Reinforcement Learning Techniques for Decentralized Self-adaptive Service Assembly

M. Caporuscio[1], M. D'Angelo[1], V. Grassi[2], and R. Mirandola[3](✉)

[1] Linnaeus University, Växjö, Sweden
{mauro.caporuscio,mirko.dangelo}@lnu.se
[2] Università di Roma Tor Vergata, Rome, Italy
vgrassi@info.uniroma2.it
[3] Politecnico di Milano, Milan, Italy
raffaela.mirandola@polimi.it

Abstract. This paper proposes a self-organizing fully decentralized solution for the service assembly problem, whose goal is to guarantee a good overall quality for the delivered services, ensuring at the same time fairness among the participating peers. The main features of our solution are: (*i*) the use of a gossip protocol to support decentralized information dissemination and decision making, and (*ii*) the use of a reinforcement learning approach to make each peer able to learn from its experience the service selection rule to be followed, thus overcoming the lack of global knowledge. Besides, we explicitly take into account load-dependent quality attributes, which lead to the definition of a service selection rule that drives the system away from overloading conditions that could adversely affect quality and fairness. Simulation experiments show that our solution self-adapts to occurring variations by quickly converging to viable assemblies maintaining the specified quality and fairness objectives.

1 Introduction

We consider a distributed peer-to-peer scenario, where a large set of peers cooperatively work to accomplish specific tasks. In general, each peer possesses the know-how to perform some tasks (offered services), but could require services offered by other peers to carry out these tasks. Scenarios of this type can be typically encountered in pervasive computing application domains like ambient intelligence or smart transportation systems, where several (from tens to thousands) services cooperate to achieve some common objectives [13].

A basic functional requirement for this scenario is to match required and provided services, so that the resulting assembly makes each peer able to correctly deliver its service(s). Besides this functional requirement, we also assume the existence of non functional requirements concerning the quality of the delivered service, expressed in terms of several quality attributes referring to different quality domains (e.g., performance, dependability, cost).

© IFIP International Federation for Information Processing 2016
Published by Springer International Publishing Switzerland 2016. All Rights Reserved
M. Aiello et al. (Eds.): ESOCC 2016, LNCS 9846, pp. 53–68, 2016.
DOI: 10.1007/978-3-319-44482-6_4

Our goal is to devise a self-assembly procedure among the peers, aimed at fulfilling both functional and non functional requirements. For the latter, we aim in particular to maximize some notion of global utility expressed in terms of the quality attributes of the services delivered by peers in the system, ensuring at the same time fairness (i.e., no peer should be excessively favored or penalized with respect to others). Challenges to be tackled to achieve this goal include: (1) the presence of several functionally equivalent services, with different values of their quality attributes, which makes non trivial determining the "best" selection of offered services to be bound to required services; (2) the intrinsic dynamism of a large distributed system, with peers entering/exiting the system, or changing the value of their quality attributes, which require to dynamically adapt the assembly to the changing system configuration; (3) the lack of global knowledge, which is difficult to achieve and maintain in a large distributed system with autonomous peers; this makes centralized service assembly policies hardly usable; (4) the need of devising a service selection and assembly procedure that scales with increasing system size (up to hundreds or thousands of services/peers); (5) the possibly load-dependent nature of service quality attributes. This obviously holds for attributes in the performance domain (e.g., response time), where load has a negative impact on their value; it may also hold for other domains like the dependability domain, where increasing load could increase the likelihood of failures [8], or the cost domain, for example in case of cost schemes based on congestion pricing [11]. Load-dependent quality attributes rule out simple greedy service selection policies, as they could easily lead to service overloading, and consequent worsening of the overall delivered quality.

To cope with these challenges we propose in this paper a self-adaptive fully decentralized solution for the service assembly problem, whose main features are: (*i*) the adoption of an unstructured peer-to-peer approach for dynamic service discovery, based on the use of a gossip protocol that guarantees resilience through self-adaptation to dynamic changes occurring in the system, and scalability with respect to the system size, thanks to the bounded amount of information maintained and exchanged among peers; (*ii*) the use of a reinforcement learning approach to make each peer able to dynamically learn from its experience the service selection rule to be followed, thus overcoming the lack of global knowledge; (*iii*) the explicit consideration of load-dependent quality attributes, which leads to the definition of a service selection rule that drives the system away from service overloading conditions.

The paper is organized as follows. In Sect. 2 we give an overview of the main features of the adopted approach. In Sect. 3 we define the system model and state the problem we intend to solve. In Sect. 4 we detail the core elements of our approach. In Sect. 5 we present experimental results obtained through simulation. In Sect. 6 we survey related work, while in Sect. 7 we present conclusions and hints for future work.

2 Adopted Approach Overview

Self-adaptive systems have been proposed to cope with the dynamic environment where large software-intensive systems typically operate, for example because of changes in the availability or quality of the resources they rely on [2].

A self-adaptive system typically consists of a *managed system* that implements the system business logic, and a *feedback loop* that implements the adaptation logic for the managed system. General architecture for the feedback loop is the MAPE-K model, with *Monitor* (M), *Analyze* (A), *Plan* (P) and *Execute* (E) activities, plus a *Knowledge* (K) that maintains relevant information for the other components (e.g., system state, adaptation rules) [17].

In our setting, *Monitor* aims at collecting information about candidate services and their quality attributes, whereas *Analyze* and *Plan* aim at selecting, among the set of known candidates, those services that best serve to resolve existing dependencies and fulfill non functional requirements. Finally, *Execute* actually implements the bindings with the selected services, so leading to the construction (and maintenance) of the required assembly. However, how the MAPE-K activities are actually architected and implemented must take into account the specific characteristics of the managed system and its operating environment. In the rest of this section, we outline the main characteristics of the approach we have adopted to this end, highlighting how it deals with the challenges described in the Introduction.

MAPE-K information sharing architectural pattern – In large distributed settings a single MAPE-K loop is hardly adequate to manage the whole system, and *monitor*, *analyze*, *plan*, and *execute* are implemented by multiple MAPE-K loops that coordinate with one another. According to the *information sharing pattern* [17], each peer self-adapts locally by implementing its own MAPE-K loop, but requires state information from other peers in the system. Apart from information sharing, peers do not coordinate other activities. Hence, this pattern supports autonomous adaptation decisions at each node, and enables scalability thanks to the loose coordination required, limited to state information exchange.

Gossip based monitoring – According to the information sharing pattern, information collected by the *monitor* at each peer is shared with other peers in the system. In the scenario we are considering, this information mainly concerns offered services the peer is aware of, and their functional and non-functional properties. To cope with some of the challenges we have outlined in the introduction, this coordinated monitoring activity should scale with increasing system size, and be able to quickly react to changes occurring in the system (e.g., new offered services, variations of their quality). To this end, we adopt a gossip-based approach [1], which exploits epidemic protocols to achieve reliable information exchange among large sets of interconnected peers, also in presence of network volatility (e.g., peers join/leave the system suddenly). Specifically, in a gossip communication model, each peer in the system periodically exchanges information with a dynamically built peer set, and spreads information epidemically, similar to a virus in biological communities. This guarantees quick, decentralized and scalable information dissemination, and makes gossip-based communication well suited for our purposes. We detail the applied algorithm in Sect. 4.1.

TD-learning based analysis and planning – *Analyze* and *Plan* are local at each peer, and do not require any explicit coordination with other peers. In our setting, these activities aim at selecting, within the set of candidates built by the monitoring activity, the offered services to resolve the dependencies of local services, trying at the same time to maximize the system quality and ensure fairness (see Sect. 3). In the dynamic scenario we are considering, fixed selection rules are hardly able to achieve satisfactory results. Indeed, we make peers learn on their own the selection rule to be applied, using a reinforcement learning approach where 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 [15]. These features fit well with the considered scenario, where peers do not know each other (and the services they offer) in advance, but discover themselves dynamically. Moreover, services can have multiple dependencies to be resolved, and are characterized by multiple load-dependent quality attributes. To this end, we focus on temporal-difference (TD) learning methods [15], which can learn directly from raw experience without a model of the environment's dynamics and are implemented in an on-line, fully incremental fashion. In particular, we base the learning on two kinds of knowledge that are incrementally acquired by each peer: information about the existence of offered services and their advertised quality, achieved through the gossip-based monitoring activity, and the direct experience of the services' quality, acquired by each peer after actually binding to the selected services. We use the 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 Sect. 4.2.

3 System Model

In this section we define the model of the system we are considering and introduce the terminology and notation used in the rest of the paper.

We consider a set of N distributed services $\mathbf{S} = \{S_1, \ldots, S_N\}$ hosted by peer nodes communicating each other through a network. A service S is a tuple $\langle \mathit{Type}, \mathit{Deps}, \mathit{In}_t, \mathit{Out}_t, \mathbf{u}, \mathbf{U}_t \rangle$, where:

- $S.\mathit{Type} \in \mathbf{T}$ denotes the type of the provided interface (we say that $S.\mathit{Type}$ is the type of S). We assume $w \geq 1$ different service types $\mathbf{T} = \{T_1, \ldots, T_w\}$.
- $S.\mathit{Deps} \subseteq \mathbf{T}$ is the set of required dependencies for S: for each $d \in S.\mathit{Deps}$, S must be bound to a service S' such that $d = S'.\mathit{Type}$, in order to be executed. If $S.\mathit{Deps} = \emptyset$, then S has no dependencies. We assume that $S.\mathit{Deps}$ is fixed for each service and known in advance.
- $S.\mathit{In}_t \subseteq \mathbf{S}$ is the set of services S is bound to 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.
- $\mathbf{u} \subseteq \mathbb{R}^m$ is a vector of m "local quality" 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 $\mathbf{u}$ gives only a partial view of the overall quality of S, which also depends on the quality of the services used to resolve them. For example, in case of a completion time attribute, the corresponding $\mathbf{u}$ entry could represent the execution time in isolation of S internal code on the hosting node, without considering the completion time of the called services.
- $\mathbf{U}_t \subseteq \mathbb{R}^m$ is a vector of m "overall quality" attributes, which express the quality of the service S at time t, depending on both local quality of S and the quality of the services it is bound to to resolve its dependencies. We show below (see Eq. 1) how $\mathbf{U}_t$ is expressed in terms of both these factors.

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 $d = S'.Type$. 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.

Given these definitions, the overall quality for a service S is defined as follows:

$$\mathbf{U}_t(S) = \begin{cases} \mathbf{L}(\mathbf{u}(S), S.Out_t), & \text{if } S.Deps = \emptyset \\ \perp & \text{if } S \text{ is partially resolved} \\ \mathbf{C}\left(\mathbf{L}(\mathbf{u}(S), S.Out_t), \mathbf{U}_t(S_1), \ldots, \mathbf{U}_t(S_k)\right) & \\ & \text{if } S \text{ fully resolved, with } S.In_t = \{S_1, \ldots, S_k\} \end{cases} \tag{1}$$

In Eq. (1) if S has no dependencies ($S.Deps = \emptyset$), then S is by definition fully resolved, and $\mathbf{U}_t(S)$ is calculated by means of a suitable function $\mathbf{L} : \mathbb{R}^m \times 2^{\mathbf{S}} \rightarrow \mathbb{R}^m$, which, given the local quality $\mathbf{u}(S)$ and the set $S.Out_t$ of services currently bound to S, returns the actual load-dependent $\mathbf{U}_t(S)$ at time t. In order to keep the model as general as possible, we use the set $S.Out_t$ to define the load-dependent nature of $\mathbf{L}$, without explicitly specifying information such as request rate and job size [12], which is application specific. However, $\mathbf{L}$ can be easily extended and instantiated to account for further specific information, without affecting our notion of overall quality. Instead, if S has a nonempty set of dependencies ($S.Deps \neq \emptyset$) and is not fully resolved, $\mathbf{U}_t(S)$ is set to $\perp$, i.e., the special value that is guaranteed to be "worse" than the quality of any fully resolved instance of S. Finally, if S has a nonempty set of dependencies and is fully resolved, $\mathbf{U}_t(S)$ is computed using a function $\mathbf{C} : \mathbb{R}^{(1+|S.In_t|)m} \rightarrow \mathbb{R}^m$, which combines the local load-dependent quality $\mathbf{L}(\mathbf{u}(S), S.Out_t)$ with the overall quality of all S dependencies. The general equation (1) can be instantiated for specific quality attributes as described, for example, in [1].

Problem formalization – Our goal is to maximize the quality globally delivered by the services hosted in the system, ensuring at the same time fairness among services. To this end, we must define our notion of global quality and fairness.

For the former, the vector $\mathbf{U}_t(S)$ details the overall quality delivered by a specific service S in terms of a set of distinct quality attributes. To facilitate dealing with multiple and possibly conflicting quality attributes, we transform $\mathbf{U}_t(S)$ into a single scalar value, using the Simple Additive Weighting (SAW) technique [18]. According to SAW, we redefine the service quality of S as a weighted sum of its normalized quality attributes, as follows:

$$GU_t(S) = \sum_{i=1}^{m} w_i \frac{V_i^{max} - U_{i,t}(S)}{U_{i,t}(S) - V_i^{min}} \tag{2}$$

where $U_{i,t}(S)$ denotes the i-th entry of $\mathbf{U}_t(S)$, V_i^{max} and V_i^{min} denote, respectively, the maximum and minimum value of $U_{i,t}$, and $w_i \geq 0$, $\sum_{i=1}^{m} w_i = 1$, are weights for the different quality attributes expressing their relative importance.

$$\xi_t = \frac{1}{|\mathbf{S}_t^{full}|} \sum_{S \in \mathbf{S}_t^{full}} GU_t(S) \tag{3}$$

$$\zeta_t = \frac{(\sum_{S \in \mathbf{S}_t^{full}} GU_t(S))^2}{|\mathbf{S}_t^{full}| \sum_{S \in \mathbf{S}_t^{full}} GU_t(S)^2} \tag{4}$$

Now, let $\mathbf{S}_t^{full} \subseteq \mathbf{S}$ be the set of fully resolved services at time t. Equation 3 defines the *global system quality* as the average quality offered by services in $\mathbf{S}_t^{full}$. Furthermore, in order to measure the *uniformity of quality* delivered in the system, we make use of the *Jain's fairness index* [6], defined as in Eq. 4.

In our load-dependent setting, the more uniform is the quality, the more uniform the load distribution tends to be. Hence, our goal can be stated as the definition of a self-adaptive assembly procedure that: (i) maximizes ξ_t, thus optimizing *quality*, and (ii) maximizes ζ_t, thus optimizing *fairness*.

To this end, next section describes the system operations for service discovery and selection that drive the system towards the achievement of this goal.

4 System Operations

This section describes the implementation of the MAPE-K information sharing pattern that drives the self-adaptive assembly process, focusing in particular on the monitoring (Sect. 4.1), and analyzing and planning (Sect. 4.2) activities.

4.1 Gossip Based Monitoring

Algorithm 1 describes the general gossip-based scheme [7] that implements the monitoring activity (M). It includes two concurrent threads: an active thread that starts an interaction by sending a message to a random set of peers[1], and a passive thread that reacts to messages received from other peers.

Every Δt time units, the *active thread* reads monitored information I_K from the knowledge base (K) (line 4), and sends a message $\mathbf{m}$ containing I_K to the current *peer set* (line 6). Specifically, $I_K = Hosted \cup Known$ contains information

[1] Provided by an underlying peer sampling protocol, e.g. NEWSCAST [16].

Algorithm 1. Gossip based information sharing

```
1: procedure ActiveThread
2:     loop
3:         Wait $\Delta t$
4:         $I_K \leftarrow \textsc{Read}_K()$
5:         for all $S_i \in \textsc{GetPeers}()$ do
6:             Send $\langle I_K \rangle$ to $S_i$
7: procedure PassiveThread
8:     loop
9:         Wait for message $\langle \mathbf{m} \rangle$
10:        for all $S_i \in \mathbf{m}$ do
11:            $\textsc{Update}_K(S_i)$
12: function $\textsc{Update}_K$(in $S \in \mathbf{S}$)
13:     if $\exists \overline{S} \in Hosted | S.Type \in \overline{S}.Deps$ then
14:         if $|Known| < N_K$ then
15:             $Known \leftarrow Known \cup \{S\}$
16:         else
17:             $m \leftarrow \min_j \{GU_t(S_j) \mid S_j \in Known \wedge S_j.Type = S.Type\}$
18:             $Known \leftarrow Known \setminus \{S_m\} \cup \{S\}$
```

about the set of services hosted locally and monitored by M, and the set of other known services discovered by means of message gossiping, respectively.

On the other hand, the *passive thread* listens for messages gossiped by other peers and, upon receiving a new message $\mathbf{m}$, it invokes the function $\textsc{Update}_K()$ for each S_i contained in $\mathbf{m}$. The function $\textsc{Update}_K()$ is in charge of updating the knowledge K with the received information. Indeed, referring to Algorithm 1, $\textsc{Update}_K()$ updates the set *Known* (stored in K) that collects the currently known N_K (or less) "best" services solving at least one dependency for the hosted services[2]. In particular, if the size of *Known* is exceeded, the S_j with the smallest $GU_t(S)$ (i.e., S_m) is replaced by the newly discovered $S_i \in \mathbf{m}$.

As a consequence, the total amount of exchanged information between a pair of peer nodes is upper bounded by $O(N_K \cdot |\textsc{GetPeers}()|)$. This makes scalable the information sharing procedure, as its complexity at each round grows at most linearly with the number of nodes in the system, assuming that $\textsc{GetPeers}()$ returns a set of peers whose cardinality is independent of the system size.

4.2 TD-learning Based Analysis and Planning

As introduced in Sect. 2, the *analysis* and *planning* activities are locally implemented at each peer. 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*), trying to maximize the global system quality ξ_t and the fairness ζ_t.

Algorithm 2 outlines the *analysis* implementation. It consists of a thread that actively checks, every Δ_t time units, the knowledge K. Whenever the analysis performed by $\textsc{Check}_K()$ notices a variation in K, then a new plan is required by calling $\textsc{Select}_K()$ that implements the P activity.[3] $\textsc{Select}_K()$ implements a

[2] The upper bound N_K is a system parameter.

[3] For the sake of simplicity, we omit the details of $\textsc{Check}_K()$, which strictly depends on the specific implementation of K.

Algorithm 2. TD-learning based analysis and planning

```
1: procedure ActiveThread
2:     loop
3:         Wait $\Delta t$
4:         $b \leftarrow$ Check$_K$()
5:         if $b =$ true then
6:             for all $S_i \in Hosted$ do
7:                 Select$_K$($S_i$)
8: function Select$_K$(inout $S \in \mathbf{S}$)
9:     for all $d \in S.Deps$ do
10:        $m \leftarrow \arg\max_j \{H_t(S_j) \mid S_j \in Known \wedge S_j.Type = d\}$
11:        if ($\exists S_k \in S.In_t$ — $S_k.Type = d$) then
12:            if $H_t(S_k) < H_t(S_m)$ then
13:                $S.In_t \leftarrow S.In_t \setminus \{S_k\} \cup \{S_m\}$
14:        else
15:            $S.In_t \leftarrow S.In_t \cup \{S_m\}$
```

selection rule that, among the set of service candidates contained in K, properly chooses the set of services of interest that best achieve the goal stated in Sect. 3, as explained below.

The service selection rules are defined by means of a TD-learning method [15] that calculates, based on historical data, a *value function* that expresses how good a particular action is in a given situation. Indeed, *value functions* are used to properly select the action that provides the best possible *reward*, in a given situation. The general formulation of a TD method is:

$$E_t \leftarrow E_{t-1} + \alpha[R_t - E_{t-1}] \tag{5}$$

where E_t is the estimated value function at step t, $\alpha \in (0, 1]$ is the learning-rate parameter, R_t is the reward obtained by taking the action, and E_{t-1} is the value function calculated at the previous step – i.e., the *historical data*. In simple incremental averaging estimation methods [15], the learning-rate parameter α changes at every step and is calculated as $1/k$, where k is the number of accumulated rewards at step t.

Its rationale is to increasingly give more weight to the accumulated experience. 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. Specifically, we adopt a service selection rule implementing a *Two-layer* Hierarchical Reinforcement Learning (2HRL) [3] technique, which considers both data monitored locally, and data shared by the monitoring activities (M) of other peers.

Learning from local data – First layer aims at learning the behaviour of service candidates in $Known$ by relying on direct experience, without considering the information shared by other peers.

Let $GU_t^R(S_j)$ be the quality obtained while interacting with a given S_j at time t. This value is used to predict the quality $GU_{t+1}^E(S_j)$ expected from the same S_j at the next time step (i.e., at time $t + 1$). Specifically, at any given time t, the planning activity P calculates for all $S_j \in Known$, the expected quality $GU_t^E(S_j)$ by instantiating Eq. 5:

$$GU^E_t(S_j) = GU^E_{t-1}(S_j) + \alpha_j[GU^R_t(S_j) - GU^E_{t-1}(S_j)] \quad (6)$$

where $GU^E_t(S_j)$ is the estimated quality (i.e., value function) at time t, $\alpha_j \in (0, 1]$ is the learning-rate, $GU^R_t(S_j)$ is the quality (i.e., the reward) obtained by directly interacting with S_j at time t, and $GU^E_{t-1}(S_j)$ is the quality estimated at the previous step – i.e., the historical data.

As in Eq. 5, the learning-rate parameter α_j could be calculated as $\alpha_j \leftarrow 1/SR(S_j)$, where $SR(S_j)$ is the number of times that S_j has been invoked. However, while this averaging method is appropriate for stationary environments, it is not well suited for dealing with highly dynamic environments such as the one considered here. In fact, it would make the method not able to promptly react to sudden changes. In these cases, literature suggests to use a constant step-size parameter $\overline{\alpha}$ to be defined at design-time [15].

We introduce instead the notion of *learning-window*, i.e., a fixed time-window of size z, in which we apply TD technique. The idea is to subdivide the non-stationary problem into a set of smaller stationary problems, which can be solved by applying the averaging method. Indeed, calculating α_j as $\alpha_j \leftarrow 1/[SR(S_j) \bmod z]$ provides us with the flexibility of averaging methods while preventing long-past rewards to be overweighted.

Learning from shared data – Second layer aims at integrating into the learning process of each peer the information remotely monitored and shared by other peers. As described in Sect. 4.1, each M activity continuously monitors the local set of hosted services $Hosted$ and for each $S \in Hosted$, gossips every Δ_t information about it, e.g., $S.In_t$, $S.Out_t$, and $GU_t(S)$. However, since the gossip-based communication is epidemic, the data sent by a given peer p_i might be outdated when received by the other peers in the system. Indeed, monitored data is strongly time- and load-dependent (see Sect. 3), and might quickly change over time, due to highly dynamic changes occurring in the system. In this scenario, understanding how much the P activity of a peer can trust the received data is crucial for selecting the services $S_j \in Known$ that best fit the goal of maximizing ξ_t and ζ_t (see Sect. 3).

To this end, let $\mathbf{m}_{\bar{t}}$ be a message received by activity P at time $\bar{t} < t$, and let $GU_{\bar{t}}(S_j)$ be the quality advertised for each $S_j \in \mathbf{m}_{\bar{t}}$, i.e., $GU_{\bar{t}}(S_j)$ is the last quality value known for S_j. We estimate the *level of trust* $\tau_t(S_j)$ of the quality advertised by S_j by instantiating Eq. 5:

$$\tau_t(S_j) = \tau_{t-1}(S_j) + \alpha_j[F_t(S_j) - \tau_{t-1}(S_j)] \quad (7)$$

where $\tau_t(S_j)$ is the estimated *level of trust* (i.e., value function) at time t, $\alpha_j \in (0, 1]$ is the learning-rate calculated within the *learning-window*, and $F_t(S_j)$ (i.e., reward) measures how much accurate is the data received about S_j. Specifically, the *accuracy* is calculated as 1 minus the relative error between the value $\overline{GU}^R_t(S_j)$ at time t and the value $GU_{\bar{t}}(S_j)$ advertised at time $\bar{t}$:

$$F_t(S_j) = 1 - \frac{|\overline{GU}^R_t(S_j) - GU_{\bar{t}}(S_j)|}{|GU_{\bar{t}}(S_j)|} \quad (8)$$

where $\overline{GU}_t^R(S_j) = min(GU_t^R(S_j), GU_{\bar{t}}(S_j))$ is the normalized value of $GU_t^R(S_j)$, which forces $F_t(S_j) \in [0, 1]$.

Finally, the two TD-learning layers are combined in a new function H that, given a service $S_j \in Known$, computes its expected quality at time t:

$$H_t(S_j) = \tau_t(S_j) \cdot GU_{\bar{t}}(S_j) + (1 - \tau_t(S_j)) \cdot GU_t^E(S_j) \tag{9}$$

Informally, if trust is high (i.e., $\tau_t(S_j) \approxeq 1$) then shared data $GU_{\bar{t}}(S_j)$ is considered highly relevant in the evaluation of S_j. Viceversa, whenever the trust in shared data is low (i.e., $\tau_t(S_j) \approxeq 0$), then local experience $GU_t^E(S_j)$ is considered more relevant than shared data for evaluating S_j.

Function Select_K in Algorithm 2 shows how the 2HRL technique is used to build, for a given $S \in Hosted$, the set of bindings $S.In_t$ that achieves the global goal: maximizing ξ_t and ζ_t. The algorithm checks, for all dependences $d \in S.Deps$, what is the service $S_m \in Known$ that matches the dependency d and evaluates the maximum value of H_t (line 10). If a service S_k matching d is already in $S.In_t$, then S_m replaces S_k only if the former provides a better H_t than the latter (line 13). On the other hand, if $S.In_t$ does not contain any service matching d, then S_m is added to $S.In_t$ (line 15).

5 Experimental Evaluation

In this section we present a set of simulation experiments to assess the effectiveness of our approach. To this end, we implemented a large-scale simulation model for the PeerSim simulator [9]. 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.

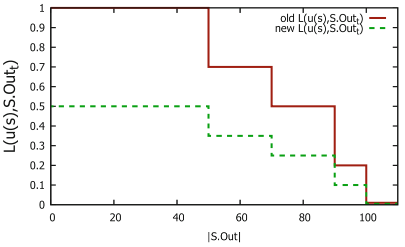
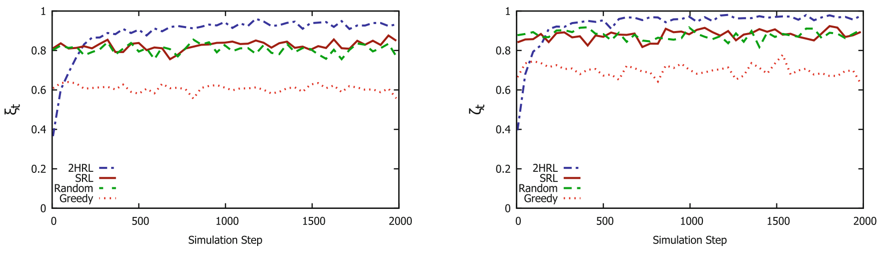


Fig. 1. Local quality functions

Model Parameters – We consider a system with N services and w different interface types $\mathbf{T} = \{T_1, \ldots, T_w\}$. For the sake of simplicity, we assume that each network node hosts a single service; hence the number of nodes inside the network is equal to the number of services, i.e., N. We create $\lfloor N/w \rfloor$ services of each type and, for each service S we randomly set the number of its dependencies. Specifically, 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



(a) Global quality, higher is better (b) Fairness, higher is better

Fig. 2. Static scenario

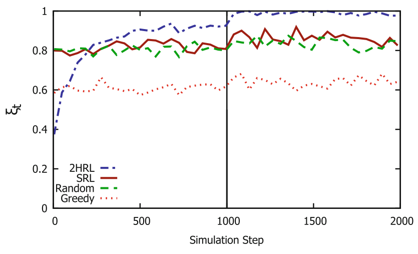
$S.Deps$ as a random subset of $\{S.Type+1, S.Type+2, ..., w\}$. Note that, according to this rule, services of type T_w have no dependencies. Finally, we assume that the load-dependent quality function $L()$ (see Sec. 3) of each service S is defined by the solid line in Fig. 1, and the *global quality* $GU_t(S)$ is defined such that it returns values in the range $(0, 1]$. Furthermore, other parameters of our HRL 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.

Performance Measures – As stated in Sect. 3 we evaluate the performance of our approach by means of the *global system quality* ξ_t, and the *fairness* ζ_t. In particular, ξ_t is computed as the average quality of all fully resolved services at step t, and ζ_t is computed as Jain's fairness index. Both ξ_t and ζ_t are higher-is-better metrics whose upper-bound is 1. All experiments are run by considering $N = 1000$ services, $w = 10$ interface types, and 2000 simulation steps. All results are computed by taking the average of 50 independent simulation runs.

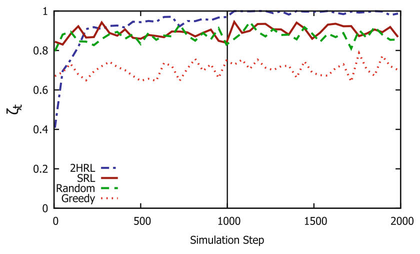
5.1 Simulation Results

Hereafter we report the simulation results obtained in different scenarios. To show the effectiveness of the proposed approach, we compare the results obtained by our approach with a set of state-of-the-art techniques based on different selection rules. Specifically, we experimented the following alternative selection rules: (i) a *Random* algorithm, which does not consider quality values but randomly selects, among the available services, those services that satisfy the required functional dependencies; (ii) a *Greedy* algorithm, which selects among the available services, those services with maximum quality; and (iii) a single-layer reinforcement learning (SRL) algorithm, which exploits past experience to predict the behavior of known services [12]. All the experiments show that our solution outperforms the results provided by these alternative selection rules.

Static scenario – This experiment considers a static scenario involving $N = 1000$ services of $w = 10$ different types. Figure 2 shows how ξ_t and ζ_t, calculated on the fully resolved assembly resulting from the application of different selection rules, vary in function of time t. In particular, it shows how our 2HRL approach

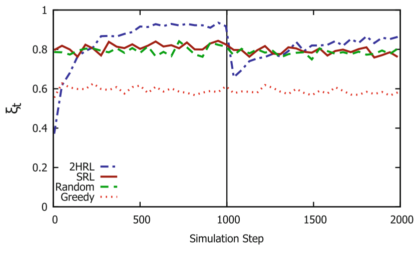


(a) Global quality, higher is better

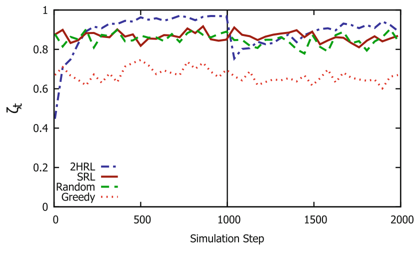


(b) Fairness, higher is better

Fig. 3. Peers leave the network



(a) Global quality, higher is better



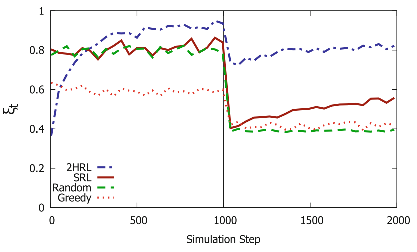
(b) Fairness, higher is better

Fig. 4. Peers join the network

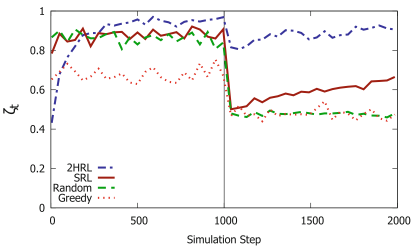
outperforms other selection rules by building a fully-resolved assembly whose ξ_t and ζ_t tend to upper-bound (see Fig. 2a and b). Next experiments aim at assessing the ability of 2HRL to self-adapt to changes that might happen in the networking environment. Indeed, open-end collections of distributed peer-to-peer nodes are necessarily prone to failures since autonomous nodes might suddenly leave/join the network at any time, as well as change their local quality.

Peers leave the network – This experiment considers a dynamic scenario where a number of nodes unexpectedly leave the networking environment. In particular, starting from the previous experimental setting – i.e., $N = 1000$ services of $w = 10$ different interface types – we randomly remove 500 nodes after $t = 1000$ simulation steps. Figure 3 reports how the different selection rules react to the environmental 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 global system quality. In fact, drastically removing half of the services (i.e., from 1000 to 500) from the network reduces the total load in the network and makes the 2HRL converging towards an optimal configuration evaluating $\xi_t \approx 1$ (see Fig. 3a) and $\zeta_t \approx 1$ (see Fig. 3b).

Peers join the network – On the other hand, this experiment considers a dynamic scenario where a number of new nodes join the networking environment. In particular, starting from the initial experimental setting – i.e., $N = 1000$



(a) Global quality, higher is better



(b) Fairness, higher is better

Fig. 5. Peers change the local quality function

services of $w = 10$ different interface types – we randomly add 500 new nodes at simulation step $t = 1000$. Figure 4 reports how the different selection rules react to the new environmental change. In this case, after an initial drop of ξ_t at time step $t = 1000$ the 2HRL selection rule allows services to learn from the new environment setting and to self-organize into fully-resolved assemblies that gradually re-establish a good level of global quality (see Fig. 4a) and fairness (see Fig. 4b). The initial drop of ξ_t at time step $t = 1000$ is mainly caused by the fact that the trust value $\tau_0(S) = 0.95$ makes the algorithm to select newly added services, which are evaluated better than the older ones. However, the 2HRL selection rule quickly learns from the new setting and converges towards a new optimal configuration within a few steps. Setting the initial trust $\tau_0(S)$ to a lower value – e.g., $\tau_0(S) = 0.5$ – would mitigate such an issue by allowing 2HRL to behave more conservatively while evaluating new discovered services.

Peers change the local quality function – Finally, this experiment considers a dynamic scenario where 500 randomly chosen services change at time step $t = 1000$ their quality function $L(\mathbf{u}(S), S.Out_t)$, as depicted by the dashed line in Fig. 1. Figure 5 reports how the different selection rules react to the new setting. Also in this case, we can notice that after an initial drop of ξ_t at time step $t = 1000$ the 2HRL selection rule allows services to quickly self-organize into fully-resolved assemblies that re-establish good level of global quality (see Fig. 5a) and fairness (see Fig. 5b).

6 Related Work

In this section we focus exclusively on approaches based on reinforcement learning. This methodology has been already used in literature to tackle service selection and load balancing problems [4,5,14,19]. Some of them (e.g.,[5,14]) are based on the approach previously presented in [12]. All these papers consider scenarios 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 peer does not know in advance the other peers (and the services they offer) in the environment, but discover them dynamically. Moreover, our services can have multiple dependencies, and we consider multiple load-dependent quality attributes.

Shaerf et al. [12] studied the process of multi-agent reinforcement learning in the context of load balancing in a distributed system, without use of either central coordination or explicit communication. They studied a system consisting of a certain number of agents using a finite set of resources, each having a time dependent capacity. The considered resource selection rules were purely local and the same for all agents. The presented experimental study considered a relatively small system of 100 agents. A notable outcome of the experiments was that making agents communicate each other to share information about the performance of resources was detrimental to the overall system performance. Galstyan et al. [4] presented a reinforcement learning model for adaptive resource allocation in a multi-agent system. The learning scheme is based on minority games on networks. Each agent learns over time the best performing strategies and use them to select the resource to be used. Zhang et al. [19] propose a multi-agent learning algorithm and apply it for optimizing online resource allocation in cluster networks. The learning is distributed to each cluster, using local information only and without access to the global system reward. Sugawara et al. [14] investigate multi-agent systems where agents can't identify the states of all other agents to assign tasks. The selection is done according to local information about the other known agents; however this information is limited and may contain uncertainty. Parent et al. [10] apply reinforcement learning for the dynamic load balancing of parallel data-intensive applications. Viewing a parallel application as a one-state coordination game in the framework of multi-agent reinforcement learning they are able to improve the classic job farming approach.

7 Conclusion

In this paper we have presented a self-organizing fully decentralized approach for the dynamic assembly of services in distributed peer-to-peer scenarios, whose goal is to guarantee a good overall quality for the delivered services, ensuring at the same time fairness among the participating peers. The core element of the proposed solution is the combined use of gossip protocols and reinforcement learning techniques. Gossip supports the decentralized information dissemination and decision making, whereas reinforcement learning enables each peer to dynamically learn from its experience the service selection rule to be followed, thus overcoming the lack of global knowledge. Besides, we explicitly take into account load-dependent quality attributes, which leads to the definition of a service selection rule that drives the system away from overloading conditions that could adversely affect quality and fairness. Thanks to these features, the system is able to build and maintain in a fully decentralised way an assembly of services that, besides functional requirements, fulfils global QoS requirements. Moreover, a set of simulation experiments shows how our solution self-adapts to occurring variations and quickly converges to feasible assemblies, which maintains the specified quality and fairness objectives.

Future work encompasses the extension of the experimental part with the inclusion of different real-world scenarios and other possible definitions of fairness. We also intend to extend 2HRL to cyber-physical systems, where a new set of challenging quality concerns have to be managed under severe resource constraints, e.g., energy consumption, real-time responsiveness.

References

1. Caporuscio, M., Grassi, V., Marzolla, M., Mirandola, R.: GoPrime: a fully decentralized middleware for utility-aware service assembly. IEEE Trans. Softw. Eng. **42**(2), 136–152 (2016)
2. Cheng, B.H.C., et al.: 08031 - software engineering for self-adaptive systems: a research road map. In: Dagstuhl Seminar Proceedings Software Engineering for Self-Adaptive Systems, vol. 08031. IBFI (2008)
3. Erus, G., Polat, F.: A layered approach to learning coordination knowledge in multiagent environments. Appl. Intell. **27**(3), 249–267 (2007)
4. Galstyan, A., Kolar, S., Lerman, K.: Resource allocation games with changing resource capacities. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems AAMAS 2003, pp. 145–152 (2003)
5. Ghezzi, C., Motta, A., Panzica La Manna, V., Tamburrelli, G.: QoS driven dynamic binding in-the-many. In: Heineman, G.T., Kofron, J., Plasil, F. (eds.) QoSA 2010. LNCS, vol. 6093, pp. 68–83. Springer, Heidelberg (2010)
6. Jain, R.K., Chiu, D.M.W., Hawe, W.R.: A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical report DEC-TR-301, Digital Equipment Corporation, September 1984
7. Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.M., van Steen, M.: Gossip-based peer sampling. ACM Trans. Comput. Syst. **25**(3) (2007). Article No. 8
8. Jiang, L., Xu, G.: Modeling and analysis of software aging and software failure. J. Syst. Softw. **80**(4), 590–595 (2007)
9. Montresor, A., Jelasity, M.: PeerSim: a scalable P2P simulator. In: Proceedings of the 9th International Conference on Peer-to-Peer (P2P 2009), Seattle, WA, pp. 99–100, September 2009
10. Parent, J., Verbeeck, K., Lemeire, J., Nowe, A., Steenhaut, K., Dirkx, E.: Adaptive load balancing of parallel applications with multi-agent reinforcement learning on heterogeneous systems. Sci. Program. **12**(2), 71–79 (2004)
11. Paschalidis, I.C., Tsitsiklis, J.N.: Congestion-dependent pricing of network services. IEEE/ACM Trans. Netw. **8**(2), 171–184 (2000)
12. Schaerf, A., Shoham, Y., Tennenholtz, M.: Adaptive load balancing: a study in multi-agent learning. J. Artif. Intell. Res. **2**, 475–500 (1995)
13. Schuhmann, S., Herrmann, K., Rothermel, K., Boshmaf, Y.: Adaptive composition of distributed pervasive applications in heterogeneous environments. ACM Trans. Auton. Adapt. Syst. (TAAS) **8**(2), 10:1–10:21 (2013)
14. Sugawara, T., Fukuda, K., Hirotsu, T., Sato, S., Kurihara, S.: Adaptive agent selection in large-scale multi-agent systems. In: Yang, Q., Webb, G. (eds.) PRICAI 2006. LNCS (LNAI), vol. 4099, pp. 818–822. Springer, Heidelberg (2006)
15. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
16. Voulgaris, S., Jelasity, M., van Steen, M.: A robust and scalable peer-to-peer gossiping protocol. In: Moro, G., Sartori, C., Singh, M.P. (eds.) AP2PC 2003. LNCS (LNAI), vol. 2872, pp. 47–58. Springer, Heidelberg (2004)

17. Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M.: On patterns for decentralized control in self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 7475, pp. 76–107. Springer, Heidelberg (2013)
18. Yoon, K.P., Hwang, C.L.: Multiple Attribute Decision Making: An Introduction, vol. 104. Sage Publications, Thousand Oaks (1995)
19. Zhang, C., Lesser, V., Shenoy, P.: A multi-agent learning approach to online distributed resource allocation. In: Proceedings of Twenty-First International Joint Conference on Artificial Intelligence (IJCAI 2009), vol. 1, pp. 361–366 (2009)