react in every setting – i.e., from 5% to 50% of nodes removed and inserted inside the system– again by self-organizing services into fully-resolved assemblies that improve the average system QoS (Fig. 9a), and fairness (Fig. 9b).

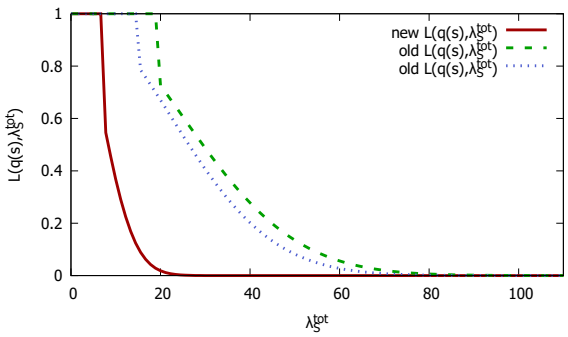


Figure 10: Quality functions change

Nodes change the quality function. Finally, this experiment considers a dynamic scenario where, starting from the static scenario experimental setting, every $\eta = 10$ learning steps 10% of randomly chosen services change their quality function $L(\vec{\mathbf{q}}(S), \lambda_S^{tot})$. In particular, for every randomly chosen service the local quality function changes as shown in Figure 10 – i.e., the local quality degrades –.

Figure 11 shows how the different selection rules react to the new setting. Also in this case, we can see that after an initial drop of ξ_t and ζ_t, the 2HRL selection rule allows services to quickly self-organize into fully-resolved assemblies that re-establish a good level of the average QoS (Figure 11a) and fairness (Figure 11b). An interesting outcome of this experiment is that, while other selection rules slowly decrease the average quality and fairness inside the system, the 2HRL rule is able to maintain

the performance level, despite more and more nodes changing their local quality function over time.

Again, we investigate also another scenario, where we show the behavior of the 2HRL rule only, with respect to the number of nodes changing the local quality function. In particular, starting from the static scenario – i.e., $N = 1000$ services of NUM_INT $= 10$ different types – we evaluate the resilience of the 2HRL rule by varying the number of nodes changing the local quality function every $\eta = 10$ learning steps. Figure 12 shows how the 2HRL selection rule is able to react in every setting – i.e., from 5% to 50% of nodes changing the local quality function – self-organizing services into fully-resolved assemblies that improve the average system QoS (Fig 12a), and fairness (Fig 12b), when changes occur.

8.3. Comments

Our simulation experiments show that the 2HRL algorithm is able to achieve globally good results in different situations, either static or dynamic, despite the fact that each agent builds its own perception of the execution environment (i.e., the local knowledge), which is then used to analyze the current setting and plan next actions.

This result is in line with a result already highlighted in [45], about the fact that, in load-dependent quality settings, populations that make choices by relying on a global knowledge built as a combination of all agents experience do not get good results. Indeed, Schaerf et al. [45] point out that the use of a shared global knowledge leads agents to be conservative and static, as they tend to make the same choice because of the same knowledge base. On the contrary, relying on local knowledge can lead to better load balancing and hence better global quality.

In this respect, our 2HRL algorithm makes each agent relying only on local knowledge built by combining the advertised QoS with the agent's personal experience. As an example of the resulting outcome, consider a lightly loaded service advertising its current good QoS. Because of this, some agents will bind to it and will actually experience the advertised QoS. Whereas, other agents (e.g., those binding later to the same service) will find the service already busy, and will likely experience a QoS lower than the advertised one, so having a negative perception. Therefore, at the next cycle, the former agents will tend to exploit the same service used before, whereas the latter will tend to explore new services. After a number of iterations, we can observe an "emergent collective behavior", where agents tend to self-balance themselves by selecting those services that locally maximize their expected reward.

9. Threats to Validity

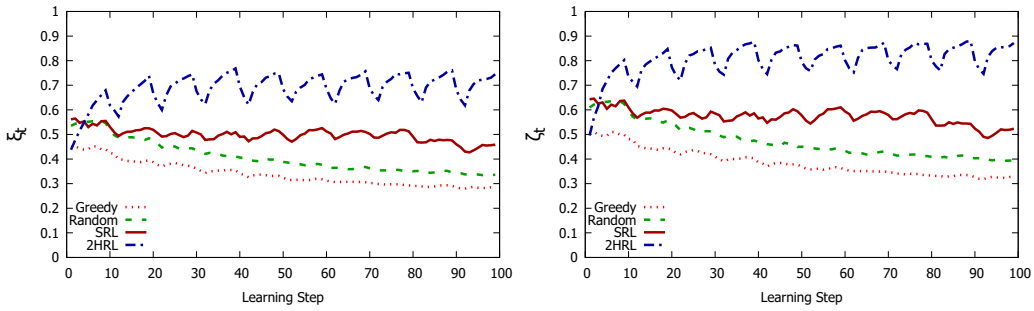
There are some potential threats to the validity [55] of the proposed approach.

A threat to external validity concerns the approach evaluation. Indeed, we adopted an evaluation based on extensive simulations, instead of considering single case studies, to perform a general analysis of the approach with respect to the main requirements stated in Section 2, namely *R2: QoS-awareness*, *R3: Resilience*, and *R4: Scalability*. However, to evaluate the practical implication of the adoption of our service assembly framework, we plan to select one of the existing service discovery platforms (e.g., GoPrime [8]) to support the actual implementation of our approach, so to validate it in a real-word settings. In particular, we plan to conduct a set of controlled experiments with software engineers and/or students, which will design and develop real fully-featured applications in the context of ongoing research projects and/or courses.

A threat to internal validity is represented by the selection of the social welfare indexes. To smooth this threat we adopted two different indexes to complement a measure of the overall QoS with a fairness index that measures how uniform is the QoS delivered by all the participating services. We are also planning to investigate the definition of other social welfare indexes to extend the validity of our approach.

10. Conclusion and Future Work

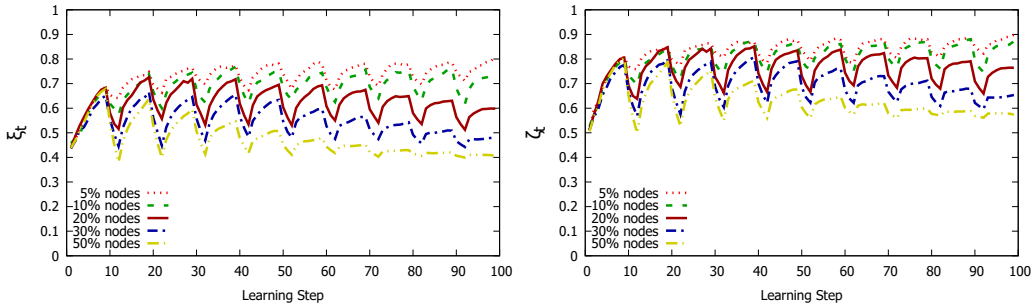
In this paper we have presented a fully decentralized service assembly framework for pervasive computing environments, where the set of services to be managed includes those offered by high-end software applications as well as those offered by low level virtualized hardware resources. The core element of the proposed solution is the combined use of (*i*) a reinforcement learning approach to cope with the lack of global knowledge and to make the system able to dynamically learn from experience the selection rule to be applied, and (*ii*) a QoS-driven service selection rule, taking into account the possible load-dependent nature of some QoS attributes.



(a) Average QoS, higher is better

(b) Fairness, higher is better

Figure 11: Self-adapting to nodes changing the local quality function: Different Rules



(a) Average QoS

(b) Fairness

Figure 12: Self-adapting to nodes changing the local quality function: 2HRL

Besides, our solution relies on a decentralized information sharing-based service discovery, which allows scalable and flexible diffusion of information in both structured and unstructured environments

Thanks to these features, the system is able to build and maintain in a fully decentralized way an assembly of services that, besides functional requirements, is able to guarantee a good level of social welfare. Moreover, we have shown through a set of simulation experiments that our solution includes self-adaptation capabilities to cope with unpredictably variable operating environments where, for example, services independently leave/enter the system.

We plan to extend this work along several lines. As concerns the adoption of reinforcement learning techniques, we intend to investigate the stability and convergence properties of our approach as well as the impact of possible different exploration-exploitation strategies. We remark that the lack of global knowledge and the dynamism of the scenarios we want to tackle rule out optimization approaches, which in turn require the global picture of the system. As future work, we also aim at comparing the performance of our decentralized learning technique with respect to optimization approaches that can serve as a benchmark.

We also envisage an extension of the experimental part with the inclusion of different scenarios and other possible definitions of fairness. Another interesting direction is combining the proposed RL method with model-based techniques at design-time to speed-up learning convergence at runtime. Previous works in the area show the feasibility of the approach [2, 49].

Finally, existing service discovery platforms (like GoPrime [8]) for pervasive computing environments do exist and have been reviewed in some papers [5, 14, 44]. These platforms will be considered as possible candidates to support the actual implementation of our approach, so to validate it in a real-word setting. Such prototype will allow us to perform accurate measurements and evaluate, for instance, the impact of runtime costs due to service

rebinding.

References

[1] S. Aghabozorgi, A. S. Shirkhorshidi, and T. Y. Wah. Time-series clustering a decade review. *Information Systems*, 53:16 – 38, 2015.

[2] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *2012 IEEE Network Operations and Management Symposium*, pages 204–212, April 2012.

[3] A. Bouguettaya, M. Singh, M. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, B. Benatallah, B. Medjahed, M. Ouzzani, F. Casati, X. Liu, H. Wang, D. Georgakopoulos, L. Chen, S. Nepal, Z. Malik, A. Erradi, Y. Wang, B. Blake, S. Dustdar, F. Leymann, and M. Papazoglou. A service computing manifesto: The next 10 years. *Commun. ACM*, 60(4):64–72, Mar. 2017.

[4] L. Buşoniu, R. Babuška, and B. De Schutter. *Multi-agent Reinforcement Learning: An Overview*, pages 183–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[5] C. Cabrera, A. Palade, and S. Clarke. An evaluation of service discovery protocols in the internet of things. In *Proc. of the Symposium on Applied Computing*, pages 469–476. ACM, 2017.

[6] M. Caporuscio, M. D'Angelo, V. Grassi, and R. Mirandola. Reinforcement learning techniques for decentralized self-adaptive service assembly. In *Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proc*, volume 9846 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2016.

[7] M. Caporuscio and C. Ghezzi. Engineering Future Internet applications: The Prime approach. *Journal of Systems and Software*, 106(0):9 – 27, 2015.

[8] M. Caporuscio, V. Grassi, M. Marzolla, and R. Mirandola. GoPrime: a fully decentralized middleware for utility-aware service assembly. *IEEE Transactions on Software Engineering*, 42(2):136–152, Feb 2016.

[9] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. L. Presti, and R. Mirandola. MOSES: A framework for qos driven runtime adaptation of service-oriented systems. *IEEE Trans. Soft. Eng.*, 38(5):1138–1159, 2012.

[10] J. Cardoso. Complexity analysis of bpel web processes. *Software Process: Improvement and Practice*, 12(1):35–49, 2007.

[11] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, Aug. 2001.

[12] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9), Sept. 2001.

[13] M. D'Angelo, S. Gerasimou, S. Ghahremani, J. Grohmann, I. Nunes, E. Pournaras, and S. Tomforde. On learning in collective self-adaptive systems: State of practice and a 3d framework. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 13–24, May 2019.

[14] W. K. Edwards. Discovery systems in ubiquitous computing. *IEEE Pervasive Computing*, 5(2), Apr. 2006.

[15] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[16] G. Erus and F. Polat. A layered approach to learning coordination knowledge in multiagent environments. *Applied Intelligence*, 27(3):249–267, Dec. 2007.

[17] P. Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29(1), Jan. 2007.

[18] S. Fagernes and A. L. Couch. Resource-sharing among autonomous agents - A comparative study of selfish versus altruistic behaviour. *Service Oriented Computing and Applications*, 12(3-4):317–331, 2018.

[19] S. Ferretti. Publish-subscribe systems via gossip: A study based on complex networks. In *Proc. of the Fourth Annual Workshop on Simplifying Complex Networks for Practitioners*, SIMPLEX '12, pages 7–12, New York, NY, USA, 2012. ACM.

[20] R. R. Filho and B. Porter. Defining emergent software using continuous self-assembly, perception, and learning. *ACM Trans. Auton. Adapt. Syst.*, 12(3):16:1–16:25, Sept. 2017.

[21] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. Edge-centric computing: Vision and challenges. *SIGCOMM Comput. Commun. Rev.*, 45(5):37–42, Sept. 2015.

[22] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *Proc. of the First Workshop on Self-healing Systems*, WOSS '02, pages 33–38, New York, NY, USA, 2002. ACM.

[23] A. Greenwald and K. Hall. Correlated-q learning. In *AAAI Spring Symposium*. AAAI Press, 2003.

[24] M. Han. *Reinforcement Learning Approaches in Dynamic Environments*. PhD thesis, 2018.

[25] P. J. Hoen, K. Tuyls, L. Panait, S. Luke, and J. A. La Poutré. An overview of cooperative and competitive multiagent learning. In *Proc. of the First International Conference on Learning and Adaption in Multi-Agent Systems*, LAMAS'05, pages 1–46, Berlin, Heidelberg, 2006. Springer-Verlag.

[26] J. Hu and M. P. Wellman. Nash q-learning for general-sum stochastic games. *J. Mach. Learn. Res.*, 4:1039–1069, Dec. 2003.

[27] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. Technical Report DEC-TR-301, Digital Equipment Corporation, Sept. 1984.

[28] C. Jatoth, G. R. Gangadharan, and R. Buyya. Computational intelligence based qos-aware web service composition: A systematic literature review. *IEEE Transactions on Services Computing*, 10(3):475–492, May 2017.

[29] L. Jiang and G. Xu. Modeling and analysis of software aging and software failure. *J. Syst. Softw.*, 80(4):590–595, Apr. 2007.

[30] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, A. Mohiuddin, R. Sepassi, G. Tucker, and H. Michalewski. Model-based reinforcement learning for atari, 2019.

[31] S. Kapetanakis and D. Kudenko. Reinforcement learning of coordination in cooperative multi-agent systems. In *Eighteenth National Conference on Artificial Intel-*

ligence, pages 326–331. American Association for Artificial Intelligence, 2002.

[32] H. B. Mahfoudh, G. Di Marzo Serugendo, A. Boulmier, and N. Abdennadher. Coordination model with reinforcement learning for ensuring reliable on-demand services in collective adaptive systems. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems*, pages 257–273, 2018.

[33] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed Hashing in a Small World. *USITS '03 Proc. of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, 4:10–10, 2003.

[34] E. Meshkova, J. Riihijarvi, M. Petrova, and P. Mahonen. A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks. *Computer Networks*, 52(11):2097 – 2128, 2008.

[35] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell. Sensing meets mobile social networks: The design, implementation and evaluation of the cenceme application. In *Proc. of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 337–350. ACM, 2008.

[36] M. Moghaddam and J. G. Davis. *Service Selection in Web Service Composition: A Comparative Review of Existing Approaches*, pages 321–346. Springer New York, New York, NY, 2014.

[37] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, Sept. 2009.

[38] A. Moustafa and T. Ito. A deep reinforcement learning approach for large-scale service composition. In T. Miller, N. Oren, Y. Sakurai, I. Noda, B. T. R. Savarimuthu, and T. Cao Son, editors, *PRIMA 2018: Principles and Practice of Multi-Agent Systems*, pages 296–311, 2018.

[39] A. Moustafa and M. Zhang. *Multi-Objective Service Composition Using Reinforcement Learning*, pages 298–312. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[40] G. Mühl. *Generic Constraints for Content-Based Publish/Subscribe*, pages 211–225. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[41] L. Panait and S. Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, Nov. 2005.

[42] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *First International Semantic Web Conference*, 2002.

[43] I. C. Paschalidis and J. N. Tsitsiklis. Congestion-dependent pricing of network services. *IEEE/ACM Trans. Netw.*, 8(2):171–184, Apr. 2000.

[44] L. Roffia, F. Morandi, J. Kiljander, A. D'Elia, F. Vergari, F. Viola, L. Bononi, and T. S. Cinotti. A semantic publish-subscribe architecture for the internet of things. *IEEE Internet of Things Journal*, 3(6):1274–1296, Dec 2016.

[45] A. Schaerf, Y. Shoham, and M. Tennenholtz. Adaptive load balancing: A study in multi-agent learning. *Journal of Artificial Intelligence Research*, 2:475–500, 1995.

[46] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proc. of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[47] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[48] D. Sykes, J. Magee, and J. Kramer. Flashmob: Distributed adaptive self-assembly. In *Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 100–109. ACM, 2011.

[49] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *2006 IEEE International Conference on Autonomic Computing*, pages 65–73, June 2006.

[50] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons Ltd., Chichester, UK, 2nd edition edition, 2002.

[51] O. Wahab, J. Bentahar, H. Otrok, and A. Mourad. A survey on trust and reputation models for web services: Single, composite, and communities. *Decision Support Systems*, 74, 04 2015.

[52] H. Wang, X. Chen, Q. Wu, Q. Yu, X. Hu, Z. Zheng, and A. Bouguettaya. Integrating reinforcement learning with multi-agent techniques for adaptive service composition. *ACM Trans. Auton. Adapt. Syst.*, 12(2):8:1–8:42, May 2017.

[53] H. Wang, G. Huang, and Q. Yu. Automatic hierarchical reinforcement learning for efficient large-scale service composition. In *2016 IEEE International Conference on Web Services (ICWS)*, pages 57–64, June 2016.

[54] D. Weyns and et al. On patterns for decentralized control in self-adaptive systems. In R. De Lemos, H. Giese, H. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *LNCS*, pages 76–107. 2013.

[55] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[56] K. P. Yoon and C.-L. Hwang. *Multiple attribute decision making: an introduction*, volume 104. Sage publications, 1995.

A. Table of Notations

Structural Model	
N	Number of services (nodes)
$\mathbf{S}$	Set of services $\mathbf{S} = \{S_1, \ldots, S_N\}$
$\mathbf{T}$	Set of service types $\mathbf{T} = \{1, \ldots, T\}$
$\mathbf{E}$	Set of resolved dependencies $\mathbf{E} \subseteq \mathbf{S} \times \mathbf{S}$
$\mathbf{A}$	Service Assembly $\mathbf{A} = (\mathbf{S}, \mathbf{E})$
m	Number of QoS and structural attributes
$\vec{\mathbf{q}}$	vector of m internal QoS
$S.Type$	Type of the provided interface by S
$S.Context$	Context of S
$S.Deps$	Set of dependencies of S
$S.In_t$	Set of nodes that S is bound to at time t
$S.Out_t$	Set of nodes that have a binding with S at time t
$\mathbf{S}_t^{full}$	Set of fully resolved services at time t
$match_T$	Function matching two interface types
$match_C$	Function matching two contexts
$match_S$	Function matching a service and dependency type
QoS Model	
σ_S	Rate of external service requests addressed to S
Θ_S	Set of transfer functions of S
λ_S^{tot}	Load addressed to S
$\mathbf{L}$	Quality function that returns the load-dependent $\vec{\mathbf{Q}}_t(S)$ at time t
$\mathbf{C}$	Function that combines $\mathbf{L}$ with the QoS of all S dependencies
$\vec{\mathbf{Q}}_t(S)$	QoS of S at time t
$Q_{i,t}(S)$	i-th entry of $\vec{\mathbf{Q}}_t(S)$
$Q_t(S)$	QoS of S at time t (scalar version of $\vec{\mathbf{Q}}_t(S)$)
V_i^{max}	Maximum value of $Q_{i,t}$
V_i^{min}	Minimum value of $Q_{i,t}$
w_i	Weights for the different QoS attributes $Q_{i,t}(S)$
Learning Model	
η	Learning-step
α_j	Learning-rate parameter used by S for S_j
t_η	Time a learning step η happens
$Q_\eta^R(S_j)$	QoS (reward) observed by a given S while interacting with S_j at step η
$Q_{\eta+1}^E(S_j)$	QoS (reward) expected by S from S_j at the step $\eta + 1$
z	Learning window
$\hat{N}(S_j)$	Number of times that S_j has been invoked by S
$\mathbf{m}_t$	Message received at time t by S
$Q_t(S_j)$	Advertised QoS by $S_j \in \mathbf{m}_t$
$\tau_{t_\eta}(S_j)$	Level of trust for the QoS advertised by S_j
$F_{t_\eta}(S_j)$	Accuracy of the advertised QoS by S_j
$H_{t_\eta}(S_j)$	Final quality function combining the two learning layers
$\perp$	Special value that is guaranteed to be worse than the QoS of any $\mathbf{S}_t^{full}$
Social Welfare Indexes	
ξ_t	Average QoS delivered by services in $\mathbf{S}_t^{full}$
ζ_t	Fairness at time t delivered by services in $\mathbf{S}_t^{full}$

B. QoS Attributes – Examples

Execution time. This attribute measures the time needed by a service S to fulfill one request at a given time t. For each dependency $d \in S.Deps$, we recall that the transfer function $\theta_{S,d}(\lambda)$ described in Section 5.2 represents the average number of times a service S' resolving d is invoked during the execution of S, when S is subject to an incoming rate λ of service requests. Let $e(S)$ denote the execution time of local code declared by S for serving a single request in isolation, and let $L_t^e(e(S), \lambda_S^{tot})$ denote the time actually spent by S for serving a request when subjected to a load λ_S^{tot}. Then, we can define the QoS $Q_t^e(S)$ of service S with respect to execution time as (we omit the arguments of $\mathbf{C}()$ for the sake of simplicity):

$$Q_t^e(S) = \begin{cases} -L_t^e(e(S), \lambda_S^{tot}) & \text{if } S.Deps = \emptyset \\ \perp & \text{if } S \text{ is partially resolved} \\ \mathbf{C}() \stackrel{\text{def}}{=} -L_t(e(S), \lambda_S^{tot}) + \sum_{S' \in S.In_t} Q_t^e(S') \times \theta_{S,(S'.Type,S'.Context)}(\lambda_S^{tot}) & \text{if } S \text{ is fully resolved} \end{cases} \tag{12}$$

$Q_t^e(S)$ expresses the average execution time of S at a given time t for serving one request when it is subject to a total load λ_S^{tot} of concurrent requests; we express $Q_t^e(S)$ as a negative value so that it is a higher-is-better metric. We point out that such an expression is based on the assumption of a sequential execution model for the dependencies of S (as we sum the execution times of all services in $S.In_t$). In the case of a parallel execution model for some of those dependencies, the expression should be modified accordingly (see for example [9, 10]).

Reliability. This attribute measures the probability that S correctly completes its task for a given service request. Let $r(S)$ be the internal reliability (that is the probability that no internal failure occurs) declared by S at design time and let $L_t^r(r(S), \lambda_S^{tot})$ denote the runtime reliability of S when serving λ_S^{tot} concurrent requests. Then, we can define the QoS $Q_t^r(S)$ of service S with respect to reliability as:

$$Q_t^r(S) = \begin{cases} L_t^r(r(S), \lambda_S^{tot}) & \text{if } S.Deps = \emptyset \\ \perp & \text{if } S \text{ is partially resolved} \\ \mathbf{C}() \stackrel{\text{def}}{=} L_t^r(r(S), \lambda_S^{tot}) \times \prod_{S' \in S.In_t} Q_t^r(S')^{\theta_{S,(S'.Type,S'.Context)}(\lambda_S^{tot})} & \text{if } S \text{ is fully resolved} \end{cases} \tag{13}$$

The reliability-based QoS $Q_t^r(S)$ for S is the joint probability that S experiences no internal failures when it is subject to a total load λ_S^{tot}, and all $\theta_{S,S'.Type}$ invocations of each dependency S' produce no failure; the joint probability of these events is the product of their probabilities [50].

Cost. The average cost of a service S is the average cost incurred for one execution of S. The cost could be expressed in monetary units, or some other suitable unit (e.g., energy consumption). Let $c(S)$ denote the cost declared by S for serving a single request in isolation, we distinguish two cases depending on how cost accumulates for multiple service invocations.

In the first case, we assume that an additive cost is incurred for each single invocation of a service $S' \in S.In$. Under this assumption, the cost-based QoS $Q_t^c(S)$ for an assembly rooted at S can be defined as:

$$Q_t^c(S) = \begin{cases} -L_t^c(c(S), \lambda_S^{tot}) & \text{if } S.Deps = \emptyset \\ \perp & \text{if } S \text{ is partially resolved} \\ \mathbf{C}() \stackrel{\text{def}}{=} -L_t^c(c(S), \lambda_S^{tot}) + \sum_{S' \in S.In_t} Q_t^c(S') \times \theta_{S,(S'.Type,S'.Context)}(\lambda_S^{tot}) & \text{if } S \text{ is fully resolved} \end{cases} \tag{14}$$

Where $L_t^c(e(S), \lambda_S^{tot})$ is the cost of S for serving a request when subjected to a load λ_S^{tot}, and $\theta_{S,S'.Type}$ is the number of invocations for each dependency S'. Note that, to force $Q_t^c(S)$ to be a higher-is-better metric, the QoS is the *negated* total cost. Alternatively, we may assume a flat cost model, where a fixed cost is paid for the use of a service, independently of the number of times it is actually invoked. In this case, the flat cost-based QoS $Q_t^c(S)$ can be defined as:

$$Q_t^c(S) = \begin{cases} -c(S) & \text{if } S.Deps = \emptyset \\ \perp & \text{if } S \text{ is partially resolved} \\ \mathbf{C}() \stackrel{\text{def}}{=} -c(S) + \sum_{S' \in S.In} Q_t^c(S') & \text{if } S \text{ is fully resolved} \end{cases} \tag{15}$$

Note that, in this case we assume the cost is not load-dependent.