A Reflective Database-Oriented Framework for Autonomic Managers

Wendy Powley and Pat Martin
School of Computing, Queen's University
Kingston, ON Canada
{wendy, martin}@cs.queensu.ca

Abstract

The trend towards autonomic systems emphasizes the need for implementation frameworks for autonomic features. In this paper we describe a general framework for the development of autonomic managers, that is, components that can manage their own behaviour. Our approach uses concepts and tools from relational database management systems as well as reflective programming techniques to implement the components of the autonomic manager.

1. Introduction

Advances in software technologies and practices have enabled developers to create larger, more complex applications to meet the ever increasing user demands. The unpredictability of how the applications will behave and interact in a widespread, integrated environment poses great difficulties for system testers and managers. Management of such systems involves not only management of individual components, but management of an environment that changes dynamically and, at the same time, strives to meet system-wide objectives. Human understanding of such an environment is complex, manual management of such a system is near impossible.

The Autonomic Computing initiative, spawned by IBM in 2001, proposes a solution to software management problems in which the responsibility for software management shifts from the human administrator to the software system itself. The IBM Architectural Blueprint for Autonomic Computing provides a set of guidelines for building autonomic capability into an on-demand computing environment [4]. The blueprint discusses some of the emerging standards and identifies several building blocks for the implementation of Autonomic Computing features. In this paper we outline a general methodology for the implementation of one of the key building blocks identified by the IBM blueprint, the autonomic manager. The autonomic manager implements the management capabilities of a resource, or a set of resources, called managed elements.

The autonomic manager consists of a feedback loop consisting of 4 components; Monitor, Analyze, Plan and Execute, sometimes referred to as the MAPE loop. Sensors provide mechanisms to collect information about the current state of an element. Effectors are mechanisms that change the state or configuration of an element. Central to all of the MAPE functions is knowledge about the system such as performance data reflecting past, present and expected performance, system topology, negotiated Service Level Agreements (SLAs), and policies and/or rules governing system behaviour.

The MAPE loop is typically implemented as a feedback control loop. Thus, there is a need for mechanisms to maintain logic flow as well as communication channels that allow the various components to communicate and pass messages. Furthermore, autonomic managers rarely operate solely in isolation. Instead they cooperate with other managers to maintain overall system performance, thus requiring communication between autonomic managers. If external management capabilities are required for a component, management interfaces to the autonomic manager must be exposed. In this paper we describe a general framework for the development of autonomic managers that meet these requirements.

Data management is an important aspect of the autonomic manager. Much of the logic flow of the MAPE loop depends upon changes or updates to the data stored in the database, or knowledge store, of the autonomic manager. Our framework for the development of autonomic managers takes advantage of the rich capabilities of the database management system, not only to manage the data, but also to control the logic flow in the system and to implement pieces of the logic of the components themselves.

A *reflective system* maintains a model of self-representation and changes to the self-representation

are automatically reflected in the underlying system. Reflection enables *inspection* and *adaptation* of systems at run-time [7] thus making reflection a viable approach to implanting autonomic features in computing systems. In our approach, we consider the autonomic element's controllable features, along with their current status, to be the component's self-representation. This information is stored as part of the system knowledge. Whenever a change is made to the self-representation data, a corresponding change is made to the actual system.

The remainder of the paper is structured as follows. Related work is presented in Section 2. Section 3 presents our proposed reflective database-oriented framework for autonomic managers and illustrates our approach using an example autonomic Web services environment. In Section 4 we summarize the contributions of the paper and outline future directions.

2. Related Work

The IBM Architectural Blueprint for Autonomic Computing [4] defines a general architecture for building autonomic systems. Many approaches to building autonomic systems are based on this blueprint document and some extend the blueprint by specifying component interfaces as well as component behaviours and interactions [12].

Several proposed approaches use control theory techniques to implement the feedback loop [3][10]. Some approaches use an ontology to extend the capabilities of autonomic components [5][10]. Liu & Parashar [6] present a component-based programming framework to support the development of autonomic self-managed applications. The concept of reflection has been used for autonomic computing systems, particularly in terms of adaptive middleware [1].

Our approach builds on the IBM Architectural blueprint and proposes a novel framework for autonomic systems that exploits the powerful capabilities of a database management system for system control as well as to manage and store the vast array of knowledge required for an autonomic system. Our approach uses common tools provided by most database management systems as well as reflective programming techniques to incorporate self-awareness into the autonomic system.

3. Framework for Autonomic Managers

In our approach, relational database tables store the knowledge related to an autonomic manager and database triggers defined on the tables control the logic flow of the system. Database triggers are routines that are stored in a database and are executed or "fired" when a table is modified. Two types of triggers are used in our approach; *insert* triggers (fired upon insert of new data into a table) and *update* triggers (fired upon update of a particular attribute in a table). The trigger action may be coded within the trigger definition or the action may be coded within external code that is called by the trigger. Logic for the autonomic manager may be implemented as a stored procedure (code that is stored within the database system), a user defined function (code that can be used within an SQL query), or as an external program that can be called by a database trigger.

In the following sections we describe our framework and outline how the various components of an autonomic manager are implemented using a reflective database-oriented approach. As an example, we introduce an autonomic Web services environment and describe in detail one of the autonomic managers employed in this prototype environment, namely an autonomic manager for the buffer pool of a database management system.

Autonomic Web Services System

An architecture for an Autonomic Web Services Environment is shown in Figure 1. A Web services environment typically consists of a collection of components including HTTP servers, application servers, database servers, and Web service applications. We refer to a *site* as a collection of components and resources necessary for hosting a Web service system. A site can span multiple platforms and can be distributed across multiple physical nodes. Each component of the site is assumed to be autonomic, that is, self-aware and capable of self-configuration to maintain a specified level of performance [11].

An autonomic environment requires some control at the system level to achieve system-wide goals. The autonomic Web services environment can be viewed as a hierarchy of autonomic managers. Each component may consist of one or more autonomic element(s) and each autonomic element employs an autonomic manager that implements the self-managing features of the autonomic element. At the highest level, a *site manager*, also an autonomic manager, provides service provisioning and management of the components, if necessary, to ensure overall system performance.

Our approach, as shown in Figure 2, is illustrated in detail by describing the autonomic manager for the buffer pool area of a database management system (DBMS), a key resource for performance in a database system. The DBMS buffer pool acts as a cache for

all data requested from the database. Given the high cost of I/O accesses in a database system, it is important for the buffer pool to function as efficiently as possible, which means dynamically adapting to changing workloads to minimize physical I/O accesses. Configuring a DBMS buffer pool involves sizing the buffer pool properly and controlling the asynchronous writes to flush "dirty" pages from the buffer pool back to disk as efficiently as possible to make room for newly requested pages.

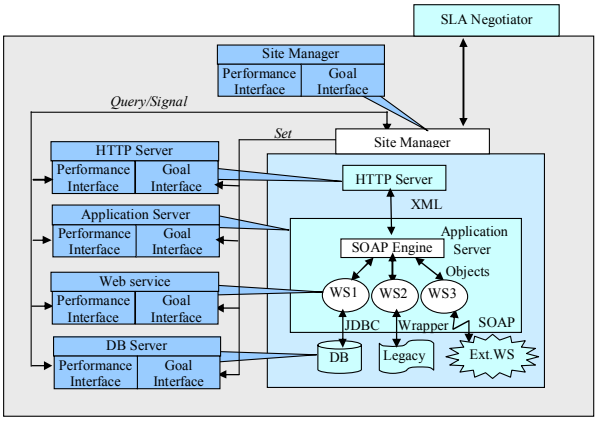

Figure 1: Autonomic Web Services Environment [11]

Knowledge

The MAPE loop requires knowledge about the system topology, performance metrics (of both its managed element and, on occasion, that of other managed elements), component-based and system wide policies, and the expectations, or system goals. Knowledge used by the MAPE loop is stored in a set of database tables that can be accessed internally by the autonomic element, or externally by other autonomic managers via standard interfaces. For our example DBMS buffer pool autonomic manager, the basic system knowledge is represented by the set of tables shown in Figure 3. These tables include BP_Perf_Data (the performance data for the buffer pools), Self-Representation (the current buffer pool configuration), Analyzer_Result (the results from the analyzer), Goals (performance expectations for the buffer pools) and Policy (policies governing the buffer pools). The BP_Perf_Data table is specific to the performance data for the DBMS buffer pools, however, the other tables are general tables shared by other autonomic managers.

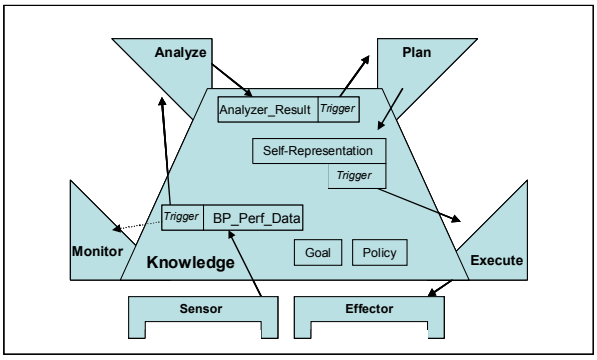

Figure 2: Autonomic Manager for DBMS Buffer Pools

For the buffer pool autonomic manager, the data that most reliably predicts and most accurately depicts potential problems related to buffer pool performance is placed in the table BP_Perf_Data (see Figure 3). It includes information about the types of I/O required by the database (data versus index reads, physical versus logical reads, asynchronous versus synchronous I/O) and, most notably, the buffer pool hit rate (the likelihood of finding a requested page in the buffer pool) which is, in most cases, a good indicator of buffer pool performance.

The self-representation of the autonomic manager reflects the status of the controllable features of the autonomic element. This information is stored in the Self_Representation table. A number of DBMS configuration parameters are related to buffer pool usage including the buffer pool size, the size of the sort area, and the number of asynchronous processes that can be spawned for pre-fetching data or for writing dirty pages back to disk. These parameters are the controllable features of the buffer pool and modifying the values of these settings has an effect on the efficiency of the buffer pool. In our approach, these parameters and their current values form the self-representation of the DBMS buffer pool. The self-representation table is initially populated using an SQL query to the DBMS catalog tables that store the DBMS configuration information.

Goals and policies for the DBMS buffer pools are stored in the Goal and Policy tables respectively. Buffer pool goals are typically specified in terms of hit rate and/or response time goals. Policies describe the rules used to adjust buffer pool sizes. For example, in our case, an increase in the average data access time of greater than 5% triggers a greedy algorithm to reduce the access time by shifting pages among buffer pools [8].

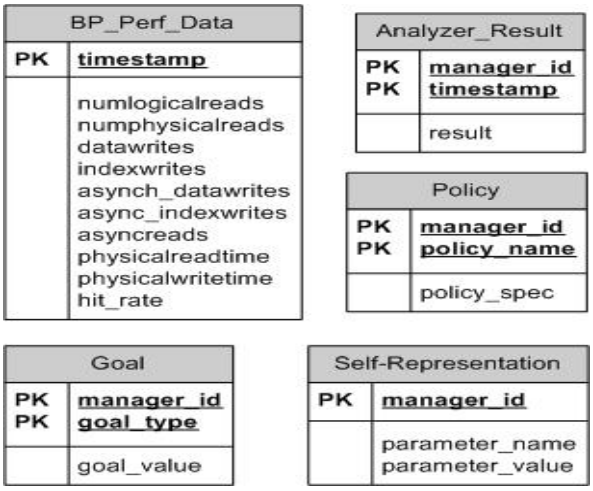

Figure 3: Database Tables for DBMS Buffer Pool Autonomic Manager

Monitor/Sensors

The implementation of the sensors for a managed element is dependent upon the type of interface and/or the instrumentation provided by the element. The sensor for our buffer pool autonomic manager is an application program that uses the monitoring API for the DBMS to retrieve the data relevant to buffer pool performance. The sensor collects the data periodically and inserts the data into the BP_Perf_Data table. The monitor component of the MAPE loop is used to filter and correlate sensor data. In our DBMS buffer pool manager, raw data is used for analysis, so a monitor component is not required.

Analyzer/Planner

The analyzer and planner modules of the MAPE loop are custom built components specific to the autonomic element. The analyzer is responsible for examining the current state of the system and flagging potential problems. The planner module determines what action(s) should be taken to correct or circumvent a problem. These components may be policy-driven, or they may require complex logic and/or specialized algorithms.

In the DBMS buffer pool example, the analyzer defines the logic that examines the performance data in light of the defined policies and goals for the buffer pools, and determines whether or not the component is meeting its expectations.

The planner implements the algorithm(s) that define the adjustment(s) required to achieve the expectations for the managed element. For the buffer pool example, specialized tuning algorithms have been developed that predict buffer pool performance under various configurations and select a configuration that minimizes the overall data access cost [8]. These algorithms are incorporated into the planner logic.

Both the analyzer and the planner for the DBMS buffer pool autonomic manager are implemented as user defined functions and are thus stored in, and are accessible from within, the database management system.

Execute Module/Effectors

The concept of reflection is used in our approach to implement the effectors for an autonomic manager. The self-representation of the system embodies the current configuration settings for the managed element. These represent the features of the managed element that can be controlled. In our example, the self-representation for the DBMS buffer pools includes the current settings for tunable parameters such as the size of the buffer pool, the number of I/O prefetchers, the number of I/O cleaners (threads to asynchronously write pages back to disk) etc. The value of each parameter is adjustable and, when changed, affects the system performance.

The self-representation information for a managed element is stored in the Self_Representation table. An update trigger on the value attribute in this table is used to implement the effectors, that is, the mechanisms that effect change to the managed element. When a change is made to the self-representation, the trigger is fired, thus invoking the execute module which in turn invokes the effector that makes the actual configuration change(s). In the DBMS buffer pool example, the trigger calls the external code which implements the logic that makes the actual change to the DBMS system configuration.

Logic Flow

The MAPE loop is implemented as a feedback control loop that repeatedly monitors the component, analyzes its status, and makes necessary adjustments to maintain a pre-defined level of performance. The control of the feedback loop in our approach is largely implemented by database triggers defined on the database tables that store the system knowledge. This is illustrated in Figure 2.

We describe the flow of control using the DBMS buffer pool autonomic manager beginning with the sensors. The sensors periodically collect data and insert this data into the performance data tables. An insert trigger is defined on the performance data table that invokes the analyzer module whenever new data arrives. Depending on the frequency of data collection the trigger may be modified to fire only periodically, as opposed to every time new data is inserted. The trigger defined on the BP_Perf_Data table is defined (using an IBM DB2 database) as:

```
CREATE TRIGGER NewDataTGR
AFTER INSERT ON BP_Perf_Data
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC  VALUES(AnalyzeBPData());
```

The analyzer code may be implemented as a stored procedure, a user defined function, or as an external program that can be called by the database trigger. In our example, the analyzer code is defined as a user defined function called "AnalyzeBPData". If the analyzer detects a problem, it places a result, or a notification in the Analyzer_Result table in the database. An insert trigger on the Analyzer_Result table calls the planner module whenever the result indicates that some action may be required.

The planner determines the appropriate action to take and, to effect the change, it updates the appropriate data in the Self-Representation table. The update trigger on this table signals the execute module to take the suggested action. In the IBM blueprint, the execute module effects change to the monitored element by way of the effectors, that is, it makes changes to the configuration of the managed element. In our reflective approach, the planner makes a change to the managed element's self-representation and an update trigger defined on this table acts as the effector, the mechanism that actually makes the change to the managed element. The implementation of the routine to make the change depends upon the nature of the managed element. Changes may be implemented via the element's API, or they may involve updating configuration files and possibly restarting the component.

Communication

System-wide management of the Web services environment is facilitated by a hierarchy of autonomic managers that query other managers at the lower level to acquire current and past performance statistics, consolidate the data from various sources, and use pre-defined policies and Service Level Agreements (SLAs) to assist in system-wide tuning. Autonomic managers, therefore, must be able to communicate to share information. In our approach, this is done via standard Web Services interfaces. Two management interfaces are defined for each autonomic element; the *Performance Interface* and the *Goal Interface*. The Performance Interface exposes methods to retrieve, query and update performance data. Each element exposes the same set of methods, but the actual data each provides varies. Meta-data methods allow the discovery of the type of data that is stored for each element. The Goal Interface provides methods to query and establish the goals for an autonomic element, thus allowing external management of a component. Meta-data methods promote the discovery of associated goals and additional methods allow the retrieval of current goals.

4. Summary and Future Directions

Autonomic Computing is a promising approach to solving the problem of effectively managing today's large and complex software systems. The IBM Architectural Blueprint for Autonomic Computing provides a set of guidelines for building autonomic capability into an on-demand computing environment. In this paper we described a general methodology for implementing autonomic managers, which are of one of the key building blocks of the blueprint.

We outlined a reflective database-oriented approach to the implementation of autonomic managers for autonomic computing systems. This approach makes use of reflective programming in which a system maintains a self-representation and changes to the self-representation are reflected in the actual system. Our approach uses relational tables to store the self-representation and the system knowledge and implements triggers to effect the changes to the managed element. The logic flow of the managed element is implemented via database triggers that invoke the appropriate components. Information is shared among cooperating autonomic managers by way of standard Web Services interfaces.

We plan to continue our work on the framework in several areas. First, we will make our interfaces and interactions compliant with the OASIS Web Services Distributed Management (WSDM) standards [9]. This will allow us to enhance the interactions among autonomic managers and will strengthen the case for autonomic management of Web services. Second, we will adopt a specification language for manager policies and develop an interpreter for these policies. The interpreter will form the skeleton for the analyzer and planner modules, which will make the modules much more generic since customization of the actions will be embodied in the policies. Third, we will develop a wider range of prototype autonomic managers using the framework.

References

[1] G. Agha (ed). Special Issue on Adaptive Middleware, Communications of the ACM, 2002, 45(6).

[2] P. Bruni, N. Harlock, M.H. Hong, and J. Webber. DB2 for z/OS and OS/390 Tools for Performance Management. IBM Redbooks, November 2001.

[3] Y. Diao, J.L. Hellerstein, G. Kaiser, S. Parekh, D. Phung. Self-Managing Systems: A Control Theory Foundation. IBM Research Report RC23374 (W0410-080) , October 13, 2004.

[4] IBM, An Architectural Blueprint for Autonomic Computing, June 2005.

[5] G. Lanfranchi, P. Della Peruta, A. Perrone, and D. Calvanese. Toward a New Landscape of Systems Management in an Autonomic Computing Environment. IBM Systems Journal, 42(1), pp. 119-128.

[6] H. Liu and M. Parashar. A Component Based Programming Framework for Autonomic Applications. Proceedings of the International Conference on Autonomic Computing (ICAC 2004), May 17-18, 2004, New York, NY, pp. 10-17.

[7] P. Maes. Computational Reflection, *The Knowledge Engineering Review,* pp.1-19, Fall 1988.

[8] P. Martin, W. Powley, M. Zheng and K. Romanufa. Experimental Study of a Self-Tuning Algorithm for DBMS Buffer Pools. Journal of Database Management, Vol 16(2), April - June, 2005, pp. 1-20.

[9] OASIS. *Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 1*, OASIS Standard, March 2005.

[10] G. Tziallas and B. Theodoulidis. Building Autonomic Computing Systems Based on Ontological Component Models and a Controller Synthesis Algorithm. Proceedings of the 14th International Workshop on Database and Expert Systems Applications (DEXA'03), Sept 1-5, 2003, Prague, Czech Republic, pp. 674-680.

[11] W. Tian, F. Zulkernine, J. Zebedee, W. Powley and P. Martin. An Architecture for an Autonomic Web Services Environment. *Proceedings of the Joint Workshop on Web Services and Model-Driven Enterprise Information Systems WSMDEIS (ICEIS 2005*), May 2005, Miami, Fl.

[12] S.R. White, J.E. Hanson, D.M. Chess and J.O. Kephart. An Architectural Approach to Autonomic Computing, *Proceedings of the International Conference on Autonomic Computing (ICAC'04),* May 17-18, 2004, New York, NY, pp. 2-9.