DIOS++: A Framework for Rule-Based Autonomic Management of Distributed Scientific Applications*

Hua Liu and Manish Parashar

The Applied Software Systems Laboratory
Dept of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ 08854, USA
{marialiu,parashar}@caip.rutgers.edu

Abstract. This paper presents the design, prototype implementation and experimental evaluation of DIOS++, an infrastructure for enabling rule based autonomic adaptation and control of distributed scientific applications. DIOS++ provides: (1) abstractions for enhancing existing application objects with sensors and actuators for runtime interrogation and control, (2) a control network that connects and manages the distributed sensors and actuators, and enables external discovery, interrogation, monitoring and manipulation of these objects at runtime, and (3) a distributed rule engine that enables the runtime definition, deployment and execution of rules for autonomic application management. The framework is currently being used to enable autonomic monitoring and control of a wide range of scientific applications including oil reservoir, compressible turbulence and numerical relativity simulations.

1 Introduction

High-performance parallel/distributed simulations are playing an increasingly important role in science and engineering and are rapidly becoming critical research modalities. These simulations and the phenomena that they model are large, inherently complex and highly dynamic. Furthermore, the computational Grid, which is emerging as the dominant paradigm for distributed computing, is similarly heterogeneous and dynamic, globally aggregating large numbers of independent computing and communication resources, data stores and sensor networks. As a result, applications must be capable of dynamically managing, adapting and optimizing their behaviors to match the dynamics of the physics they are modeling and the state of their execution environment, so that they continue to meet their requirements and constraints. Autonomic computing [3] draws on the mechanisms used by biological systems to deal with such complexity, dynamics and uncertainty to develop applications that are self-defining, self-healing, self-configuring, self-optimizing, self-protecting, contextually aware and open. This paper presents the design, prototype implementation and experimental evaluation of DIOS++, an infrastructure for supporting the autonomic adaptation and control of distributed scientific applications. While a number of existing systems support interactive monitoring

* Support for this work was provided by the NSF via grants numbers ACI 9984357(CAREERS), EIA 0103674 (NGS) and EIA-0120934 (ITR), DOE ASCI/ASAP (Caltech) via grant numbers PC295251 and 1052856.

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 66–73, 2003.

and steering capabilities, few existing systems (e.g. CATALINA [4] and Autopilot [2]) support automated management and control. DIOS++ enables rules and polices to be dynamically composed and securely injected into the application at runtime so as to enable it to autonomically adapt and optimize its behavior. Rules specify conditions to be monitored and operations that should be executed when certain conditions are detected. Rather than continuously monitoring the simulation, experts can define and deploy appropriate rules that are automatically evaluated at runtime. These capabilities require support for automated and controlled runtime monitoring, interaction, management and adaptation of application objects.

DIOS++ provides: (1) abstractions for enhancing existing application objects with sensors and actuators for runtime interrogation and control, (2) a control network that connects and manages the distributed sensors and actuators, and enables external discovery, interrogation, monitoring and manipulation of these objects at runtime, and (3) a distributed rule engine mechanism that enables the runtime definition, deployment and execution of rules for autonomic application management. Access to an object's sensors and actuators is governed by access control polices. Rules can be dynamically composed using sensors and actuators exported by application objects. These rules are automatically decomposed, deployed onto the appropriate processors using the control network, and evaluated by the distributed rule-engine.

DIOS++ builds on the DIOS [1], a distributed object substrate for interactively monitoring and steering parallel scientific simulations, and is part of the Discover [1] computational collaboratory. Discover enables geographically distributed clients to collaboratively access, monitor and control Grid applications using pervasive portals. It is currently being used to enable interactive monitoring, steering and control of a wide range of scientific applications, including oil reservoir, compressible turbulence and numerical relativity simulations.

2 DIOS++ Architecture

DIOS++ is composed of 3 key components: (1) autonomic objects that extend computational objects with sensors (to monitor the state of an object), actuators (to modify the state of an object), access policies (to control accesses to sensors and actuators) and rule agents (to enable rule-based autonomic self-management), (2) mechanisms for dynamically and securely composing, deploying, modifying and deleting rules, and (3) a hierarchical control network that is dynamically configured to enable runtime accesses to and management of the autonomic objects and their sensors, actuators, access policies and rules.

2.1 Autonomic Objects

In addition to its functional interfaces, an autonomic object exports three aspects: (1) a *control aspect*, which defines sensors and actuators to allow the object's state to be externally monitored and controlled, (2) an *access aspect*, which controls accesses to these sensors/actuators and describes users' access privileges based on their capabilities,

[1] http://www.discoverportal.org

and (3) a *rule aspect*, which contains rules that can autonomically monitor, adapt and control the object. These aspects are described in the following subsections. A sample object that generates a list of random integers (*RandomList*) is used as a running example. The number of integers and their range are allowed to be set at run time.

Control aspect. The control aspect specifies the sensors and actuators exported by an object. Sensors provide interfaces for viewing the current state of an object, while actuators provide interfaces for processing commands to modify the object's state. For example, a *RandomList* object would provide sensors to query the current length of the list or the maximum value in the list, and an actuator for deleting the current list. Note that sensors and actuators must be co-located in memory with the computational objects and must have access to their internal state, since computational objects may be distributed across multiple processors and can be dynamically created, deleted, migrated and redistributed.

DIOS++ provides programming abstraction to enable application developers to define and deploy sensors and actuators. This is achieved by deriving computational objects from virtual base object provided by DIOS++. The derived objects can then selectively overload the base object methods to specify their sensors and actuators interfaces. This process requires minimal modification to the original computational objects and has been successfully used by DIOS++ to support interactive steering.

Access aspect. The access aspect addresses security and application integrity. It controls the accesses to an object's sensors and actuators and restricts them to authorized users. The role-based access control model is used, where users are mapped to roles and each role is granted specific access privileges defined by access policies.

The DIOS++ access aspect defines three roles: owner, member, and guest. Each user is assigned a role based on her/his credentials. Owner can modify access policies, define access privileges for members and guests, and enable or disable external accesses. Polices define which roles can access a sensor or actuator and in what way. Owners can also enable or disable a sensor or actuator. Access polices can be defined statically during object creation using the DIOS++ API, or can be injected dynamically by the owner at runtime via secure Discover portals.

Rule aspect. The DIOS++ framework uses user-defined rules to enable autonomic management and control of applications. The rule aspect contains rules that define actions that will be executed when specified conditions are satisfied. The conditions and actions are defined in terms of the control aspect (e.g. sensors and actuators). A rule consists of 3 parts: (1) Condition part, defined by the keyword "IF" and composed of conditions which are conjoined by logical relationships (AND, OR, NOT etc.), (2) Then action part, defined by the keyword "THEN" and composed of operations that are executed when the corresponding condition is true, (3) Else action part, defined by the keyword "ELSE" and composed of operations that are executed when condition is not fulfilled.

For example, consider the *RandomList* object with 2 sensors: (1) *getLength()* to get the current length of the list and (2) *getMaxValue()* to get the maximal value in the list, and 1 actuator *append(length, max, min)* that creates a list of size *length* with random integers between *max* and *min*, and appends it to the current list.

```
IF RandomList.getLength()<10 AND RandomList.getMaxValue()<=50
THEN RandomList.append(10, 100, 0)
```

Note that rules are separated from the application logic. This provides flexibility and allows users to dynamically create, delete and change rules without modifying the application. Users use these rules to monitor and control their applications at run time. Rules can be added, deleted, changed on the fly without stopping and restarting the application. Rules are handled by rule agents and the rule engine, which are part of the control network (described in the following subsection) and are responsible for storing, evaluating and executing rules.

2.2 Control Network

The DIOS++ control network (see Figure 1) is a hierarchical structure consisting of a rule engine and gateway, autonomic objects (composed of computational objects and rule agents), and computational nodes. It is automatically configured at run time using the underlying messaging environment (e.g. MPI) and the available processors.

The lowest level of the control network hierarchy consists of computational nodes. Each node maintains a local autonomic object registry containing references to all autonomic objects currently active and registered. At the next level of hierarchy, the Gateway represents a management proxy for the entire application. It combines the registries exported by the nodes and manages a registry of the interaction interfaces (sensors and actuators) for all the objects in the application. It also maintains a list of access policies related to each exported interface and coordinates the dynamic injection of rules. The Gateway interacts with external interaction servers or brokers such as those provided by Discover.

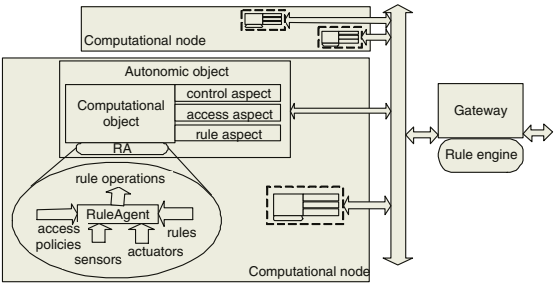

Fig. 1. DIOS++ architecture.

Co-located with Gateway, the rule engine accepts and maintains the rules for the application. It decomposes these rules and distributes them to corresponding rule agents, coordinates the execution of rule agents, keeps track of rule execution and reports them to the user. Each rule agent executes its rules using an execution script, and reports the rule execution status to the rule engine. The execution script is also defined by the rule engine and specifies the rule execution sequence and the rule agent's behavior. The specification and execution of scripts and the coordination between the rule engine and rule agents are illustrated in the following subsections.

In DIOS++, although rule execution is coordinated by the rule engine, rules are evaluated and executed in parallel. This central-control and distributed-execution mechanism has the following advantages: (1) Rule execution which can be compute-intensive is done in parallel by rule agents. This reduces the rule execution time as compared to

a sequential rule execution. (2) Rule agents are created dynamically and delegated to autonomic objects. This solution requires less system resources than static rule agents as the agents are created only when need. It also leads to more efficient rule execution. (3) Rule agent's behavior is based on script, which allows it to adapt to the execution environment and the rules that it needs to execute. Rule agent scripts can be calibrated at runtime by the rule engine to make rule agents more adaptive.

The operation of the control network is explained below using an example. Consider a simple application that generates a list of integers and then sorts them. This application contains two objects: (1) *RandomList* that provides a list of random integers, and (2) *SortSelector* that provides several sorting algorithms (bubble sort, quick sort, etc.) to sort integers.

Initialization. During initialization, the application uses the DIOS++ APIs to create and register its objects, and to export its aspects, interfaces and access policies to the local computational node. Each node exports these specifications for all its objects to the Gateway. The Gateway then updates its registry. Since the rule engine is co-located with Gateway, it has access to the Gateway's registry. The Gateway interacts with the external access environment (Discover servers in our prototype) and coordinates accesses to the application's sensor/actuators, policies and rules.

Interaction and rule operation. At runtime the Gateway may receive incoming interaction or rule requests from users. The Gateway first checks the user's privileges based on her/his role, and refuses any invalid access. It then transfers valid interaction requests to corresponding objects and transfers valid rule requests to the rule engine. Finally, the responses to the user's requests or from rules executions are combined, collated and forwarded to the user. Once again we use the example to describe this process.

Rule definition: Suppose *RandomList* exports 2 sensors: *getLength()*, and *getList()*. *SortSelector* exports no sensors, and 2 actuators: *sequentialSort()* and *quickSort()*. The owner can access all these interfaces. Members can only access *getLength()* and *getList()* in *RandomList*, and *sequentialSort()* in *SortSelector*. Guests can only access *getLength()* in *RandomList*.

Using DIOS++, users can view, add, delete, modify and temporarily disable rules at runtime using a graphical rule interface integrated with the Discover portal. An application's sensors, actuators and rules are exported to the Discover server and can be securely accessed by authorized users (based on access control polices) via the portal. Authorized users can compose rules using the sensors and actuators. Note that rules may be defined for individual objects or for the entire application and span multiple objects. Users specify a priority for each rule, which is then used to resolve rule conflicts.

Rule deployment: Consider the following rules defined by a user. Let Rule1 have a higher priority than Rule2:

```
Rule1: IF RandomList.getLength()<100 THEN RandomList.getList()
       ELSE RandomList.getLength()
Rule2: IF RandomList.getLength()<50 THEN SortSelector.sequentialSort()
       ELSE SortSelector.quickSort()
```

Rule1 is an object rule, which means that the rule only applies to one object. Rule2 is an application rule, which means that the rule can affect several objects. When the Gateway receives the two rules, it will first check the user's privileges. If the rules are defined by member users, Rule2 will be rejected by Gateway since member users do not have the privilege to access *quickSort()* interface in *SortSelector*.

The Gateway transfers valid rules to the rule engine. The rule engine dynamically creates rule agents for the objects if they do not already exist. It then composes a script for each agent, which defines the rule agent's lifetime and rule execution sequence based on rule priorities. For example, the script for the rule agent for *RandomList* may specify that this agent will terminate itself when it has no rules, and that Rule1 is executed first. Note that this script is extensible.

In the case of an object rule, the rule engine just injects the object rule into its corresponding rule agent. In the case of an application rule, the rule engine will first decompose the rule into triggers and then inject triggers into corresponding agents. So Rule2 is decomposed into 3 triggers: (1) *SortSelector.sequentialSort()*, (2) *SortSelector.quickSort()*, and (3) *RandomList.getLength()* < 50. These triggers are injected into corresponding agents as shown in Figure 2.

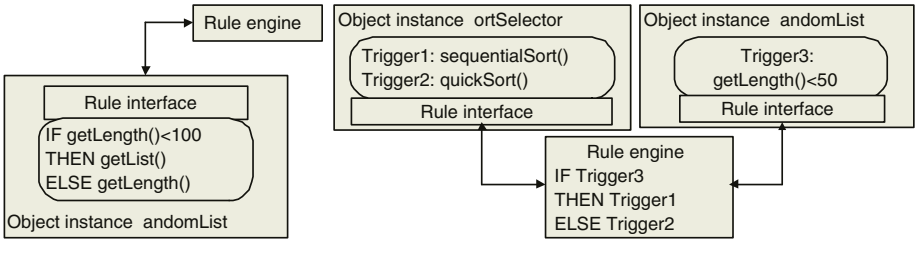

Fig. 2. Left: rule engine deploys an object rule to its corresponding rule agent. Right: rule engine decomposes an application rule into triggers and injects triggers to corresponding rule agents.

Rule execution and conflicts: During interaction periods in the application's execution, the rule engine fires all the rule agents at the same time and rule agents work in parallel. Rule agents execute object rules and return the results to the rule engine. The rule engine then reports them to the user through the Gateway. Rule agents also execute triggers, which are part of application rules, and report corresponding results to the rule engine. The rule engine collects the required trigger results, evaluates condition combinations, and then issues corresponding actions if the conditions are fulfilled. Application rule results are also reported to the user through the Gateway.

While typical rule execution is straightforward (actions are issued when their required conditions are fulfilled), the application dynamics and user interactions make things unpredictable. As a result, rule conflicts must be detected at runtime. In DIOS++, rule conflicts are detected at runtime and are handled by simply disabling the conflicting rules with lower priorities. This is done by locking the required sensors/actuators. For example, suppose that a user defines two rules for the object instance *RandomList*. Rule1 requires setting the minimal integer value to 5 when the list length is less than 100 and

larger than 50, and Rule2 requires the minimal value to be 6 when the list length is larger than 30 and less than 70. Rule1 has higher priority than Rule2. The two rules conflict with each other, for example, when the list length is 60.

```
Rule1: IF RandomList.getLength()>50 AND RandomList.getLength()<100
       THEN RandomList.setMinInt() = 5
Rule2: IF RandomList.getLength()>30 AND RandomList.getLength()<70
       THEN RandomList.setMinInt() = 6
```

The script asks rule agent to fire Rule1 first. After Rule1 is executed, the interface of *setMinInt()* is locked during the period when the length is less than 100 and larger than 50. When Rule2 is issued, it cannot be executed as the required interface is locked. The interface will be unlocked when the length value is not within the range of 50 to 100.

The rule agent at an object will continue existing or destroy itself according to the lifetime specified in its script. Rule engine can dynamically modify the scripts to change the behavior of rule agents.

3 Experimental Evaluation

DIOS++ has been implemented as a C++ library. This section summarizes an experimental evaluation of the DIOS library using the IPARS reservoir simulator framework on a 32 node beowulf cluster. IPARS is a Fortran-based framework for developing parallel/distributed reservoir simulators. Using DIOS++/Discover, engineers can interactively feed in parameters such as water/gas injection rates and well bottom hole pressure, and observe the water/oil ratio or the oil production rate. The evaluation consists of 3 experiments:

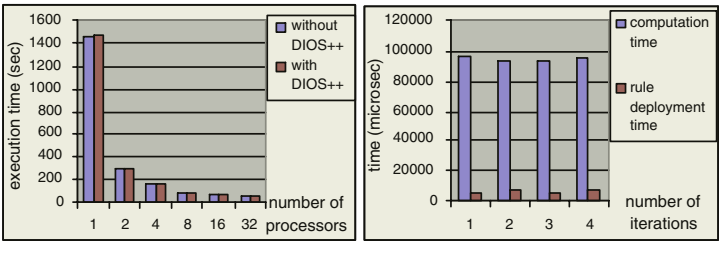

Fig. 3. Left: Runtime overheads introduced in the DIOS++ minimal rule execution mode. Right: Comparison of computation and rule deployment time.

Experiment 1 (Figure 3, left): This experiment measures the runtime overhead introduced to the application in DIOS++ minimal rule execution mode. In this experiment, the application automatically updates the Discover server and its connected clients with current state of autonomic objects and rules. Explicit interaction and rule execution are disabled during the experiment. The application's run time with and without DIOS++

are plotted in the left figure in Figure 3. It can be seen that the runtime overheads due to DIOS++ are very small and are within the error of measurements.

Experiment 2 (Figure 3, right): This experiment compares computation time and rule deployment time for successive iterations. In this experiment, we deployed two object rules and two application rules in 4 successive iterations (object rules are deployed in the first and third iterations; application rules are deployed in the second and fourth iterations). The experiment shows that object rules need less deployment time than application rules. This is true since rule engine only has to inject object rules to corresponding rule agents, while it has to decompose application rules to triggers, and inject triggers to corresponding rule agents.

Experiment 3 (Figure 4): This experiment compares computation time, object rule execution time and application rule execution time for successive application iterations. The experiment shows that an application rule requires a larger execution time than an object rule, since rule engine has to collect results from all the triggers, check whether the conditions are fulfilled and invoke corresponding actions.

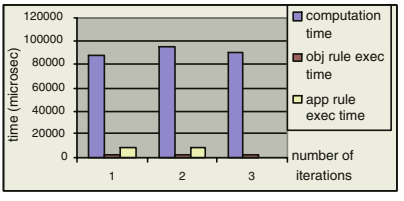

Fig. 4. Comparison of computation, object rule execution and application rule execution times.

4 Summary and Conclusions

This paper presented the design, prototype implementation and experimental evaluation of DIOS++, an infrastructure for supporting the autonomic adaptation and control of distributed scientific applications. DIOS++ enables rules and polices to be dynamically composed and securely injected into application at runtime so as to enable it to autonomically adapt and optimize its behavior. The framework is currently being used, along with Discover, to enable autonomic monitoring and control of a wide range of scientific applications, including oil reservoir, compressible turbulence and numerical relativity simulations.

References

1. R. Muralidhar and M. Parashar, “A Distributed Object Infrastructure for Interaction and Steering”, Concurrency and Computation: Practice and Experience, John Wiley and Sons, 2003 (to appear)
2. R. Ribler, J. Vetter, H. Simitci, and D. Reed, “Autopilot: Adaptive Control of Distributed Applications”, Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, IL, July 1998
3. P. Horn, “Autonomic Computing: IBM’s perspecitive on the State of Information Technology”, IBM Corp., October 2001. http://researchweb.watson.ibm.com/autonomic/
4. S. Hariri, C.S. Raghavendra, Y. Kim, M. Djunaedi, R. P. Nellipudi, A. Rajagopalan, P. Vadlamani, Y. Zhang, “CATALINA: A Smart Application Control and Management”, the Active Middleware Services Conference 2000