Declarative Policies for Web Service Selection

Massimo Marchi, Alessandra Mileo
DSI-DICO, Università di Milano, Italy
{marchi@dsi, mileo@dico}.unimi.it

Alessandro Provetti
Dip. di Fisica, Università di Messina, Italy
ale@unime.it

Abstract

We describe a modified Grid architecture that allows the specification and enforcement connection policies to Grid on Web Services. This is accomplished by interposing a policy enforcement engine between a calling application and the relative client stubs. Only service requests need be analyzed and filtered. Connection policies are conveniently expressed in the declarative policy specification language PPDL which allows expressing simple preferences and integrity constraints. PPDL policies are evaluated by translating them into a Logic Program with Ordered Disjunctions and calling the psmodels interpreter. This process is completely transparent to both client applications and Web/Grid services. There are clear advantages in having the connection logic expressed declaratively and externally to applications.

1 Introduction

In this article we describe how the standard Grid service/Web service architecture can be improved by interposing a policy enforcement engine between a calling application and the relative client stubs. Our policies can specify, among others, preferences and prohibitions in the routing of remote invocations to Web services (WS). Therefore, with our solution WS preference and invocations is not hard-coded into client applications but (declaratively) defined and enforced independently to clients, so that they can be (de)activated and modified at runtime. Hence our architecture so remains transparent to both client application and the invoked Web service.

The Policy Description Language with Preferences (PPDL)[1] is a recent development of the PDL language. (P)PDL policies are high-level, i.e., abstract from the device they are applied to. Even though PPDL has rather simple constructs and is prima-facie a less expressive language than those traditionally considered in knowledge representation, it allows us to capture the essence of routing control and to keep the so-called business logic outside the code; so it can be inspected and changed any time transparently from the applications, which won’t need re-writing. Hence, by adopting PPDL we keep policies in a declarative, almost documentation form; yet, policies are automatically translated and evaluated into Logic Program with Ordered Disjunctions and calling the psmodels interpreter. Psmodels is a version of smodels, an inferential engine for Answer Set Programming (ASP).

2 The Grid Service Architecture

Web services is a distributed technology that permits, in a worldwide network environment, the building of effective client/server applications. A set of well-defined protocols, built mainly on XML and Uniform Resource Identifier (URI), is used to describe, address and communicate with services, thus achieving a high level of interoperability between different client and server implementations.

A typical WS may be viewed as a service dispenser. A generic client application, can consult a directory of available services, called Universal Description, Discovery and Integration (UDDI) Registry, invoke one of such services and retrieve the results in a fashion similar to that of usual Web sessions.

In our experimental architecture we adopt Grid Services, an extension of Web Services available in the Globus Toolkit 3 Framework[8]. Grid services provide some graceful features not always supported by general Web Services, such as dynamic instance service creation, lifetime management and notification.

In order to explain our architecture, we need to describe in detail how Web/Grid service invocation works. Typically, communication between client and server is made through...
a coupled object: client stub and server stub, that hides all low-level communication activity. Starting from a service description (in the WSDL language), it is possible to automatically generate the code for client and server stubs. The policy module, which will be described in detail below, is inserted in the architecture by modifying the class that implements the client stub interface (see Figure 1 below).

In order to use a WS, client applications go through two steps.

In the first step, the application creates a handler for managing communication with the chosen service. Such handlers are in fact instances of the Java class that implements the so-called client stubs. For each service hosted by a given server an instance must be created that represents the service toward client applications. In fact, each service is addressed by an URI which reads something like http://server.domain/Service/Math. Such URI says that server server.domain is hosting service Service/Math.

In the second step, the client application actually calls the service by invoking the corresponding instance and passing all the call arguments. The called instance performs all the needed operations, it communicates with the service, retrieves the results and returns them to the client.

Our policy module enters into play at step 1, where it monitors the communication between policy module and client application. Thus, the policy module will catch all service calls, enforce the policy by invoking the external pmodels solver and finally will route the call according to the policy results.

![Figure 1. PPDL module location on WS client.](image)

3 Declarative route policies: PDL and PPDL

PDL has been recently proposed by Chomicki, Lobo, Naqvi [6, 7] with the Policy Description Language (PDL). In that context, a (device-independent) policy is a description of how events received over a network (e.g., queries to data, connection requests etc.) are served by some given network terminal or data server. PDL allows managers to specify policies independently from the details of the particular device executing it. We refer the reader to works by Chomicki et al. [7] for a complete introduction and motivation for PDL in network management. In order to introduce our PPDL, we will now give an overview of PDL.

PDL can be described as an evolution of the Event-Condition-Action schema of active databases. A PDL program is defined as a set of policy rules \( P_i \) plus a consistency maintenance mechanism called monitor:

\[
P_i : e_1, \ldots, e_m \text{ causes } a \text{ if } C\quad M_i : \text{never } a_1, \ldots, a_n \text{ if } C'
\]

where \( C, C' \) are Boolean conditions, \( e_1, \ldots, e_m \) are events, which can be seen as input requests\(^3\) and \( a \) is an action, which is understood to be a configuration command that can be executed by the network manager and actions \( a_1 \ldots a_n \) of \( M \) cannot execute simultaneously.

If the application of policies yields a set of actions that violate one of the rules in the monitor then the PDL interpreter will cancel some of them, but notice that selection of a particular action(s) to drop cannot be specified in PDL\(^4\).

Our PPDL language\(^2\) addresses this issue and allows the specification of preferences on how to enforce constraints. This is done by reconstructing Brewka’s ordered disjunction connective [3] into PDL, thus obtaining an output based on degrees of satisfaction of a preference rule.

The resulting language is called PPDL: PDL with Preferences and it enables users to specify preferences in policy enforcement (cancellation of actions) To describe a preference relation on actions to be blocked when a constraint violation occurs, we introduced constraints with the following syntax:

\[
\text{never } a_1 \times \ldots \times a_n \text{ if } C.
\]

which means that actions \( a_1, \ldots, a_n \) cannot be executed together and—in case of constraint violation—\( a_1 \) should be blocked. If this is not possible (i.e. \( a_1 \) must be performed), block \( a_2 \), else block \( a_3 \) etc.; if all of \( a_1, \ldots, a_{n-1} \) must be executed, then block \( a_n \).

PPDL policies receive a declarative semantics and are computed by translating them into Brewka’s Logic Programs with Ordered Disjunctions (LPDs).

By lack of space, the declarative and procedural semantics are given in the companion article [1]. Notice that, both in PDL and PPDL translations to ASP, minimality of answer

\(^3\)Also, non-occurrence of an event may be in the premise of the rule. To allow for that, for each event \( e \) a dual event \( \bar{e} \) is introduced, representing the fact that \( e \) has not been recorded. This is called negation as failure (NAF) and it is different than asserting \( \neg e \), which means that an event corresponding to the negation of \( e \) has been recorded. In this paper we will not consider negated events.

\(^4\)However, Chomicki et al. describe two general solutions, called action-cancellation and event-cancellation, respectively.
sets corresponds to minimal of the set of actions that get canceled in case of violations.

4 Web/Grid Service Selection in PPDL

This section gives a complete example of a Web Service scenario based on our architecture. We used the Hotels On Line Search Service, which was found through the IBM UDDI registry, and simulated an analogous Service called AllHotels, which inherits some methods from HOL and is organized as a gateway to hotel information from the Web. The AllHotels service consists, essentially, of hotel search and reservation functions. Clearly, the results obtained can vary according to the business, but we expect the interaction schema to remain fundamentally the same.

In our scenario, several details regarding location and interface of the service are known and are made available for policy enforcement through tables. We also want to define routing policies based on some parameters expressing reliability and performance of the service.

We make this information available through a local Web Service called InfoCollector. The InfoCollector does nothing but lookup some selected Web service to collect information about availability of hotel rooms in certain cities. Its operation is automated thanks to the Lixto suite application by Lixto Gmbh. Thanks to the set up of InfoCollector, our service lookup table look as follows: where Rel and Perf represent, respectively, a reliability and a performance indicator. These parameters are updated at the end of each local call to a hotel reservation WS. Currently, the InfoCollector is invoked by the application before the routing policy enforcement to get more recent data. In some sense this system learns meaning as more applications connects to a certain hotel reservation service, performance and reliability can be evaluated more and more accurately.

In the context of the lookup table above, we have designed the PPDL policy described next. The goal here is to maximize reliability of retrieved data, while taking into account the sharp differences in performance among available servers.

\[ P_1: \begin{align*}
\text{req}(L,M) \quad & \text{causes} \quad \text{send}(URL1,M,Rel1,Perf1) \times \\
& \quad \text{send}(URL2,M,Rel2,Perf2) \quad \text{if} \quad \text{Rel1} \leq \text{Rel2}, \text{Perf1} \leq \text{Perf2}, \text{not today(holiday)}.
\end{align*} \]

\[ P_2: \begin{align*}
\text{send}(URL1,M,Rel1,Perf1) \times \\
& \quad \text{send}(URL2,M,Rel2,Perf2) \quad \text{if} \quad \text{Rel1} = \text{Perf1} \leq \text{Rel2}, \text{Perf2}, \text{not today(holiday)}.
\end{align*} \]

\[ P_3: \begin{align*}
\text{send}(URL1,M,Rel1,Perf1) \times \\
& \quad \text{send}(URL2,M,Rel2,Perf2) \quad \text{if} \quad \text{today(holiday)}.
\end{align*} \]

Rule \( P_1 \) simply says that an invocation of a method \( M \) can be sent to address \( URL \) if, according to the lookup table, \( M \) is provided by the Web Server at \( URL \), and this is the general policy to decide whether a request should be sent to a server.

Monitor rules \( P_1 \) and \( P_2 \) tell how routing should be preferably performed according to the value of Rel and Perf indexes. In particular, \( P_1 \) says that method \( M \) should be preferentially invoked on the Web Server with greater value of both indexes, and \( P_2 \) says that method \( M \) should be preferentially invoked on the Web Server where the sum of the two indexes is higher.

Last monitor rule \( P_3 \) specify that on holidays, AccessPoint2 should be preferred over all other servers providing the same service.

4.1 The software layers

In general, a PPDL policy specification can be animated by the following step-by-step procedure outlined in Figure 2 below.

First, the PPDL policy is translated into an Answer Set Program[2]. Second, the resulting LPOD program is fed to a solver that computes one of its answer sets. Such answer set will contain, among other uninteresting atoms, a set of instances of the execute(a1, ..., an) predicate that describe the actions that should be executed next. An extractor takes the ASP solver output and extracts the a1, ..., an actions to be executed, then it examines the action name and calls the appropriate routines, that will invoke the chosen client stub.

\[ \begin{align*}
M_1: \text{never} \quad & \text{send}(URL1,M,Rel1,Perf1) \times \\
& \quad \text{send}(URL2,M,Rel2,Perf2) \quad \text{if} \quad \text{Rel1} \leq \text{Rel2}, \text{Perf1} \leq \text{Perf2}, \text{not today(holiday)}.
\end{align*} \]

\[ \begin{align*}
M_2: \text{never} \quad & \text{send}(URL1,M,Rel1,Perf1) \times \\
& \quad \text{send}(URL2,M,Rel2,Perf2) \quad \text{if} \quad \text{Rel1} + \text{Perf1} \leq \text{Rel2} + \text{Perf2}, \text{not today(holiday)}.
\end{align*} \]

\[ \begin{align*}
M_3: \text{never} \quad & \text{send}(URL1,M,Rel1,Perf1) \times \\
& \quad \text{send}(AccessPoint2,M,Rel2,Perf2) \quad \text{if} \quad \text{today(holiday)}.
\end{align*} \]

\[ \begin{align*}
P_1: \text{req}(L,M) \quad & \text{causes} \quad \text{send}(URL1,M,Rel1,Perf1, today(holiday)) \times \\
& \quad \text{send}(URL2,M,Rel2,Perf2, today(holiday)) \quad \text{if} \quad \text{Rel1} \leq \text{Rel2}, \text{Perf1} \leq \text{Perf2}, \text{not today(holiday)}.
\end{align*} \]

\[ \begin{align*}
P_2: \text{req}(L,M) \quad & \text{causes} \quad \text{send}(URL1,M,Rel1,Perf1, today(holiday)) \times \\
& \quad \text{send}(URL2,M,Rel2,Perf2, today(holiday)) \quad \text{if} \quad \text{Rel1} + \text{Perf1} \leq \text{Rel2} + \text{Perf2}, \text{not today(holiday)}.
\end{align*} \]

\[ \begin{align*}
M_1: \text{never} \quad & \text{send}(URL1,M,Rel1,Perf1) \times \\
& \quad \text{send}(URL2,M,Rel2,Perf2, today(holiday)) \quad \text{if} \quad \text{Rel1} \leq \text{Rel2}, \text{Perf1} \leq \text{Perf2}, \text{not today(holiday)}.
\end{align*} \]

\[ \begin{align*}
M_2: \text{never} \quad & \text{send}(URL1,M,Rel1,Perf1) \times \\
& \quad \text{send}(URL2,M,Rel2,Perf2, today(holiday)) \quad \text{if} \quad \text{Rel1} + \text{Perf1} \leq \text{Rel2} + \text{Perf2}, \text{not today(holiday)}.
\end{align*} \]

\[ \begin{align*}
M_3: \text{never} \quad & \text{send}(URL1,M,Rel1,Perf1, today(holiday)) \times \\
& \quad \text{send}(AccessPoint2,M,Rel2,Perf2, today(holiday)) \quad \text{if} \quad \text{today(holiday)}.
\end{align*} \]
4.2 The complete architecture

As we have mentioned above, the architecture in Figure 3 is obtained by modifying the GT3 class, ServiceLocator, that creates an object instance for each client-grid connection. Each service is identified by an URL and provides a set of operations, or methods. In our architecture, the Trapper routine described in Figure 3 catches all outgoing calls made by the client application.

![Figure 3. The new architecture.](image)

The Trapper method stores the URLs of available services in a lookup table and uses them to translate from the Java object that represents the stub to the relative URL and vice versa. When the client application perform a method invocation, Trapper extracts from that call i) the URL of the service, ii) the requested interface and method and iii) the arguments that should be passed to the remote method.

Next, Trapper translates all real names to symbolic values used in the PPDL policy. In our solution this step is performed by means of an external environment specification file, `environment.h`.

Now, the PPDL policy specification, `policy.ppd1`, needs to be translated to an LPPOD program in order to apply it. This step is performed by Translator.

Next, Decorator assembles call data, the policy and the environment specifications together into a complete LPOD program. The inferential engine, the `lpars+psmodels` box (see Fig.3), interprets this program and extracts one (or more) answer sets. The answer set contains action atoms that describe executable, non-blocked actions.

Extractor extracts from the answer set a subset of non-redundant actions by non-deterministically choosing an action from tie-breaks. Finally Trapper translates back the solution into a real client-stub call.

5 Conclusions and Open problems

In this article we have described a new, experimental-yet-functional Grid service architecture that, in our opinion, has several advantages, thanks to having the connection logic expressed outside the application and in declarative format. Our solution is transparent to the standard Grid service architecture and can be described as bringing to Web services the same advantages that triggers and constraints bring to relational databases and their client applications. Our implementation is still in its infancy and several of the software layers may be improved with more sophisticated implementation and optimization. Our experiments also suggest that an important issue that needs to be investigated further is related to method calls routing when multiple solution are obtained.

References


