

Software Architecture Design of an Autonomic System

Yan Zhang Anna Liu Wei Qu

School of Information Technologies, the University of Sydney, Sydney NSW 2006 Australia

E-mail: yanzhang@it.usyd.edu.au

annali@it.usyd.edu.au

wequ3946@it.usyd.edu.au

Abstract

Autonomic Computing is an exciting research direction that aims to provide self-configuration, self-optimization, self-healing and self-protection capabilities to computer systems.

Currently, performance tuning of J2EE application server is a complex manual task. This is unfortunately a necessary task in order to achieve optimal performance under dynamic workload environment.

We propose generic self-configurable software architecture for enabling the automatic tuning of application servers. The proposed architecture encompasses both feed-forward control and feed-back control. We demonstrate the self-configurable software architecture by applying it to J2EE application servers.

1. Introduction

J2EE-based application servers are popular servers-side solutions for many Web-enabled e-business applications. They aim to provide the scalable, high-performance Java infrastructure for processing many simultaneous requests from users for Internet application services, and also offer a consistent design and deployment model, and various software components that aim to ease the development of complex Internet-based applications. However, due to a lack of self-tuning capability in J2EE-based application servers, J2EE-based application servers have to be configured manually to achieve good performance. This kind of manual tuning is time-consuming and requires highly skilled personnel. In addition, since the workloads of e-business sites tend to vary dynamically and exhibit short-term fluctuations [1], even if the system is tuned well at one point in time, it will show poor performance at other times. That is, a statically configured system will perform

poorly in the time-varying workloads as present in typical e-business environment. The end result is that systems performing badly will frustrate customers. Customers will leave this site and move to other sites with better performance.

Autonomic computing [2], which derives its name from the autonomic nervous system, seeks to bring automated self-configuring, self-healing, self-optimizing and self-protecting capabilities into computing systems. The development of autonomic computing systems has attracted much research efforts. In particular, control theory is emerging as a promising approach. In this paper, we propose a novel self-configuration architecture which combines feed-forward control with feed-back control for tuning the performance of J2EE application servers automatically.

The rest of this paper is organized as follows. Section 2 introduces fundamentals of autonomic computing, and describes the J2EE multi-tiered architecture, where components across the tiers are amenable to self-configuration. We also discuss some existing challenges in performance tuning for commercial off-the-shelf (COTS) J2EE products. Section 3 introduces our approach and discusses a specialization of a generic control system model for the purpose of automatic performance tuning in the J2EE component-based system context. To realize such an architecture, a number of design issues are addressed in section 4. Section 5 describes a simple scenario of performance tuning on J2EE platform. Section 6 discusses related works. Finally, conclusion and future works are presented in section 7.

2. Background

This section introduces the fundamentals of autonomic computing and the J2EE multi-tiered architecture. We also highlight the challenges in

performance tuning J2EE application servers and components.

2.1. Autonomic computing

Over time, computing systems are becoming more and more complex. Heterogeneous infrastructures comprised of many different applications, system components have a multitude of essential tuning parameters. In order to meet the business availability requirement 24*7 (24 hours a day, 7 days a week), the cost of managing such computing systems is very high, and the complexity in the tuning task is highly prone to error [3]. To ease this problem, the right software with the right architecture and the right tuning mechanisms need to be developed. Recently, an analogy is made with the *autonomic* nervous system. The autonomic vision seeks to solve problems by using a number of principles of system design to overcome current limitations [2]. The goal of these principles is to make the systems become much more self-managing, that is, self-configuring, self-healing, self-optimizing, and self-protecting.

Full autonomic computing systems will be:

Self-configuring: Able to adapt to dynamically changing environments

Self-healing: Able to discover, diagnose, and act to prevent disruptions

Self-optimizing: Able to tune resources and balance workloads to maximize use of IT resources

Self-protecting: Able to anticipate, detect, identify, and protect against attacks

Table 1 shows the four aspects of self-management as they are now and would be with realization of autonomic computing.

Autonomic computing is an exciting new research direction in computing. Existing research on autonomic computing not only focuses on servers, networks, databases and storage management, but also touches on the human-computer interaction aspects [3]. Every aspect of autonomic computing offers grand challenges on engineering and science [16].

Our work focuses on the self-configuration and self-optimization aspects.

Table 1. Four aspects of self-management as they are now and would be with autonomic computing

Concept	Current computing	Autonomic computing
Self-configuration	Corporate data centers have multiple vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone.	Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
Self-optimization	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their own performance and efficiency.
Self-healing	Problem determination in large, complex systems can take a team of programmers weeks.	System automatically detects, diagnoses, and repairs localized software and hardware problems.
Self-protection	Detection of and recovery from attacks and cascading failures is manual.	System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures.

From "The Vision of Autonomic Computing", IEEE Computer, January 2003. [16]

2.2. J2EE Multi-Tiered Application Architecture

A simple depiction of the J2EE multi-tier architecture is shown in Figure 1. The role of each tier is as follows:

Client Tier: In a Web application, the client tier is composed of an Internet browser that submits HTTP (hypertext transfer protocol) requests and downloads HTML (hypertext markup language) pages from a Web Server. In an application not deployed using a browser, stand-alone Java clients or applets can be used, and these would communicate directly with the Business Component tier, using the Java Remote Method Invocation (RMI) as the underlying protocol.

Web Tier: The Web tier runs a World Wide Web server to handle client requests, and invokes J2EE

servlets or Java Server Pages (JSPs). Servlets are invoked by the Web server depending on the type of user request and will query the business logic tier to get the required information to satisfy the request. The servlets then format the information for return to the user via the Web server. JSPs are basically static HTML pages that contain snippets of servlet code. The code is invoked by the JSP mechanism, which also takes responsibility for formatting the dynamic portion of the page.

Business Component Tier: The business components constitute the core business logic for the application. The business components are realized by Enterprise JavaBeans, the software component model supported by J2EE. EJBs receive requests from servlets in the Web tier, or directly from Java clients. EJBs then satisfy the request usually by accessing some data sources, and return the results to the servlet or the Java client. EJB components are hosted by J2EE environment known as an EJB container. The container supplies a number of services to the EJBs that it hosts. These services include transaction and lifecycle management, state management, security, multi-threading, and resource pooling. EJBs simply specify the type of behaviour they require from the container at run time, and then rely on the container to provide the services. This frees the application programmer from cluttering the business logic with code to handle system and environmental issues.

Enterprise Information Systems Tier: This tier typically consists of one or more databases and back-end applications like mainframes and other legacy systems. EJBs must query these data stores to process requests. The Java Database Connectivity (JDBC) drivers are typically used for accessing databases, and the Java Connector Architecture (JCA) standard protocol is used to access packaged applications such as enterprise resource planning (ERP) systems and customer relationship management (CRM) systems, as well as various mainframe-based transaction processing systems.

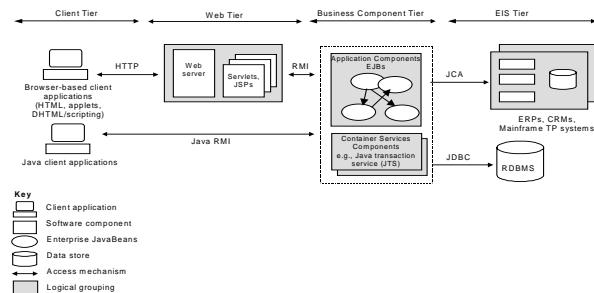


Figure 1: J2EE Multi-Tiered Application Architecture

2.3. Challenges in Performance Tuning J2EE COTS components

With the ubiquity of Internet access, limitless numbers of users may converge at a system at the same time. The performance of systems then quickly became the crucial concern of the developers of the systems. Because J2EE-based application server plays the key roles in distributed systems, its performance is of utmost importance in building distributed systems. Failure to meet the performance requirements can lead to entire system failure. To meet the e-business performance requirement, most of the J2EE-based products have various configurable system parameters, such as thread pool size, database connection pool size, and application component cache etc. Software architects and performance tuning engineers need to configure those parameters in order to tune the performance behaviour of the J2EE-based applications. Manual performance tuning using tuning values as advised by J2EE product vendors does not always result in the most optimal system performance. In practice, software architects experimentally discover configurations that provide the desired levels of performance. This approach has some drawbacks. The experimental approach is a time-consuming, expensive and non-trivial method and it must be done late in the project life-cycle when the application has been built. Besides, the software systems configured by the experimental approach do not possess the flexibility and adaptation capability. Software architects configure their systems according to their experimental results. Once software systems have been built, the configurations were fixed. The performance characteristic is only observed afterwards. Further, if the external environment changes, for example, the workloads change with the time, the performance will not necessarily be most optimal overtime. Hence, current static approach to performance tuning cannot result in an optimal performance.

Our project is proposing to build a new architectural mechanism, which dynamically monitors and measures the system behaviours, independently feed these measures into a behavioural model, and use the results to adaptively self-configure the systems for optimal performance.

3. Self-configuration and Self-optimisation architecture

This section introduces our architecture design approach and discusses a specialization of a generic

control system model for the purpose of self-configuration for optimal performance in a J2EE component-based system context.

3.1. Approach

We begin with a generic control system model. Then we refine it to enable automatic performance tuning in J2EE systems. This specialized model becomes our architectural framework, which is independent of any specific performance tuning strategy or control algorithm. Therefore, different tuning strategy and algorithms can be compared and evaluated within this architectural framework. For a detailed discussion of performance modeling and tuning strategy, see our other work in [17].

In this framework, we combine the predictive autonomy approach with reactive autonomy approach. Predictive autonomy approach is based on feedforward control while reactive autonomy approach is based on feedback control [4].

3.2. Generic Control System

In general, a control system is composed of controller and controlled system, shown in Figure 2 [5].

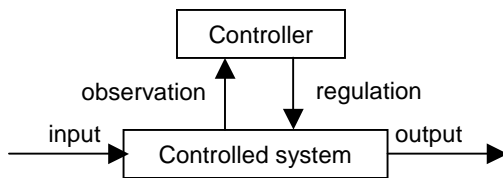


Figure 2. Generic control system

The controller can control the controlled system via feed-forward control strategy or feed-back control strategy. With a feed-forward control strategy, regulation is determined based on the input to the controlled system. With a feed-back control strategy, regulation is determined based on the comparison of the difference between a desired objective and measured output delivered by the controlled system.

3.3. Self-configuration and Self-Optimisation architecture for J2EE application server

In the context of this work, the “controlled system” is the J2EE application server and the “controller” provides dynamic performance regulation. Figure 3 shows the specialized architecture of the control model for autonomic performance tuning of J2EE systems.

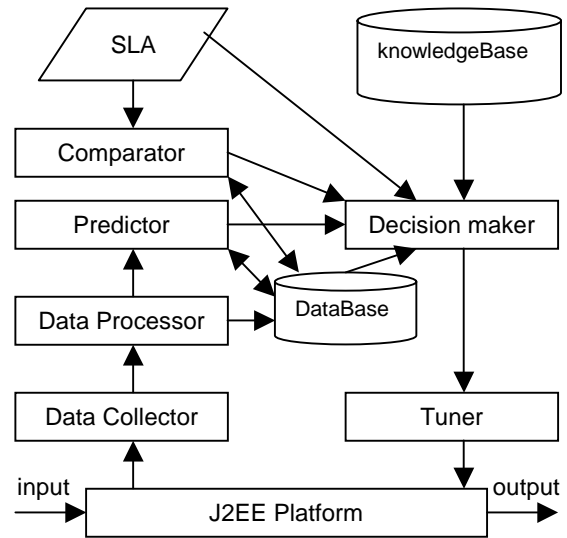


Figure 3. Autonomic performance tuning architecture

The Data Collector is to obtain observations. It collects real-time performance data which will be further processed by Data Processor to obtain average response time or throughput values etc. These real-time performance data stored in the database will be used by other components. The Predictor uses the performance data to predict future workloads to help the Decision Maker to proactively deal with the time-varying workloads. The comparator will compare the difference between the real-time performance data such as throughput and Service Level Agreement (SLA)¹. The Decision Maker applies some algorithm based on the result from Predictor, Comparator and SLA to choose an appropriate performance tuning strategy in the Knowledge Base. The Tuner will translate the strategy into actions and inject the actions into the J2EE application server to realize the tuning strategy to attain the system performance requirements.

4. Design Issues

This section identifies and elaborates a number of technical issues that have to be addressed in order to realize the above architecture.

¹ Service Level Agreement (SLA) is a contract between a consumer and a provider of an IT service that specifies the levels of quality (e.g. availability, performance, security etc) of the service.

4.1. Data Collection

Firstly, we need to consider what kind of performance data need to be collected. Example data to collect include request arrival rate and CPU utilization. This decision is closely related to the actual performance tuning model and strategy to be employed [17].

Once we understand which performance data we need to collect, we need to think about how to get these data. Obtaining these data is a non-trivial issue. Firstly, we need to develop our own monitoring component to plug into the J2EE component-based system. This brings us to two additional design issues: one is that how do we minimize the impact of the instrumentation code on the J2EE component-based system; the other is how should we in fact insert the monitoring code into the J2EE component-based system's code to ensure code modularity. One idea in addressing the first issue is to use reflection-based approach to help with collecting some of the required data. Reflection is the ability of a system to watch its computation and possibly change the way it is performed [6]. The second above mentioned issue is addressed by the Aspect-Oriented Programming community, in the area of cross-cutting concerns [7].

4.2. Data Processing

Data processing relies on many factors: the collected data, the required measurements, and the rules for data processing. A generic model should be developed to define how data collected are translated to measurements so that it can be reused or generated automatically as much as possible. Due to statistical measurements requirements, lots of data need to be stored. Therefore, the cost of processing store data should be reduced as much as possible. The data processing can be done through calculations, heuristics (logic rules) or conversions etc. We need to model these different techniques in a uniform way.

4.3. Workload and Performance Prediction

This component uses the historical performance data and prediction algorithm to predict future workload and system performance and provides corresponding confidence levels, which helps Decision Maker to choose the reasonable performance tuning strategies.

The techniques applied most often to workload prediction are moving averages, exponential smoothing, and linear regression ([8], [9]). The pattern found in historical data has a strong influence on the

choice of the technique. Performance prediction modeling is addressed in [10].

4.4. Decision Making

Comparator compares the real-time performance measurements with the Service Level Agreement and determines the difference. Based on the results from Comparator and Predictor, Decision Maker will choose appropriate tuning strategies and balance conflict strategies according to some rules and control algorithm to help the system to achieve optimal performance while maintaining overall system stability.

4.5. Auto Tuning

When the Tuner translates the strategy into actions and inject the actions into the J2EE component-based system, we will meet the same issues described in 4.1. In addition, we should notice that tuning may bring the problem of system instability. Therefore, we need to augment the algorithm and injection method to avoid the instability.

4.6. Performance Cost

It is recognized that collecting, storing and processing data is a performance overhead to the system. Factors affecting these overheads include frequency of data collection and the number of rules. Therefore, the instrumentation framework for gathering runtime performance statistics should be as non-intrusive as possible, and incurs minimal performance penalty.

5. A Performance Tuning Case Study

This section describes a scenario of performance tuning on J2EE platform. This motivating case study provides us the foundation for examining the challenges of performance tuning and the optimization problem. The insights gained through this experience will assist us with constructing the self-configuring and self-optimising architecture.

5.1. Assumption

There are two ways to improve the performance of an e-business application built on J2EE. One is tuning the application itself. The other is tuning the J2EE application server through the application server's configuration parameters. Here we focus on the latter.

It is assumed that the application has been reasonably tuned.

5.2. Types of Tuning Parameters for J2EE Component-based System

There is a myriad of tuning parameters for J2EE application servers. These parameters can be adjusted in order to specific applications requirements. These parameters can be classified into two types: those with high impact on performance; and those for avoiding and recovering from failure. Here we focus on the performance parameters.

5.3. A Performance Tuning Scenario

In this J2EE application server performance tuning scenario, we will focus on the EJB container component and the data source component. When a client invokes a server component, the container automatically allocates a thread and invokes an instance of the component. The container manages all resources on behalf of the component and manages all interactions between the component and the external systems such as database management systems. According to our experimental results [17], Data source connection pool size and thread pool size are two of the performance parameters which have high performance impacts. To demonstrate and validate our architectural framework, we choose them as key tuning sensitivity points to start with. We use JBoss Application Server for our research because it is not only freely available but also is representative of the commercial J2EE products. In addition, JBoss is built on the top of JMX infrastructure. It is easy for us to get run-time system performance data and it is also possible for us to add our own component to dynamically tuning the system. The new Aspect oriented programming features of JBoss also holds promises for being a good platform for injecting cross-cutting concerns and code.

As shown in Figure 4, EJB clients arrive at the EJB container. Remote enterprise Java beans communicate using the RMI/IIOP protocol. Method invocations initiated over RMI/IIOP will be processed by a server side server. If the requests can not be served by a server thread, they will be put in the waiting queue. To obtain the ideal performance result, the good performance setting practice on thread pool and Database connection pool is to let them have a threading “funnel effect” [11]. That is, the size of Database connection pool is usually smaller than that of thread pool.

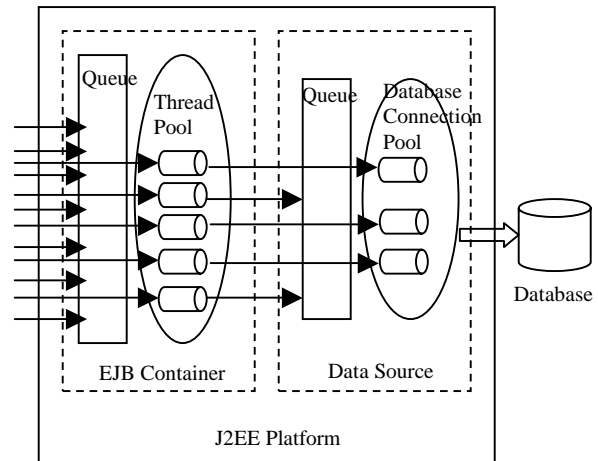


Figure 4. A simple scenario for performance tuning

In our research, we will set up and refine the quantitative relationship between the two parameters via combining control theory with our previous experimental results [12] [15]. The quantitative formula describing the system performance characteristic as well as the logic rules and application-specific factors will be embedded in the knowledgebase to help the Decision maker to choose appropriate tuning strategies.

6. Related Work

Control theory originally used in traditional engineering community is now being tried and applied to autonomic computing in the area of software configuration. [13] and [14] are based on feedback control approach. In the work of [13], a web server QoS provisioning architecture is proposed for the purpose of adapting web contents to provide overload protection of web servers. The framework provided by [14] is used for distributed multimedia applications to dynamically control and reconfigure their internal parameters and functionalities. Different from the above work, [4] proposes a method called “clockwork”, which is based on feed-forward control approach. It uses statistical modeling, tracking and forecasting techniques borrowed from econometrics to self-tune system to dynamically adapt to the time-varying workloads. Our work is different from the above work. Our framework combines feedback control approach with feed-forward control approach. Further, our research does not focus on distributed applications. We focus on server side component self-configuration.

7. Conclusions and future work

We have proposed an architecture for implementing autonomic behavior in J2EE based systems. It uses a hybrid feed-forward and feedback approach. To demonstrate and validate our proposed architecture, JBoss application server is used in our research. Based on CSIRO empirical results [12] [15], we are applying control theory to quantify the relationship between performance metric and two performance tuning parameters which have high performance impact. The quantitative formula describing the system performance characteristic will be embedded in the knowledgebase to help the Decision maker to choose appropriate tuning strategies.

At the same time, we will try to use statistical methods such as auto regression analysis to predict future workloads. A tuning algorithm will be developed based on workload prediction and system performance behavior.

After completing such prototype, we will incorporate more tuning parameters into our model and set up more tuning strategies. A decision mechanism will be set up to choose appropriate strategies and balance the conflict strategies. The ultimate goal of our self-reconfiguration architecture is to eliminate manual re-configuration effort, enable system self-optimisation and to minimize system downtime.

8. Acknowledgement

We acknowledge CSIRO for the assistance and support for this work.

9. References

- [1] D. A. Menascé, V. A. F. Almeida, R. Riedi, F. Pelegrinelli, R.Fonseca, and W. Meira Jr., "In Search of Invariants for E-Business Workloads," Proc. Second ACM Conference on Electronic Commerce, Minneapolis, October 17-20, 2000.
- [2] http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf
- [3] A. G. Ganek and T. A. Corbi, "The Dawning of the Autonomic Computing Era", IBM SYSTEMS JOURNAL, VOL 42, NO 1, 2003, pp5-18.
- [4] L. W. Russell, S. P. Morgan and E. G. Chron, "Clockwork: A new movement in autonomic systems", IBM SYSTEMS JOURNAL, VOL 42, NO 1, 2003, pp77-84.
- [5] William L. Brogan, *Modern control theory 3rd ed*, Englewood Cliffs, N.J. , Prentice Hall, c1991.
- [6] F., Kon, et al, "Monitoring, Security and Dynamic Configuration with the DynamicTAO Refeective ORB", in Procs IFIP/ACM Int. Conf. On Distributed System Platforms and Open Distributed Processing, (Middleware 2000), Germany 2000, Springer-Verlag, pages 121-143.
- [7] G., Kiczales, et al., "Aspect-Oriented Programming", Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.
- [8] H. Letmanyi, *Guide on Workload Forecasting*, Special Publication 500-123, Computer Science and Technology, National Bureau of Standards, Washington, D. C., 1985.
- [9] T. Lo, "Computer workload forecasting techniques: a tutorial", Proceedings of the International Conference on Computer Capacity Management, San Francisco, 1980.
- [10] Daniel A. Menasce, Virgilio A. F. Almeida and Larry W. Dowdy, *Capacity Planning and Performance Modeling*, PTR Prentice Hall, Englewood cliffs, New Jersey 07632, 1994
- [11] Stacy Joines, Ruth Willenborg and Ken Hygh, *Performance Analysis for Java™ Web Sites*, Person Education, Inc. 2003.
- [12] I. Gorton, A. Liu, P. Brebner, "Rigorous Evaluation of COTS Middleware Technology", IEEE Computer, March 2003.
- [13] T. Abdelzaher and N. Bhatti, "Web Server QoS Management by Adaptive Content Delivery," in *Proceedings of Seventh International Workshop on Quality of Service*, May 1999.
- [14] B., Li, K. Nahrstedt, "A control-based middleware framework for quality of service adaptations", IEEE J. on Sel. Areas in Comms. Vol. 17, No. 9, pp1632-1650, Sept. 1999.
- [15] CSIRO Middleware Technology Evaluation Series: Evaluating J2EE Application Servers
- [16] Jeffrey O. Kephart and David M. Chess, "The Vision of Autonomic Computing", IEEE Computer, January 2003, pp41-50.
- [17] S. Chen, Y. Liu, I. Gorton, A. Liu, "Performance Prediction of Component-based Applications", The Journal of Systems and Software, December 2003.