Specification and Execution of Declarative Policies for Grid Service Selection*

Massimo Marchi[1], Alessandra Mileo[2], and Alessandro Provetti[3]

[1] DSI - Università degli studi di Milano
Milan, I-20135 Italy
marchi@dsi.unimi.it.

[2] DICo - Università degli studi di Milano
Milan, I-20135 Italy
mileo@dico.unimi.it.

[3] Dip. di Fisica - Università degli studi di Messina
Messina, I-98166 Italy
ale@unime.it

Abstract. We describe a modified Grid architecture that allows to specify and enforce connection policies with preferences and integrity constraints. This is done by interposing a policy enforcement engine between a calling application and the relative client stubs. Such policies are conveniently expressed in the declarative policy specification language PPDL. In particular, PPDL allows expressing preferences on how to enforce constraints on action execution. PPDL policies are evaluated by translating them into a Logic Program with Ordered Disjunctions and calling the psmodels interpreter. We describe an experimental architecture that enforces connection policies by catching and filtering only service requests. The process is completely transparent to both client applications and Grid services. There are clear advantages in having the connection logic expressed declaratively and externally to applications.

Keywords: Grid Services. Customization. User preferences. Declarative Policies. Answer Set Programming.

1 Introduction

In this article we how the standard Grid 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).

* This work has been supported by i) MIUR COFIN project *Formal Languages and Automata: Methods, Models and Applications* and ii) the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-37004 WASP project. The authors participate in M^2AG: Milan-Messina actions group: http://mag.dsi.unimi.it/

Therefore, with our solution WS preference and invocation is not hard-coded into client applications but (declaratively) defined and enforced outside the clients, so that they can be (de)activated and modified at runtime. Our architecture is implemented so as remain transparent to both client application[1] and the invoked Web service.

The *Policy Description Language with Preferences* (PPDL), which is formally described below, is a recent development of the PDL language. PDL is a result of applied research at Bell Labs [9,12] on automated tools for network administration. 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 (Logic Programming, Description logics and others), it allows capturing the essence of routing control and allows us 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 rewriting. Finally, by adopting PPDL we keep policies in a declarative, almost documentation form yet with automated and relatively efficient enforcement. Performance is the result of adopting Answer Set Programming solvers [16] to execute the policy evaluation.

2 The Grid Service Architecture

Web services is a distributed technology that permits, in a worldwide network environment, to build 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 large interoperability between different client and server implementations.

A typical WS may be viewed as a service dispenser (please see Figure 1 below). 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.
While we cannot dwell on the details of the WS architecture, let us just notice that each WS can be addressed by an URI. For our purposes, we will consider URIs that are simple URLs. The *Web Service Description Language* (WSDL) is used to describe how to communicate with WS, e.g., the format that service requests should have, that of service responses, the possible error messages and so on.

3 Our Experimental Architecture

In our experimental architecture we adopt *Grid Services,* an extension of Web Services available in the *Globus ToolKit 3 Framework* [11]. Grid services provide

[1] So far, however, we have considered only Java applications.

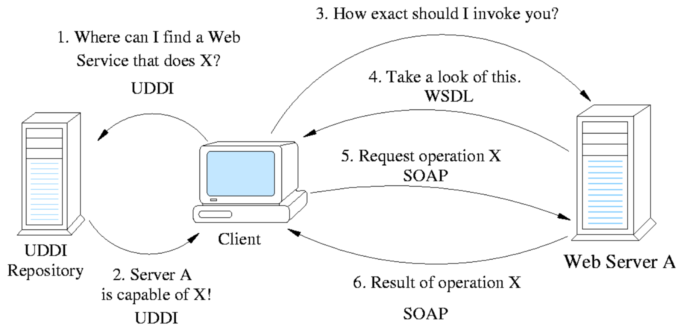

Fig. 1. The standard Grid architecture.

some graceful features not always supported by general Web Services, such as dynamic instance service creation, lifetime management and notification.

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 WSDL service description, 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 2 below).

In order to use a WS, a client application must go through two steps, which are now described in some detail.

In step 1, 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 step 2, 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 operation to communicate with the service, retrieves the results and return them to the client.

Our policy module enters into play at step 1, where it catches all creations and keeps a look-up table containing all available services. At that stage, URIs are translated into corresponding instance handlers during communication between policy module and client application. Moreover, the policy module catches all service calls, it enforces the policy by invoking the external psmodels solver and finally routes the call according to the policy results, thus achieving goals such as reliability, load balancing etc.

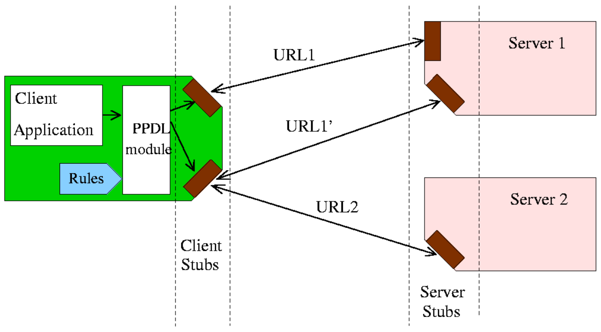

Fig. 2. PDL module location on WS client.

4 Introduction to PPDL

Developing and analyzing policies made of several provisions can be quite a complex tasks, in particular if one wants to ensure some sort of *policy-wide consistency*, that is for example, that no two different provisions of the policy result in conflicting actions to be executed. In order to guarantee *policy consistency*, declarative approaches to policy specification appear to be a promising viable solution.

One such approach to specify policies in network context has been recently proposed by Chomicki, Lobo, Naqvi [7,8,9] 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. This feature is a great advantage when dealing with heterogeneous networks, which is most often the case nowadays. We refer the reader to works by Chomicki et al. [9] for a complete introduction and motivation for PDL in network management. In order to introduce our PPDL, we will now give an overview of its parent language PDL.

4.1 Overview of PDL

PDL can be described as an evolution of the *Event-Condition-Action* (ECA) schema of active databases. A PDL program is defined as a set of policy rules P_i and a consistency maintenance mechanism called *monitor*, composed by a set of rules M_i of the form

$$\begin{aligned} P_i:&\ e_1, \ldots e_m \ \textbf{causes}\ a\ \textbf{if}\ C \\ M_i:&\ \textbf{never}\ a_1,\ \ldots,\ a_n\ \textbf{if}\ C' \end{aligned}$$

where C, C' are Boolean conditions, $e_1, \dots e_m$ are events, which can be seen as input requests[2] and a is an action, which is understood to be a configuration command that can be executed by the network manager and actions $a_1 \dots a_n$ of M_i are forbidden from executing *simultaneously.*

PDL assumes that events and actions are syntactically disjoint and that rules are evaluated and applied in parallel. One may notice the lack of any explicit reference to time. In fact, PDL rules are interpreted in a discrete-time framework as follows. If at a given time t the condition is evaluated true and all the events have been received from the network, then at time $t+1$ action a is executed. As a result, we can see PDL policies as describing a *transducer.*

If the application of policies yields a set of actions that violates 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 by the language as is. However, Chomicki et al. describe two general solutions, called action-cancellation and event-cancellation, respectively.

The declarative semantics of PDL policies is given by means of translation into Answer Set Programming (ASP), namely in the expressive framework of *disjunctive logic programs.* Also, thanks to that translation one can actually *run* a PDL policy against a set of input events by feeding the translated version to an ASP solver (see [16]) and inspecting the computed answer sets to find the actions dictated by the policy. In our language PPDL we retain and extend Chomicki et al. translation to get the same appealing features of a concise declarative semantics and interpretation via ASP solvers.

4.2 PPDL: Policy Description Language with Preferences

It should be observed that in PDL it is possible to specify which actions cannot execute together but it is not possible to specify what should be done in order to avoid violations. In other words, the administrator cannot specify which actions should preferentially be dropped, and what actions should be preferentially executed even in case of a violation. Indeed, in PDL the choice of which action to drop is non-deterministic.

We believe that flexible policy languages, by which one can specify whether and how to enforce constraints, are required. We have moved closer to achieve such result by defining an extension of PDL [2,1] that allows users to express preferences. This is done by reconstructing Brewka's ordered disjunction connective [4] 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

[2] 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 $\overline{e}$ 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.

describe a preference relation on action to be blocked when a constraint violation occur, we introduced constraints with the following syntax:

$$\textbf{never } a_1 \ \times \ldots \times \ a_n \textbf{ if } C. \tag{1}$$

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 (LPODs). Ordered disjunctions are a relatively recent development in reasoning about preferences with Logic Programming and are subject of current work by [4,5], [6], [15] and others. One important aspect of Brewka's work is that preferred answer sets need not be minimal. This is a sharp departure from traditional ASP and in [1] we have investigated how adding preferences to PDL implies a trade-off between user-preferences and minimality of the solutions. Notice that, both in PDL and PPDL translations to ASP, minimality of answer sets corresponds to *minimality of the set of actions that get canceled in case of violations.*

4.3 Overview of LPOD

As mentioned earlier, Logic Programs with Ordered Disjunctions have been introduced by [4] in his work on combining Qualitative Choice Logic and Answer Set Programming. A new connective, called *ordered disjunction* and denoted with "×," is introduced. An LPOD consists of rules of the form

$$C_1 \ \times \ \ldots \ \times \ C_n \ :- \ A_1, \ \ldots, \ A_m, \ not \ B_1 \ \ldots, \ not \ B_k. \tag{2}$$

where the C_i, A_j and B_l are ground literals. The intuitive reading [4] of the rule (2) is:

when $A_1, \ldots, A_m$ are observed and $B_1, \ldots, B_k$ are not observed, then
if possible deduce C_1, but if C_1 is not possible, then deduce C_2,
...
if all of $C_1, \ldots, C_{n-1}$ are not possible, then deduce C_n instead.

The × connective is allowed to appear in the head of rules only; it is used to define a preference relation so as to *select* some of the answer sets of a program by using ranking of literals in the head of rules, on the basis of a given strategy or context. The answer sets of a LPODs program are defined by Brewka as sets of atoms that maximize a preference relation induced by the "×-rules" of the program. Before describing the semantics, let us consider a simple example.

Example 1. (from [5]) Consider the Linux configuration domain, and the process of configuring a workstation. There might be several kinds of different preference criteria. First, there are usually several available versions for any given software package. In most cases we want to install the latest version, but sometimes, we have to use an older one. We can handle these preferences by defining a new atom for each different version and then demanding that at least one version should be selected if the component is installed. Second, a component may have also different variants (e.g. a normal version and a developer version). A common user would prefer to have the normal variant while a programmer would prefer the developer version. Suppose there are three versions of emacs available. This preferences can be modeled using rules expressed by LPODs syntax:

1. $emacs - 21.1 \ \times \ emacs - 20.7.2 \ \times \ emacs - 19.34 \ :- \ installed - emacs.$
2. $dev - library \ \times \ usr - library \ :- \ need - library, \ developer.$
3. $usr - library \ \times \ dev - library \ :- \ need - library, \ not \ developer.$

4.4 The Declarative Semantics of LPODs

The semantics of LPOD programs is given in terms of a model preference criterion over answer sets. [4] shows how Inoue and Sakama's split program technique can be used to generate programs whose answer sets characterize the LPOD preference models. In short, a LPOD program is rewritten into several split programs, where only one head appears in the conclusion. Split programs are created by iterating the substitution of each LPOD rule (2) with a rule of the form:

$$C_i :- A_1, \ldots, A_m, not\ B_1, \ldots, not\ B_k, not\ C_1, \ldots, not\ C_{i-1} \tag{3}$$

Consequently, Brewka defines answer sets for the LPOD program Π as the answer sets of any of the split programs generated from Π.

There is one very important difference between Gelfond and Lifschitz's answer sets and LPOD semantics: in the latter (set-theoretic) minimality of models is not always wanted, and therefore not guaranteed. This can be better explained by the following example.

Example 2. Consider these two facts:

1. $A \ \times \ B \ \times \ C.$
2. $B \ \times \ D.$

To *best satisfy* both ordered disjunctions, we would expect $\{A, \ B\}$ to be the single preferred answer set of this LPOD, even if this is not even an answer set of the corresponding disjunctive logic program (where "$\times$" is replaced by "$\vee$"). Indeed, according to the semantics of [10] $\{B\}$ satisfies both disjunctions and is *minimal.*

To sum it up, since minimality would preclude preferred answer sets to be considered dealing with preferences implies adopting non-minimal semantics.

LPOD programs are be interpreted by a special version of the solver *Smodels,* called *Psmodels,* which is presented in [5]. In a nutshell, LPOD programs are translated (by the *lparse* parser) into equivalent (but longer) ASP programs and then sent to *Psmodels.*
Now, we can go back to policies and describe how PPDL is mapped into LPOD.

4.5 Translating PPDL Policies into Answer Set Programming

Starting from a set of preference cancellation rules (1) we define LPOD *ordered blocking rules* as follows:

$$block(a_1) \;\times\; \ldots \;\times\; block(a_n) \;:\!-\; exec(a_1), \ldots,\; exec(a_n),\; C. \tag{4}$$

Since the PPDL-to-LPOD translation described above does not provide a mechanism for avoiding action block, the resulting program is deterministic: we would obtain answer sets where the leftmost action of each rules of the form (4) that fires is always dropped.

As a result, in [1] we argued that a simplified version of rule (4) can be formulated as follows:

$$block(a_1) \;:\!-\; exec(a_1), \ldots,\; exec(a_n),\; C. \tag{5}$$

This translation realizes a simple, deterministic preference criteria in canceling action violating a constraint, according to the given strategy: for each constraint, we put as leftmost action an action that shall always be dropped.

However, ordered disjunctions are appealing precisely when some actions *may not be blocked.* This can be specified by using a new rule called *anti-blocking rule* which is added to the language.

Anti-blocking rules. This rules allow users to describe actions that *cannot be filtered* under certain conditions. The syntax of *anti-blocking rule* is as follows:

$$\textbf{keep}\; a \;\textbf{if}\; C. \tag{6}$$

where a is an action that cannot be dropped when the boolean condition C is satisfied. This rule is applied whenever a constraint of the form (1) is violated, and a is one of the conflicting actions. In ASP, anti-blocking rules are mapped in a constraint formulated as follows:

$$:\!-\; block(a),\; C. \tag{7}$$

which is intended as *action a cannot be blocked if condition C holds.* Notice that if we want to control the execution of action a, postulating that under condition C action a is executed *regardless*, then we should write, in PPDL:

$$\begin{array}{l} \emptyset\;\textbf{causes}\; a \;\textbf{if}\; C. \\ \textbf{keep}\; a \;\textbf{if}\; C. \end{array}$$

that will be translated in LPOD as follows:

$$\begin{array}{l} exec(a) \;:\!-\; C. \\ :\!-\; block(a),\; C. \end{array}$$

Unlike in traditional PDL, where actions are strictly the consequence of events, by the **causes** described above we allows *self-triggered* or *internal* actions. We should mention that, even without internal events, a PPDL policy with monitor, blocking and anti-blocking rules, may be inconsistent. Consider the following example referred to allocation of resource $res1$ among two different users $usr1$ and $usr2$.

Example 3. Take policy P_{res}:

$$\begin{aligned} P_{res} = \{\ & \mathit{need\_usr1\_res1}\ \mathbf{causes}\ \mathit{assign\_usr1\_res1}. \\ & \mathit{need\_usr2\_res1}\ \mathbf{causes}\ \mathit{assign\_usr2\_res1}.\ \} \end{aligned}$$

and a preference monitor M_{res} saying that resource $res1$ cannot be assigned both to $usr1$ and $usr2$. In particular, it is preferable to drop the request of $usr2$, supposed he/she is less important than $usr1$. Moreover, if one of the users has an urgent need, than his/her request should not be dropped.

$$\begin{aligned} M_{res} = \{\ & \mathbf{never}\ \mathit{assign\_usr1\_res1}\ \times\ \mathit{assign\_usr2\_res1}. \\ & \mathbf{keep}\ \mathit{assign\_usr1\_res1}\ \mathbf{if}\ \mathit{urgent\_usr1}. \\ & \mathbf{keep}\ \mathit{assign\_usr2\_res1}\ \mathbf{if}\ \mathit{urgent\_usr2}.\ \} \end{aligned}$$

where $\mathit{urgent\_usr1}$ and $\mathit{urgent\_usr2}$ stand for Boolean conditions. Both P_{res} and M_{res} are translated the following LPOD, named π_{res}:

$$\begin{aligned} & \mathit{exec}(\mathit{assign\_usr1\_res1})\ :\!-\ \mathit{occ}(\mathit{need\_usr1\_res1}). \\ & \mathit{exec}(\mathit{assign\_usr2\_res1})\ :\!-\ \mathit{occ}(\mathit{need\_usr2\_res1}). \\ & \mathit{block}(\mathit{assign\_usr2\_res2}) \times\ \mathit{block}(\mathit{assign\_usr2\_res1})\ :\!-\ \mathit{exec}(\mathit{assign\_usr1\_res1}), \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{exec}(\mathit{assign\_usr2\_res1}). \\ & :\!-\ \mathit{block}(\mathit{assign\_usr1\_res2}), \mathit{urgent\_usr1}. \\ & :\!-\ \mathit{block}(\mathit{assign\_usr2\_res2}), \mathit{urgemt\_usr2}. \end{aligned}$$

Now, suppose that events $\mathit{need\_usr1\_res1}$ and $\mathit{need\_usr2\_res1}$ has occurred. It is clear that if both the clients have urgent requests, π_{res} is inconsistent so the policy+monitor application yields an error and the requests should be transmitted again.

The simple example above shows that if we want to use prioritized semantics in extended PPDL, we have to be careful in introducing anti-blocking rules, in order to ensure that at least one action can be blocked whenever a constraint is violated.

5 The PPDL Specification of Grid Service Selection

This section gives a complete example of a Grid service scenario based on our architecture. In our Department there are three servers that implement grid services. Here we consider a grid service called *math* available on all three servers. The *math* service consists, essentially, of arithmetic functions. Clearly, we get the exact same service from all services, even though the implementation can vary to i) optimize performance on certain inputs and ii) adapt to the particular platform where the service is run.

In our scenario, several details regarding location and interface of the service are known and are made available for policy enforcement through tables. The following lookup table[3] is an example of a PPDL specification of the services we have access to:

Table 1. A service lookup table

URL	service
mag.usr.dsi.unimi.it/math	mag.math
zulu.usr.dsi.unimi.it/math	mag.math
grid001.usr.dsi.unimi.it/math	mag.math
grid002.usr.dsi.unimi.it/math	mag.math

In the context of the lookup table above, we have designed the PPDL policy described next: The goal is is to maximize computation per time unit, while keeping into account the sharp differences in performance among our servers.

P_1: req(I,M,L1,L2) **causes** send(Url,I,M,L1,L2)
if table(Url,I), M≠m-plus.
P_2: req(I,M,L1,L2) **causes** send(Url,I,m-plus,L1,L2)
if table(Url,I), M=m-plus, L1≤10, L2≤10.
P_3: request(I,M,L1,L2) **causes** send(Url,I,m-fastplus,L1,L2)
if table(Url,I), M=m-plus, $L1 > 10$.
P_4: request(I,M,L1,L2) **causes** send(Url,I,m-fastplus,L1,L2)
if table(Url,I), M=m-plus, $L2 > 10$.

M_1: **never** send(grid001,I,M,L1,L2) × send(grid002,I,M,L1,L2)
if M=m-plus.
M_2: **never** send(zulu,I,M,L1,L2) **if** M=m-fastplus.
M_3: **never** send(grid002,I,M,L1,L2) × send(grid001,I,M,L1,L2)
if M=m-fastplus, $L1 > 20$.
M_4: **never** send(grid002,I,M,L1,L2) × send(grid001,I,M,L1,L2)
if M=m-fastplus, $L2 > 20$.
M_5: **never** send(grid001,I,M,L1,L2) × send(grid002,I,M,L1,L2)
if M=m-fastplus, L1≤20, L2≤20.

Rule P_1 simply says that an invocation of a method other than *m-plus* method, is sent to Web service where, according to the lookup table, such service is available. Policy rules P_2 to P_4 practically define the method *m-plus* and say that for such method, if at least one parameter is greater than 10, then a faster method called *m-fastplus* should be (transparently) invoked.

Monitor rules M_1 to M_5 tell how routing should be preferably performed according to the size of the parameters and the computational power of the

[3] There are several ways for creating the lookup table. For instance, it may be obtained by consulting the UDDI directory on the Web.

server. In particular, M_1 says that the *m-plus* method invocation should be blocked on server *grid001* with higher preference with respect to *grid002*, as the first one is faster than the second one, and we want it not to be busy with simple computation. M_2 prevents the client from sending a *m-fastplus* method invocation to the slow *zulu* server.

Rules M_3 to M_5 tell the client how to route *m-fastplus* method invocation among the faster servers, according to the size of the parameter: if both the parameters are less or equal to 20, a method invocation is send to *grid002*. Otherwise, it is routed to *grid001*, which is supposed to perform better with high values of the parameters.

5.1 The Software Layers

In general, a PPDL policy specification can be *animated* by the following step-by-step procedure outlined in Figure 3) below.

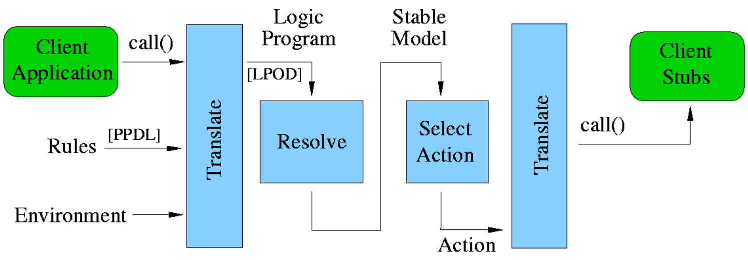

Fig. 3. The software layers of our architecture

First, the PPDL policy is translated into an Answer Set Program, following the encoding defined in [2]. Second, the resulting ASP program is fed to a solver that computes one of its answer sets. These answer set will contain, among other uninteresting atoms, a set of instances of the *execute(a_i)* predicate that describe the actions that should be executed next. An extractor takes the ASP solver output and extracts the $a_1, \ldots a_n$ actions to be executed, then it examines the action name and calls the appropriate routines, that will invoke the chosen client stub.

5.2 The Complete Architecture

As we have mentioned above, the architecture in Figure 4 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 operation, or *methods*. In our architecture, the *Trapper* routine described in Figure 4 catches all outgoing calls made by the client application.

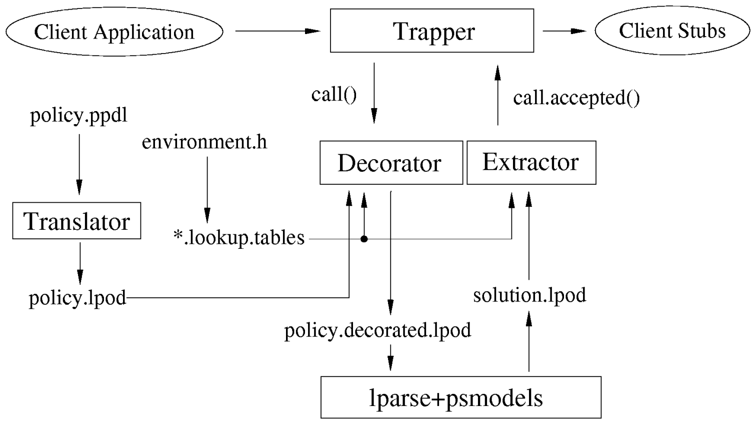

Fig. 4. The standard Grid architecture.

The *Trapper* method stores the URLs of available services in a lookup table and use them to pass from the Java object that represent the stub to the relative URL and vice versa. When the client application perform a method invocation, *Trapper* extract 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.ppdl,* needs to be translated to an LPOD program in order to apply it. This step is performed by *Translator.*

Next, *Decorator* assembles all call data, the policy and environment specifications together into a complete LPOD program. An external module, the *lparse+psmodels* box seen in Figure 4, interprets this program and extracts one (or more) answer set. The answer set contains the a set *accept* atoms describing executable, non-blocked actions.

Extractor extracts from the solution 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.

6 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 Grid services the same advantages that triggers and constraints bring to relational databases and their client applications.

Our implementation of the architecture is still in its infancy and several of the software layers may be improved with more sophisticated implementation and optimization. However, automatic mapping from PPDL to LPOD has already been described in details, and it is rather straightforward. We are working on a simple user-friendly interface to help users in writing and compiling their PPDL policies. Meanwhile, our experiments are suggesting that one important issue that needs to be investigated further is related to method calls routing when multiple solution are obtained.

The PPDL module may return different routing possibilities[4] for each method call, all this solution being equally preferred according to the PPDL semantics. Only one server invocation should be done for each call. In our prototype the one invocation to execute is chosen non-deterministically among all the possible ones. Clearly, when several parallel method calls are requested by the client application, it is important to have a method to distribute the calls among all available servers according to some criteria, e.g., performance improvement, overload avoidance or reliability. This may be done by using some planning techniques or by defining a further (internal) level of policy specification.

Acknowledgments. Thanks to A. Bertoni, E. Bertino, G. Gonzales and M. Ornaghi for useful discussions on this subject.

References

1. Bertino, E., Mileo, A. and Provetti, A., 2003. User Preferences VS Minimality in PPDL. In Buccafurri F. (editor), Proc. of *AGP03,* APPIA-GULP-PRODE. Available from *http://mag.dsi.unimi.it/PPDL/*
2. Bertino E., Mileo A. and Provetti A., 2003. *Policy Monitoring with User-Preferences in PDL.* Proc. of *NRAC 2003* IJCAI03 Workshop on Reasoning about Actions and Change.
   Available from *http://mag.dsi.unimi.it/PPDL/*
3. Bertino, E., Mileo, A. and Provetti, A., 2003. PDL with Maximum Consistency Monitors. Proc. of *Int'l Symp. on Methodologies for Intelligent Systems (ISMIS03).* Springer LNCS. Available from *http://mag.dsi.unimi.it/PPDL/*
4. Brewka, G., 2002. *Logic Programming with Ordered Disjunction.* Proc. of AAAI-02. Extended version presented at NMR-02.
5. Brewka, G., Niemelä I and Syrjänen T., 2002. *Implementing Ordered Disjunction Using Answer Set Solvers for Normal Programs.* Proc. of JELIA'02. Springer Verlag LNAI.
6. Buccafurri F., Leone L. and Rullo P., 1998. Disjunctive Ordered Logic: Semantics and Expressiveness. Proc. of KR'98. MIT Press, pp. 418-431.

[4] Notice that each possibility is represented by an *accept(...)* atom.

7. Chomicki J., Lobo J. and Naqvi S., 2000. *A logic programming approach to conflict resolution in policy management.* Proc. of KR2000, 7th Int'l Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufmann, pp 121–132.
8. J. Chomicki, J. Lobo, 2001. *Monitors for History-Based Policies.* Proc. of Int'l Workshop on Policies for Distributed Systems and Networks. Springer, LNCS 1995, pp. 57–72.
9. Chomicki J., Lobo J. and Naqvi S., 2003. *Conflict Resolution using Logic Programming.* IEEE Transactions on Knowledge and Data Engineering 15:2.
10. Gelfond, M. and Lifschitz, V., 1991. Classical negation in logic programs and disjunctive databases. New Generation Computing: 365–387.
11. Web location related to Web Services technologies.
    Globus Toolkit Framework: *http://www.globus.org/*
    World Wide Web Consortium: *http://www.w3c.org/*
12. Lobo J., Bhatia R. and Naqvi S., 1999. A Policy Description Language, in *AAAI/IAAI, 1999*, pp. 291–298.
13. Marchi M., Mileo A. and Provetti A., 2004. *Specification and execution of policies for Grid Service Selection.* Posters at ICWS2004 conference. IEEE press. Available from *http://mag.dsi.unimi.it/PPDL/*
14. Marchi M., Mileo A. and Provetti A., 2004. *Specification and execution of policies for Grid Service Selection.* Poster at Int'l Conference on Logic Programming (ICLP04) Spinger LNCS. Available from *http://mag.dsi.unimi.it/PPDL/*
15. Schaub T., and Wang K., 2001. *A comparative study of logic programs with preference.* Proc. of Int'l. Joint Conference on AI, IJCAI-01.
16. Web location of the most known ASP solvers.
    Aspps: *http://cs.engr.uky.edu/ai/aspps/*
    CMODELS: *http://www.cs.utexas.edu/users/tag/cmodels.html*
    DLV: *http://www.dbai.tuwien.ac.at/proj/dlv/*
    NoMoRe: *http://www.cs.uni-potsdam.de/~linke/nomore/*
    Smodels: *http://www.tcs.hut.fi/Software/smodels/*
    PSmodels: *http://www.tcs.hut.fi/Software/smodels/priority/*