QoS-Aware Fully Decentralized Service Assembly

Vincenzo Grassi
Dip. di Ingegneria Civile e
Ingegneria Informatica
Università di Roma Tor Vergata, Italy
vgrassi@info.uniroma2.it

Moreno Marzolla
Dip. di Informatica–Scienza e Ingegneria
Università di Bologna, Italy
marzolla@cs.unibo.it

Raffaela Mirandola
Dip. di Elettronica e Informazione
Politecnico di Milano, Italy
mirandola@elet.polimi.it

Abstract—**Large, distributed software systems are increasingly common in today geographically distributed IT infrastructures. A key challenge for the software engineering community is how to efficiently and effectively manage such complex systems. Extending software services with autonomic capabilities has been suggested as a possible way to address this challenge. Ideally, self-management capabilities should be based on fully distributed, peer-to-peer (P2P) architectures in order to try to overcome the scalability and robustness problems of centralized solutions. Within this context, we propose an approach for the adaptive self-assembly of distributed services, based on a simple epidemic protocol. Our approach is based on the three-layer reference model for adaptive systems, and is centered on the use of a gossip protocol to achieve decentralized information dissemination and decision making. The goal of our system is to build and maintain an assembly of services that, besides functional requirements, is able to fulfill global quality of service (QoS) and structural requirements. A set of simulation experiments is used to assess the effectiveness of our approach in terms of convergence speed towards the optimal solution, and resilience to failures.**

I. Introduction

One of the major challenges for today software engineering is the management of large and complex computing systems characterized by a high degree of physical distribution. Examples of such systems arise in many application domains, such as sensor networks, intelligent transportation systems, ambient intelligence [1].

The high number of components and the intrinsic dynamism of these systems (where the quality and number of available resources can rapidly change) push scalability and complexity issues well beyond traditional scenarios. The use of autonomic capabilities has been suggested as a possible solution to overcome these problems [2]–[4]. The autonomic computing paradigm enables the system to automatically self-configure in response to variations of operating conditions, thus guaranteeing short reaction times and avoiding slow, costly and error-prone manual interventions. Scalability and robustness considerations call for fully decentralized solutions to the implementation of these autonomic capabilities [5]. Indeed, for the systems we are considering, centralized control can seriously hinder scalability and fault-tolerance, and can also be difficult or even impossible to achieve.

The goal of this paper is thus to provide a contribution towards the design and implementation of decentralized solutions for the autonomic management of large and highly dynamic distributed systems. To abstract from characteristics of specific application domains, we model the considered system as a multi-agent system, where a set of peers cooperatively work to accomplish specific tasks. Each peer possesses the know-how to perform some tasks (offered services), but generally requires services offered by other peers to carry out these tasks. Hence, we focus on the problem of devising a self-assembly procedure among services offered by the peers, whose goal is to match required and provided services. Besides the functional requirement of matching required services with offered services, we assume that the system operates under non functional requirements concerning the quality of the offered services (QoS) (e.g. performance, dependability, cost). Hence, the multi-agent system must be able to select, among the set of functionally feasible assemblies, an assembly that makes it able to fulfill a global QoS goal.

According to decentralization principles, the solution we propose to achieve these goals is characterized by the following properties [6]:

- absence of external control, so that the self-(re)configuration process is initiated internally;
- dynamic operation, so that the system is continuously able to adapt itself to requirement or environment changes (including arrival of new peers or failure of existing ones);
- absence of central control, where the assembly among peers is only maintained through local decision-making with dissemination of information in order to improve the decision-making process.

The general architectural scheme we adopt to design a system able to achieve these properties is based on the three-layers architecture for self-adaptive systems proposed in [7] and applied to a distributed system in [8]. Within this framework, we focus on the intermediate layer and architect it according to a *gossip protocol* [9]–[11] to support information dissemination and decentralized decision-making. Simulation results show that our solution is able to produce a fully resolved service assembly very quickly; also, the whole system can quickly restructure itself to cope with node failures.

The paper is organized as follows. Sections II and III describe the system model, whose reference self-adaptive architecture is presented in IV. Then, we illustrate in Section V the system operations that implement the decentralized self-assembly, and present in Section VI experimental results obtained through simulation. We survey related works in

Section VII. Finally, we present conclusions and future works in Section VIII.

II. System Model

In this section we define the system model and introduce the terminology and notation used in the rest of the paper. Then, we introduce the notion of *compound utility* and show through some examples how it can be used to enforce both QoS and structural constraints.

We consider a system consisting of a set of distributed peers offering different services. In the rest of the paper, for the sake of simplicity and readability, we assume that each peer hosts a single service, so we will use quite interchangeably the terms peer and service, and with a slight abuse of notation will denote by the same name both a service and the peer offering it. The model can be extended to peers offering different types of services in quite a straightforward way.

The system we consider thus consists of a set $\mathcal{S}$ of N distributed services $\mathcal{S} = \{S_1, \dots, S_N\}$. Services can be located anywhere and can communicate each other through a network. Each service has a *provided interface*, which is an interface through which it provides functionalities to clients. Also, each service has a set of *required interfaces*, that must be bound to the provided interfaces of other services. Formally, a service S is a tuple $S = (t, \delta, u, \mathit{Pred}, \mathit{Succ})$ where:

- $t \in \mathcal{T}$ is the type of the provided interface (we say that t is the type of S). We assume that there are $T \geq 1$ different service types $\mathcal{T} = \{1, \dots, T\}$; each service type basically defines a set of input/output types, and represents the kind of service provided by S.
- $\delta \subseteq \mathcal{T}$ is the set of dependencies of S (if $\delta = \emptyset$, then S has no dependencies). The set δ contains the types of the required dependencies of S; therefore, for each $d \in \delta$, S must be bound to a service of type d in order to be executed. Note that the dependency set δ does not contain duplicates, meaning that a service may depend at most once on any specific interface type. We assume that the set of dependency types δ is fixed for each service and known in advance.
- u is the utility of service S in isolation, which can be considered as a measure of one or more QoS (e.g., reliability, cost, execution time) or structural attributes, depending only on the internal characteristics of S. If S has a non empty set of dependencies, then u gives only a partial view the overall utility of S, which also depends on the utility of the services used to resolve them; therefore, the concept of *compound utility* will be introduced shortly.
- $\mathit{Pred} \subseteq \mathcal{S}$ is the set of services which are currently bound to S to resolve some of its dependencies.
- $\mathit{Succ} \subseteq \mathcal{S}$ is the set of other services S is currently bound to, to resolve some of their dependencies.

In the following, given a service $S \in \mathcal{S}$ we denote with $t[S]$, $\delta[S]$, $u[S]$, $\mathit{Pred}[S]$ and $\mathit{Succ}[S]$ the service type, the set of dependencies, the local utility and the sets Pred and Succ of S, respectively.

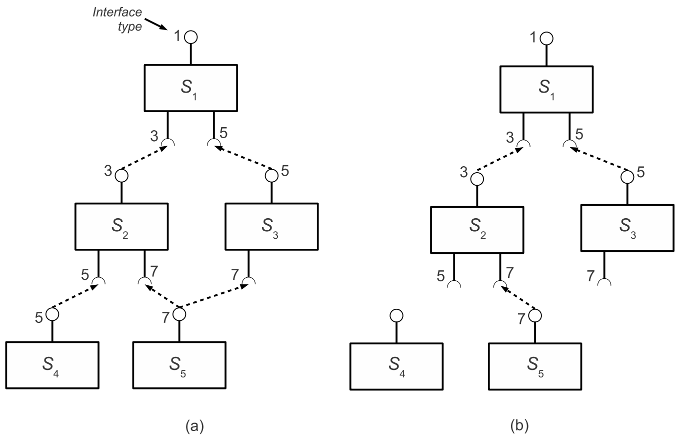

Fig. 1. Assembly examples

We point out that while $t[S]$ and $\delta[S]$ represent static information that do not change throughout the service lifetime, $u[S]$, $\mathit{Pred}[S]$ and $\mathit{Succ}[S]$ represent dynamic state information whose value can change because of variations of some internal S characteristic or of the services it is bound to.

A service assembly $\mathcal{A}$ is an acyclic graph $\mathcal{A} = (\mathcal{S}, \mathcal{E})$, where $\mathcal{E} \subseteq \mathcal{S} \times \mathcal{S}$ is the set of resolved dependencies. Specifically, a directed edge $(S_i, S_j) \in \mathcal{E}$ denotes that S_i is used by S_j to resolve one of its dependencies. It is required that S_j has a dependency on $t[S_i]$, that is $t[S_i] \in \delta[S_j]$. We allow the same service to be used as a dependency multiple times by different other services. By definition, $\mathit{Pred}[S] = \{S' \in \mathcal{S} \mid (S, S') \in \mathcal{E}\}$ and $\mathit{Succ}[S] = \{S'' \in \mathcal{S} \mid (S'', S) \in \mathcal{E}\}$.

A service S is *fully resolved* in a given assembly $\mathcal{A}$ if either:

- S has no dependencies ($\delta[S] = \emptyset$); or
- for all $d \in \delta[S]$ there exists a service $S' \in \mathit{Pred}[S]$ of type d that is itself fully resolved.

A service S is *partially resolved* in a given assembly $\mathcal{A}$ if it is not fully resolved in $\mathcal{A}$. This means that S has a non empty list of dependencies, and at least one dependency is either not matched, or is matched by some service which itself is not fully resolved.

As an example, in Fig. 1 we show two assembly involving the services $\{S_1, S_2, S_3, S_4, S_5\}$ using the standard UML 2.0 component diagram notation. Service S_1 in Fig. 1 (a) is fully resolved, while service S_1 in 1 (b) is not, since it is bound to S_2 and S_5 which have missing dependencies.

III. Compound Utility Computation

The last important part of our model is the definition of a *compound utility* function $U(S)$, that associates a scalar value to any assembly rooted at service S, based on both the local utility of S and the utilities of the services it is bound to. The goal of the system we are proposing is to build and maintain fully resolved assembly such that the compound utility of each service is maximized (or at least greater than a specified threshold). In the next subsection we show how, by suitably defining the meaning of the local utility $u[S]$ and the function $U(S)$, it is possible in this way to drive the system towards the construction of assembly with specific non-functional properties.

$U(S)$ is defined as a function that recursively calculates the compound utility of S in terms of $u[S]$ and the compound utility of the services it is bound to. If S is not fully resolved, then it must be $U(S) = -\infty$. We give below some concrete examples of definition of $u[S]$ and $U[S]$ for some specific QoS or structural attributes.

Reliability-based Compound Utility: The reliability of a service S is the probability that S correctly completes its task, for a given service request. Let $u[S]$ denote the internal reliability of S, that is the probability that no internal failure occurs when S is executed. Furthermore, for each dependency $S' \in \mathit{Pred}[S]$ bound to S, let $n[S']$ be the average number of times service S' is invoked during the execution of S. The value of $n[S']$ can be estimated by a monitoring activity performed locally by the peer hosting S; $n[S']$ will likely only depend on the type $t[S']$. If we define

$$U_r(S) = \begin{cases} -\infty & \text{if } S \text{ not fully resolved} \\ u[S] \times \prod_{S' \in \mathit{Pred}[S]} U_r(S')^{n[S']} & \\ & \text{otherwise} \end{cases} \tag{1}$$

then $U_r(S)$ represents the overall reliability of the assembly rooted at S (if $\mathit{Pred}[S] = \emptyset$, the product is set to one and therefore $U_r(S) = u[S]$).

Cost-based Compound Utility: The average cost of a service S is the overall 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). We distinguish two cases. In the first one, we assume that an additive cost is incurred for each single invocation of a service $S' \in \mathit{Pred}[S]$ resolving one of the S dependencies. This is reasonable for costs referring, for example, to energy consumption. Let $-u[S]$ represent the cost of S; note that $u[S]$ is negative, to ensure that the utility is a *higher is better* metric. The cost-based compound utility $U_c(S)$ of S can be computed as:

$$U_c(S) = \begin{cases} -\infty & \text{if } S \text{ not fully resolved} \\ u[S] + \sum_{S' \in \mathit{Pred}[S]} n[S'] \times U_c(S') & \\ & \text{otherwise} \end{cases} \tag{2}$$

(if $\mathit{Pred}[S] = \emptyset$, the sum is set to zero and $U_c(S) = u[S]$). In the second case, modeling other types of cost (e.g., monetary cost), a flat cost model could be adopted, 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 compound utility $U_f(S)$ of S can be computed as:

$$U_f(S) = \begin{cases} -\infty & \text{if } S \text{ not fully resolved} \\ u[S] + \sum_{S' \in \mathit{Pred}[S]} U_f(S') & \\ & \text{otherwise} \end{cases} \tag{3}$$

Response Time Compound Utility: The average response time of a service S is the overall average time needed to fulfill one service request addressed to S. Let $-u[S]$ denote the average time spent within service S; again, the utility $u[S]$ is negative so that the compound utility $U(S)$ is a higher is better metric. The response time compound utility of S can be computed by defining:

$$U_t(S) = \begin{cases} -\infty & \text{if } S \text{ not fully resolved} \\ u[S] + \sum_{S' \in \mathit{Pred}[S]} n[S'] \times U_t(S') & \\ & \text{otherwise} \end{cases} \tag{4}$$

Structural Requirements: Besides QoS requirements, one could be interested also in structural requirements about the resulting assembly of services (enforcing, for example, some specific architectural style). These requirements could concern *local* properties (e.g., the number of dependencies solved by a given service S should not be greater than a given threshold, to avoid service overloading), or *global* properties (e.g., the overall assembly should conform to a pipeline structure, where each offered service is bound to only one required service). Global constraints seem more difficult to be enforced in a system where each peer has only a limited local knowledge of the whole structure. However, we give below examples showing that by suitably defining $u[S]$ and $U(S)$, a systems that tries to maximize $U(S)$ actually drives itself towards the construction of an assembly the fulfills local or global structural constraints, in the latter case limited to those constraints that can be recursively defined.

Let us consider a local constraint on the maximum number of dependencies that can be resolved by S, meaning that S can be used by at most $D_{\max}$ other services to resolve their dependencies. To achieve this, S uses the set $\mathit{Succ}[S]$ to keep track of the number of other services it is currently bound to. S reports a local utility $u[S] = 1$ to the first $D_{\max}$ services that are using S as a dependency; any further service enquiring the local utility of S will get $u[S] = 0$. If the compound utility $U_m(S)$ is defined as:

$$U_m(S) = \begin{cases} -\infty & \text{if } S \text{ not fully resolved} \\ u[S] \times \prod_{S' \in \mathit{Pred}[S]} U_m(S') & \\ & \text{otherwise} \end{cases} \tag{5}$$

then the system will be driven towards an assembly where each service is used no more than $D_{\max}$ (if possible).

Let us consider instead a global structural constraint enforcing a pipeline structure on a fully resolved assembly. The utility $u[S]$ is set to 1 if S has at most one dependency ($\delta[S] \leq 1$), and 0 otherwise:

$$u[S] = \begin{cases} 1 & \text{if } \delta[S] \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

Then, the compound utility $U_p(S)$ can be defined as:

$$U_p(S) = \begin{cases} -\infty & \text{if } S \text{ not fully resolved} \\ u[S] \times \prod_{S' \in \mathit{Pred}[S]} U_p(S') & \text{otherwise} \end{cases} \quad (6)$$

Eq. (6) yields $U_p(S) = 1$ if and only if either S has no dependencies, or S is fully resolved and its direct and indirect dependencies are organized as a chain (pipeline structure).

We point out that in the case of the pipeline structural requirement described by the utility function of Eq. (6), a value $U_p(S) = -\infty$ denotes that no fully resolved assembly rooted at S has been built so far (irrespective of any structural constraint). A value $U_p(S) = 0$ denotes that a fully resolved assembly has been built, but the structural constraint has not been satisfied; a better assembly may or may not be identified as the algorithm progresses. Finally, a value $U_p(S) = 1$ denotes that a fully resolved assembly satisfying the pipeline constraint has been found.

Other Considerations: We have given above some practical examples of utility-based QoS or structural measures; additional metrics can be defined in similar ways, and also some useful extensions can be considered. The metrics we introduced are based on the implicit assumption that the utility $u[S]$ of a service S does not depend on the number of services S is bound to; in other works, $u[S]$ does not depend on the size or content of $\mathit{Succ}(S)$. While this may be appropriate for some metrics (e.g., cost, reliability), it may be too restrictive for others. In fact, the response time of a service S may actually depend on the number of clients, which can be different over time. As we will see in Section V, our algorithm already deals with variable, state-dependent utility values, since the compound utility computed by each node is periodically sent to all neighbors.

Additionally, the above definitions allow the computation of the compound utility of a service in terms of a single QoS attribute. In some cases, one could be interested in a definition of utility depending on the value of more than one QoS attribute. For these cases, it is possible to define a *multi-attribute compound utility* of S by taking the weighted sum of single-attribute utility values. If M QoS attributes $U_1(S), \ldots, U_M(S)$ are computed for a single service S, it is possible to compute a single value $U(S)$ as:

$$U(S) = \sum_{i=1}^{M} w_i U_i(S)$$

for a suitable choice of weights such that $\sum_{i=1}^{M} w_i = 1$, $0 \leq w_i \leq 1$. If the QoS attributes take values in different domains, they should be normalized in the same range (e.g., $[0, 1]$) before computing the weighted average.

Table I summarizes the notation used in this paper; the table also includes additional symbols which will be introduced in the next sections.

TABLE I
SYMBOLS USED IN THIS PAPER

$\mathcal{T}$	Set of service types $\mathcal{T} = \{1, \ldots, T\}$
$t[S]$	Service type of S
$\delta[S]$	Set of dependencies of S
$\mathit{Pred}[S]$	Set of peers that S is bound to, to resolve its own dependencies
$\mathit{Succ}[S]$	Set of peers that have a binding with S to resolve one of their dependencies
$u[S]$	Utility of service S
$U(S)$	Compound utility of the assembly rooted at S
D	Number of dependencies per node
N	Number of services (peers)
K	Number of neighbors
C	Cache size
N_{opt}	Number of services of each type with utility 1
R_t	Fraction of fully resolved services at step t
U_t	Average utility of fully resolved services at step t

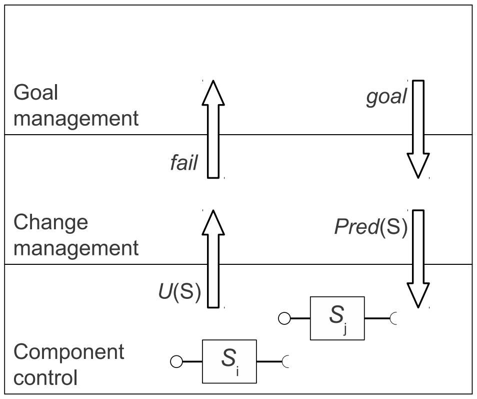

Fig. 2. Reference Architecture

IV. System Architecture

A reference model for the architecture of a self-adaptive software system has been presented in [7], where it is suggested to architect the system along three interacting layers. The bottom layer (*component control*) is concerned with adaptation at the level of single components (*i.e.*, peers in our settings). The middle layer (*change management*) is responsible for globally managing the system consisting of components at the lower layer, by issuing adaptation plans that make the system able to meet some pre-defined goal. The definition of new goals and possibly of new adaptation plans is under the responsibility of the upper layer (*goal management*).

We adopt this reference model for the architecture of our multi-agent P2P system, as illustrated in Fig. 2 and detailed below.

Bottom Layer: Each single peer S at the *component control* layer is responsible for possibly implementing suitable internal adaptation actions aimed at maintaining its local utility level, or improving it. In any case, S keeps updated the value of $u[S]$ through some suitable monitoring activity, and communicates with all its peers in *Pred*$[S]$ and *Succ*$[S]$ to keep updated the value of $U(S)$, as detailed in Section V. Information about variations in the value of $U(S)$ flows also toward the middle layer. A variation of $U(S)$ can trigger at the middle layer the construction of a new assembly, if it makes the current assembly no longer able to fulfill a given non functional requirement.

Middle layer: The *change management* layer is instead responsible in our architecture for building and maintaining at each peer S the two state components *Pred*$[S]$ and *Succ*$[S]$, so that the resulting whole assembly is able to fulfill the non functional requirements coming from the upper layer. To this end, when the compound utility of some peer at the bottom layer changes, the middle layer must check whether this impairs the ability of fulfilling the existing requirements. In the positive case, the middle layer determines a new suitable assembly. Similarly, when the upper layer notifies a new goal, the middle layer must check whether the already determined assembly is able to fulfill it. In the negative case, the middle layer determines a new suitable assembly. In case no such assembly exists, a fail notification is sent to the upper layer.

Upper layer: The *goal management* layer manages the functional and non functional goals to be fulfilled and inform the middle layer about them. In particular, in our architecture two different kinds of non functional goals can be communicated to the middle layer:

- *max* goals, that are fulfilled by determining an assembly that maximizes some suitably defined compound utility of the offered services;
- *more-than* goals, that are fulfilled by determining an assembly whose compound utility is greater than a specified threshold.

If a fail notification is received from the middle layer for some specific goal, the upper layer is responsible for implementing a suitable response according to its own policies (e.g., by issuing some different goal).

In this work, we focus on the design of the operations of the middle layer and its interactions with the bottom layer. Operations of the upper layer are instead out of the scope of this work. Our design is based on a fully decentralized approach, that makes the resulting system robust and scalable in the presence of events like: arrival of new requirements from the upper layer, upgrade/downgrade of peer utility (including the extreme case of peer failure) or arrival of new peers at the bottom layer. To this end, the middle layer of our architecture operates according to a gossip schema, as described in Section V. This allows a fully decentralized information dissemination and decision-making about the construction and maintenance at each peer of the two sets *Pred*$[S]$ and *Succ*$[S]$.

V. System Operations

In this section we describe how the middle and bottom layer of our architecture operate to dynamically build and maintain in a fully decentralized way an assembly of services able to fulfill a specified QoS goal. We first present in Section V-A the gossip-based algorithm implemented at the middle layer. Then, we present in Section V-B further details on operations implemented at the middle and bottom layer, and discuss some issues concerning the system operations.

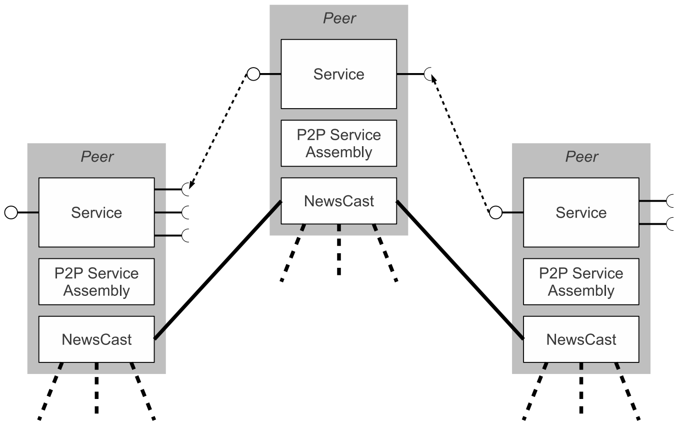

Fig. 3. High level structure of peers

A. Middle Layer Core Algorithm

Given a set of services $\mathcal{S}$ and a utility function $U(\cdot)$, the middle layer decentralized algorithm builds an assembly where each service $S \in \mathcal{S}$ is fully resolved and the value of $U(S)$ is monotonically increased, until it reaches its maximum value or, at least, overcomes a given threshold. As described in Section III, by suitably defining the utility function it is possible to fulfill the QoS or structural goals coming from the upper layer. The algorithm is based on a gossip scheme [9], [11] which iteratively resolves the dependencies of each service using local information only.

Figure 3 shows the internal structure of peers in our P2P network. Each peer consists of two P2P protocols (labeled NEWSCAST and *P2P Service Assembly*), and the actual service offered by the peer on top of them.

The NEWSCAST protocol is responsible for maintaining an overlay network over the set of peers. Each peer maintains a *local view* of the system that consists of a set of peers it can exchange messages with. The local view is constantly updated, so that a peer is always provided with “fresh” list of neighbors. Updating the local views is also necessary to maintain a fully connected overlay in presence of node and link failures. Maintenance of the overlay is accomplished by the NEWSCAST algorithm described in [10]. Each peer maintains a set of K neighbors, where K is a predefined constant; periodically, each peer merges its list with that of a randomly chosen neighbor, keeping the most K recently added links and dropping older ones. This simple protocol exhibits many useful features: the list of neighbors it produces is a good approximation of a true random sample among all peers; moreover, the protocol is highly resilient and can maintain a fully connected overlay in presence of node or link faults.

Algorithm 1 P2P Service Assembly algorithm executed by S_i

```
    // Local variables
 1: $i \leftarrow$ GETPROCID()
 2: $Pred[S_i] \leftarrow \emptyset$

 3: procedure ACTIVETHREAD
 4:     loop
 5:         Wait $\Delta t$
 6:         for all $S_j \in$ GETNEIGHBORS() do
 7:             Send $\langle Pred[S_i] \cup \{S_i\} \rangle$ to $S_j$

 8: procedure PASSIVETHREAD
 9:     loop
10:         Wait for message $\langle \mathcal{B} \rangle$ from $S_j$
11:         for all $S \in \mathcal{B}$ do
12:             if $t[S] \notin \delta[S_i]$ then
13:                 continue
14:             if there exists $S_j \in Pred[S_i]$ s.t. $t[S_j] = t[S]$ then
15:                 if $U(S) > U(S_j)$ then
16:                     $Pred[S_i] \leftarrow Pred[S_i] \setminus \{S_j\} \cup \{S\}$
17:             else
18:                 $Pred[S_i] \leftarrow Pred[S_i] \cup \{S\}$
```

Algorithm 1 describes the core of the gossip algorithm for service assembly executed by each peer. By executing this algorithm all peers collectively implement the operations of the reference architecture middle layer. The goal of Algorithm 1 is to find a binding of the service to its required dependencies, such that the compound utility of the assembly is maximized. With a small modification, described later, the algorithm can be stopped as soon as the compound utility of a service reaches a given threshold.

Algorithm 1 includes an initialization phase and two concurrent threads: an active thread which starts an interaction by sending a message to a neighbor, and a passive thread which responds to messages received from other peers. The set of neighbors is provided by the underlying NEWSCAST service.

During initialization (lines 1–2) the variable i is initialized with the ID of the peer executing the algorithm; also, the set $Pred[S_i]$ of services bound to S is set to empty.

The active thread is extremely simple: every Δt time units, S_i sends a message to all its neighbors. The message payload is a set of services, containing the list of currently bound dependencies $Pred[S_i]$ plus S_i itself.

The passive thread listens for messages coming from other peers. Upon receiving a message containing the set L, S_i checks all services $S \in L$ to see whether some of them can be used to fill its own dependencies. Note that the cardinality of L is bounded by $1 + \max_i |\delta[S_i]| \leq 1 + T$, since each service can have at most T dependencies. If the type of $S \in L$, $t[S]$, is not required as a dependency for S_i (line 12), then service S is ignored. If $t[S]$ is required as a dependency, additional conditions must be checked. If no service of type $t[S]$ is currently bound to S_i, then S is added to $Pred[S_i]$ (line 18). If service S_j of the same type of S is already bound to S_i, then S replaces S_j only if the former provides a better compound utility than the latter (line 16).

Figure 4 shows an example of algorithm execution over a set of eight services $S_1, \ldots, S_8$. UML 2.0 component diagrams represent the services, with provided and required interfaces labeled with the interface type. The compound utility of some of the services is shown inside each block; note that S_8 is the only service which is not fully resolved, therefore $U(S_8) = -\infty$. The initial situation is shown in Fig. 4 (a); we assume that S_2 executes the active thread, and S_1 and S_3 are its neighbors according to the overlay network (not shown). First, S_2 sends the list $Pred[S_2] \cup \{S_2\} = \{S_2, S_5, S_6\}$ to its first neighbor S_1. S_1 observes that it can replace its dependency S_4 with S_2 (both have type 2), since S_2 provides a higher compound utility than S_4. Therefore, the services are rewired according to Fig. 4 (b). Now S_2 sends the same list $\{S_2, S_5, S_6\}$ to the other neighbor S_3. S_3 then discovers that it can replace its dependency S_7 with S_6, since it provides higher compound utility than the existing dependency. Fig. 4 (c) shows the final wiring of the services.

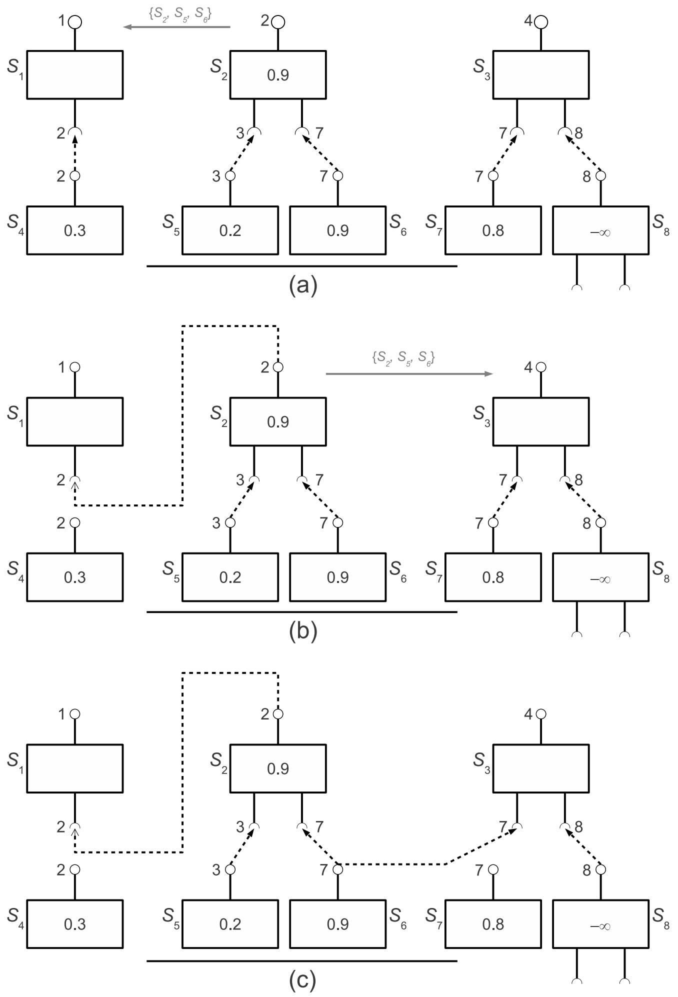

Fig. 4. Example of execution of Algorithm 1. At the beginning, service S_2 has S_1 and S_3 as neighbors.

B. Implementation Details

In this section we present operations that are locally implemented at each peer (i.e., at the bottom layer of the reference architecture), and are related with the execution of the middle layer algorithm. Moreover, we give some implementation details which are omitted from the pseudocode shown in Algorithm 1 and discuss some further issues about system operations.

Compound Utility Computation: We discuss here how each peer can efficiently compute its compound utility. In principle, each service S_i may compute $U(S_i)$ by asking the compound utility to all dependencies (which are kept in the *Pred*$[S_i]$ set) and aggregating the results (see the examples from Section II). However, this is extremely inefficient since lots of messages would be sent during the execution of the P2P algorithm.

The message traffic can be reduced by allowing each service to cache its compound utility, and notify changes to all services currently bound to it. To do so, each service S_i maintains the set *Succ*$[S_i]$ of all services which are currently using S_i to resolve one of their dependencies. Each time the value $U(S_i)$ changes, S_i sends an update to all services in *Succ*$[S_i]$ as in the Observer pattern [12]. The value of $U(S_i)$ changes when $u[S_i]$ changes, or a service is added/removed from the set *Pred*$[S_i]$ (lines 16 and 18 of Algorithm 1). Note that $U(S_i)$ may also change when some dependency stops working: when this happen, S_i may become partially resolved, and $U(S_i)$ becomes $-\infty$. Service S_i can discover failures by periodically polling all dependencies in *Pred*$[S_i]$.

When a new dependency S is added (line 18), then *Succ*$[S]$ must be updated to include the new client. If service S replaces the existing dependency S_j (line 16), then the client S_i must be removed from *Succ*$[S_j]$ and added to *Succ*$[S]$.

Multiple Queries: In general, the system may allow multiple users to request creation of fully resolved assemblies satisfying different QoS goals. For example, a user may require a service with a given external interface type, providing the higher reliability; another user may require a service with a different interface type providing the lower cost; and so on. Therefore, in general multiple instances of the P2P protocol must be active at the same time, each one building an assembly with the requested features. The information about which kind of utility must be maximized can be injected by the user requesting the assembly, and disseminated across the network using a flooding algorithm.

Stopping and Failure Criterion: Since the utility of the services are not known in advance, the system has no way to know what the maximum compound utility is; furthermore, the maximum utility which can be achieved also depends on the root service. Therefore, a suitable stopping criterion should be devised in order to eventually interrupt the active thread. Stopping the loop after a fixed timeout is simple but shifts the burden on the appropriate selection of the timeout value. A more suitable stopping criterion can be that of monitoring the compound utility of the assembly, and stop when the increment over time becomes sufficiently small (Section VI gives an intuitive justification to that). If the user is not interested in maximizing the utility of the assembly, but only looks for an assembly providing a compound utility above a given threshold, then the root service can interrupt the loop on the active thread of Algorithm 1 as soon as the compound utility reaches the threshold.

The definition of a stopping criterion is also related with the identification of the occurrence of a failure with respect to a given goal (i.e. no assembly exists that fulfills it). Given a goal, it is easy to prove that if an assembly exists that fulfills it, Algorithm 1 is eventually able to build it. However, things are more complex when such an assembly actually does not exist. In this respect, we note that a failure with respect to a *max* goal can only occur when no fully resolved assembly can be built. On the other hand, a failure with respect to a *more than* goal can only occur when the utility achievable with the assembly that currently fulfills the corresponding *max* goal is below the required threshold. The occurrence of such failures (to notify them to the goal management layer of our architecture) could be stated according to considerations similar to those discussed above for stopping criteria.

VI. Experimental Evaluation

In order to test the effectiveness of Algorithm 1, we implemented it using the PeerSim [13] simulator. PeerSim is a free Java package designed for efficient simulation of Peer-to-Peer protocols. PeerSim provides two types of simulation engines: *discrete-event* and *cycle-based*. The discrete-event engine can be used to analyze scenarios where the duration of simulated actions is important, e.g., to accurately model end-to-end message propagation delays using specific network technologies. The cycle-based engine implements the time-stepped simulation model, in which all interactions happen at specific time steps. 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 performance metric (number of interactions) has the advantage of being independent from the details of the underlying hardware and network infrastructure.

Model Parameters: We consider a system with N services with T different interface types $\mathcal{T} = \{1, \ldots, T\}$. Unless stated otherwise, we create N/T services of each type. Each peer hosts a single service.

For each service we define D random dependencies. To avoid loops in the dependency trees, we allow a service S of type $t[S]$ to only depend on services of type strictly greater than $t[S]$. Therefore, for each service S we initialize the dependency set $\delta[S]$ as a random subset of $\{t[S]+1, t[S]+2, \ldots, T\}$ of size $\min(D, T - t[S])$. Note that, according to this rule, services of type T have no dependencies.

Each service S is assigned a utility $u[S]$ that is uniformly distributed in $(0, 1)$. For each type $t \in \mathcal{T}$ we randomly choose $N_{\text{opt}} \geq 1$ services of type t and set their utility to 1. We compute the utility of a fully resolved assembly as the product of utilities of individual services, divided by the utility of

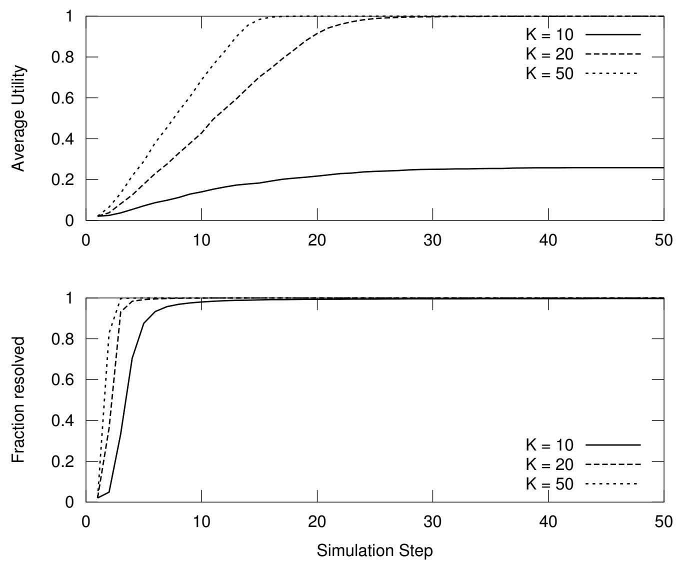

Fig. 5. Average utility (top) and fraction of resolved services (bottom) for different values of the number of neighbors K; $N = 1000$, $D = 10$, $N_{\text{opt}} = 1$. Higher is better.

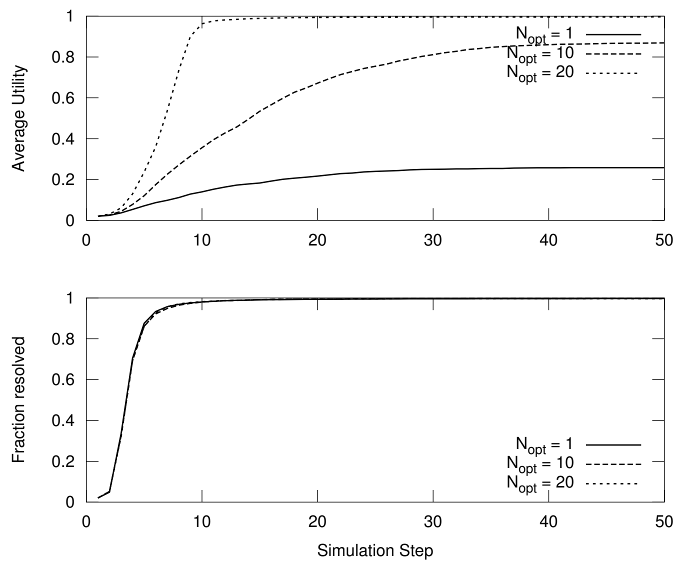

Fig. 6. Average utility (top) and fraction of resolved services (bottom) for different values of the number of services with optimal utility N_{opt}; $N = 1000$, $T = 50$, $K = 10$. Higher is better.

the root. This guarantees that the optimal (maximum) service utility of an assembly is always 1.

Performance Measures: We consider two metrics: (i) the fraction R_t of fully resolved services at simulation step t, $0 \leq R_t \leq 1$, and (ii) the average utility U_t of fully resolved services at step t, $0 \leq U_t \leq 1$. Both are higher-is-better metrics. R_t is computed by counting the fraction of fully resolved assemblies at the end of each simulation step; the optimal value of R_t is 1 (all services are fully resolved). U_t is computed as the average utility of all fully resolved services at step t (there are NR_t such services). As already explained above, the optimal value of U_t is 1. All results shown in the graphs are computed by taking the average of ten independent simulation runs.

We now report the simulation results in different scenarios.

a) Number of Neighbors K: In the first experiment we examine the impact of the value of K (number of neighbors on the P2P overlay) on R_t and U_t; specifically we consider $K \in \{10, 20, 50\}$. We consider $N = 1000$ services of $T = 50$ different types; each service has $D = 10$ randomly chosen dependencies. For each type $t \in \mathcal{T}$, there are is a single service with optimal utility ($N_{\text{opt}} = 1$).

Figure 5 shows the average utility (top part) and fraction of resolved services (bottom part) after each simulation step, for the different values of K. If each peer has $K = 10$ neighbors, we observe that all dependencies are resolved in about 10 interactions (bottom part of the figure). However, the average utility of the assemblies grows slowly, and stabilizes around a value which is well below the maximum–which, by construction, is 1. This can be explained by observing that, in order to build an assembly with optimal utility, the algorithm needs to locate the (unique) component of each needed type with utility set to 1. Since only interactions with neighbors is allowed, this process is very slow over networks with limited degree. The situation improves by increasing the number of neighbors, or if multiple optimal services are available. To prove the latter point, we examine again the scenario with $K = 10$ with increasing values of N_{opt}.

Larger values of N_{opt} imply that there exists multiple different ways to build an assembly with optimal utility. Figure 6 shows the results with $N_{\text{opt}} \in \{1, 10, 20\}$. Increasing N_{opt} allows the algorithm to produce service assemblies with higher utility; note that the value of N_{opt} has basically no impact on the speed at which fully resolved services are produced (bottom part of Fig. 6).

b) Number of Dependencies: In this experiment we study how the number of dependencies D influences the algorithm convergence speed. We set $N = 1000$, $T = 50$, $K = 20$ and $N_{\text{opt}} = 1$. We set $D \in \{5, 10, 20\}$ random dependencies on each service.

The results are shown in Figure 7. Quite surprisingly, we observe that, as the number of dependencies increase, so does the convergence speed towards the optimal utility of the service assemblies. This may appear counterintuitive at first, but can be explained by considering that each peer sends its list of resolved (immediate) dependencies to its neighbors during interactions. Since the goal of each peer is to maximize its compound utility, it will likely bind to services with high utilities as well. Therefore, if peers have larger lists of dependencies to exchange, then the gossip protocol has a better chance to locate dependencies with higher utility faster.

c) Handling Failures: Every large collection of distributed components is necessarily prone to failures: individual peers may crash at any time, and new peers may join the system. Many gossip-based algorithms exhibit the ability to handle massive failures gracefully [9]. We study the resilience of Algorithm 1 by considering again a set of $N = 1000$

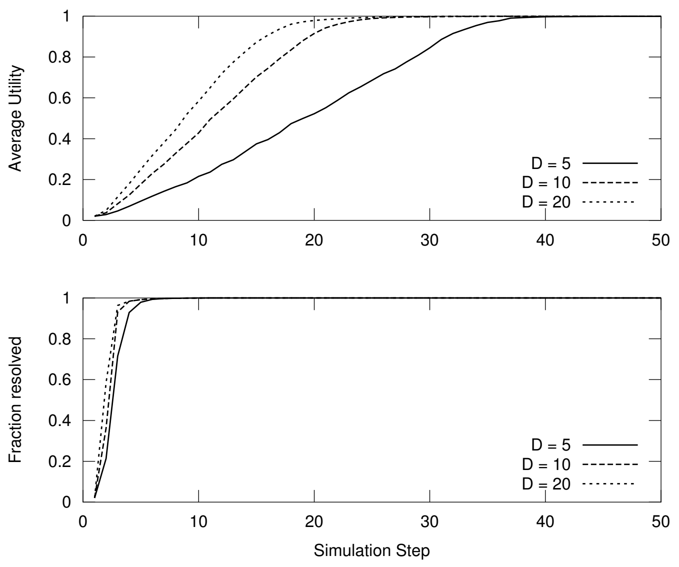

Fig. 7. Average utility (top) and fraction of resolved services (bottom) for $N = 1000$, $T = 50$, $K = 20$, $N_{\text{opt}} = 1$ and different values of the number of dependencies D. Higher is better.

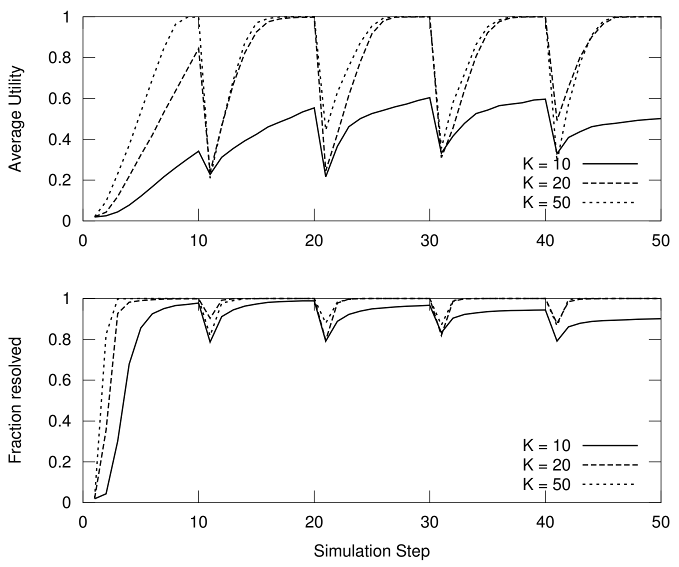

Fig. 8. Average utility (top) and fraction of resolved services (bottom) when 40 random services are removed every 10 simulation steps; $N_{\text{opt}} = 10$. Higher is better.

services of $T = 50$ different types. Each service has $D = 10$ random dependencies. For each service type, we assign maximum utility (1.0) to $N_{\text{opt}} = 10$ different peers. Every ten simulation steps we remove 40 random services.

Figure 8 shows the average utility and fraction of fully resolved services for different values of K, the number of neighbors of each peer. After each failure, we clearly see a sharp reduction of both the average utility and the fraction of resolved components. However, the algorithm quickly works around failed nodes and stabilizes itself near a new optimal configuration within a few steps. Again, if $K = 10$ the algorithm provides assemblies with sub-optimal utility. As we discussed above, when each node has a limited number of neighbors, then information diffusion slows down and the system stabilizes around a suboptimal configuration. Despite that, almost all services become quickly fully resolved, as shown in the bottom part of Fig. 8.

VII. Related Work

The problem of managing a dynamic service composition has been dealt with in literature by proposing approaches mainly based on dynamic service assembly (e.g., [8], [14]) or on dynamic service planning (e.g., [15], [16]). In this section we shortly review the papers based on dynamic service assembly, which are the ones closest to our approach. In particular we highlight the aspects concerning the centralized or decentralized control of self-adaptive systems and the self-assembly capability.

The work in [5] clearly contrasts decentralized self-adaptive systems with their centralized counterparts, and highlights the importance of decentralized control to achieve quality requirements such as resilience, robustness and scalability in large distributed systems. That work also evidences some key research challenges for the realization of decentralized self-adaptation. A deep investigation of possible patterns for decentralized control in self-adaptive systems has been presented in [17].

Besides the three-layer architecture proposed in [7] and already discussed in Section IV, other proposals have been presented in the past for the design of centralized or hierarchical self-adaptive systems, as, for example, the well-known autonomic approach proposed by IBM [2], and the Rainbow framework [18]. A recent overview of some of these approaches can be found in [17], [19]. As already discussed in Section IV, we adopt the three-layer model for the architecture of our system, and propose a decentralized approach to the management of its operations.

The goal of the system we propose is to achieve and maintain global QoS guarantees in a distributed assembly of services, despite the uncertainty caused by agents dynamically entering or leaving the system, and without requiring global state information at each agent. Since our focus is on the development of a fully decentralized solution, in the following we only review papers adopting a similar approach.

The work in [14] presents an approach where a dynamic set of agents cooperate to preserve some architectural constraints. All agents rely on a group membership service and reliable broadcast to achieve a consistent view of the accumulated knowledge. Moreover, adaptation actions are 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 proposed control architecture.

FlashMob [8] overcomes some of the limits of [14]. This work is also the closest to ours, as it 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 requires a backtracking phase to explore alternative solutions in case the assembly does not fulfill some requirement.

Differently from [8], our decentralized self-assembly procedure does not maintain an explicit knowledge of the whole assembly at each peer. This reduces the size of messages and of local state. Moreover, the achievement of global QoS goals is one of the drivers of the procedure we propose.

Some works [6], [20], [21] deal with the problem of managing dynamic organizations of agents in a decentralized way, i.e. agents that may dynamically form specific subsets (organizations) to cooperate towards some common goal. The problem considered in these papers has thus a wider scope than managing a dynamic assembly of services. However, this latter problem is part of the more general problem they consider. [20], [21] present the MACODO organization model and the related middleware for the management of dynamic organizations of agents. To simplify synchronization issues, MACODO currently adopts an architecture which is only partially decentralized. Indeed, each agent organization is based on a master-slave schema, where the master has complete knowledge of the organization state and controls the organization dynamics in a centralized way. The masters of different organizations can then cooperate to achieve some common goal (for example by merging their respective set of agents into a single organization), exchanging to this end some reduced state information. [6] presents a decentralized approach where each agent periodically contacts a subset of its peers to determine the composition of the organization it should refer to for the accomplishment of some specific task. In principle, the subset to be contacted could include the whole set of peers but, for scalability reasons, [6] suggests to randomly select a limited subset. This guarantees that, eventually, all peers will be contacted.

VIII. Conclusions

In this paper we have presented a decentralized approach to the dynamic adaptive self-assembly of distributed services. The system we propose to this end is architected according to the three-layer model proposed in [8], and its core element is a gossip-based protocol for information dissemination and decision making. Thanks to this, the system is able to build and maintain in a fully decentralized way an assembly of services that, besides functional requirements, is able to fulfill global quality of service (QoS) and structural requirements. The system operations require a bounded amount of information to be exchanged and maintained at each peer, independently of the overall number of peers in the system, thus guaranteeing the scalability of the proposed approach. Moreover, we have shown through a set of simulation experiments the effectiveness of our approach in terms of convergence speed towards the required solution, and resilience to failures.

We plan to implement a prototype of a software middleware realizing the proposed algorithm for service assembly, such that its effectiveness could be evaluated on real-world application scenarios.

Acknowledgment

We would like to thank Alberto Montresor for insightful discussions about gossip protocols.

References

[1] I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Z. Kwiatkowska, J. A. McDermid, and R. F. Paige, "Large-scale complex it systems," *Commun. ACM*, vol. 55, no. 7, pp. 71–77, 2012.

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

[3] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing - degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, 2008.

[4] B. H. C. Cheng and al., "08031 – software engineering for self-adaptive systems: A research road map," in *Software Engineering for Self-Adaptive Systems*, ser. Dagstuhl Seminar Proceedings, vol. 08031. IBFI, 2008.

[5] D. Weyns, S. Malek, and J. Andersson, "On decentralized self-adaptation: lessons from the trenches and challenges for the future," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '10. New York, NY, USA: ACM, 2010, pp. 84–93.

[6] R. Kota, N. Gibbins, and N. R. Jennings, "Decentralized approaches for self-adaptation in agent organizations," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 1, pp. 1:1–1:28, May 2012.

[7] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *FOSE*, 2007, pp. 259–268.

[8] D. Sykes, J. Magee, and J. Kramer, "Flashmob: distributed adaptive self-assembly," in *SEAMS*. ACM, 2011, pp. 100–109.

[9] M. Jelasity, A. Montresor, and Ö. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Trans. Comput. Syst.*, vol. 23, no. 3, pp. 219–252, 2005.

[10] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Trans. Comput. Syst.*, vol. 25, August 2007.

[11] D. Shah, *Gossip Algorithms*, ser. Foundations and trends in networking. Now Publishers, 2009.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented design*. Addison-Wesley Reading, MA, 1995.

[13] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Proc. of the 9th Int. Conference on Peer-to-Peer Computing (P2P'09)*, Seattle, WA, Sep. 2009, pp. 99–100.

[14] I. Georgiadis, J. Magee, and J. Kramer, "Self-organising software architectures for distributed systems," in *Proceedings of the first workshop on Self-healing systems*, ser. WOSS '02. New York, NY, USA: ACM, 2002, pp. 33–38.

[15] M. E. Falou, M. Bouzid, A.-I. Mouaddib, and T. Vidal, "A distributed planning approach for web services composition," *2012 IEEE 19th International Conference on Web Services*, vol. 0, pp. 337–344, 2010.

[16] S. Kalasapur, M. Kumar, and B. Shirazi, "Dynamic service composition in pervasive computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 7, pp. 907–918, 2007.

[17] *On Patterns for Decentralized Control in Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, vol. 7485. Springer, 2012.

[18] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Computer*, vol. 37, pp. 46–54, October 2004.

[19] 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. Software Eng.*, vol. 38, no. 5, pp. 1138–1159, 2012.

[20] D. Weyns, R. Haesevoets, A. Helleboogh, T. Holvoet, and W. Joosen, "The macodo middleware for context-driven dynamic agent organizations," *TAAS*, vol. 5, no. 1, 2010.

[21] D. Weyns, R. Haesevoets, and A. Helleboogh, "The macodo organization model for context-driven dynamic agent organizations," *TAAS*, vol. 5, no. 4, p. 16, 2010.