GoPrime: A Fully Decentralized Middleware for Utility-Aware Service Assembly

Mauro Caporuscio, Vincenzo Grassi, Moreno Marzolla, *Member, IEEE*, and Raffaela Mirandola

Abstract—Modern applications, e.g., for pervasive computing scenarios, are increasingly reliant on systems built from multiple distributed components, which must be suitably composed to meet some specified functional and non-functional requirements. A key challenge is how to efficiently and effectively manage such complex systems. The use of self-management capabilities has been suggested as a possible way to address this challenge. To cope with the scalability and robustness issues of large distributed systems, self-management should ideally be architected in a decentralized way, where the overall system behavior emerges from local decisions and interactions. Within this context, we propose GoPrime, a fully decentralized middleware solution for the adaptive self-assembly of distributed services. The GoPrime goal is to build and maintain an assembly of services that, besides functional requirements, fulfils also global quality-of-service and structural requirements. The key aspect of GoPrime is the use of a gossip protocol to achieve decentralized information dissemination and decision making. To show the validity of our approach, we present results from the experimentation of a prototype implementation of GoPrime in a mobile health application, and an extensive set of simulation experiments that assess the effectiveness of GoPrime in terms of scalability, robustness and convergence speed.

Index Terms—Service-oriented architecture, pervasive computing, runtime adaptation, quality of service, gossip protocol

1 Introduction

The pervasive computing paradigm aims to develop information processing infrastructures that seamlessly integrate into everyday activities. Systems built under this paradigm, like ambient intelligence or intelligent transportation systems, consist of several (from tens to thousands) services that cooperatively contribute to the achievement of some common goal [1].

How to properly architect and manage such systems is one of the major challenges for today's software engineering. Indeed, the high number of services 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], [3], [4]. The autonomic computing paradigm enables the system to automatically self-configure and self-adapt in response to variations of operating conditions, thus guaranteeing short reaction times and minimal or no human intervention at all. Scalability and robustness considerations call for fully decentralised solutions to the implementation of these autonomic capabilities [5]. Indeed, for the systems we are considering, centralised control can seriously hinder scalability and fault-tolerance, and can also be difficult or even impossible to achieve.

This paper provides a contribution towards the design and implementation of decentralised solutions for the autonomic management of large and highly dynamic distributed pervasive systems. Specifically, we propose GoPrime,[1] a fully decentralised middleware for the adaptive self-assembly of distributed services. Abstracting from characteristics of specific application domains, GoPrime is intended to manage distributed systems where a 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. In this context, the GoPrime goal is to drive a self-assembly procedure among the peers, aimed at matching required and provided services. Moreover, we assume that the system operates under non-functional requirements concerning the quality of the offered services (QoS) (e.g., performance, dependability, cost) and/or the structure of the resulting assembly. Hence, GoPrime is able to drive the systems it manages towards the selection, among the set of functionally feasible assemblies, of an assembly that fulfills global non-functional requirements.

According to decentralisation principles, GoPrime operations are characterised by the following properties [7]:

- absence of external control, so that the self-(re)configuration process is initiated and managed internally;
- dynamic operation, so that GoPrime is continuously able to adapt the system to requirement or environment changes (including arrival of new peers or failure of existing ones);
- absence of central control, where GoPrime maintains the assembly among peers through local decision-

- *M. Caporuscio is with the Department of Computer Science, Linnaeus University, Växjö, Sweden. E-mail: mauro.caporuscio@lnu.se.*
- *V. Grassi is with the Informatica, Sistemi e Produzione, University of Roma "Tor Vergata," Roma, Italy. E-mail: vgrassi@disp.uniroma2.it.*
- *M. Marzolla is with the Dipartimento di Informatica Scienza e Ingegneria, University of Bologna, Bologna, Italy. E-mail: moreno.marzolla@unibo.it.*
- *R. Mirandola is with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy. E-mail: raffaela.mirandola@polimi.it.*

1. GoPrime stands for GOssip-based Prime, an extension, centered around the use of a gossip protocol, of the Prime middleware [6].

making only, with dissemination of information in order to improve the binding process.

A key element of GoPrime is the use of a *gossip protocol* [8], [9], [10] to support information dissemination and decentralised decision-making. Experimental results show that this 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.

This paper extends the early results presented in [11] with substantial new contributions along two directions: (*i*) we extend the methodology proposed in [11] to deal with the case of multiple (possibly conflicting) non-functional requirements; (*ii*) we present GoPrime, a middleware implementing this methodology, and results from its experimentation in a practical case study.

The paper is organised as follows. Section 2 describes the system model and introduces the concept of *compound utility*, a quantitative measure of the "quality" of a service assembly. In Section 3 we illustrate the architecture of GoPrime, and describe the gossip-based fully decentralised algorithm that is used to build an assembly fulfilling QoS and structural requirements. Section 4 describes the implementation of GoPrime. In Section 5 we show a practical case study of GoPrime in an e-Health scenario, and evaluate the scalability and robustness of the proposed algorithms using simulation. We survey related work in Section 6 and present conclusions and future work in Section 7.

2 System Model

In this section we define the model of the system managed by GoPrime and introduce the terminology and notation used in the rest of the paper. As a part of the model, we illustrate the assumptions and formalise the notion of *compound utility*, a unified mechanism that GoPrime uses to enforce both functional and non-functional requirements on the assembly to be built and maintained.

2.1 Model Definition

We consider a system containing N distributed services $\mathbf{S} = \{S_1, \dots, S_N\}$, with each service having type $d \in \mathbf{T} = \{1, \dots, T\}$. Services are hosted on peer nodes, each node containing one or more services. Nodes can be located anywhere and communicate with each other through a network.

Each service $S \in \mathbf{S}$ has a *provided interface*, through which S 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 $(Type, Deps, Util, In, Out)$, where:

- $S.Type \in \mathbf{T}$ denotes the type of the provided interface (we say that $S.Type$ is the type of S). We assume that a function exists $matches : \mathbf{T} \times \mathbf{T} \rightarrow [0,1]$ such that $matches(d_1, d_2) = 0$ if type d_1 does not match type d_2 and $matches(d_1, d_2) > 0$ if some matching exists according to some suitable matching criterion [12].
- $S.Deps \subseteq \mathbf{T}$ denotes the set of dependencies of S (if $S.Deps = \emptyset$, then S has no dependencies). The set $S.Deps$ contains the types of the required dependencies of S; therefore, for each $d \in S.Deps$, S must be bound to a service S' such that $matches(d, S'.Type) > 0$, in order to be executed. Note that the dependency set $S.Deps$ does not contain duplicates, meaning that a service may depend at most once on any specific interface type. We assume that the set of dependency types $S.Deps$ is fixed for each service and known in advance.
- $S.Util \subseteq \mathbb{R}^m$ is a vector of m real values belonging to $\mathbb{R}^m$ representing QoS (e.g., reliability, cost, response time) or structural attributes, which express the utility of service S in isolation, depending only on the internal characteristics of S. If S has a non empty set of dependencies, then $S.Util$ gives only a partial view of the overall utility of S, which also depends on the utility of the services used to resolve them; therefore, we introduce in Section 2.2 the concept of *compound utility*.
- $S.In \subseteq \mathbf{S}$ is the set of services S is currently bound to, to resolve its dependencies.
- $S.Out \subseteq \mathbf{S}$ is the set of other services that are bound to S, to resolve one of their dependencies.

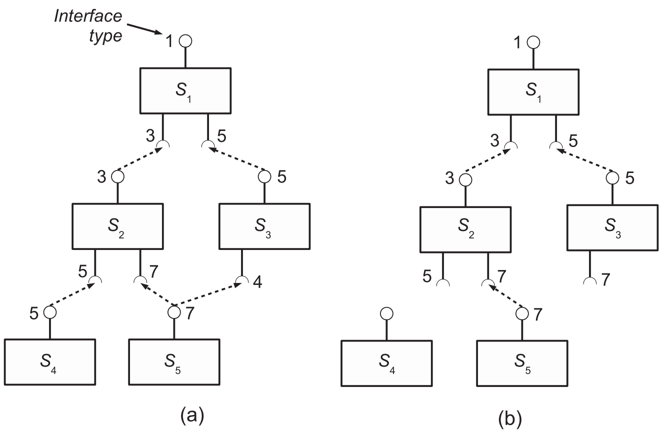

Fig. 1. Assembly examples with $N = 5$ services and $T = 7$ interface types.

We point out that while $S.Type$ and $S.Deps$ represent static information that does not change throughout the service lifetime, $S.Util$, $S.In$ and $S.Out$ represent dynamic state information whose value can change because of internal changes of S or of the services it is bound to.

A *service assembly* $\mathbf{A}$ is a graph $\mathbf{A} = (\mathbf{S}, \mathbf{E})$, where $\mathbf{E} \subseteq \mathbf{S} \times \mathbf{S}$ is the set of resolved dependencies. Specifically, a directed edge $(S_i, S_j) \in \mathbf{E}$ denotes that S_i is used by S_j to resolve one of its dependencies.

In general, we allow multiple simultaneous bindings to the same service S by other services that use S to resolve one of their dependencies. The number of simultaneous bindings to a service can be upper bounded (e.g., to avoid service overloading).

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

- S has no dependencies ($S.Deps = \emptyset$); or
- for all $d \in S.Deps$ there exists a fully resolved service $S' \in S.In$ such that $matches(S.Type, d) > 0$.

A service S is *partially resolved* in a given assembly $\mathbf{A}$ if it is not fully resolved in $\mathbf{A}$. 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.

As an example, in Fig. 1 we show two assemblies involving the services $\{S_1, S_2, S_3, S_4, S_5\}$ using the

TABLE 1
Symbols Used in this Paper

N	Number of services (peers)
$\mathbf{S}$	Set of services $\mathbf{S} = \{S_1, \ldots, S_N\}$
$\mathbf{T}$	Set of service types $\mathbf{T} = \{1, \ldots, T\}$
$\mathbf{A}$	Service Assembly $\mathbf{A} = (\mathbf{S}, \mathbf{E})$
m	Number of QoS and structural attributes of service S
$S.Type$	Service type of S
$S.Deps$	Set of dependencies of S
$S.In$	Set of peers that S is bound to, to resolve its own dependencies
$S.Out$	Set of peers that have a binding with S to resolve one of their dependencies
$S.Util$	"Local" utility vector of service S
$\mathbf{U}(S)$	Compound utility of the assembly rooted at S (a scalar compound utility is denoted with $U(S)$)
$\mathbf{F}$	Function $\mathbb{R}^{(1+\|S.Deps\|)m} \to \mathbb{R}^m$ that combines $S.Util$ with the compound utility of all dependencies of S

standard UML 2.0 component diagram notation. Service S_1 in Fig. 1a is fully resolved, while service S_1 in Fig. 1b is not, since it is bound to S_2 and S_5 which have missing dependencies.

Since our model adheres to *Service Statelessness* design principle [13], services do not maintain the interaction state between service invocations, i.e., a consumer's request is served in complete isolation, without relying on information from previous requests. Hence, we assume that the state of the interaction between the consumer and the service is kept on the user's side, and requests include all information necessary for their processing.

Service statelessness enhances (*i*) decoupling of interacting services, (*ii*) flexibility of the model, since it allows for easily rearranging the assembly at run time and, (*iii*) scalability, by exploiting service caching and replication. On the other hand, since the whole state of the interaction must be transferred at each request, a greater network capacity is required in order to keep the quality of service acceptable.

Table 1 summarises the notation used in this paper; the table also includes additional symbols which will be introduced in the next sections.

2.2 Compound Utility

The last important part of our model is the definition of the *compound utility* function that associates a utility value to each service $S \in \mathbf{S}$. In general, the compound utility is a vector-valued function $\mathbf{U} : \mathbf{S} \to \mathbb{R}^m \cup \{\bot\}$, such that $\mathbf{U}(S)$ is a vector of m *compound utility values*, each of them referring to some specific QoS or structural property. Such utility values are a function of: (*i*) the local utility $S.Util$, (*ii*) the compound utilities of the services S is bound to, and, possibly, (*iii*) other domain-dependent parameters that could be useful to better characterise the compound utility evaluation (for example, parameters capturing environment information for a context-aware utility calculation).

The compound utility function $\mathbf{U}(S)$ for a service S is recursively defined according to the following structure[2]:

2. For the sake of simplicity, we omit the indication of context parameters $\mathbf{U}(S)$ could depend on, thus showing only the dependence on $S.Util$ and on the utility of services S is bound to.

$$\mathbf{U}(S) = \begin{cases} S.Util & \text{if } S.Deps = \emptyset \\ \bot & \text{if } S \text{ is partially resolved} \\ \mathbf{F}\big(S.Util, \mathbf{U}(S_{d_1}), \ldots, \mathbf{U}(S_{d_k})\big) & \\ & \text{if } S \text{ fully resolved,} \\ & S.Deps = \{d_1, \ldots, d_k\}. \end{cases} \tag{1}$$

If S has no dependencies ($S.Deps = \emptyset$), then S is by definition fully resolved, and $\mathbf{U}(S)$ coincides with its local utility vector $S.Util$. If S has a nonempty set of dependencies and is not fully resolved, $\mathbf{U}(S)$ is set to $\bot$, i.e., the special value that is guaranteed to be "worse" than the utilities of any fully resolved instance of S. Finally, if S has a nonempty set of dependencies and is fully resolved, $\mathbf{U}(S)$ is computed using a function $\mathbf{F} : \mathbb{R}^{(1+|S.Deps|)m} \to \mathbb{R}^m$, which combines $S.Util$ with the compound utility of all S dependencies.

We now provide some examples to show how the general expression (1) can be specialised to express specific QoS or structural attributes; for simplicity, we focus on a single attribute at a time.

Reliability-based compound utility. We can define the reliability of a service S as the probability that S correctly completes its task, for a given service request. For each dependency $d \in S.Deps$, let $S'.n$ be the average number of times a service S' resolving d is invoked during the execution of S. The value of $S'.n$ can be estimated by the peer hosting S through a monitoring task; $S'.n$ will likely only depend on the type d, and not on the specific identity of S'. Let $r(S)$ be the internal reliability of S, that is the probability that no internal failure occurs when S is executed. Therefore, the reliability-based compound utility $U_r(S)$ for S is the joint probability that S experiences no internal failures, and all $S'.n$ invocations of each dependency S' produce no failure; the joint probability of these events is the product of probabilities [14]. $U_r(S)$ can be defined as (we omit the arguments of $F()$ for the sake of simplicity):

$$U_r(S) = \begin{cases} r(S) & \text{if } S.Deps = \emptyset \\ \bot & \text{if } S \text{ is partially resolved} \\ F() \overset{\text{def}}{=} r(S) \times \prod_{S' \in S.In} U_r(S')^{S'.n} & \\ & \text{if} S \text{ is fully resolved} \end{cases} \tag{2}$$

(since each dependency S' is executed $S'.n$ times, its contribution to the compound reliability is $U_r(S') \times \cdots \times U_r(S')$ ($S'.n$ times) $= U_r(S')^{S'.n}$).

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). Let $c(S)$ denote the cost for each invocation of S. We distinguish two cases, depending on how cost accumulates for multiple service invocations. In the first case, we assume that an additive cost is incurred for each single invocation of a service $S' \in S.In$. This is reasonable for costs referring, for example, to energy consumption. Under this assumption, the cost-based compound utility $U_c(S)$ for an assembly rooted at S can be defined as:

$$U_c(S) = \begin{cases} -c(S) & \text{if } S.Deps = \emptyset \\ \bot & \text{if } S \text{ is partially resolved} \\ F() \overset{\text{def}}{=} -c(S) + \sum_{S' \in S.In} U_r(S') \times S'.n & \\ & \text{if } S \text{ is fully resolved.} \end{cases} \tag{3}$$

Note that, to force $U_c(S)$ to be a higher-is-better metric, the compound utility is the *negated* total cost. Alternatively, we may assume a flat cost model, where a fixed cost is paid for the use of a service, independently of the number of times it is actually invoked (this could be the case, for example, of monetary cost). In this case, the flat cost-based compound utility $U_c(S)$ can be defined as:

$$U_c(S) = \begin{cases} -c(S) & \text{if } S.Deps = \emptyset \\ \bot & \text{if } S \text{ is partially resolved} \\ F() \overset{\text{def}}{=} -c(S) + \sum_{S' \in S.In} U_c(S') & \\ & \text{if } S \text{ is fully resolved.} \end{cases} \tag{4}$$

Response time-based 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 $s(S)$ denote the average time spent within service S for internal operations only. Then, we can define the response-time compound utility $U_t(S)$ of service S as:

$$U_t(S) = \begin{cases} -s(S) & \text{if } S.Deps = \emptyset \\ \bot & \text{if } S \text{ is partially resolved} \\ F() \overset{\text{def}}{=} -s(S) + \sum_{S' \in S.In} U_t(S') \times S'.n & \\ & \text{if } S \text{ is fully resolved.} \end{cases} \tag{5}$$

$U_t(S)$ expresses the overall average response time of S; in this case too $U(S)$ is negative so that the compound utility $U(S)$ is a higher-is-better metric. We point out that equation (5) is based on the assumption of a sequential execution model for the dependencies of S. In the case of a parallel execution model for some of those dependencies, the definition should be modified accordingly (see for example [15], [16]).

Structural compound utility. 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 properties 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 $U(S)$, a systems that tries to maximise it actually drives itself towards the construction of an assembly that fulfills local or global structural properties, in the latter case limited to those properties that can be recursively defined. Let us consider first a local constraint on the maximum number of dependencies that can be resolved by S, meaning that S can be used by at most $S.Dmax$ other services to resolve their dependencies, e.g., to avoid overload. The compound utility $U_l(S)$ defined as:

$$U_l(S) = \begin{cases} \bot & \text{if } S \text{ is partially resolved} \\ 0 & \text{if } |S.Out| > S.Dmax \\ 1 & \text{if } |S.Out| \leq S.Dmax \end{cases} \tag{6}$$

returns 1 if and only if the structural constraint above is satisfied, i.e., at most $S.Dmax$ other services are currently using S. Note that recursion (i.e., definition of $F()$) is not present in (6) as this utility depends on a local property only, and thus $U_l(S)$ actually corresponds to $S.Util$.

Let us consider instead a global structural constraint enforcing a pipeline structure on a fully resolved assembly. We define the compound utility $U_p(S)$ as follows:

$$U_p(S) = \begin{cases} \bot & \text{if } S \text{ is partially resolved} \\ 0 & \text{if } |S.Deps| > 1 \\ 1 & \text{if } S.Deps = \emptyset \\ F() \overset{\text{def}}{=} U_p(S') & \text{if } |S.Deps| = 1 \wedge S.In = \{S'\}. \end{cases} \tag{7}$$

From (7), $U_p(S)$ yields 1 if and only if either S has no dependencies, or S is fully resolved and its direct and indirect dependencies are organised as a chain (pipeline structure). Hence, $U_p(S) = \bot$ 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.

2.3 Comparing Compound Utilities

GoPrime encodes the set of QoS and structural attributes associated with a service S in a suitably defined utility function $\mathbf{U}(S)$, as shown in Section 2.2. In the given examples, each $U(S)$ function can be considered as an indication of "how good" service S is with respect to a given QoS or structural attribute. In case of a vector-valued utility $\mathbf{U}(S)$, the question arises of how good a service S is (including the assembly that resolves its dependencies) with respect to the whole set of considered and possibly conflicting attributes. Hence, an important issue is how to compare the compound utilities of different assemblies to determine which one is "better" with respect to a given set of requirements.

Towards this aim, let us denote by S_1, S_2 two functionally equivalent services, i.e., such that $matches(S_1.Type, S_2.Type) > 0$ (actually, S_1, S_2 could denote the same service with its dependencies solved by a different assembly). GoPrime can manage the comparison between $\mathbf{U}(S_1)$ and $\mathbf{U}(S_2)$ in two different ways.

A first way is to map both $\mathbf{U}(S_1)$ and $\mathbf{U}(S_2)$ to a single scalar value using, for instance, the Simple Additive Weighting (SAW) technique [17]. According to SAW we can map $\mathbf{U}(S) = (U_{S,1}, \dots U_{S,m})$ to a scalar value $U(S)$ by taking the weighted sum of $\mathbf{U}(S)$ components, as follows:

$$U(S) = \sum_{i=1}^{m} w_i U_{S,i} \tag{8}$$

for a suitable choice of weights such that $\sum_{i=1}^{m} w_i = 1$, $0 \leq w_i \leq 1$. The relative values of weights w_i are intended to specify the relative importance associated with each attribute. If the structural or QoS attributes expressed by the $U_{S,i}$'s take values in different domains, they should be normalised in the same range (e.g., $[0, 1]$) before computing the weighted average.

Normalisation can be done by considering instead of the attribute value $U_{S,i}$, $1 \leq i \leq m$, the value:

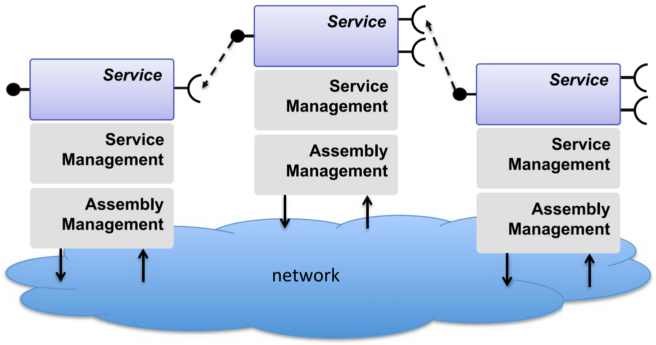

Fig. 2. Reference architecture.

$$\frac{Umax_i - U_{S,i}}{U_{S,i} - Umin_i}, \tag{9}$$

where $Umax_i$ and $Umin_i$ denote, respectively, the maximum and minimum value of the considered ith attribute.

However, assigning meaningful values to the weights w_i is not an easy task, and determining $Umax_i$ and $Umin_i$ requires additional effort in a distributed environment (it can be done using gossip-based aggregation [8], at the cost of increased complexity). Besides, the SAW technique could be impossible to apply when some attributes are measured on an ordinal scale [18].

In this case, GoPrime can compare the compound utilities of different assembly within a Pareto optimality framework, as suggested in [19]. Under Pareto optimality, S_1 is better than S_2 if $\mathbf{U}(S_1)$ *dominates* $\mathbf{U}(S_2)$, i.e., $\mathbf{U}(S_1)$ is better than $\mathbf{U}(S_2)$ for at least one of its entries and no worse according to all of the others.

The use of Pareto optimality actually leads to the identification of a set of services that are non-dominated with respect to a given compound utility definition, thus forming a *Pareto front*. This raises the question of what service must be selected to solve a dependency from a Pareto front [20]. The answer to this question is in general domain-dependent.

3 GoPrime Architecture

The GoPrime goal is to drive the distributed system we are considering towards the construction of a service assembly that is increasingly good (in the sense discussed in Section 2.3) with respect to a given set of attributes. In this section we present the GoPrime fully decentralised architecture that allows achieving this goal. We outline in Section 3.1 the overall architecture, in Section 3.2 we detail the algorithm that implements its core functions and in Section 3.3 we discuss the algorithm's properties.

3.1 Architecture

Fig. 2 shows the two GoPrime macrocomponents, namely *Service Management* and *Assembly Management*. Each peer node hosts an instance of this pair of macrocomponents, besides the services it offers. Overall, these pairs, by cooperating among them as outlined below and detailed in Section 3.2, give rise to a fully decentralised implementation of the GoPrime operations. In particular, the *Assembly Management* macrocomponents cooperate according to a gossip schema that allows fully decentralised information dissemination and decision-making about the assembly construction and maintenance. This makes the system robust and scalable in the presence of events like arrival of new requirements, upgrade/downgrade of service utility (including the extreme case of service failure), or arrival/departure of new peers (and corresponding hosted services).

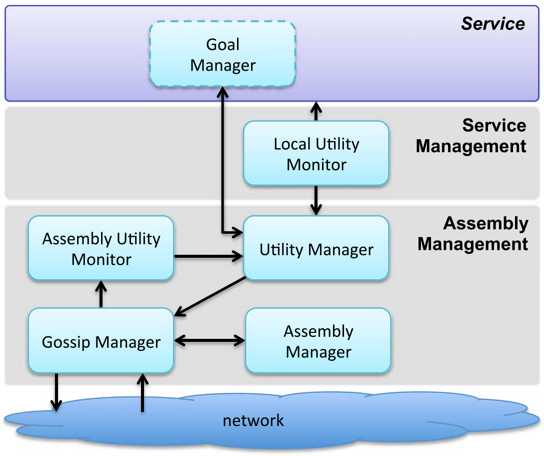

Fig. 3. GoPrime high-level architecture.

Each of the macrocomponents in Fig. 2 is actually architected as a set of interacting components, as shown in Fig. 3. We give below some details about these components and their functions.

Service Management includes *Local Utility Monitor*, which monitors the local utilities of hosted services and notifies detected changes to the *Utility Manager* hosted by the same node. *Service Management* could also include other components, possibly concerned with the implementation of internal adaptation actions aimed at maintaining local QoS attributes, but they are not part of the current GoPrime implementation.

The *Assembly Management* macrocomponent includes *Gossip Manager*, *Assembly Manager*, *Utility Manager* and *Assembly Utility Monitor*.

Gossip Manager is the core component that implements the GoPrime decentralised information dissemination and decision-making. We detail in Section 3.2 its operations. It activates and manages one or more gossip algorithm instances, based on directives received from Utility Manager, building a suitable $S.In$ set for a local S service. Moreover, it notifies the current value of $S.In$ to Assembly Manager and Assembly Utility Monitor.

Utility Manager receives requirements—expressed in terms of utility definition and related metrics—from the above Goal Manager, and sends to Gossip Manager directives about corresponding "gossips" to be started (one new gossip is possibly activated for each newly received requirement), to build and maintain an assembly able to fulfill the requirement. For each maintained assembly, Utility Manager receives from the Local Utility Monitor and the Assembly Utility Monitor information about the utilities of local and remote services, respectively, which are used to build the assembly. Based on this information, Utility Manager keeps updated the value of the compound utility for the locally hosted services, and notifies it to *Goal Manager*. Note that *Goal Manager* is an abstract architectural entity whose definition and implementation are strongly tied with the service it refers to. Therefore, it is not part of GoPrime and its specific implementation is left to the service developer.

Assembly Manager receives from Gossip Manager the current composition of the set $S.In$ that specifies which services should currently be used to solve the dependencies of a local service S, and manages the corresponding bindings. Moreover, it receives notifications of incoming binding requests for each local service S, and keeps updated the corresponding set $S.Out$.

Assembly Utility Monitor receives from Gossip Manager the current composition of the set $S.In$, for each local service S, and monitors the QoS of services in $S.In$, sending to Utility Manager notifications about observed changes.

3.2 Core Algorithm

In this section we describe in detail how GoPrime operates to dynamically build and maintain in a fully decentralised way a suitable assembly of services.

To achieve this goal, the various instances of the *Gossip Manager*, located on the different nodes, implement a decentralised algorithm, based on a gossip communication model [8], [10] for the dissemination of local information about the system state. Gossip communication builds upon epidemic protocols to achieve reliable information exchange among large sets of interconnected peers, also in presence of network instability (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.

Algorithm 1 describes the core of the gossip algorithm for service assemblies cooperatively executed by the Gossip Managers hosted by peer nodes. This algorithm iteratively resolves the dependencies of each service, thus leading to the construction of an assembly where each service $S \in \mathbf{S}$ is (possibly) fully resolved and the value of $\mathbf{U}(S)$ is monotonically increased until it reaches its maximum value or, at least, a given threshold is exceeded (see Section 3.3).

Algorithm 1. Algorithm Executed by the Agent Responsible for Service S

```
        // Input parameters
 1:     $S; \mathbf{U}(); \textsc{SelectFromBest}_d(); \textsc{UpdateBest}_d()$, (for all $d \in S.Deps$)
        // Local variables
 2:     $S.In \leftarrow \emptyset$
 3:     $Best_{S,d}\quad \emptyset$;  for all $d \in S.Deps$
 4:     procedure ActiveThread
 5:        loop
 6:           Wait $\Delta t$
 7:           for all $S_j \in \textsc{GetPeers}()$ do
 8:              Send $\langle S.In \cup \{S\} \rangle$ to $S_j$
 9:     procedure PassiveThread
10:        loop
11:           Wait for message $\langle \mathbf{B} \rangle$ from $S_j$
12:           for all $S_k \in \mathbf{B}$ do
13:              if there exists $d \in S.Deps$ s.t.
                     $matches(d, S_k.Type) > 0$ then
14:                 $Best_{S,d}\quad \textsc{UpdateBest}_d(Best_{S,d}; S_k; \mathbf{U}())$
15:                 $S.In \leftarrow \textsc{SelectFromBest}_d(Best_{S,d}; S.In; \mathbf{U}())$
```

Besides service S, the algorithm takes as input parameters the functions $\mathbf{U}()$, $\textsc{UpdateBest}_d()$, $\textsc{SelectFromBest}_d()$, for each $d \in S.Deps$. We gave in Section 2 the general definition and examples of possible instantiations of function $\mathbf{U}()$, while functions $\textsc{UpdateBest}_d()$ and $\textsc{SelectFromBest}_d()$ are described below.

Function $\textsc{UpdateBest}_d()$ keeps the set $Best_{S,d}, d \in S.Deps$ updated, where $Best_{S,d}$ collects the currently known H (or less) "best" services with respect to $\mathbf{U}(S)$ that can be used to solve dependency d. The upper bound H on the cardinality of $Best_{S,d}$ is a system parameter. As a consequence, the specific definition of $\textsc{UpdateBest}_d()$ depends on the definition of $\mathbf{U}()$ and of a suitable metric for it, as discussed in Section 2.3. Algorithms 2 and 3 show possible definitions of $\textsc{UpdateBest}_d()$ in case of a scalar or Pareto-based metric for $\mathbf{U}()$, respectively.

Algorithm 2. $\textsc{UpdateBest}_d()$ for Scalar $\mathbf{U}()$

```
      // Input parameters
1:    $Best_{S,d} \subseteq \mathbf{S}; S_i \in \mathbf{S}; \mathbf{U}()$
      // Algorithm
2:    $Best_{S,d}\quad Best_{S,d} \cup \{S_i\}$
3:    if $|Best_{S,d}| \leq H$ then
4:       continue
5:    else /* drop the worst service to keep $|Best_{S,d}| \leq H$ */
6:       $jmin\quad \arg\min_j\{\mathbf{U}(S_j) \mid S_j \in Best_{S,d}\}$
7:       $Best_{S,d}\quad Best_{S,d} \setminus \{S_{jmin}\}$
   return $Best_{S,d}$
```

Algorithm 3. $\textsc{UpdateBest}_d()$ for Pareto-Based $\mathbf{U}()$

```
       // Input parameters
 1:    $Best_{S,d} \subseteq \mathbf{S}; S_i \in \mathbf{S}; \mathbf{U}()$
       // Algorithm
 2:    $Best_{S,d}\quad Best_{S,d} \cup \{S_i\}$
 3:    for all $S_j \in Best_{S,d}$ do
 4:       if $\mathbf{U}(S_i)$ dominates $\mathbf{U}(S_j)$ then
 5:          $Best_d \leftarrow Best_{S,d} \setminus \{S_j\}$
 6:       if $\mathbf{U}(S_j)$ dominates $\mathbf{U}(S_i)$ then
 7:          $Best_d \leftarrow Best_{S,d} \setminus \{S_i\}$
 8:          break
 9:    if $|Best_{S,d}| \leq H$ then
10:       continue
11:    else
12:       /* remove a service from $Best_{S,d}$ according to some
          domain dependent criterion, to keep $|Best_{S,d}| \leq H$ */
   return $Best_{S,d}$
```

On the other hand, each function $\textsc{SelectFromBest}_d()$ takes as input the set $Best_{S,d}$ and the set $S.In$, and selects from $Best_{S,d}$ (according to some suitable criterion) the service that must actually be used to solve the dependency d. This service is added to $S.In$, possibly substituting a previously selected service. Algorithm 4 gives a possible definition of $\textsc{SelectFromBest}_d()$ in case of a scalar metric for the compound utility $\mathbf{U}()$. In case of Pareto-based compound utility $\mathbf{U}()$ the definition of $\textsc{SelectFromBest}_d()$ is domain dependent.

Finally, Algorithm 1 describes the general gossip-based scheme implemented by each Gossip Manager. It includes an initialisation phase and two concurrent threads: an active thread that starts an interaction by sending a message to a peer, and a passive thread that responds to messages received from other peers. The set of peers is provided by the underlying gossip communication protocol (more details will be given in Section 4). During initialisation (lines 2-3) the sets $S.In$ of services bound to S and $Best_{S,d}, d \in S.Deps$ are set to empty.

Algorithm 4. SELECTFROMBEST$_d$() for Scalar **U**()

```
    // Input parameters
1:  $Best_{S,d} \subseteq \mathbf{S}; S.In \in \mathbf{S}; \mathbf{U}()$
    // Algorithm
2:  $jmax \quad \arg\max_j\{\mathbf{U}(S_j) \mid S_j \in Best_{S,d}\}$
3:  if there exists $S_k \in S.In$ s.t. $matches(S_k.Type, d) > 0$ then
4:      if $\mathbf{U}(S_k) < \mathbf{U}(S_{jmax})$ then
5:          $S.In \quad S.In \setminus \{S_k\} \cup \{S_{jmax}\}$
6:  else
7:      $S.In \quad S.In \cup \{S_{jmax}\}$
    return $S.In$
```

The active thread is extremely simple: every Δt time units, Gossip Manager sends a message to its *peer set*. The message payload is a set of services, containing the list of currently bound dependencies $S.In$ plus S itself.

The passive thread listens for messages coming from other peers. Upon receiving a message containing the set **B**, Gossip Manager checks all services $S_k \in \mathbf{B}$ to see whether some of them can be used to fill its own dependencies. If $S_k.Type$ is required as a dependency, then S_k is considered as a candidate to be added to $Best_{S,d}$ (line 13). The decision whether to include S_k in $Best_{S,d}$ is taken by function UPDATEBEST() (line 4), possibly dropping from $Best_{S,d}$ some other service whose utility is worse than S_k (see examples given in Algorithms 2 and 3). The update of the sets $Best_{S,d}$ can lead to a substitution of the service currently used to solve dependency d (as specified in the set $S.In$) with a new "better" service taken from $Best_{S,d}$. The decision about this possible substitution is taken by function SELECTFROMBEST$_d$() (line 1).

As is typical with gossip-based protocols, a new instance of Algorithm 1 is created at each node for each query entered into the system, where a query essentially specifies a service S and one or more requirements that the assembly rooted at S needs to fulfill. Each node can define its own policies for deciding if and when a new instance of some of the hosted services can be created in response to the arriving stream of queries.

It is worth noting that, by maintaining a set $Best_{S,d}$ with $|Best_{S,d}| > 1$, GOPRIME allows for a quick local recovery from the loss of the binding with the service currently used by S to solve dependency d (e.g., because of a failure, or the hosting peer leaving the system). Indeed, in this case, a new service can be locally selected from $Best_{S,d}$. This recovery action can at least keep S fully resolved, even if some non-functional requirement could no longer be fulfilled.

Fig. 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. For simplicity, the compound utility is a single scalar value ($m = 1$), so that the

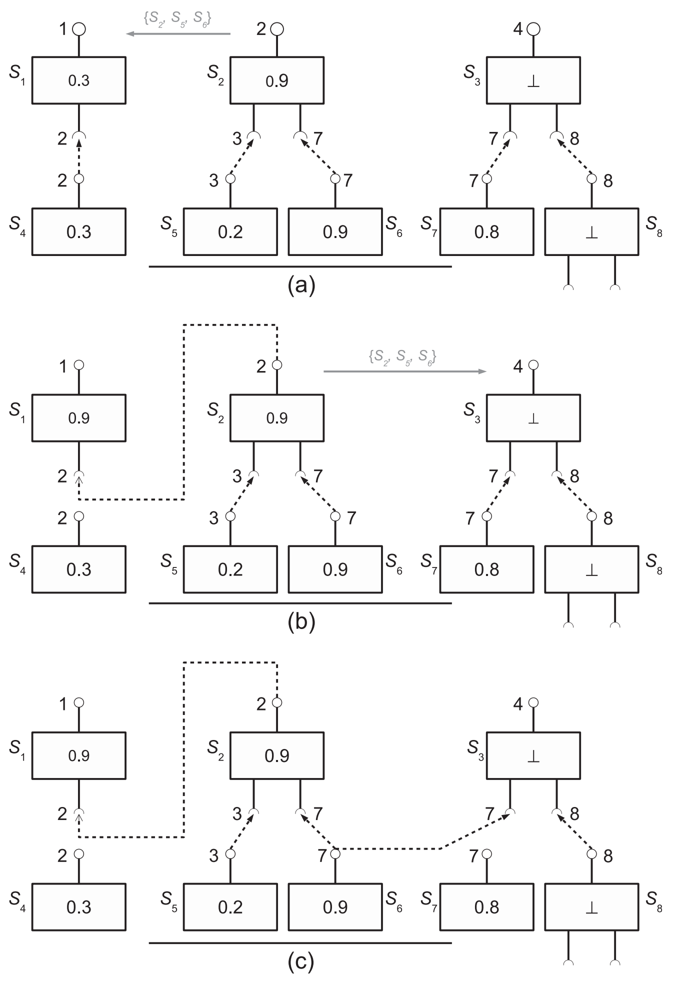

Fig. 4. Example of execution of Algorithm 1. At the beginning, service S_2 has $\{S_1, S_3\}$ as neighbours.

scalar versions of functions UPDATEBEST$_d$() and SELECTFROMBEST$_d$() can be used (see Algorithms 2 and 3).

The compound utility of some of the services is shown inside each block, and is assumed to be simply the maximum of the compound utilities of each dependency; note that S_3 and S_8 are not fully resolved, therefore $U(S_3) = U(S_8) = \perp$. The initial situation is shown in Fig. 4a; we assume that S_2 executes the active thread, and S_1 and S_3 are in its *peer set*. First, S_2 sends the list $S_2.In \cup \{S_2\} = \{S_2, S_5, S_6\}$ to its first peer 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. 4b. Now S_2 sends the same list $\{S_2, S_5, S_6\}$ to the other peer 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. 4c shows the final wiring of the services.

3.3 Algorithm Properties

In this section we argue that, thanks to Algorithm 1, GOPRIME is able to guarantee the construction and maintenance of an assembly fulfilling functional and non-functional requirements, defined as follows:

- *functional* requirement: service S must be fully resolved;

- *non-functional* requirement: the value of a_S must be maximised (or, also, it must hold $a_S \geq a_{min}$), where a_S is a given QoS or structural attribute of S and a_{min} is some suitable threshold value for that attribute.

Let us denote by $\mathbf{U}(S)_k$ and $\mathbf{U}(S)_{k+1}$ the compound utility of a service $S \in \mathbf{S}$ at two consecutive rounds of Algorithm 1. The central element of our argument is that $\mathbf{U}(S)_{k+1}$ is possibly better and in any case no worse than $\mathbf{U}(S)_k$, under the hypothesis that:

1) no service leaves the system;
2) the local utility of each service does not change;
3) function $\mathbf{F}()$ in equation 1 is non decreasing with respect to any of its arguments.

Actually, hypotheses (1) and (2) above could be too strong in the dynamic environment we are considering. Hence, after discussing the case when they hold, we generalise our argument to the case where they are released.

To prove the non-decreasing monotonicity of $\mathbf{U}(S)_k$ with respect to k when all the three hypotheses above hold, let us consider first the case where S does not change any of its bindings from round k to round $k+1$. In this case, hypotheses (1) and (2) guarantee that $\mathbf{U}(S)_{k+1} = \mathbf{U}(S)_k$.

Let us consider instead the case where S does change from round k to round $k+1$ its binding to solve a dependency $d \in S.Deps$, and let us denote by S' and S'' the old and new service S is bound to solve d. The non-decreasing monotonicity of $\mathbf{U}(S)_k$ is immediately evident in the case of a scalar metric. Indeed, looking at Algorithms 2 and 4, we see that S changes its current binding from S' to S'' only if $\mathbf{U}(S'') > \mathbf{U}(S')$. This, together with hypothesis (3) above, guarantees that in case of change of binding, we have $\mathbf{U}(S)_{k+1} \geq \mathbf{U}(S)_k$.

In case of a Pareto-based metric, we can see from Algorithm 3 that a new service S_i is added to the set $Best_{S,d}$ at round $k+1$ only if $\mathbf{U}(S_i)$ is not dominated by $\mathbf{U}(S_j)$ for any other service $S_j \in Best_{S,d}$ at round k, and possibly $\mathbf{U}(S_i)$ dominates $\mathbf{U}(S_h)$ for some service $S_h \in Best_{S,d}$. As a consequence, $\mathbf{U}(S'')$ can never be dominated by $\mathbf{U}(S')$, however function $\textsc{SelectFromBest}_d()$ is defined. This, together with hypothesis (3), guarantees that, in case of change of binding, $\mathbf{U}(S)_{k+1}$ cannot be dominated by $\mathbf{U}(S)_k$.

The arguments above show that the algorithm implemented by the Gossip Manager makes $\mathbf{U}(S)_k$ a monotonic non-decreasing function, with respect to k. We now discuss how, thanks to this property, GoPrime guarantees the fulfilment of functional and non-functional requirements.

Let us consider functional requirements first. We know from equation (1) that $\mathbf{U}(S)_k$ takes the lowest value when S is not resolved, for any service $S \in \mathbf{S}$. Hence, if services exist in $\mathbf{S}$ able to fully solve the dependencies of S, then the monotonicity of $\mathbf{U}(S)_k$ together with the properties of a gossiping scheme, guarantee that the system will be eventually driven towards the construction of an assembly where all S dependencies are resolved, for any definition of function $\mathbf{F}()$ in equation (1).

For non-functional requirements, we have presented in Section 2.2 example definitions of function $\mathbf{F}()$ for specific QoS or structural attributes. By instantiating the general definition of $\mathbf{U}(S)$ according to these definitions, the monotonicity of $\mathbf{U}(S)_k$ together with the properties of a gossiping scheme, guarantee that an assembly will be eventually built for a service S where maximisation or threshold-based non-functional requirements are fulfilled (in the latter case, provided that the specified threshold is below the achievable maximum). For requirements involving the maximisation of the compound utility value, the gossip protocol actually only guarantees that the compound utility value of the assembly will be non-decreasing, so that some user-defined criteria must be set to stop the protocol when the utility is considered "good enough". The simulation experiments, which will be presented in Section 5.2, suggest that, as the rate of improvements of the compound utility value slows down, the assembly is approaching its optimal configuration.

It remains to be discussed the case where hypotheses (1) or (2) above do not hold. Let us consider hypothesis (1). If a service that is currently part of an assembly rooted at a service S leaves the system, then $\mathbf{U}(S)$ suddenly drops to the lowermost value $\perp$, since the S dependencies are no longer fully resolved. GoPrime will recover from this situation thanks to its continuous effort in monotonically increasing the $\mathbf{U}()$ value, as discussed above. Let us consider now hypothesis (2). If the local utility of some service belonging to an assembly rooted at S increases, then $\mathbf{U}(S)$ will increase by hypothesis (3), thus having no negative impact, as expected. If the local utility of some service decreases, then $\mathbf{U}(S)$ could decrease too, by hypothesis (3). In this case, the situation is similar (even if less extreme) to the case where a service leaves the system, and GoPrime recovers in an analogous way from this situation.

Hence, in general, $\mathbf{U}(S)_k$ will present a piecewise monotonic non-decreasing behaviour, as a result of services leaving the system or decreasing their local utility (e.g., because of some internal failure), and the parallel continuous operation of GoPrime. We will present in Section 5 experiments providing evidence of this behaviour.

4 GoPrime Implementation

GoPrime is implemented as an extension of Prime, a support middleware for developing pervasive applications that adhere to the Pervasive-REST (P-REST) architectural style [6]. We give first a quick overview of Prime, then we present its GoPrime extension.[3]

Prime exploits a two-layer software architecture to provide engineers with a set of enhanced functionalities (white boxes in Fig. 5) facilitating the design and development of P-RESTful applications—i.e., applications adhering to the P-REST style [6]. Specifically:

Communication layer – To deal with the inherent instability of pervasive environments, Prime arranges devices in an `overlay network` built on top of low-level network technologies (e.g., Bluetooth, Wi-Fi). Such an overlay is exploited to provide two basic communication facilities, namely `point-to-point` and `point-to-multipoint`. Point-to-point communication grants a given node direct communication with a remote node, whereas point-to-multipoint communication allows communication with many different nodes at the same time. Furthermore, Prime implements a `DNS` facility for managing device mobility [21].

3. GoPrime is available at http://github.com/maurocaporuscio/prime-middleware-extensions

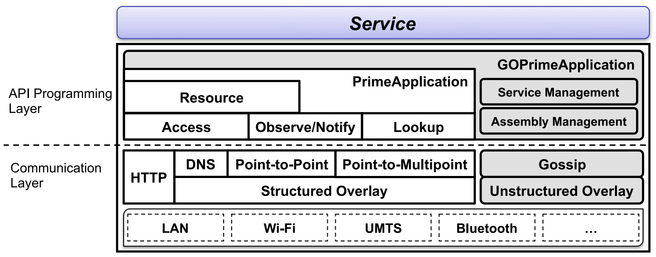

Fig. 5. GoPrime software architecture.

API programming layer – Prime provides the programming abstractions to implement P-RESTful applications. `PrimeApplication` acts as container for exposed `Resources`—i.e., it handles both resource life-cycle and provision—and provides the set of operations allowed on resources: (i) `Access`, which gathers the set of operations to access and manipulate resources according to the REST uniform interface, (ii) `Observe/Notify`, which allows resources to declare interest in a given resource and to be notified whenever changes occur, and (iii) `Lookup`, which supports resource discovery.

These functionalities grant `PrimeApplication` a set of key characteristics, such as: (i) *loose coupling*: `resources` are deployed and executed independently of other resources, (ii) *flexibility*: `resources` can be added and removed into the running application, (iii) *dynamism* `resources` of interests are discovered and bound into the running application, and (iv) *serendipity*: unforeseen `resources` are accommodated into the running application.

GoPrime exploits such characteristics and extends Prime with the set of capabilities discussed in previous sections, namely `Gossip` communication, `Assembly Management`, and `Service Management`. Indeed, the `GOPrimeApplication` (light-grey boxes in Fig. 5) uses the extended *communication layer* and *API programming layer* to provide up-level `Services` with the *Utility-Aware Service Assembly* functionality. Specifically, referring to the high-level architecture described in Section 3, GoPrime includes three macrocomponents, namely `Gossip` communication (Section 4.1), `Assembly Management` (Section 4.2), and `Service Management` (Section 4.3).

4.1 Gossip Communication

`Gossip` macrocomponent extends the Prime *communication layer* by providing `GOPrimeApplication` with the ability of gossiping information of interest.

To optimise information dissemination, `Gossip` builds and maintains the *peer set* over which information of interest is disseminated. To this end, `Gossip` implements the Newscast epidemic protocol [9], and maintains a *local view* of a set of peers it can exchange messages with. The local view is constantly updated, so that a node is always provided with "fresh" list of peers. Updating the local views is also necessary to maintain an updated *peer set* in presence of node and link failures. Specifically, each peer maintains a set of K peers, where K is a predefined constant; periodically, each peer merges its list with that of a randomly chosen peer, keeping the most K recently added links and dropping older ones. This protocol exhibits useful features: (i) the *peer set* it produces is a good approximation of a true random sample among all peers, and (ii) the protocol is highly resilient and can maintain a full *peer set* in presence of node or link faults.

4.2 Assembly Management

`Assembly Management` implements *assembly construction* and *maintenance* functionalities.

Referring to Fig. 6, the `GOPrimeApplication` class makes use of `Assembly Management`, which in turn includes `AssemblyManager`, `UtilityManager`, `AssemblyUtilityMonitor`, and `GossipManager`.

`AssemblyManager` is in charge of managing the assembly specified by the local service by satisfying the set of dependencies and resolving the corresponding bindings. To this end, `AssemblyManager` interacts with `GossipManager`, which in turn provides *gossip communication* facility: (i) sending/receiving messages to/from the underlying network, and (ii) implementing Algorithm 1, as well as the UpdateBest$_d$() and SelectFromBest$_d$() functions (see Section 3.2), to keep the `Assembly Management` updated.

Finally, `UtilityManager` is implemented as a supporting abstract class used to map the set of high-level requirements, specified by `Service Management`, to low-level directives needed to instruct `GossipManager`. Specifically, `UtilityManager` is extended by the `AssemblyUtilityMonitor` to combine local and remote utility and keep the compound utility updated.

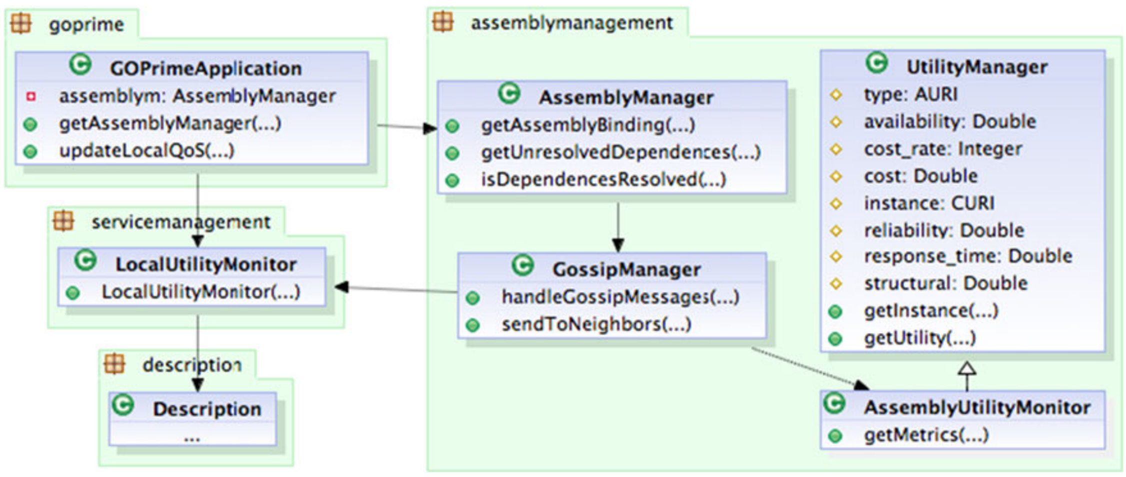

Fig. 6. An excerpt of the GoPrime class diagram.

```
@prefix   :<http://www.prime.org/ResourceDescription#>.
@prefix ex:<http://www.prime.org/Example#> .

[] a :Resource ;
  :aURI [ a ex:Type2 ] ;
  :cURI "S2" ;
  :offers [ a :Interface ;
          :method [ a :GET ;
                   :output "text/html" ;
                   :semanticRef "getSomeData" ] ] ;
  :hasQoS [ a :QoSSpecification ;
          :utility "0.9" ] ;
  :requires [ a :ReqSpecification ;
          :resType ex:Type3 ; :times "1" ] .
            [ a :ReqSpecification ;
          :resType ex:Type7 ; :times "1" ] .
```

Fig. 7. Resource description.

4.3 Service Management

`Service Management` monitors the utility of the local service through `LocalUtilityMonitor`, and forwards detected changes to `UtilityManager`, which in turn recomputes the compound utility and notifies the `GossipManager`.

It is worth noting that `LocalUtilityMonitor` exploits the *semantic-aware* PRIME Resource Description mechanism for implementing the matching function $matches : \mathbf{T} \times \mathbf{T} \rightarrow [0, 1]$ (defined in Section 2). Specifically, following the P-REST architectural style imposed by the PRIME middleware, a *Service* is implemented as a `GOPrimeApplication`, which exposes information of interest through the instantiation of Resources. In turn, Resources are provided/consumed through the P-REST uniform interface, and must be described by means of the PRIME Resource Description Ontology, which specifies the set of concepts needed to properly advertise/retrieve resources of interest to/from the networking environment. In particular, a `Description` is composed of (i) `aURI` and `cURI`, which define the *Service Type* implemented by the resource and its concrete identifier, respectively; (ii) the functional description, which describes the set of functionalities offered by the resource, and (iii) the contextual properties of the resource (e.g., the geographic coordinates).

Exploiting such a mechanism, the matching function is redefined as $matches : \textbf{aURI} \times \textbf{aURI} \rightarrow [0, 1]$ and exploits the *signature matching* algorithm [12] to check whether provided and required *Service Type* satisfy one of the following subsumption relationships: (i) if no subsumption relation exists between the two types (*fail* matching) then $matches$ returns 0, (ii) if the required type subsumes the provided one (*subsume* matching) then $matches$ returns $1/3$, (iii) if the provided type subsumes the required one (*plugin* matching) $matches$ returns $2/3$, and (iv) if the types are equivalent (*exact* matching) then $matches$ returns 1.

GOPRIME extends the PRIME Resource Description Ontology by defining the set of concepts needed to specify the Utility of the local service, as well as the set of its dependencies. For instance, referring to the example presented in Fig. 4c, Fig. 7 shows the semantic-aware description for the Resource implementing S_2. In particular, the `Description` defines a service identified as `S2` (the `cURI` attribute) of type `Type2` (the `aURI` attribute), which implements a `GET` method. Further, `S2` declares the current compound utility (`hasQoS` attribute), and its dependence on two service types, `Type3` and `Type7` respectively. Each dependency

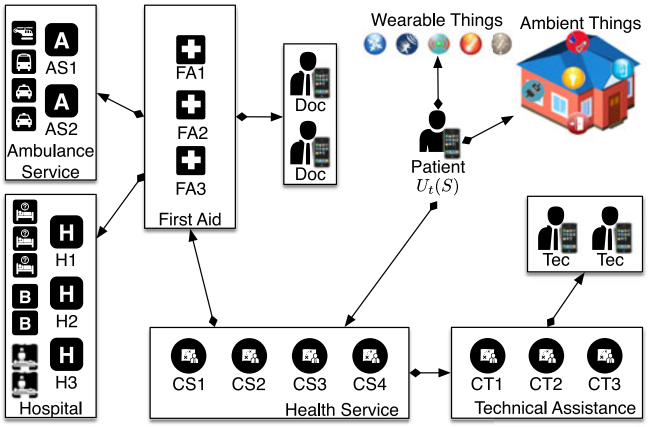

Fig. 8. Case study: the eHealth scenario.

declares a `times` attribute that specifies the number of times such a dependency is resolved during the execution of the local service (see Section 2.2).

5 EXPERIMENTAL RESULTS

The GOPRIME assessment carried out in this section is twofold and concerns (i) its suitability when dealing with a real world application (Section 5.1), and (ii) its scalability, convergence speed and robustness (Section 5.2).

5.1 GOPRIME in Action: eHealth Application

The experiment presented in this section aims to validate the utility-aware decentralised service assembly approach implemented by GOPRIME through a real world case study, namely the eHealth application.

The eHealth application aims at (i) monitoring elder people's *health parameters* (e.g., weight, blood pressure, heart rate, etc ...), as well as their *daily activity* (e.g., sleeping, eating, walking, etc ...), and (ii) raising *health alarms* whenever either the health parameters or the activity deviate from usual (e.g., the patient is sleeping too much and/or the blood pressure is too low). To this end, the eHealth application is built as composition of a set of services. Fig. 8 depicts the eHealth scenario by highlighting the set of involved service types (the labeled boxes) and service instances (the icons within each box). For each service type, arrows point to other service types it depends on. Specifically:

1) Wearable Things sense health parameters and provide them to *Patient*.
2) Ambient Things monitor daily activities and provide them to *Patient*.
3) Patient analyses sensed data and, in case of anomaly detection, issues a query for a (medical or technical) *Health Service*, specifying its non-functional requirement. In this experiment, we assume that it is "minimise the Response Time", which corresponds to $max(U_t(HealthService))$ according to Equation (5).
4) Health Service receives the alarm and, depending on its type (i.e., medical or technical), alerts *First Aid* or *Technical Assistance*, accordingly.
5) Technical Assistance, upon receiving a Technical alarm, sends a *Technician* to the patient's home.

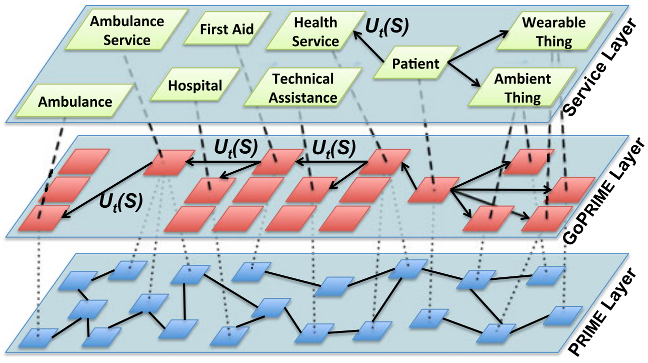

Fig. 9. The eHealth system model.

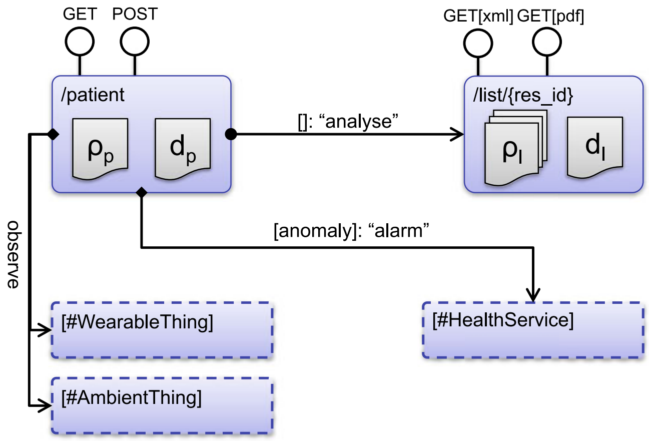

Fig. 10. PRIME-based eHealth software architecture.

6) First Aid contacts the *Ambulance Service*, and makes a reservation at the *Hospital*.
7) Hospital reserves *Analysis Laboratory* and *Operating Rooms* to efficiently manage the alarm.
8) Ambulance Service selects an *Ambulance* according to the requirements for the ongoing emergency.

Fig. 9 describes the eHealth system model, which is defined according to a three layer schema composed of *Service Layer*, GoPRIME *Layer*, and PRIME *Layer*.

In general, the *Service* layer specifies the set of Service Types (defined as aURI) involved in the running application. Referring to the eHealth scenario, *Service* layer in Fig. 9 describes (i) the set of services composing the eHealth system (i.e., *Wearable Thing*, *Ambient Things*, *Patient*, etc.), and (ii) the non-functional requirement $max(U_t(HealthService))$ and the dependencies declared by *Patient*, i.e., the service initiating the interaction.

The GoPRIME layer manages the selection of actual services satisfying the non-functional requirement. GoPRIME encapsulates the descriptions of those services that match each type. GoPRIME layer provides a vitalisation layer used to introduce a further degree of indirection, enabling for loose binding between *Service Types* and their implementing PRIME instances. Indeed, the Service types are not directly bound to their concrete implementations. Besides, decoupling the Service Types from their concrete implementations, achieves the flexibility degree required for supporting run-time adaptation. Once specified the Service types and the set of descriptions, services are bound to instances at run time by means of the PRIME binding mechanisms. Still referring to the eHealth scenario, GoPRIME layer in Fig. 9 maps the abstract set of Service Types defining the eHealth system to the set of Service Descriptions matching such types. For example, the *Wearable Thing* type is matched by two different Service Descriptions (e.g., d_1 and d_2), whereas the *Patient* type is matched by one Service Description (e.g., d_7).

The PRIME layer manages the life-cycle of components implementing the service descriptions. PRIME layer contains the set of all possible component instances implementing the Service Types specified within the *Service layer*, as well as other companion components that might be used to support the computation. Each Service Type might be implemented by several components that vary from each other in terms of QoS properties (e.g., availability and reliability). Note that the GoPRIME layer plays the role of filtering layer: (i) a description specifies both functional and non-functional requirements for the service, and (ii) the PRIME binding mechanism makes use of such descriptions for selecting the component instance, among all the available ones, which provides the needed functionality and satisfy QoS requirements. For example, referring to Fig. 9 the Service Descriptions d_1 and d_2 refer to, respectively, component c_{14} and c_{11}, whereas the description d_7 refers to the component c_8.

Since the underlying components are implemented by means of PRIME, services must be designed and implemented by adhering to the P-REST architectural style. To this end, Fig. 10 presents an excerpt of the eHealth Application software architecture, specified according to the P-REST metamodel [6], which shows the architectural design of the `patient` service: let `/patient` be the GoPRIME Resource representing the Patient service in Fig. 8. `/patient` includes a `/list` resource, which can be accessed by following the link labeled as `store`. Indeed, `/list` is defined as a `Resource Set` of resources `{res_id}`. Moreover, `/patient` makes use of three abstract resources, namely `#WearableThing`, `#Ambient-Thing` and `#HealthService`.

At run-time, data will be actually `read` from concrete resources belonging to the classes `#WearableThing`, and `#AmbientalThing`, and stored as a list of values: the actual URIs of the resources identified by `{res_id}`, are derived in two steps: (1) considering the `include` relation from `/list/{res_id}` to `/patient`, then obtaining the `/list/{res_id}` URI Template, and (2) substituting the `res_id` variable with actual values. This results in a set of n distinct resources identified by `/patient/list/1`, `/patient/list/2`, ...`/patient/list/n` URIs, respectively. Patient keeps analysing data in `list` and, in case of anomaly detection, sends an `alarm` to the actual concrete resource belonging to `#HealthService` and matching the non-functional requirement $max(U_t(HealthService))$.

Fig. 11 shows the service assembly made by GoPRIME from the Patient perspective, with response time-based compound utility defined in equation (5). Specifically, once the Patient has declared the $max(U_t(HealthService))$ requirement for the final assembly, GoPRIME gossips compound utilities until all dependencies are iteratively resolved and the response time of the resulting assembly is minimised.

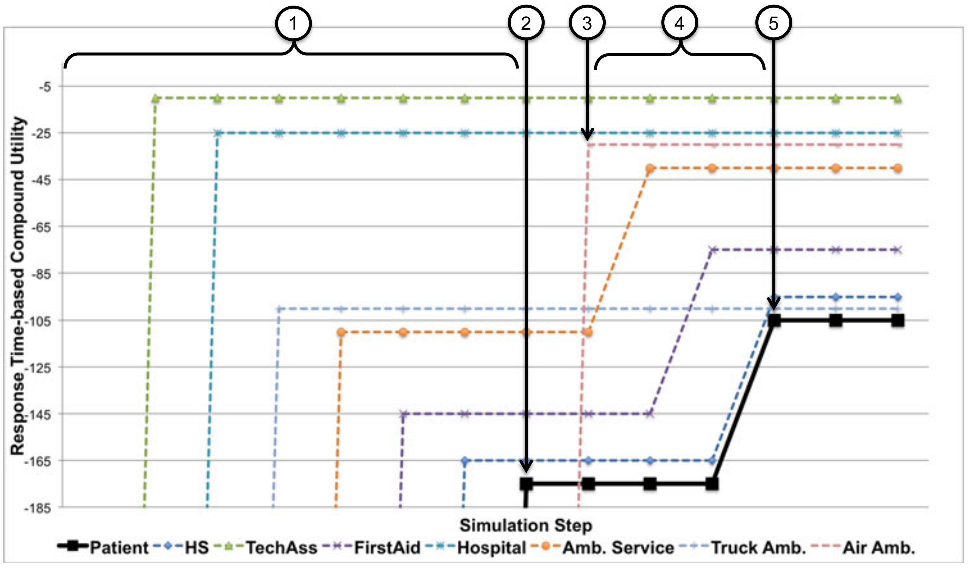

Fig. 11. GoPrime-based patient assembly.

Initially (at step s_0), the compound utilities (expressed in units of time ut) are: *Truck Ambulance* $= -100ut$, *Ambulance Service* $=\perp$, *Hospital* $= -25ut$, *First Aid* $=\perp$, and *Technical Assistance* $= -10ut$. Hence, applying the above equation it results *Health Service* $=\perp$, as some dependencies are still not resolved. From step s_0 to step s_6 (see ①), GoPrime gossips local utilities and resolves the dependencies by recursively calculating compound utilities. At step s_6, the compound utilities are: *Truck Ambulance* $= -100ut$, *Ambulance Service* $= -100 - 10 = -110ut$, *Hospital* $= -25ut$, *First Aid* $= -10 - 25 - 110 = -145ut$, *Technical Assistance* $= -10ut$, and *Health Service* $= -10 - 10 - 145 = -165ut$. Hence, at step s_7 (see ② the dependencies for *Patient* are resolved, and its compound utility is $U_t = -10 - 165 = -175ut$.

Fig. 11 also shows how the Patient's compound utility changes as soon as a faster ambulance, namely *Air Ambulance* with $U_t = -30ut$, becomes available (see ③). When *Air Ambulance* appears, GoPrime gossips its response time-based utility, and reassembles the service, accordingly (see ④). Specifically, *Air Ambulance* $= -30ut$, *Ambulance Service* $= -10 - 30 = -40ut$, *Hospital* $= -25ut$, *First Aid* $= -10 - 25 - 40 = -75ut$, *Technical Assistance* $= -10ut$, and *Health Service* $= -10 - 10 - 75 = -95ut$. Hence, at step s_7 (see ⑤ the new compound utility for *Patient* is $U_t = -10 - 95 = -105ut$.

It is worth noting that, for the sake of simplicity, such experimentation has been carried out by considering a single attribute for the compound utility, namely U_t as defined in equation (5). Alternatively, Patient can specify a multi-attribute requirement that aims to balance response time and some other quality attribute (e.g., cost). For example, to avoid using the costly Air Ambulance when not needed, Patient can combine U_t and U_c (i.e., the cost-based compound utility defined in Equation (3)) by means of either SAW or Pareto technique (as defined in Section 2.3), to specify a non-functional requirement that balances cost and response time.

5.2 Scalability and Robustness Analysis

In order to test the effectiveness of GoPrime on a larger scale than our case study allows, we implemented a simulation model using the cycle-based engine of the PeerSim [22] simulator. PeerSim is a free Java package designed for efficient simulation of Peer-to-Peer protocols; the cycle-based engine it provides 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 a performance metric (number of interactions) has the advantage of being independent from the details of the underlying hardware and network infrastructure.

Model parameters. Using the same notation from Table 1, we consider a system with T interface types and $C \geq 1$ services of each type, so that $N = C \times T$. Moreover, to perform the experiments in the least favourable conditions, we assume that each peer hosts a single service. This implies that N is also equal to the number of peers; hence, in the following we will refer to N as the "system size".

For each service we define D random dependencies. Abstracting from specific utility definitions (like those described in Section 2.2), each service S is assigned a scalar utility $S.Util$ that is uniformly distributed in $(0, 1)$, and we compute the compound utility of a fully resolved assembly as the product of utilities of individual services. For each type $d \in \mathbf{T}$ we randomly choose $N_{\text{opt}} \geq 1$ services of type t and set their utility to 1; this ensures that a maximum compound utility value of 1 can always be achieved, by binding together those services.

We point out that, even if we adopt a scalar utility in our experiments, the obtained indications extend also to the case of a non-scalar utility compared according to Pareto dominance. Indeed in this case GoPrime would drive the system towards the construction of an assembly whose utility belongs to the corresponding Pareto front. Hence, the Pareto front would play the role of the maximum achievable utility value of 1 in the scalar setting of our experiments, and the cardinality of the Pareto front would correspond to the N_{opt} value.

TABLE 2
Simulation Parameters

D	Number of dependencies per node
K	Number of neighbours returned by GetPeers() function
C	Number of instances of each service type
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

Performance measures. We consider two metrics: the fraction R_t of fully resolved services at simulation step t, $0 \leq R_t \leq 1$, and 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 maximum value of U_t is 1. Unless stated otherwise, all results are computed by taking the average of ten independent simulation runs.

Table 2 summarises the simulation parameters. We now report the results in different scenarios.

System size. We first evaluate the mean number of iterations that are necessary to produce fully resolved assemblies, for increasing values of the system size N, $1{,}000 \leq N \leq 15{,}000$. This is important for understanding the scalability of the proposed gossiping scheme. We performed a simulation with $T = 50$ different service types, each service having $D = 10$ dependencies. 25 percent of the instances of each type have maximum utility 1. At each iteration, each node exchanges state information with $K = 20$ other nodes.

Fig. 12 shows the average number of iterations that are required to resolve all dependencies and achieve a compound utility greater than 0.99 (out of the maximum value 1), for each service in the system. As it can be expected, more iterations are required to achieve the desired maximum utility for increasing system size. However, the number of iterations increases quite slowly, suggesting the logarithmic growth typical of gossip protocols [8].

Besides the number of iterations to fulfill the system requirements, another important factor that can affect scalability is the amount of exchanged information among nodes in the network. From line 8 of Algorithm 1, we see that each node sends a message whose size is proportional to $K \times (1 + |S.Deps|) \leq K \times (1 + T)$, independent of the system size N. Hence, the overall amount of information exchanged at each round grows linearly with the system size.

The ability of GoPrime to resolve dependencies quickly depends on the number of instances of the service types in the system: if there are only a few instances of each service type, the gossip protocol requires more iterations to build fully resolved assemblies. This is shown in Fig. 13 where we consider a system with $N = 5{,}000$ services and a variable number of instances C for each type, $C \in \{2, 5, 10, 50, 100\}$. The number of iterations required to produce a (not necessarily optimal) fully resolved assembly steeply increase as the number of instances of each type decreases. This can be improved by tuning the value of parameter K (number of peers of each node) as described next.

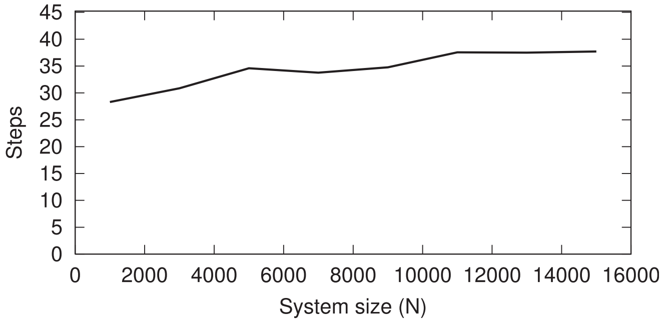

Fig. 12. Mean number of iterations required to produce fully resolved assemblies with utility at least 0.99. There are $T = 50$ service types Each data point is the average of 50 independent simulation runs. Lower is better.

Number of neighbours. We now examine the impact of the value of K (number of neighbour peers returned by the GetPeers() function) on R_t and U_t; specifically we consider $K \in \{10, 20, 50\}$. We consider $N = 1{,}000$ services of $T = 50$ different types; each service has $D = 10$ randomly chosen dependencies. For each type $t \in \mathbf{T}$, there is a single service with maximum utility ($N_{\mathrm{opt}} = 1$).

Fig. 14 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 communicates with $K = 10$ peers, we observe that all dependencies are resolved in about 10 interactions (bottom part of the figure). The average utility grows monotonically, as expected (recall the discussion in Section 3.3); however, the growth is slow, and the utility tends to stabilise around a value that is below the maximum, which, by construction, is 1. This can be explained by observing that, in order to build an assembly with maximum utility, the algorithm needs to locate the (unique) service of each needed type with utility set to 1. Since only interactions with peers are allowed, this process is very slow over networks with limited degree. The situation improves by increasing the number K of peers to communicate with at each iteration, or if multiple services with maximum utility 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 exist multiple different ways to build an assembly with maximum utility. Fig. 15 shows the results with $N_{\mathrm{opt}} \in \{1, 10, 20\}$. Increasing

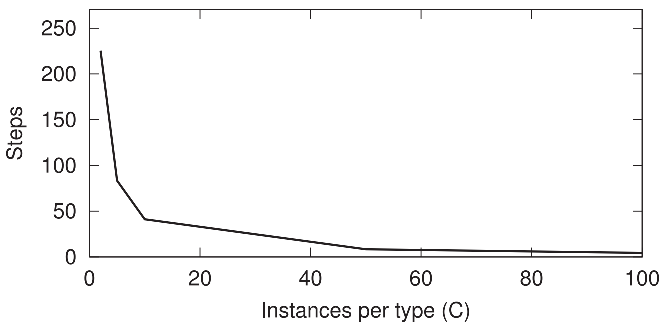

Fig. 13. Mean number of iterations required to produce fully resolved assemblies with $N = 5{,}000$ services and variable number of instances per type C. Lower is better.

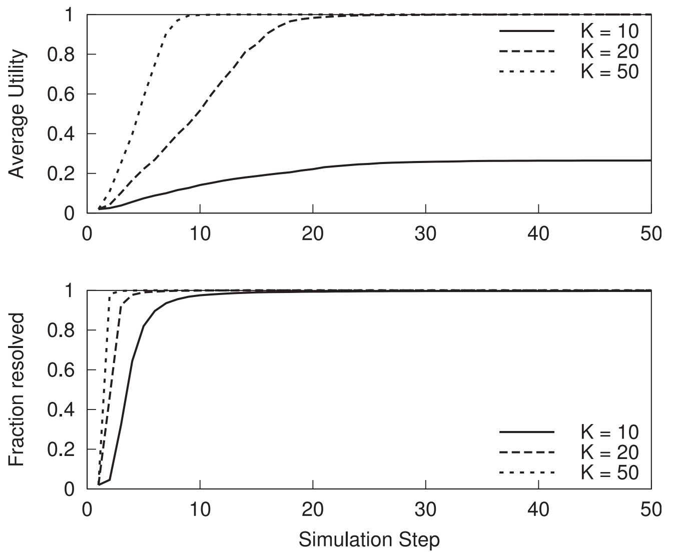

Fig. 14. Average utility (top) and fraction of resolved services (bottom) for different values of the number of peers K; $N = 1{,}000$, $D = 10$, $N_{\text{opt}} = 1$. Higher is better.

N_{opt} values allow the algorithm to produce service assemblies more quickly with higher utility; on the other hand, 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. 15).

Number of dependencies. In this experiment we study how the number of dependencies D influences the algorithm convergence speed. We set $N = 1,000$, $T = 50$, $K = 20$ and $N_{\text{opt}} = 1$. We set $D \in \{5, 10, 20\}$ random dependencies on each service.

The results are shown in Fig. 16. We observe that, as the number of dependencies increase, so does the convergence speed towards the maximum utility of the service assembly. This may appear counterintuitive at first, but can be explained by considering that each peer sends its list of resolved (immediate) dependencies to its peer set during interactions. Since the goal of each peer is to maximise its

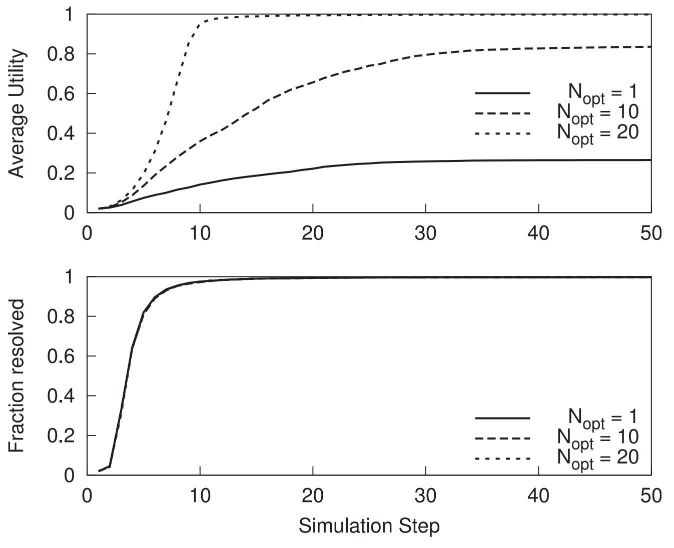

Fig. 15. Average utility (top) and fraction of resolved services (bottom) for different values of the number of services with maximum utility N_{opt}; $N = 1{,}000$, $T = 50$, $K = 10$. Higher is better.

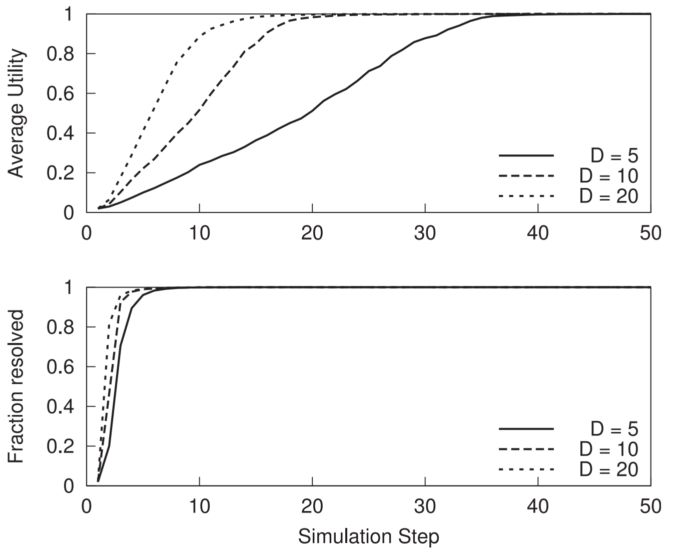

Fig. 16. Average utility (top) and fraction of resolved services (bottom) for $N = 1{,}000$, $T = 50$, $K = 20$, $N_{\text{opt}} = 1$ and different values of the number of dependencies D. Higher is better.

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 faster locate dependencies with higher utility.

Handling failures. Every large collection of distributed services 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 [8]. We study the resilience of Algorithm 1 by considering again a set of $N = 1{,}000$ services of $T = 50$ different types. Each service has $D = 10$ random dependencies. For each service type, we assign utility 1 to $N_{\text{opt}} = 10$ different peers. Every ten simulation steps we remove 40 randomly selected services.

Fig. 17 shows the average utility and fraction of fully resolved services for different values of K, the number of neighbours of each peer returned by GetPeers() function. After each failure, we clearly see a sharp reduction of both the average utility and the fraction of resolved services. However, the algorithm quickly works around failed nodes and stabilises itself near a new optimal configuration within a few steps. This gives rise to a piecewise monotonic non-decreasing behaviour, as discussed in Section 3.3. Again, we see that for the smallest considered value of K ($K = 10$) the algorithm provides assembly with utility below the maximum, within the considered time window. As we discussed above, when each node has a limited number of peers to talk with, then information diffusion slows down and the system tends to stabilise around a suboptimal configuration (with respect to the maximum achievable utility). Despite that, almost all services become quickly fully resolved, as shown in the bottom part of Fig. 17.

6 Related Work

6.1 Architectures for Self-Adaptation

It has been widely recognised that the architecture of self-adaptive software systems should include one or more control loops to perform self-adaptation tasks [4]. A notable example of a general approach based on this idea is the

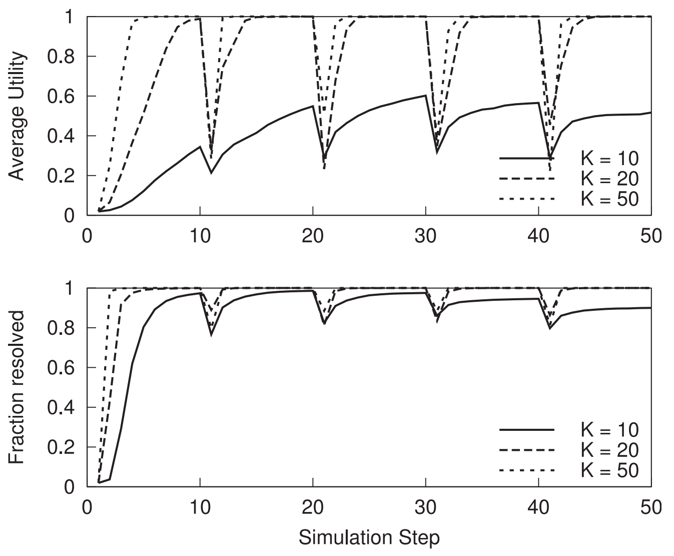

Fig. 17. Average utility (top) and fraction of resolved services (bottom) when 40 random services are removed every 10 simulation steps; $N_{\mathrm{opt}} = 10$. Higher is better.

autonomic computing framework and the related MAPE-K (Monitor, Analyse, Plan, Execute, and Knowledge) reference model of an autonomic system [2], [3]. The MAPE-K architecture proposed in [2] adopts a centralised hierarchical organisation. However, the work in [5] clearly contrasts decentralised self-adaptive systems with their centralised counterparts, and highlights the importance of decentralised control to achieve quality requirements such as resilience, robustness and scalability in large distributed systems. That work also discusses some key research challenges for the realisation of decentralised self-adaptation. A deep investigation of possible architectural patterns for decentralised management of MAPE-K loops in self-adaptive systems has been presented in [23]. In this respect, GoPrime follows the *information sharing* pattern presented in [23], where the MAPE-K loops executed at each node interact through their Monitor components for information sharing, while the Analyse, Plan and Execute components operate independently of corresponding components at other peers, for local analysis, planning, and execution of adaptations. In particular, Algorithm 1, which implements the gossip-based information dissemination, corresponds to a decentralised implementation of the Monitor operations, while functions $\textsc{SelectFromBest}_d()$ and $\textsc{UpdateBest}_{S,d}()$ locally implement the operations of the Analyse and Plan components (see Algorithms 2, 3 and 4).

Another general reference model for the architecture of a self-adaptive software system has been presented in [24]. This paper suggests to architect the system along three different layers, which interact with each other by reporting status information to the above layer and issuing adaptation directives to the layer below. The bottom layer (*component control*) is concerned with adaptation at the level of single components (i.e., services in the SOA domain). The middle layer (*change management*) reactively uses a pre-specified set of plans to adapt the system consisting of components at the lower layer. When these plans are no longer able to meet the system goals, or when new goals are introduced, the upper layer (*goal management*) determines new adaptation plans. From the viewpoint of this three-layer reference model, GoPrime basically corresponds to a decentralised implementation of the middle layer, which interacts on the one side with the bottom layer consisting of the managed services and on the other side with the upper layer that, in the GoPrime architecture described in Section 3.1, is implemented by the Goal Manager component.

6.2 Dynamic Service Assembly

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., [25], [26]) or on dynamic service planning (e.g., [27], [28]). In this section we briefly review papers based on dynamic service assembly, which are the ones closest to our approach. In particular, since our focus is on the development of a fully decentralised solution, we only consider papers adopting a similar approach.

The work in [25] 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 [26] overcomes some of the limits of [25]. This work is also the closest to GoPrime as it adopts a gossip-based adaptive decentralised 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 [26], our decentralised 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.

While FlashMob employs a top-down approach to resolve dependencies, GoPrime uses a bottom-up strategy to build fully resolved assemblies satisfying structural and QoS requirements. To do so, some extra work is done by each peer, also those that will not be part of the final "best" assembly. This extra work is, however, paid back by the ability to operate without global knowledge, and the robustness properties that can be obtained.

Some works [1], [7], [29], [30] deal with the problem of managing dynamic organisations of agents in a decentralised way, i.e., agents that may dynamically form specific subsets (organisations) 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. The MACODO organisation model and the related middleware for the management of dynamic organisations of agents adopts an architecture that is only partially decentralised [29], [30]. Indeed, each agent organisation is based on a master-slave schema, where the master has complete knowledge of the organisation state and controls the organisation dynamics in a centralised way. The masters of different organisations can then cooperate to achieve some common goal (for example by merging their respective set of agents into a single organisation), exchanging to this end some reduced state information. Kota et al. [7] presents a decentralised approach where each agent periodically contacts a subset of its peers to determine the composition of the organisation 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, [7] suggests to randomly select a limited subset. This guarantees that, eventually, all peers will be contacted. Schuhmann et al. [1] deals with distributed pervasive applications and proposes configuration algorithms for homogeneous and heterogeneous environments. The goal of these algorithms is to choose the most efficient configuration method for a given environment while minimising the configuration latency.

7 Conclusions

In this paper we have presented GoPrime, a fully decentralised middleware solution for the adaptive self-assembly of distributed services. The core element of GoPrime 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 decentralised way an assembly of services that, besides functional requirements, is able to fulfill global quality of service 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.

GoPrime relies on a suitably defined, application-specific *utility function* to steer the system towards a state where all dependencies are resolved, and the utility of the whole assembly (*compound utility*) is maximised. The utility function must be defined recursively: the utility of a non-leaf instance depends on its local utility, and the utilities of its dependencies. We therefore do not allow a service instance to be part of a cycle, since in that case the assembly would never be resolved. Note that we do allow different instances of the same service *type* to appear in a fully resolved assembly; we do not, however, support the possibility for the same service *instance* to be part of a cycle. How this limitation can be relaxed is subject of ongoing research.

We have shown the validity of our approach presenting results from the experimentation of a prototype implementation of GoPrime in a real world e-health application, and an extensive set of simulation experiments that assess the effectiveness of GoPrime in terms of scalability, robustness and convergence speed towards the optimal assembly.

Future work includes a validation of the approach in a real industrial setting. We plan also to extend GoPrime with the introduction of load balancing mechanisms to make it able to deal with load-dependent utility. Another direction of research includes the extension of GoPrime with the context-aware adaptation capabilities, e.g., based on physical proximity of nodes used in the assembly.

References

[1] S. Schuhmann, K. Herrmann, K. Rothermel, and Y. Boshmaf, “Adaptive composition of distributed pervasive applications in heterogeneous environments,” *ACM Trans. Auto. Adaptive Syst.*, vol. 8, no. 2, pp. 10:1–10:21, Jul. 2013.

[2] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *IEEE Comput.* vol. 36, no. 1, pp. 41–50, Jan. 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, pp. 7:1–7:28, 2008.

[4] B. H. C. Cheng, et al., “08031—software engineering for self-adaptive systems: A research road map,” in *Software Engineering for Self-Adaptive Systems*, series. Dagstuhl Seminar Proceedings, vol. 08031. Berlin, Germany: Springer-Verlag, 2008, pp. 1–26.

[5] D. Weyns, S. Malek, and J. Andersson, “On decentralized self-adaptation: Lessons from the trenches and challenges for the future,” in *Proc. Workshop Softw. Eng. Adaptive Self-Manag. Syst.*, 2010, pp. 84–93.

[6] M. Caporuscio and C. Ghezzi, “Engineering future internet applications: The PRIME approach,” *J. Syst. Soft.*, vol. 106, pp. 9–27, 2015.

[7] 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.

[8] 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.

[9] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, “Gossip-based peer sampling,” *ACM Trans. Comput. Syst.*, vol. 25, no. 3, pp. 8:1–8:36, Aug. 2007.

[10] D. Shah, *Gossip Algorithms*, series. Foundations and trends in networking. Delft , The Netherlands: Now Publishers, 2009.

[11] V. Grassi, M. Marzolla, and R. Mirandola, “QoS-aware fully decentralized service assembly,” in *Proc. 8th Int. Symp. Softw. Eng. Adaptive Self-Manage. Syst.*, 2013, pp. 53–62.

[12] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara, “Semantic matching of web services capabilities,” in *Proc. 1st Int. Semantic Web Conf.*, 2002, pp. 333–347.

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

[14] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, 2nd ed. Chichester, U.K.: Wiley, 2002.

[15] J. Cardoso, “Complexity analysis of BPEL web processes,” *Softw. Process: Improvement Practice*, vol. 12, no. 1, pp. 35–49, 2007.

[16] 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. Softw. Eng.*, vol. 38, no. 5, pp. 1138–1159, Sep./Oct. 2012.

[17] C. Hwang and K. Yoon, *Multiple Criteria Decision Making*. Lecture Notes in Economics and Mathematical Systems. New York, NY, USA: Springer, 1981.

[18] M. J. Shepperd, *Foundations of Software Measurement*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1995.

[19] M. Harman, S. A. Mansouri, and Y. Zhang, “Search-based software engineering: Trends, techniques and applications,” *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, Dec. 2012.

[20] M. Ehrgott, *Multicriteria Optimization*, series. Lecture Notes in Economics and Mathematical Systems. New York, NY, USA: Springer-Verlag, 2000.

[21] G.-C. Roman, G. P. Picco, and A. L. Murphy, “Software engineering for mobility: A roadmap,” in *Proc.Conf. Future Softw. Eng.*, 2000, pp. 241–258.

[22] A. Montresor and M. Jelasity, “PeerSim: A scalable P2P simulator,” in *Proc. 9th Int. Conf. Peer-to-Peer Comput.*, Seattle, WA, USA, Sep. 2009, pp. 99–100.

[23] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. Goeschka, “On patterns for decentralized control in self-adaptive systems,” in *Proc. Int. Seminar Softw. Eng. Sel-Adaptive Syst.*, 2012, pp. 76–107.

[24] J. Kramer and J. Magee, “Self-managed systems: An architectural challenge,” in *Proc. Future Softw. Eng.*, 2007, pp. 259–268.

[25] I. Georgiadis, J. Magee, and J. Kramer, “Self-organising software architectures for distributed systems,” in *Proc. 1st Workshop Self-Healing Syst.*, 2002, pp. 33–38.

[26] D. Sykes, J. Magee, and J. Kramer, "Flashmob: Distributed adaptive self-assembly," in *Proc. 6th Int. Symp. Softw. Eng. Adaptive Self-Manage. Syst.* 2011, pp. 100–109.

[27] M. E. Falou, M. Bouzid, A.-I. Mouaddib, and T. Vidal, "A distributed planning approach for web services composition," in *Proc. IEEE 19th Int. Conf. Web Serv.*, 2010, pp. 337–344.

[28] 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, Jul. 2007.

[29] D. Weyns, R. Haesevoets, A. Helleboogh, T. Holvoet, and W. Joosen, "The MACODO middleware for context-driven dynamic agent organizations," *ACM Trans. Auton. Adaptive Syst.*, vol. 5, no. 1, pp. 3:1–3:28, 2010.

[30] D. Weyns, R. Haesevoets, and A. Helleboogh, "The MACODO organization model for context-driven dynamic agent organizations," *ACM Trans. Auton. Adaptive Syst.*, vol. 5, no. 4, p. 16, 2010.