Abstract –
This deliverable describes the approach and the architecture of a distributed infrastructure for discovery engines. In this architecture, several independent discovery engines can interact by exchanging goals and by leveraging over the mutual service descriptions and matchmaking capabilities. This architecture is based upon a reputation model that is described in D4.1.2 (version 2)
Contents

1 Introduction 4

2 State of the Art 5
  2.1 Web Service Discovery ........................................... 5
    2.1.1 UDDI ......................................................... 5
    2.1.2 ebXML ..................................................... 6
    2.1.3 WSMO, OWL-S, SAWSDL .................................. 6
    2.1.4 Dedicated Service Portals .................................. 7
  2.2 Distributed Web Service Discovery .............................. 7
    2.2.1 UDDI Distribution Model .................................. 7
    2.2.2 ebXML Registry Federation ................................ 8
    2.2.3 Service-Finder ............................................ 9

3 Distributed Discovery 10
  3.1 Scenario ......................................................... 10
  3.2 Metamodel design ................................................ 12
    3.2.1 Public information representation .......................... 14
  3.3 Distributed discovery process .................................. 17

4 Conclusions 18

A Distributed Glue Ontology 19
1 Introduction

The Web is moving from a collection of static documents to a collection of services. For realizing service interchange in a business to business setting, the service-oriented architecture together with the Web Services technology are widely seen as the most promising foundation. As a result, considerable attention has been given in both research and industry to Web Services and related technologies.

Within a Web of services in general and in any service-oriented-architecture (SOA), in particular the discovery of services, is the essential building block for creating and utilizing dynamically created applications. And discovery becomes a more critical aspect in Web scenarios where the scale of the problem raises dramatically.

Existing Web Service discovery approaches focus in setting up registries to manage service descriptions and to matchmake those with consumer's requests. Such matchmaking can be based on syntactic aspects (e.g. in UDDI or ebXML) or semantic aspects (e.g. in OWL-S, WSMO, SAWSDL). However, when scaling to the Web size, current approaches fail in two main aspects:

- they rely on logically centralized registries that expose an interface to publish web service descriptions and to submit discovery requests; and
- they don't consider the heterogeneous nature of the Web where a single request can be fulfilled by similar services that are just described accordingly slightly different points of view.

Some research efforts have been made to overcome the first problem by providing a distributed infrastructure for registries that enable distributed storage, federated queries or intelligent routing toward the best discovery engine. Nevertheless, those efforts don't treat with different perspectives since they don't consider the capabilities of dedicated "mediators" able to mediate discovery requests from one point of view to others.

This deliverable describes an architecture for distributed discovery focused on mediators and a reputation model. Mediators translate discovery requests expressed in terms of functional and non-functional requirements among different point of view maintaining the compliance with constraints and QoS aspects. The reputation model is used to drive the discovery approach among the various discovery engines participating to the distributed infrastructure.

Chapter 2 summarizes the state of the art of Web Service discovery and distributed discovery. Chapter 3 describes the distributed discovery architecture we designed, depicting the main aspects of the reputation model that is further detailed in the scope of deliverable D4.1.2 version 2.
2 State of the Art

This chapter describes the most relevant approaches for Web Service Discovery and the trends to bring such discovery to a larger scale.

2.1 Web Service Discovery

UDDI and ebXML are the two most popular attempts to create Web Service registries for Web service Discovery. Further to those approaches, others exist focusing on other aspects. WSMO, OWL-S and SAWSDL focus on semantic matchmaking. Dedicated service portals leverage over user contributions to collect and provide high-quality information in specific domains.

2.1.1 UDDI

The Universal Description, Discovery and Integration (UDDI)[2, 1, 8] project is a worldwide project and initiative comprising companies such relevant as Computer Associates, IBM, Microsoft, Oracle and SAP.

Basically, it allows to programmatically publish and retrieve a set of structured information belonging to a Web Service by defining open and standard specifications for key Web Services Registry functionalities:

- Description of services and their providers
- Publication of information about services and their providers
- Discovery of services and/or providers of those services
- Creation and deployment of XML-based distributed registries

The UDDI family of specifications is not a set of specifications developed exclusively to solve the registry problem in the Web Services area. In fact, the UDDI specifications are platform agnostic and can interact with businesses and services deployed on a variety of platforms. This much broader approach is clearly stated in the launching documents of the consortia set up to define the UDDI specifications: "The UDDI Project is an open industry initiative in which any organization can participate and implement the specifications. The specifications build on core Internet standards including TCP/IP, HTML, and XML and are independent of any underlying platform, language, object model, business application, or marketplace."

To manage this general purpose registry solution, UDDI introduces the concept of information model in the sense that every entity susceptible of being published in a UDDI registry has to be previously modeled using the tools and mechanisms defined by the UDDI specifications and presented further on. These tools and mechanisms allow the modeling of businesses and services respecting the platform agnostic principle characteristic of the UDDI specifications.

The UDDI information model is based in the concept of categorization. Every entity published in a UDDI registry can be categorized in depth using a fairly basic mechanism based on key-value pairs. To facilitate the categorization of the entities, the UDDI specifications
include a set of standard categories that can be directly referenced and used like, for example, UNSPSC for product and service categorizations and ISO 3166 for the characterization of geographical regions. Anyway, nothing impedes the publisher of an entity to define new category sets or even to enrich existent ones.

Several companies have operated public UDDI repositories, however due to several shortcomings of the approach such as complicated registration, missing monitoring facilities, its success was limited and only a few repositories are still publicly available. At the same time a number of Portals dedicated to providing a repository of services have appeared. However, all of them rely on a manual registration and review process, which implies limited coverage as well as inherently outdated information. Alternatively, one can use the classical search engines; however, they do not provide effective means to identify Web Services. For now, there exists no standardized file suffix, such that a query like "filetype:wsdl" does not match all service descriptions (e.g. the wsdl description Microsoft Services will have the ending ".asmx?wsdl"). Moreover, a standard search engine does not make any pre-filtering based on availability and other service-related parameters; their retrieval model is optimized for finding content and not dynamic services.

2.1.2 ebXML

ebXML is a set of standards intended to provide an open, interoperable and secure electronic business infrastructure for trading partners layered on top of Web Service standards. Thereby the ebXML specifications are focused on defining concepts and common methodologies that can be applied to enable efficient and interoperable implementation of eBusiness solutions by covering functionality from various aspects of B2B interactions [10].

Since an ebXML registry essentially provides generic services for content and metadata management [3], it can support a number of potential use cases. In the context of a service-oriented architecture (SOA), an ebXML registry can be used for instance as (i) a Web Service Registry to support publication and discovery of Web Service descriptions (e.g. in form of WSDL [7] descriptions) and (ii) as a registry and repository for business process descriptions (e.g. in form of BPEL [4] processes). Furthermore, the ebXML registry can be used for serving a controlled vocabularies such as taxonomies or XML schema definitions.

2.1.3 WSMO, OWL-S, SAWSDL

Several Description Frameworks (WSMO[11], OWL-S[12], SAWSDL [9]) provide the semantic descriptions needed for dynamic location of Web Services that fulfill a given request. They allow, given a goal, the dynamic location of a Web Service. The focus so far has been to capture the functionality of the service in a sufficient level of detail, to ground the accurate matching of requester goals and Web Services.

Little research has been conducted on the aspect of how to obtain those descriptions and on other aspects of a service description such as its non-functional properties.

Moreover, the approaches developed so far are based on two assumptions: the discovery engine knows all the Web Service descriptions and, the Web Service descriptions are completely correct because an expert draws them up.

However, in a Web scenario these assumptions are unreasonable. First, the services are located in different locations that could be unknown a priori, and sometimes they are not
available. Secondly, most of the services are described only syntactically, so that their semantics have to be deduced from other available sources, such as service’s documentation, Web pages and so on. For this reason, we cannot assume the correctness of the description of a service. Due to these reasons, current approaches for discovering services are not suitable in a Web scenario.

**Glue2**  
Glue2 is a Semantic Web Service discovery engine that enable providers to describe concisely the functional and non-functional capabilities of their services and consumers to describe their goals in terms of capability requests and preferences. Glue2 is conceptually based on the WSMO meta-model that is extended to distinguish between classes, subclasses and instances of goals and web service respectively. Glue2 is focused on wgMediators that provide the semantic rules to matchmake the requests and preferences expressed in the consumers’ goals with the capabilities and QoS aspects described in the providers’ web service descriptions.

### 2.1.4 Dedicated Service Portals

Meanwhile, Semantic Web Service research continues, a number of dedicated portals for service repository appeared (e.g., xMethods and programmableweb). These portals leverage community contributions to collect and manage services and their descriptions and to provide highly relevant services that are well described in several aspects (including business details). However, all of them rely on a manual registration and review process, which implies limited coverage and inability to reach Web scale.

### 2.2 Distributed Web Service Discovery

Several approaches proposed extensions of UDDI and ebXML to go a step beyond their limitations and to provide efficient and scalable distributed web service discovery. There are also independent projects that adopted different strategies.

#### 2.2.1 UDDI Distribution Model

The UDDI specifications state that a set of Web services that implements at least one UDDI API set (for instance, the "publisher" API) is termed a UDDI node. One or more UDDI nodes may form a UDDI registry, with the restriction that a node may belong to one, and only one, registry. Registries can moreover form hierarchical relationships. While a UDDI node corresponds to a company branch office that maintains phone numbers for its employees, a registry is similar to a company-wide phone directory, of which the branch office directories are a part.

UDDI has two main drawbacks. First, it replicates all public service publications in all UBR (Universal Business Registry) nodes, which is not scalable and efficient, and second, it collects service information in a passive manner, which means it waits for service publication, updating or discovery request passively and thus cannot guarantee the real-time validity of the services information.
As the purpose of this standard is to provide a lightweight registry for SOA, not all the issues of distribution have been tackled yet. In particular, all the resource identifiers used by the registry are defined as local with respect to the registry itself, and this affects how distribution is achieved. Rather than distribution, the approach chosen in the UDDI specifications is indeed replication of data among the nodes forming a registry. Once a new node joins the registry, it asks all the available information from all the other nodes and then synchronizes its data with those of the other nodes.

The issue of overcoming the limitations of UDDI distribution, by either distributing queries or content, is discussed in several works. In [15] a P2P system is proposed to enable query distribution to the nodes of the registry, while the heterogeneity of service descriptions is tackled by using DAML-S ontological description. [13] describes a slightly different approach where registries are federated so that advertisement entries in one registry can be discovered by service consumers who query via other remote registries. In [14], a Web service Discovery Infrastructure (METEOR-S) is described, which provides an ontology-based infrastructure to access a group of registries that are divided based on business domains and grouped into federations. [5] also proposes a P2P architecture, but instead of using it to be aware of the nodes of the registry, it is used to better route the queries to the right UDDI node. [] proposes an active and distributed UDDI architecture, which extends and organizes the private UDDIs based on industry classifications. That approach adopts an active monitoring mechanism, so that service information can be updated automatically and the service requestors may find the latest service information conveniently.

To embrace the UDDI vision, the NeP4B scenario should be formed by a series of UDDI nodes federated into registries, but, being decentralised by definition, each service provider should also maintain its own UDDI node. Moreover, in case of aggregating nodes and registries by domain, as each node can belong to one and only one registry, a service provider offering two services belonging to two different domains should also maintain two different registries. A third argument against the adoption of UDDI as the mean to implement distributed discovery is that, given the two previous arguments, a new node joining a registry should obtain a permission to interact with the other nodes, but no central authority can take the decision in a decentralised network.

### 2.2.2 ebXML Registry Federation

ebXML registry/repository specifications deals with the issues of distributing a registry on multiple nodes, or rather, with the issues of allowing a group of registries to voluntarily form a loosely coupled union. The registries federations can be based on common business/domain interests and specialties that the registries might share, and are based on a peer-to-peer (P2P) model where all participating registries are equal.

This enables operations such as:

- Cross-registry associations
- Federated queries
- Local caching of data from another registry
- Object relocation
While in UDDI replication was the only form of distribution, in ebXML it is just an option. Replication of RegistryObjects in other registries within a federation can improve access time and fault tolerance through local caching of remote objects, as it involves creation of a “local replica” that can be kept current using the event notification feature, or through periodic polling.

Even if ebXML specifications allow more complex ways of distributing content among the participants of a federation, there still exists the problem of deciding whether a new node is allowed to join, as there must be an explicit agreement among the participants of the federation for accepting a new node. Moreover, as in UDDI, it is not clear which actor in our scenario should be responsible of maintaining the infrastructure (either each provider maintain its node in a pure P2P fashion or there is some sort of central or local authority). These issues prevents from adopting this standard for distributed discovery in the project scenario.

2.2.3 Service-Finder

Service-Finder\(^1\) aims at developing a platform for service discovery in which Web Services are embedded in a Web 2.0 environment.

Service-Finder leverages over the paradigm shift in the usage of the Web in general has happened. The role of a Web user is shifting from a passive, only consuming function to one where he actively participates: the so-called “Web 2.0” phenomena. Both Web Service’s technologies and Web 2.0’s technologies have been used in order to develop applications. However, up to now these two areas have been kept separate: the applications that use Web Services do not use the Web 2.0 approach, and vice versa.

Service-Finder created an public Web portal\(^2\) capable of:

- Employing automated methods to gather WSDLs and all related resources such as wikis, blogs or any webpage in which useful information are given;
- Leveraging semi-automatic means to create semantic service descriptions of a Web Service out of the information gathered on the Web;
- Index the semantic aggregated information to enable fast searches and allow reasoning and matchmaking over them;
- Providing a Web 2.0 portal to support users in searching and browsing for Web Services, and facilitating community feedbacks to improve semantic annotations;
- Giving recommendations to users by tracking their behaviour.

---

\(^1\)http://www.service-finder.eu

\(^2\)The Service-Finder Portal is accessible at http://demo.service-finder.eu
3 Distributed Discovery

3.1 Scenario

The NeP4B vision depicts an Internet-based structured marketplace scenario where companies can access the huge amount of information already present in vertical portals, service and corporate databases and use it for dynamic, value-adding collaboration purposes. In this scenario, different Discovery engines allow a user to obtain ranked lists of services that are able to fulfill a user's desire (Goals). In this decentralised scenario there can be multiple discovery engines, each one understanding just a subset of the possible user's Goals, a particular category of services also exist, namely the mediation services. Those services are able, given a Goal, to produce another Goal that is more or less equivalent in its content, but expressed with a different structure. The aim of these services is to enable Goals interoperability just like a conversion filter transforms a document from its native format into a different format, while keeping the meaning of the content of the document itself.

In this scenario the concept of a Goal class represents an interface used to express a desire to a discovery engine, and as there exist different actors providing services, different actors can provide different Goal classes. Purchase groups or heavy buyers (large enterprises) are a first clear example of an actors category which can define a “standard interface” thus forcing service providers to write compatible service descriptions and discovery engine to accept the related Goal instances. At the opposite side of the producer-consumer relationship, Service providers, alone or grouped in a consortium, can also define a Goal class as a mean to be discovered, the description of the Goal class fully representing all the features offered by the service provider. A third category of Goal class provider is represented by service brokers, which can transparently aggregate homogeneous services (and thus providing a description of the common features inside the Goal class) and provide value-added services. Once a certain number of Goal classes linked to the same domain is deployed, new service providers can provide mediator services to enable goal instances to be processed and consumed by existing discovery engines.

Given the described distributed scenario, the problem of allowing service discovery turns into finding which discovery engines are “reachable” (that is, which discovery engine can understand the goal instance expressed by the user) either directly or via a series of mediation services that transform a goal instance into an equivalent one. Proceeding step-by-step doesn’t beforehand guarantee any success of the overall discovery process, and as each transformation can be costly both in time and in money, it is important to reconstruct the connections graph to navigate it in order to choose the right path of mediators and avoid dead ends. Because of the various parameters affecting the scenario, we cannot calculate an optimal route from the user’s goal to the discovery engine by just considering the elements of the connection graph, as:

- each mediator can introduce an information loss when processing the input goal instance;
- each mediator has a different cost both in terms of service fee and of processing time;
- when more than one discovery engine can be reached, there is no hint about the quality of the services pool used by the single discovery engine before actually performing the actual discovery.
In the following, we deal with the process of providing the user with a list of services which, invoked in the provided order, allow the user to fulfill his goals, while with distributed discovery we mean allowing service discovery in a distributed scenario with different actors responsible of different services rather than distributing the discovery process. Being a dynamic scenario, new actors will be able to appear on the “market” and existing ones can leave it. Furthermore, existing actors can improve or worsen the quality of their offered services over time. This means that no actor can rely on a static and a-priori knowledge of the scenario, resulting in the requirement of an infrastructure to advertise the services offered by the actors in order for the clients to find and use them.

Discovery engines, goal classes and ggMediators can be represented with a directed graph, as shown in figure 1. This graph shows the “connections” between these elements, i.e.:

- an arc from a goal to a discovery engine means that the discovery engine understand that goal class, i.e. it is able to discover services that satisfy a goal belonging to that class;

- arcs between goal classes represent ggMediators, meaning that the mediator is able to translate a goal belonging to one class to a goal belonging to the other class.

Using the graph, a client having a goal is able to find out which discovery engines it can use, either directly (because the discovery engine understands that goal class) or indirectly (through ggMediators). For example, a client that has a goal belonging to class $C$ in figure 1 can submit its goal to discovery engine $\alpha$, or it can use a mediation service to translate the goal to class $D$ and then submit it to discovery engine $\beta$.

Figure 1: Graph representation of discovery engines, goals, and ggMediators.
The client may actually not be interested in the intermediate goal classes, but only in the discovery engines that can be “reached” starting from a goal, and the mediation services (if any) that need to be invoked in order to reach a discovery engine. For this reason, partial “views” of the graph can be used, representing only this subset of information and related to each one of the goals. For example, figure 2 shows the views related to goals A and C in the graph shown in figure 1. In this representation, a part from the starting point\(^3\), all nodes correspond to services (discovery services and mediation services) and are thus related to peers in NeP4B network that offer them\(^4\).

Figure 2: Different views of the graph in figure 1, w.r.t. to goal A (left) and goal C (right).

The complete graph can contain cycles, because a chain of ggMediators can lead to the starting goal class. This is possible, at least in principle, also for the views. The client will anyway want to avoid going through such cycles, because each mediation step will probably imply some loss of quality. Therefore, when building a view from the graph, possible cycles can be cut in order to obtain an acyclic graph.

### 3.2 Metamodel design

The concept of a connection graph, roughly described in the previous section, can be formally described in terms of the relations between the concepts of the distributed discovery scenario. Furthermore, the formal model has to take into account the usage of user-generated feedbacks that are used to calculate the reputation of a certain actor in performing one of the tasks. The starting point for our metamodel are the WSMO metamodel (represented in fig. 3 on the right) as it describes the relations between mediators, goals and web services, and the Glue metamodel (represented in fig. 3 on the left), which adds the distinction between classes and instances of goals and web services.

\(^3\)The starting point is a goal class, and thus does not correspond to a service or a peer in NeP4B network.

\(^4\)This does not mean, however, that nodes in this graph represents peers (a peer can offer many services), and this graph does not represent NeP4B P2P network.
Figure 3: Glue metamodel (on the left) and WSMO metamodel (on the right)

Given our goal of navigating the graph to discover a series of paths from the starting goal to the reachable discovery engines, in figure 3 we can notice a first limitation of the basic metamodel, as it is not possible to navigate the graph following the arcs from the Goal to the Web Service because the Mediators have only outgoing arcs. To overcome this limitation and thus allow navigability, we add the missing relations.

A second issue that can be noticed is that the WSMO metamodel doesn’t take into account the existence of the discovery engine(s). Glue metamodel is derived from the WSMO specifications, but while the services described using the WSMO metamodel are meant to be distributed on the network, the descriptions themselves are intended to be all known by a single WSMO execution engine. The main difference w.r.t. our scenario is that our major requirement is not to have a central entity that is responsible of collecting the service descriptions, so we must enrich the metamodel with entities that comes from an “infrastructural” layer, such as the Discovery Engines, which is a component that is able to accept instances of one or more Goal Classes.

The complete metamodel resulting from the analysis of service discovery in a distributed scenario is shown in figure 4. The presence of both the Discovery Engine and the entities that can be reached from the GoalClass is partly redundant, because once the DE is introduced it masks these other entities. Even if redundant, this metamodel allows the existence both of Discovery Engines that performs crawling in order to find the services on the network (as the whole required information is represented) and of Discovery Engines where the inclusion of a service inside the service pool is ruled by an explicit agreement between the service owner and the DE owner.

In a feedback-based system a subject is allowed to give a feedback about an object only if it actually interacted with that object. This is easy to do if, like in EBay, all interactions can be monitored through a central system, which also manages the reputation system. But there is no such central authority in NeP4B network, so the initial version of the reputation system will rely just on locally stored feedbacks, as explained in [6]. Further improvements will design an algorithm for securely exchange reputation values between the peers of the scenario. The presence of locally-stored data introduces a further, private, information layer, which must integrate with the public information layer described before.

Reputation is related to discovery engines and mediators, as they are the “active” components of the scenario (goal classes can be seen as just specifications of the input or the output of the other two classes of components). Figure 5 shows the overall information that is required in order to model the distributed discovery scenario.
3.2.1 Public information representation

The public information layer, which is shared among all the participants of a discovery scenario, contains connectivity information and allows a client to retrieve the paths from a starting goal instance to the discovery engines that are able to handle its request. Figure 6 shows the concepts of the metamodel that are advertised on the public infrastructure, whereas Figure 7 shows the representation of the metamodel in RDF triples.

A client, in order to discover which discovery engines can be reached and how (i.e., through which ggMediators), needs to obtain the graph introduced in the previous paragraph (or, more precisely, the view of the graph w.r.t. to its goal class). However, as discussed in section 3.1, in our decentralised scenario we cannot rely on any centralised system to store this information. Moreover, the graph cannot be associated with a particular peer of the network, because it represents information related to many peers. A distributed, peer-to-peer, infrastructure is thus a natural choice for the storage and retrieval of the information represented in the graph.

The graph information defined by the complete metamodel described in the previous paragraph can be stored in a DHT (Distributed HashTable). The different actors operating in the scenario are responsible of providing information about the components that they main-
Each of these actors will use the following key-value pairs to advertise the presence of a service and the location of its description:

- *Entity.URI rdf:type* → glue:GoalClass — wsml:GGMediator — glue:DiscoveryEngine
- *Entity_URI glue:hasDescription* → Entity_URI

GG Mediator owners are responsible of publishing information regarding which Goal classes are connected with the mediator, using the following data:

- *GC_URI glue:acceptedBy GG* → GG_URI
- *GG_URI glue:ggSource* → GC_URI
- *glue:ggTarget GC_URI* → GG_URI
- *GG_URI glue:hasGoalOutput* → GC_URI

Finally, the discovery engines providers will advertise that their components are able to handle requests containing instances of certain goal classing by using the following key-value pairs:

- *glue:acceptedBy DE_URI* → GC_URI
- *glue:hasGoalClass GC_URI* → DE_URI

In order to evaluate the cost of retrieving the information linked to the metamodel design choices, we have to identify a best case and a common case. The best case is represented by a direct connection between the goal class and a discovery engine, meaning that there exists a discovery engine that is able to directly accept the goal instance provided by the user. The usual case is instead represented by the presence of at least one mediator between the given goal instance and the target discovery engine. The test queries reported below try to obtain a path between a Goal class and the Web Services that are reachable according to the published information (as shown in figure 8):
In the best case, a single query to the DHT allows the client to obtain the identifier of the discovery engine, while in the usual case the number of required queries is 3×(GG mediators in the path)+1.

We can further optimize the number of interactions that allow the user to obtain a series of paths by allowing a GG Mediator to publish information regarding an entire path rather than publishing the existence of a single node in the graph by using Med_URI glue:leadsTo → Mediator1_URI, Mediator2_URI, Mediator3_URI, DE_URI. When published, this information allows a user to directly obtain a path from the mediator to the discovery engine. As the mediator provider is responsible for the information, the user can then decide whether to trust to information and then stop continuing the search or to continue exploring the graph to obtain other mediators to reach one or more discovery engines.
3.3 Distributed discovery process

The distributed discovery process can be seen as a two-step process, which involves both
the client and the discovery engine as main actors.

In the first part of the discovery process, a client owns a goal instance and has to find
which discovery engines are able to fulfill its request, either directly or indirectly. When a
series of GG mediators lies between a goal and a discovery engine, the client must invoke the
mediation services offered by those mediators in order to obtain an equivalent goal instance
that can be submitted to the right discovery engine. This first part of the discovery process
is client-based, and relies on a public infrastructure, used by all the parties to advertise the
services they offer. Such infrastructure will contain data conforming to the model described
in 3.2.

The client starts without any prior knowledge about the scenario, and has to reconstruct
a view of the connectivity graph that allows it to find the discovery engines that are able
to accept its goal. When performing the discovery engine search, the client cannot rely on
reputation values, as they are internally stored by the single discovery engines. This way,
the client will likely start to connect to the discovery engines following the shortest paths first,
and will iteratively connect to the engines on longer paths as long as the obtained results are
not satisfying from the user's point of view.

After obtaining a path from the starting goal to the discovery engine, the client has to call a
series of GG mediators in order to obtain an equivalent goal instance that is accepted by the
target engine. If we allow each mediator to be invoked using an arbitrary function signature,
we obtain a semantic lifting and lowering problem, as original data has to be transformed
in the format accepted by the mediation and results data have to be transformed back in
the format accepted by the client. This is a non trivial problem that cannot be resolved in
a completely automatic way. In the following, we will assume that both the client and the
discovery engine know how to invoke the GG mediators and how to interpret the results of
the invocation. Moreover, all the GG mediators will expose their service using the same
function signature.

The second part of the distributed discovery process starts once the client submits a
goal instance to a discovery engine. The discovery engine can decide to forward the initial
goal instance to other discovery engines, acting as a client itself. This behaviour can be
triggered in two different ways: when the client is not satisfied with the results of the initial
discovery process and asks the discovery engine for more matching services, and when the
discovery engine has no results to show. In order to avoid cycles in the distributed discovery
process, we limit the number of recursions of forwarding the initial goal request to one. This
means that the initial discovery engine can forward the goal to other discovery engines,
which in turns cannot forward the goal to other peers. When acting as a client in searching
other engines, the discovery engine will also have to retrieve the connection graph and use
the reputation values collected by means of previous users’ interactions to choose the best
reachable discovery engines. Differently from the normal client, the discovery engine has
some sort of prior knowledge of the scenario, as it can cache the results of the queries used
when reconstructing the connectivity graph. Moreover, the discovery engine internally stores
the reputation values and the feedbacks about the other reachable mediators and DEs. This
allows using criteria other than the simple path length. The algorithm that uses the reputation
values in order to calculate the best path is explained in [6].
4 Conclusions
A  Distributed Glue Ontology

Classes

\( \text{DE\_Reputation} \sqsubseteq \text{Reputation} \)
\( \text{DiscoveryEngine} \sqsubseteq \text{Resource} \)
\( \text{GG\_Reputation} \sqsubseteq \text{Reputation} \)
\( \text{GlueResource} \sqsubseteq \text{Resource} \)
\( \text{Goal} \sqsubseteq \text{GlueResource} \)
\( \text{GoalClass} \sqsubseteq \text{Goal} \)
\( \text{GoalInstance} \sqsubseteq \text{Goal} \)
\( \text{Ontology} \sqsubseteq \text{GlueResource} \)
\( \text{Reputation} \sqsubseteq \text{Resource} \)
\( \text{StaticMatch} \sqsubseteq \text{Resource} \)
\( \text{WcGcMediator} \sqsubseteq \text{wgMediator} \)
\( \text{WebService} \sqsubseteq \text{GlueResource} \)
\( \text{WebServiceClass} \sqsubseteq \text{WebService} \)
\( \text{WebServiceInstance} \sqsubseteq \text{WebService} \)
\( \text{wgMediator} \sqsubseteq \text{GlueResource} \)

Object properties

\( \text{hasGoalClass} \equiv \text{accepted\_by}^- \)
\( \exists \text{accepted\_by} \sqsubseteq \text{GoalClass} \)
\( T \sqsubseteq \forall \text{accepted\_by.\_DiscoveryEngine} \)
\( \exists \text{dependsOn} \sqsubseteq \text{GlueResource} \)
\( T \sqsubseteq \forall \text{dependsOn.\_GlueResource} \)
\( \exists \text{goal\_memberOf} \sqsubseteq \text{GoalInstance} \)
\( T \sqsubseteq \forall \text{goal\_memberOf.\_GoalClass} \)
\( \exists \text{goal\_subClassOf} \sqsubseteq \text{GoalClass} \)
\( T \sqsubseteq \forall \text{goal\_subClassOf.\_GoalClass} \)
\( \text{hasGoalClass} \equiv \text{accepted\_by}^- \)
\( \exists \text{hasGoalClass} \sqsubseteq \text{DiscoveryEngine} \)
\( T \sqsubseteq \forall \text{hasGoalClass.\_GoalClass} \)
\( T \sqsubseteq \forall \text{has\_potential\_match\_mediator.\_WcGcMediator} \)
\( \exists \text{imports} \sqsubseteq \text{GlueResource} \)
\( T \sqsubseteq \forall \text{imports.\_Ontology} \)
\( T \sqsubseteq \forall \text{match\_group.\_GlueResource} \)
\( \exists \text{mediator} \sqsubseteq \text{StaticMatch} \)
\( T \sqsubseteq \forall \text{mediator.\_wgMediator} \)
\( \exists \text{origin} \sqsubseteq \text{StaticMatch} \)
\( T \sqsubseteq \forall \text{origin.\_GoalInstance} \)
\( \exists \text{path\_element} \sqsubseteq \text{StaticMatch} \)
\( T \sqsubseteq \forall \text{path\_element.\_GlueResource} \)
\( \exists \text{rep\_object} \sqsubseteq \text{Reputation} \)
\( T \sqsubseteq \forall \text{rep\_object.\_DiscoveryEngine} \)
∃ rep_source ⊑ Reputation
T ⊑ ∀ rep_source.GlueResource
∃ rep_target ⊑ Reputation
T ⊑ ∀ rep_target.GlueResource
∃ storedIn ⊑ GlueResource
T ⊑ ∀ storedIn.Resource
∃ wg_source ⊑ WcGcMediator
T ⊑ ∀ wg_source.GoalClass
∃ wg_target ⊑ WcGcMediator
T ⊑ ∀ wg_target.WebServiceClass
∃ ws_memberOf ⊑ WebServiceInstance
T ⊑ ∀ ws_memberOf.WebServiceClass
∃ ws_subClassOf ⊑ WebServiceClass
T ⊑ ∀ ws_subClassOf.WebServiceClass

Data properties

∃ hard_query ⊑ StaticMatch
T ⊑ ∀ hard_query
∃ hasDescription ⊑ GlueResource
∃ hasDescription ⊑ DiscoveryEngine
T ⊑ ∀ hasDescription.string
∃ hasHardQuery ⊑ WcGcMediator
T ⊑ ∀ hasHardQuery
∃ hasSoftQuery ⊑ WcGcMediator
T ⊑ ∀ hasSoftQuery
∃ rep_value ⊑ Reputation
T ⊑ ∀ rep_value.int
∃ soft_query ⊑ StaticMatch
T ⊑ ∀ soft_query
References


