Grid Service Selection with PPDL*

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

[1] DSI-DICO, Univ. degli Studi di Milano. Milan, I-20135 Italy
marchi@dsi.unimi.it, mileo@dico.unimi.it
[2] Dip. di Fisica, Univ. degli Studi di Messina. Messina, I-98166 Italy
ale@unime.it

Abstract. The application of the novel PPDL (Policy Description Language with Preferences) specification language to Grid-Service Selection is presented. We describe an architecture based on interposing the policy enforcement engine between the calling application and the client stubs. This way, our solution is fully declararive and remains transparent to the client application.

This poster article reports on our experimental application of PPDL to the standard Grid Service architecture. PPDL, which is formally described below, is a declarative language that extends the Policy Description Language PDL [2] by permitting the specification of preferences on how to enforce integrity constraints. The declarative semantics of PPDL policies is given by translation into Brewka's Logic Programs with Ordered Disjunctions (LPOD) [3]. Translated LPOD programs will then be fed to the *Psmodels* solver [4], which has shown a reasonable efficency.

Grid Services Architecture. Web Services is a distributed technology using a set of well-defined protocol derived from XML and URI that achieves a large interoperability between different client/server implementations. In our experiments, we use GT3 Grid Services, an extension of Web Services (WS) coded in Java. Typically, each communication between client and server is made through a coupled object, called *stub*. When a client application needs a service, it queries the UDDI Registry to retrieve a list of Web Services that fit its request. The query response passed to the client application can be i) *the first* in the list of results, ii) *randomly* chosen or iii) *user-chosen* through some *static* meta-information stored in the UDDI.

In contrast, our approach allows user-defined policies that are evaluated *dynamically* and *client-side*. The policy module shown in Figure 1 catches all starting invocations from client, stores all available servers returned by the UDDI, applies the connection policy by translating it into a LPOD program, invokes *Psmodels* and, according to the result, *routes* the call.

* Thanks to E. Bertino, S. Costantini, J. Lobo and M. Ornaghi. Work supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-37004 WASP project.

B. Demoen and V. Lifschitz (Eds.): ICLP 2004, LNCS 3132, pp. 464–466, 2004.

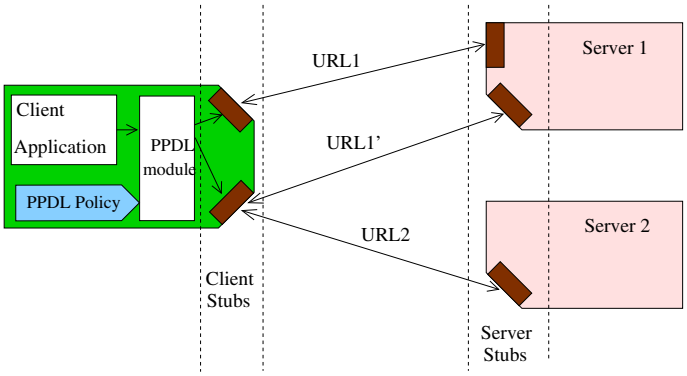

Fig. 1. PPDL module on WS client.

Policies Specification and Enforcement. Some of these authors have developed PPDL [5] as an extension of PDL [2]. Even though PPDL has rather simple constructs and is prima-facie less expressive than the traditional Knowledge Representation languages, it allows capturing the essence of routing control while keeping the so-called *business logic* outside the client applications; (P)PDL policies can be changed at any time transparently from the applications, which do not need rewriting.

A PPDL policy is defined as a set of ECA-style rules P_i and a set of consistency-maintenance rules M_i:

$P_i:\ e_1, \ldots e_m$ **causes** a **if** C
$M_i:$ **never** $a_1\ \times\ \ldots\ \times\ a_n$ **if** C'

where C, C' are Boolean conditions, $e_1, \ldots e_m$ are *events* (requests), a is an *action* to be executed and $a_1 \ldots a_n$ are actions that, intuitively, cannot execute *simultaneously*. Notice that PPDL rules are evaluated and applied in parallel and in a discrete-time framework. If applying the policy yields a set of actions that violates one of the M_i (for monitor) rules, then the PPDL interpreter will *cancel* some of the actions. The decision on which action to drop has been addressed in our work [5, 6] and it corresponds to applying the ordered disjunction operator ($\times$) of LPODs [3] into the M_i rules. Both the declarative and operational semantics of PPDL policies are given by translation into LPODs.

To sum it up, Fig. 2 shows how PPDL policies are employed in our architecture.

An Example Specification. Suppose we want to i) send calls to the *add(x,y)* function to a server providing it (to be found on a look-up table) but ii) *prefer* sending calls to host *zulu* over host *mag* whenever x is greater than 100. Assuming that we have a look-up table mapping each WS *interface* into a PPDL constant, e.g. *iMath*, the PPDL rules are as follows:

request(iMath.M(L)) **causes** send(URL, iMath.M(L)) **if** table (URL, iMath).
never send(*mag*, iMath.M(L)) $\times$ send(*zulu*, iMath.M(L)) **if** M=add, L[0]> 100.

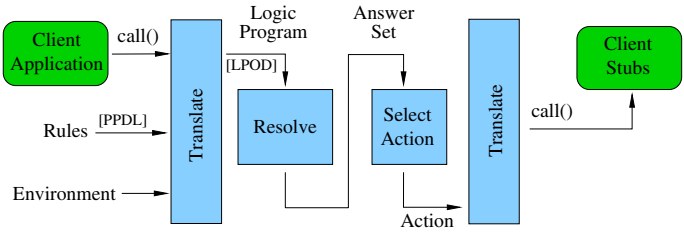

Fig. 2. Our software architecture

References

1. Marchi M., Mileo A. and Provetti A., 2004. *Specification and execution of policies for Grid Service Selection.* Poster at ICWS2004 conference. IEEE press.
2. Chomicki J., Lobo J. and Naqvi S., 2003. *Conflict Resolution using Logic Programming.* IEEE Transactions on Knowledge and Data Engineering 15:2.
3. Brewka, G., 2002. *Logic Programming with Ordered Disjunction.* Proc. of AAAI-02.
4. PSMODELS: *http://www.tcs.hut.fi/Software/smodels/priority/*
5. Bertino E., Mileo A. and Provetti A., 2003. *Policy Monitoring with User-Preferences in PDL.* Proc. of NRAC 2003 Workshop. Available from *http://mag.dsi.unimi.it/*
6. Bertino, E., Mileo, A. and Provetti, A., 2003. User Preferences VS Minimality in PPDL. Proc. of *AGP03,* Available from *http://mag.dsi.unimi.it/*