Spring 5-13-2016

Survey of Autonomic Computing and Experiments on JMX-based Autonomic Features

Adel R. Azzam
The University of New Orleans, arazzam@uno.edu

Survey of Autonomic Computing and Experiments on JMX-Based Autonomic Features

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by

Adel Azzam

B.S. University of New Orleans, December 2010

May 2016

Dedication

I dedicate this to my family, for all their love and support.

Acknowledgement

There are many people who have helped me along my journey. I feel so blessed and fortunate to have the family, friends, colleagues, classmates and teachers that I do. I don't think I can remember everyone but the following people come to mind.

I would firstly like to thank my advisor, Dr. Shengru Tu. He has been very supportive, patient, and helpful through it all. He has gone above and beyond, sometimes even meeting with me on weekends to help me with the work. I am very appreciative of his mentorship and wisdom. I cannot express my gratitude adequately in writing.

Also, I would like to express my gratitude to my other committee members - Dr. Mahdi Abdelguerfi and Dr. Minhaz Zibran. They have both been supportive and helpful. Many thanks to both committee members for taking time out of their busy schedules to help me.

Furthermore, I would like to thank Dr. Adlai DePano , Dr. Christopher Summa, and Dr. Christopher Taylor, and Dr. Walter Michaelis for being great teachers and mentors for me through the years.

I would like to thank fellow UNO alumni Daniel Stocker (MS, class of 1996) and James Sauvinet (MS, class of 2013) for their help and mentoring.

Other students who deserve thanks are Jorge Chao, Md. Rakibul Islam, Lu Yang, and Nathan Cooper.

I would like to thank my parents and my siblings. They have always been there for me. My father, Dr. Rasheed Azzam, helped me with editing, reviewing, and mentoring. My brother Omar Azzam also helped. I also must acknowledge the support of my extended family overseas and in the United States.

Finally, I thank all of the staff of UNO - from the Aramark employees to the RIS center custodial staff. I must especially thank Ms. Melanie Bopp, the Circulation Services Coordinator of EKL library, for her excellent support in helping me obtain a carrel.

Table of Contents

List of Figures

List of Tables

Abstract

Autonomic Computing (AC) aims at solving the problem of managing the rapidly-growing complexity of Information Technology systems, by creating self-managing systems. In this thesis, we have surveyed the progress of the AC field, and studied the requirements, models and architectures of AC. The commonly recognized AC requirements are four properties - self-configuring, self-healing, self-optimizing, and self-protecting. The recommended software architecture is the MAPE-K model containing four modules, namely - monitor, analyze, plan and execute, as well as the knowledge repository.

In the modern software marketplace, Java Management Extensions (JMX) has facilitated one function of the AC requirements - monitoring. Using JMX, we implemented a package that attempts to assist programming for AC features including socket management, logging, and recovery of distributed computation. In the experiments, we have not only realized the powerful Java capabilities that are unknown to many educators, we also illustrated the feasibility of learning AC in senior computer science courses.

Keywords: Autonomic Computing; Java; distributed computing; multi-agent systems; process control theory; grid computing

Chapter 1. Introduction

Autonomic computing is a concept that brings together many fields of computing with the purpose of creating computing systems that manage themselves (Huebscher 2008). Autonomic computing is a paradigm for dealing with the growing complexity of computer systems, which currently requires system administration. This paradigm contributes to a number of other technologies such as grid computing, virtualization, and service-oriented architecture (SOA) (Ganek 2007). Autonomic computing is inspired by the human autonomic nervous system (ANS), which regulates our body functions.

IBM forecasts that the number of computing devices will grow at the rate of 38% per year ("IBM" 2). While business productivity is rising, these advances are creating management challenges for IT staff. Applications often span multiple resources – application servers, Web servers, integration middleware, and legacy systems. These composite applications are difficult to manage. At the same time there is a growing volume of data, escalating demand and requiring more complex IT environments. Estimates indicate that up to 80% of an average company's IT budget goes to the maintenance of existing software (Ganek 2007).

As demand for skilled IT personnel easily outstrips supply, labor costs now exceed equipment costs by a large ratio, up to 18:1. IBM started calling for the grand challenge of autonomic computing in 2001. This initiative aimed to develop computing systems capable of self-management, to overcome the rapidly growing complexity of systems management (Kephart 2003).

In this thesis, we will review the autonomic computing landscape, the system requirements, the essential properties, and the architectural models of autonomic computing. At the heart of autonomic computing is the concept of software adaptation which can be achieved either at the operating system levels or at the specific programming levels. We will discuss the techniques at both levels, briefly including program-level adaptation in light of the Java language. The thesis will pay more attention to the relation of autonomic computing to Service-Oriented Architecture (SOA). Particularly, we will study the SOA-approach to autonomic computing for pervasive systems, as well as a software architecture for the dynamic configuration and composition of communication services, which applies the composition design pattern.

The literature surveyed has clearly indicated that autonomic computing has been in the requirements of all the modern critical enterprise systems. Autonomic features have been implemented in numerous distributed applications. A trend of increasing demand for autonomic features has been demonstrated in both Web and mobile applications, as well as cloud-based information systems. However, such a critically important aspect of computation has not been studied enough in computer science education. A goal of this thesis is to investigate the feasibility of adopting autonomic computing concepts and techniques in advanced computer science courses. A particular focus will be on distributed programs because of the inherently unreliable nature of network communication. Through a series of experimental programming exercises, we hope to develop a software library that can simplify implementations of autonomic features.

Chapter 2. Survey of Autonomic Computing

2.1 Individual Implementations of Autonomic Computing

Even before IBM’s Autonomic Computing initiative came into existence, inroads into the subject area were being made by the US Department of Defense and NASA. In this chapter, we review Autonomic Computing history, and also look at current software that demonstrates Autonomic Computing abilities.

2.1.1 DARPA Projects

Many self-management research projects were launched by DARPA for military applications. The first set of DARPA projects were focused on enabling a generation of self-repairing, self-forming, self-defending and heterogeneous networks, in order to provide security and critical advantages in unpredictable and dangerous environments. Some of these projects included the Small Unit Operations-Situational Awareness System (SUO-SAS) program, the Optical RF Combined Link experiment (ORCLE), Future Combat Systems Communications (FCS-C), and the Wireless Networks after Next (WNaN) program.

A series of DARPA programs was launched for addressing autonomy issues in battery-powered wireless systems, such as unattended ground sensor (UGS) networks (Lalanda 2014).

One of the more interesting projects launched by DARPA was the Dynamic Assembly for System Adaptability, Dependability and Assurance (DASADA), which commenced in 2000. The goal of DASADA was the development of technology for ensuring the dependability of mission-critical systems. Some of the research motivated by this project has initiated architecture-driven solutions for self-managing large-scale distributed systems. These solutions rely on the extraction of runtime software "gauges." These gauges are used for the monitoring of system dependability properties (e.g. safety, security, and architectural coherence), analyzing the collected information for detecting variance from the predicted behavior, and dynamically adapting the system in order to prevent violations of acceptable behavior (for example, self-repairing running systems) (Lalanda 2014).

2.1.2 NASA

NASA has a natural interest in the field of autonomic capabilities, due to the nature of its unmanned space missions. In these missions, communication between land-based control centers and spacecraft is frequently unavailable and continuously affected by long round-trip delays. In these contexts, the success of costly explorations has been totally dependent on the autonomic capabilities of spacecraft devices, which allow rapid control decisions to be made in real-time. Self-repair and self-reconfiguration are also critical capabilities in these missions, since direct communication with the devices is impossible.

The autonomic ability in NASA space missions includes star-tracking based navigation, self-directing antennas, automatic fault reactions, and data storage and retransmission. One example is the AutoNav autonomous navigation system, which was employed on board the Deep Space 1 (DS1) and Deep Impact spacecrafts for enabling high-speed encounter missions to small bodies, like comets and asteroids.

The popular Mars rover was another example of an autonomic system. Planet surface explorations demand the autonomous rovers to find their way through often difficult and rugged territories.

Yet another project with a futuristic slant that NASA unveiled is called the Autonomous Nano Technology Swarm (ANTS) program (2000). ANTS uses the idea of a generic architecture for human/robotic space mission, which is based on an ant colony analogy. The ant-colony provides an excellent example of self-organization that engenders a resilient and relatively cheap system. The ANTS system also utilizes high social interaction capabilities that enable self-organization into various structures in order to achieve predefined goals (Lalanda 2014).

2.2 System Approach to Autonomic Software

2.2.1 IBM's Grand Vision of 2001

On October 15th, 2001, the vision of autonomic computing was made explicit by Paul Horn, the then senior vice president of IBM Research. In his speech he suggested to build computer systems that "regulate themselves much in the same way our autonomic nervous system regulates and protects our bodies" (Ghosh 2007). In other words, there should be a minimum of human interference (IBM 2).

To solidify and prove their mission, IBM released an IBM Autonomic Computing Toolkit through its DeveloperWorks website. It was a free add-on to the Eclipse development environment. This was implemented in Java and had features such as "Common Base Events" and a "Generic Log Adapter", along with an "Autonomic Management Engine." However, support for this toolkit was discontinued by IBM in 2007.

2.2.2 Self-CHOP Properties of an Autonomic Computing System

There are four key properties for any given autonomic computing system: self-configuration, self-healing, self-optimization and self protection. These properties are often abbreviated as the "Self-CHOP properties" (also termed "Self-X"). Below we summarize what these terms mean.

Self-Configuration

Self-Configuration means the system’s ability for automatic configuration of components, according to high-level goals. More specifically, this capability gives the system the power to adapt to unpredictable situations. For example, it may remove or add new components, or install software changes (without stopping the current service) (Huebscher 2008).

Self-Healing

Self-healing denotes the ability to automatically discover, and subsequently correct, faults. For example, a service disruption in a major website can be prevented if our system has self-healing abilities. This is of interest to all major websites, as they can increase profits by minimizing their downtime (Huebscher 2008).

Self-Optimization

This refers to the automatic control of resources, and monitoring them to ensure peak functioning with respect to the defined requirements. In other words, this enables the system to tune itself on-the-fly to proactively improve the current processes, and to reactively respond to environmental conditions. This quality is of interest in the security domain, where hackers may compromise the system at any time.

Self-Protecting

The system must be able to proactively identify and protect against any arbitrary attacks or vulnerabilities. For example, any large enterprise system may be susceptible to denial-of-service attacks. This is vital in the current era of ever-increasing connectivity, as exemplified by the rapid adoption of smart phones.

Self-protection also includes a system's ability to prevent *physical* harm – such as the motion detection in modern laptops, which protects disk drives by temporarily parking the disk-drive head upon sensing that it is being dropped.

Self-
Configuring
Self-
Healing
Self-
Optimizing
Self-
Protecting

Figure 2.1 - Self-CHOP properties (Jacob)

2.2.3 Evolutionary Levels in Pursuit of Autonomic Computing

IBM introduced five evolutionary levels in the autonomic computing spectrum. These levels are defined as follows in Table 1 (Gusworld 1):

Table 1 - Evolutionary Levels in the Autonomic Computing Spectrum

Level	Description
Level 1	Basic
Level 2	Managed
Level 3	Predictive
Level 4	Adaptive
Level 5	Autonomic

At the very beginning, Level 1 is the *Basic* level. This level corresponds to the situation today - where systems are mostly managed manually. Here, we have many sources of system generated data, that require a highly skilled staff.

The next level is *Managed*, and here we have greater system awareness, improved productivity and consolidation of data and management tools.

In Level 3, the *Predictive* level, the system monitors and recommends actions, and the staff will approve or disapprove those actions. Thus, it “figures out” the plan and awaits a human’s approval.

Next is Level 4 – the *Adaptive* level where the system monitors and initiates action, and then the staff manage that against service level agreements.

Finally, Level 5 is fully *Autonomic*. The system will monitor and initiate actions based on business processes. In this stage the IT staff can focus on implementing business requirements.

2.2.4 Key Features of Autonomic Computing

IBM's autonomic computing manifesto identified eight key characteristics that define an autonomic system (Lalanda 2014):

1. Holding self-knowledge and consisting of elements that have system identification information
2. Being able to reconfigure in reaction to environmental changes (which are potentially unpredictable)
3. Always in a state of striving to optimize functioning, in order to reach predefined criteria
4. To first detect, and then recover, from component failures in order to maintain global dependency
5. In regards to various threats, being able to anticipate, detect and avoid them in order to maintain integrity and security.
6. Acquiring environment knowledge and behaving in a context-sensitive manner
7. Implementing open standards in order to be able to survive in a heterogeneous ecosystem
8. Hiding complexity through bridging the gap between underlying IT resources and business goals.

2.2.5 MAPE-K Model

Central to the autonomic computing architecture is the idea of an autonomic manager and a MAPE-K loop. This is simply a logical architecture, not a mandatory blueprint, which defines the various activities to be carried out in order to have autonomic loops. MAPE-K is an acronym for monitor, analyze, plan, execute, and knowledge. These are the aspects involved within any autonomic cycle.

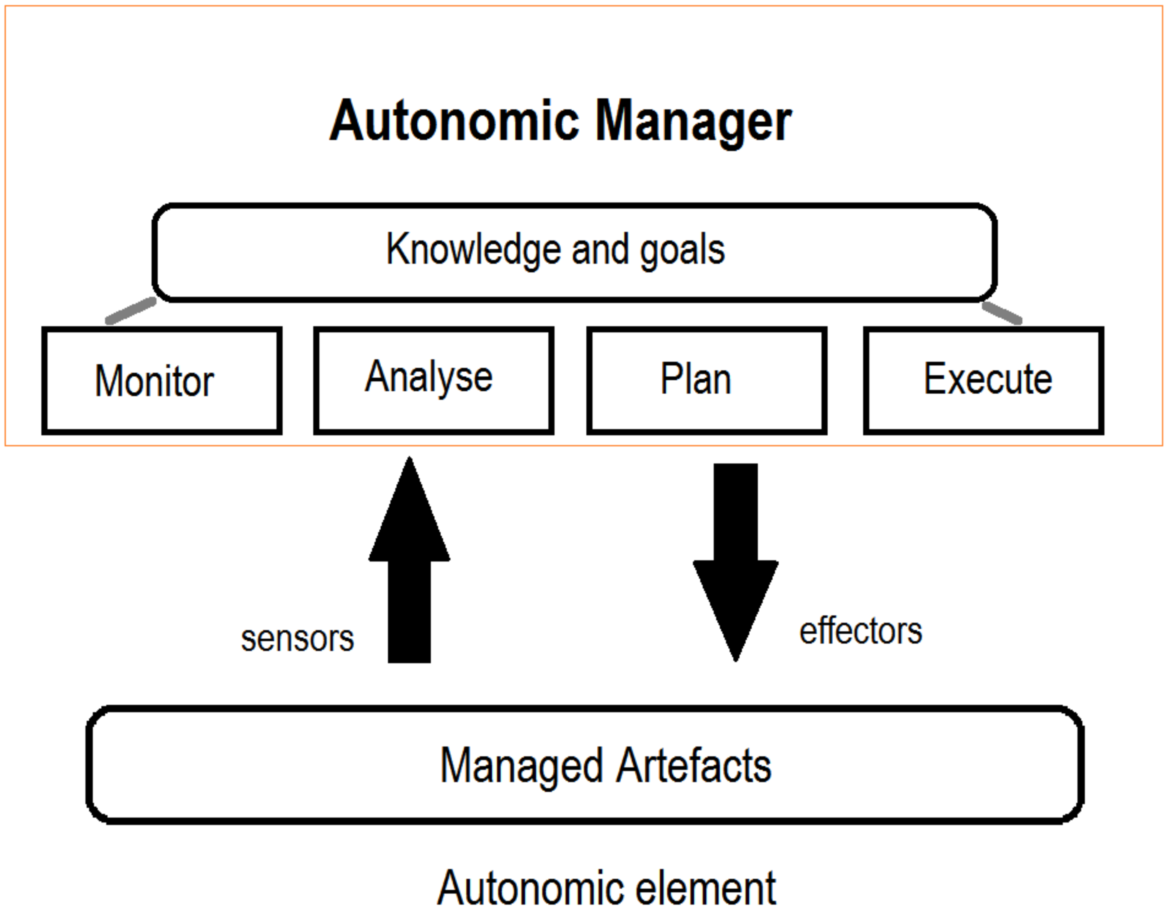


Figure 2.2 – MAPE-K Model (Parashar)

In the first stage, monitoring, we build a model of the managed artefacts and execution context. Following this, analysis uses the blueprint built by the monitoring to assess the situation and determine any anomalies (Lalanda 2014).

The planning stage comes after analysis. Its purpose is to figure out a set of management actions that allow the passage from a current state to a desired state. And lastly, the execution stage carries out plans, dealing directly with the effectors that are provided through the managed artifacts (Lalanda 2014).

The MAPE-K logical architecture is extremely important in the field of autonomic computing. It gives a structural framework to begin with when creating an autonomic system. The architecture is scalable because the activities can be run on different machines (Lalanda 2014).

2.2.6 Research on Decision-Making Techniques

There is significant research in the area of decision making, in regards to autonomic computing. A paper by Maggio et al. describes various approaches and techniques, and outlines five main techniques for them: heuristic solutions, standard control–based solutions, advanced control-based solutions, model-based machine learning solutions, and model-free machine learning solutions (Maggio 2011).

Heuristic solutions begin with a guess about application needs, and adjust it. They are designed for simplicity and performance, and thus sacrifice precision. They usually cannot be proven to converge to the optimum value. For example, the greedy approach optimizes resource allocation and energy management in a hosting center (Maggio 2011).

Standard control-based solutions use canonical models and apply standard control techniques such as Proportional Integral (PI) controllers, Proportional Integral and Derivative (PID) controllers, optimal controllers, or Petri nets. Two examples of these models are discrete-time linear models and discrete event systems. Important properties may be enforced such as convergence time and stability (Maggio 2011).

Advanced control-based solutions involve complex models (involving some unknown parameters like machine workload) that may be estimated online, to provide Adaptive Control (AC). Adaptive Control requires the ability to change controller parameters in real-time, and an identification mechanism. Another advanced control strategy is Model Predictive Control (MPC). In MPC the controller selects the next actions based on the prediction of the future system reactions. Although the overhead of such solutions is greater than that of standard controls, one may still be able to formally analyze parameter-varying systems. Thus, one may prove stability and obtain formal guarantees even in the case of unknown operating conditions. For example, one real-world example adjusts the CPU percentage dedicated to a web server adaptively (Maggio 2011).

In model-based machine learning solutions, a framework is required in which to learn system behavior and adjust tuning points online. Neural Networks (NN) are often used to build control-oriented nodes. Once trained, NNs may predict the system reaction to various inputs. It should be noted that the structure of the network and the training data quality are paramount.

2.3 IBM's Autonomic Computing Architectural Concepts and Toolkit

In IBM's 2005 white paper ("An architectural blueprint for autonomic computing") a common approach was outlined, along with terminology, to describe a self-managing autonomic computing system. This approach shows how all parts of an autonomic system are connected using enterprise service bus patterns. This enterprise service bus integrates the various building blocks, which include:

- Touchpoints for managed resources
- Knowledge Sources
- Autonomic Managers
- Manual Managers

Figure 2.3 below shows this autonomic computing reference architecture:

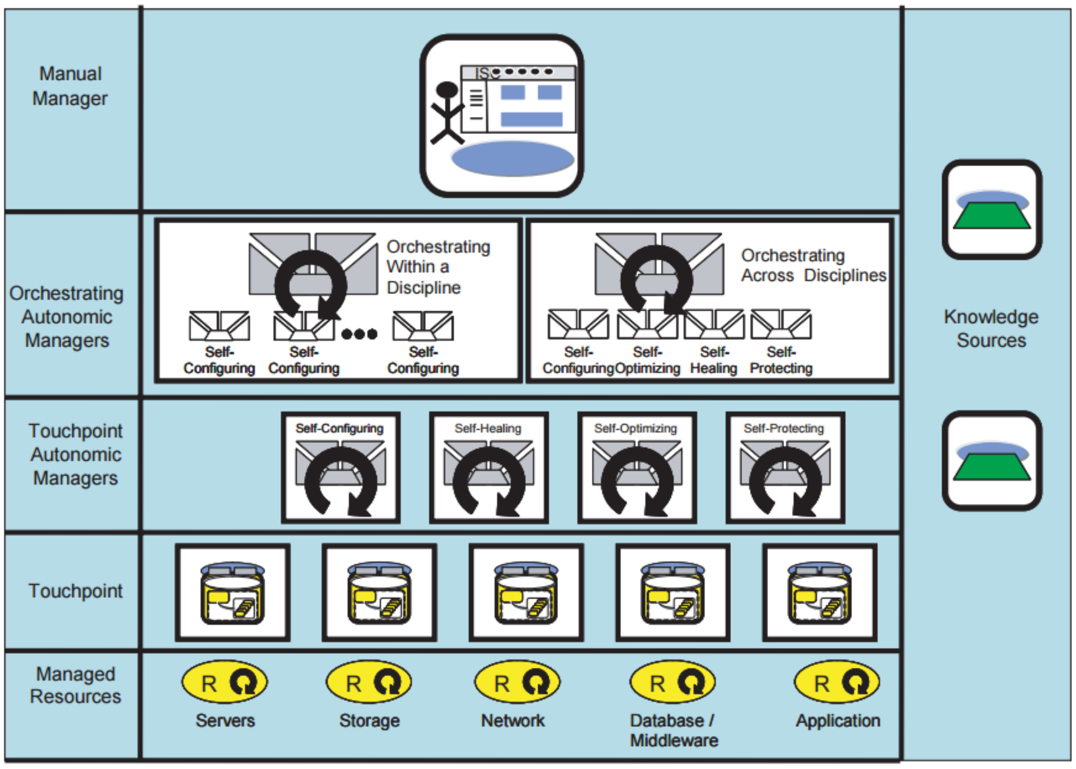


Figure 2.3 – Autonomic Computing Reference Architecture ("An architectural blueprint")

IBM's software framework for Autonomic Computing was released in 2004, and was called the "Autonomic Computing Toolkit." It was built in Java, and distributed through the IBM developerWorks website.

2.3.1 Key Component Areas

The content of the IBM Autonomic Computing Toolkit can be divided into four main categories:

- Technologies
- Tools
- Scenarios
- Information and documentation

2.3.2 Technologies

System capabilities that utilize the technologies in the Autonomic Computing Toolkit include common systems administration, problem determination, and solution installation and deployment.

Problem determination capabilities can be enhanced via the Autonomic Management Engine (AME), the Generic Log Adapter, the Log and Trace Analyzer, and Common Base Events. The Integrated Solutions Console is used to create effective common systems administration capabilities. Finally, the Solution Install technologies provide capabilities for solution configuration and deployment (Jacob 2004).

2.3.3 Tools

The Autonomic Computing Toolkit also provides the tooling needed to customize the technologies so that solutions can be created for specific needs. Tools like the Integrated Solutions Console Toolkit, Resource Model Builder, Adapter Rule Editor, and other Eclipse plug-ins are used for the creation of custom solutions.

2.3.4 Scenarios

The Autonomic Computing Toolkit also provides scenarios that show how the technologies work together, and how they may be used in realistic solutions. The scenarios are used as testing environments. All scenarios are demonstrated using the technologies and tools available in the Autonomic Computing Toolkit. There is a problem determination scenario performing self-healing, as well as two automated installation scenarios performing self-configuring tasks.

2.3.5 Information and Documentation

The Autonomic Computing Toolkit also has useful material for educating users. It provides detailed individual technology and tooling documentation, to assist with developing autonomic solutions (Jacob 2004).

2.3.6 Autonomic Computing Toolkit Technologies

The tools and technologies contained in the Autonomic Computing Toolkit are intended to help product developers create autonomic capabilities in their products. One example of an implementation of an autonomic manager is provided by the Autonomic Management Engine (AME). The AME includes built-in representations of the four parts of the control loop (monitor, analyze, plan, execute).

Developers use several technologies to create touchpoints to enable managed resources to communicate with autonomic managers. One example of these technologies is the Generic Log Adapter, which is included in the Toolkit to translate product log messages into a Common Base Event (CBE) data format. The Common Base Event is an XML structure which can be consumed by an autonomic manager (Jacob 2004).

The Log and Trace Analyzer is then used to process these messages. Afterward, it analyzes them and presents a view of the log events.

Self-configuration is enabled also, via the Solution Install components of the Toolkit, while self-management capabilities are enabled via the Integrated Solutions Console. Administrative console functions vary from run-time monitoring and control to setup and configuration (Jacob 2004).

Referring again to the MAPE-K figure (Figure 2.2), let us consider how information is passed to the autonomic manager. In the Autonomic Computing Toolkit this would be accomplished through the sensor interface. For unsolicited events, a current resource might generate its own log file. Then the Generic Log Adapter facility would be utilized to convert log entries to Common Base Events. It is also possible for an application to create its own Common Base Events directly (Jacob 2004).

The Generic Log Adapter (GLA) is one example of a facility that helps adapt a product to participate in the autonomic computing architecture. It does this by creating Common Base Events which can then be consumed by an autonomic manager. For example, using the GLA we can use a product log file to generate Common Base Event data.   First, a rule-based parser processes a log file, and then the log file is translated into the Common Base Event format. The Autonomic Computing Toolkit includes the GLA to help products adapt to the autonomic architecture without having to alter the way it creates its log files (Jacob 2004).

One single GLA runtime may be used to parse the log files of multiple products, as long as the rules have been defined for each log message format. The adapter includes a handler that can pass the Common Base Event information to the autonomic manager (using the effector and sensor interfaces) (Jacob 2004).

In conjunction with the GLA, the Adapter Rule Editor tool is used. This provides the tooling to create the specific parser rules, which are used by the GLA at runtime to create CBE objects.

The next MAPE-K component is monitoring. This component can either consume unsolicited Common Base Events, or can request specific sensor information. The building blocks for monitoring the IT environment are resource models, which are leveraged via automated best practices. These resource models contain specific metrics, thresholds, events, and parameters which are used to gauge the health of IT resources. They also contain specifications for corrective actions, if failures and error conditions come up (Jacob 2004).

The Resource Model Builder provides a standard Eclipse-based interface that provides a wizard to build resource models. This facility uses predefined resource models to specify which resource data gets accessed from the system at runtime and how this data is processed. For example, the Process resource model retrieves data related to running system processes. The resource model automatically collects performance data, and this gets processed by an algorithm to determine whether or not the system is performing up to expectations (Jacob 2004).

When a resource model is run (at a managed resource), it gathers data at regular intervals, known as cycles. A resource model with a cycle time of 100 seconds gathers information every 100 seconds, and the data collected is a snapshot of the status of the resources (which are specified in the resource model). The default cycle time of each supplied resource model can be modified as required.

Furthermore, each resource model has thresholds. A threshold is a named property of the resource (with a default value that can be modified).

The autonomic manager consists of the monitoring, analysis, plan, and execute components. The Autonomic Computing Toolkit provides an implementation called the Autonomic Management Engine. It is intended primarily for testing the components of the autonomic environment, and is a black-box implementation (Jacob 2004).

AME monitors system resources using resource models. It also sends aggregated events and performs corrective actions for problems. AME constantly monitors the system, checking for events to handle.

The Log and Trace Analyzer (LTA) may be considered a partial implementation of the autonomic manager, covering the monitor and analyze parts of the control loop. The LTA enables analysis, viewing, and correlation of log files. It therefore makes it easier to resolve problems within multi-tier systems by consuming data in the Common Base Event format and providing specialized visualization and analysis of the data (Jacob 2004).

The LTA contains a log-analysis engine, which provides an algorithm that takes an incident which is recorded in a log file as an input parameter. Then it matches this incident based on predefined rules against the symptoms of an available symptom database, and finally returns an array of objects representing the directives and solutions for the matched symptoms. The LTA provides a default implementation of an analysis engine, and a set of instruments which could be used to implement a custom analysis engine (Jacob 2004).

Rather than using the rules-based parser provided by the GLA, the LTA enables the writing of Java code-based parsers. Parsers written in Java code are usually used when the individual log messages are very complex. The Autonomic Computing Toolkit provides a number of parsers and rules for several existing IBM products (Jacob 2004).

2.3.7 Autonomic Management Engine

The Autonomic Management Engine (AME), illustrated in Figure 2.4 below, supplies a hosting environment for the resource model decision algorithms. These autonomic resource models are created using JavaScript. The core decision algorithm of the resource model is the `VisitTree()` method which is run by the AME at a given time cycle defined by the resource model descriptor, which itself is an XML file (Jacob 2004).

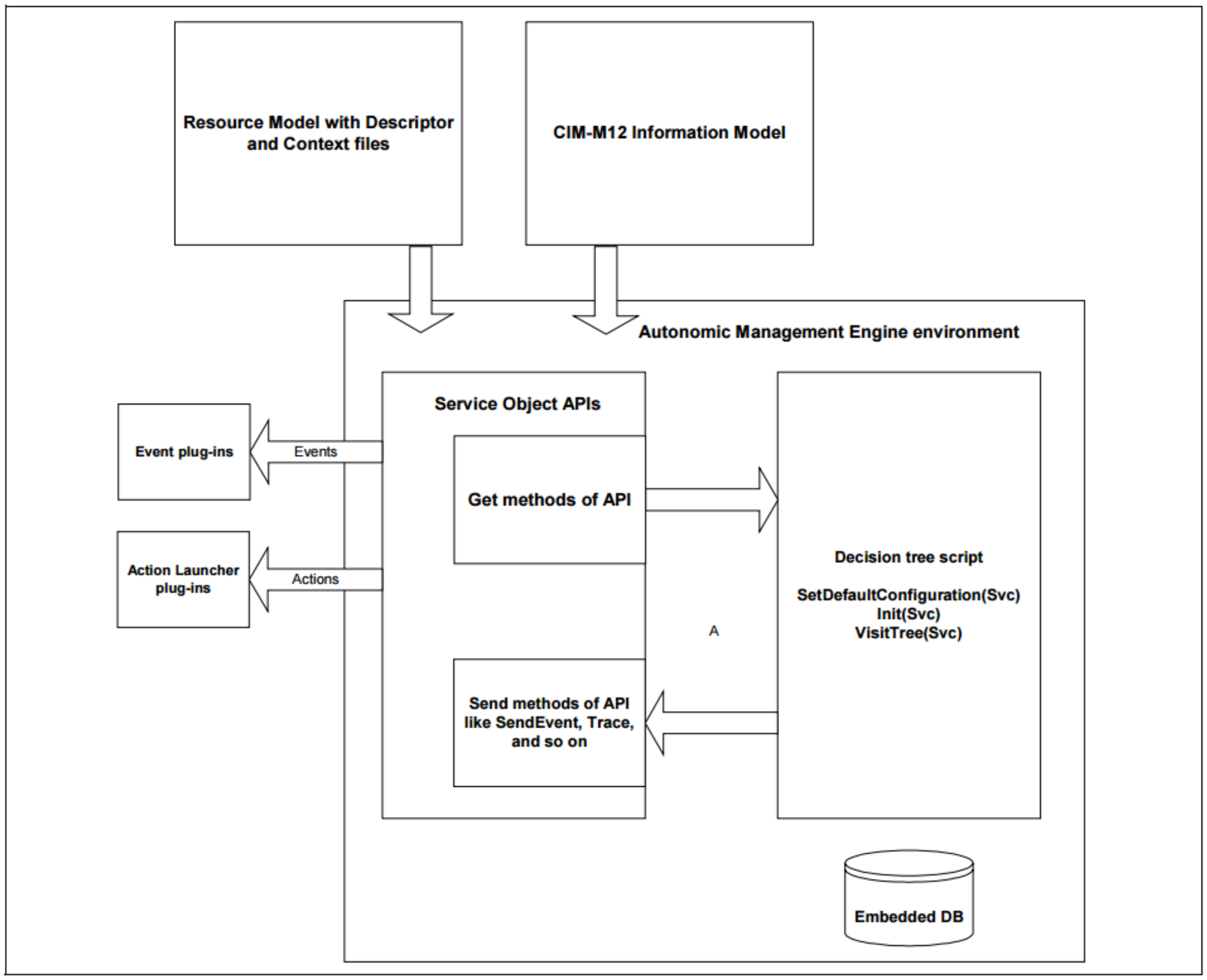

Resource Model with Descriptor and Context files
CIM-M12 Information Model
Autonomic Management Engine environment
Service Object APIs
Event plug-ins
Events
Get methods of API
Action Launcher plug-ins
Actions
Decision tree script
SetDefaultConfiguration(Svc)
Init(Svc)
VisitTree(Svc)
A
Send methods of API like SendEvent, Trace, and so on
Embedded DB

Figure 2.4 - The Autonomic Management Engine Hosting Environment (Jacob)

2.4 CASCADAS ACE

A European consortium picked up where IBM left off, implementing their own version of an autonomic computing platform. It is called the Component-ware for Autonomic Situation-aware Communications, and Dynamically Adaptable Services (CASCADAS) Autonomic Communication Elements (ACE) Toolkit, and is an open-source platform used to set up autonomic services in a distributed environment (ACE Autonomic Toolkit).

2.4.1 The ACE Theoretical Principles

2.4.1.1 Semantic Self-Organization

The Autonomic Communication Element (ACE) framework stores algorithms that support semantic self-organization of ACEs. These algorithms come in the form of autonomous clustering, synchronization, and differentiation. The commonality among all three aggregation methods is that the choice of aggregation partners is influenced by knowledge (and hence, evaluation) of the context they are operating in ("ACE Autonomic Toolkit").

2.4.1.2 Clustering

If an ACE detects a discrepancy in its list of required or available functionality (perhaps because of a change in the local environment due to a surge in demand), then it starts a “rewiring” procedure. The initiating node’s algorithm can pick one or more of the first ACE neighbors as match-makers. Also, the constraint on the conservation of the number of links may be relaxed ("ACE Autonomic Toolkit").

2.4.1.3 Differentiation

The purpose of differentiation is to allow ACEs to decide whether to terminate themselves (locally) under an inappropriate workload. Thus, resources can be transferred between applications in a way such that released resources can be re-assigned to another application. The term differentiation is derived from morphogenesis

– a biological phenomenon which means “the development of form and structure in an organism during its growth from embryo to adult'' (“morphogenesis”).

2.4.1.4 Synchronization

The purpose of synchronization is to find or create partnerships based on the time activity patterns of the constituent ACEs. The synchronization requires two main prerequisites:

1. Establishing a collaborative overlay which combines components that feature activity patterns that are compatible *a priori* starting from a random bootstrap configuration
2. Finding ways of adjusting individual time-cycles in order to create opportunities for collaborating (which would not exist if every individual activity pattern was set from the beginning).

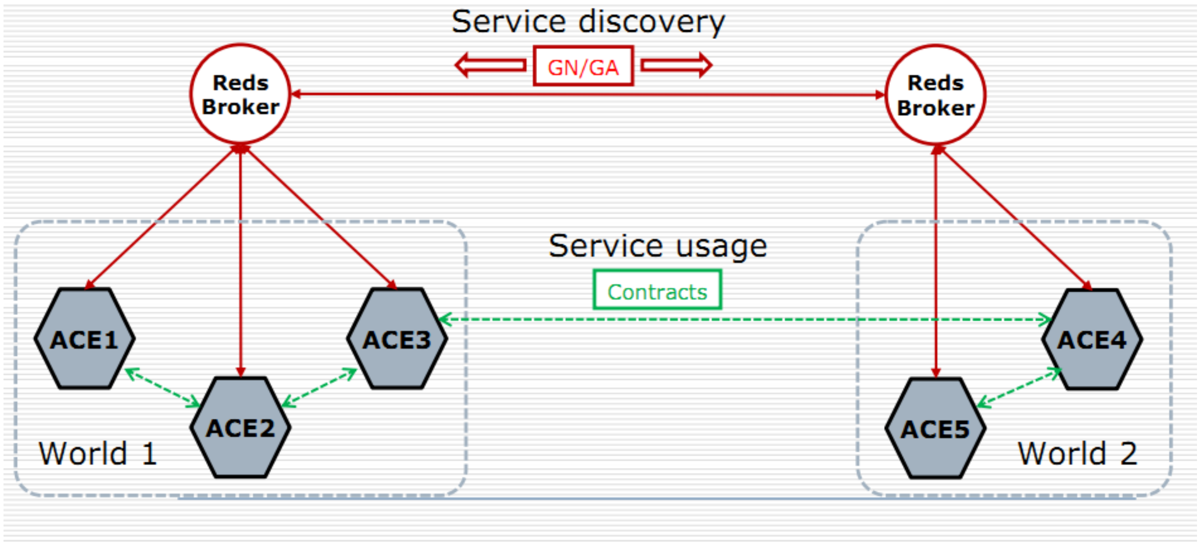


Figure 2.5 - Two types of Inter-ACE communication: Connection-less (Service-discovery) and Connection-oriented(Service Usage) ("ACE Autonomic Toolkit")

2.4.1.5 Situation Awareness

Situation-awareness denotes the ability of refining decisions according to the specific contextual situation, and this requires tools and models for analyzing and organizing pieces of information. There, the ACE Framework defines a Knowledge Network (KN) service, which is accessible (through aggregation) by ACEs as a system-wide service. Knowledge Networks process information in order to make a collection of Knowledge Atoms (KAs). These KAs structure the processed information into a data model which is built on the basis that any amount of contextual knowledge is created as a result of an event occurring. Furthermore, a dedicated data model is made to represent any fact as a 4-tuple of the form `(who, what, where, when)`.

2.4.1.6 Pervasive Supervision

To detect and correct hazard situations, an ACE needs a supervision system. This is an ongoing system with a *Sensor* that links the supervision system with the ACE. Every ACE uses the dedicated supervision organ to export a management interface through which internal state and session objects can be obtained.

Whenever a new event arrives, the Executor adds the input of each PEX to it. Processing speeds of individual PEXs may differ, but the whole system is unaffected because PEXs are independent (run on separate threads and have separate input queues).

It is important to note that there are two PEX types: a “Normal” PEX, which at start-up gets a new input queue and a new Execution Session; and a “Child” PEX, which at start-up will inherit the Execution Session of its parent PEX, and also inherits the content in the parent’s input queue. Child Plans may be used for starting separate processes/Plans for each client. Parent and Child share the Execution Session and so they can synchronize through reading/writing into it.

2.4.1.7 Functionality Repository

The Functionality Repository is the ACE organ which takes care of deploying and managing functionalities (common and specific functionalities) within the ACE. It also provides features for calling these functionalities using Events.

The main purpose of the Functionality Repository is to enable common and specific functionalities to be deployed into the ACE instance. Once deployed into the ACE they can be accessed through ACE events. Thus the Functionality Repository has two sides: as a storage facility and as a provider of an accessible interface.

As a storage facility, the Functionality Repository keeps track of the deployed functionalities, creates and stores instances of the underlying classes, and maintains call-related variables.

As an accessible interface, it listens to specific ACE events and then interprets them as calls to the functionalities. The ACE Functionality Repository also follows the Event-based access model as the ACE organs. Therefore, the input of each invocation is an internal ACE event (called `FunctionalityCallEvents`) that is produced by the Executor when the execution of a transition begins (and is sent directly or through the Bus). Also, the output of a call is an arbitrary set of events (that may be fixed or varying) created by the functionality. The Functionality Repository transmits these events to the Bus or Gateway, depending on the event type.

2.5 Autonomic Computing in Practice

The autonomic computing principles have been applied to many enterprise software systems. A number of software utility packages are available in the market today.

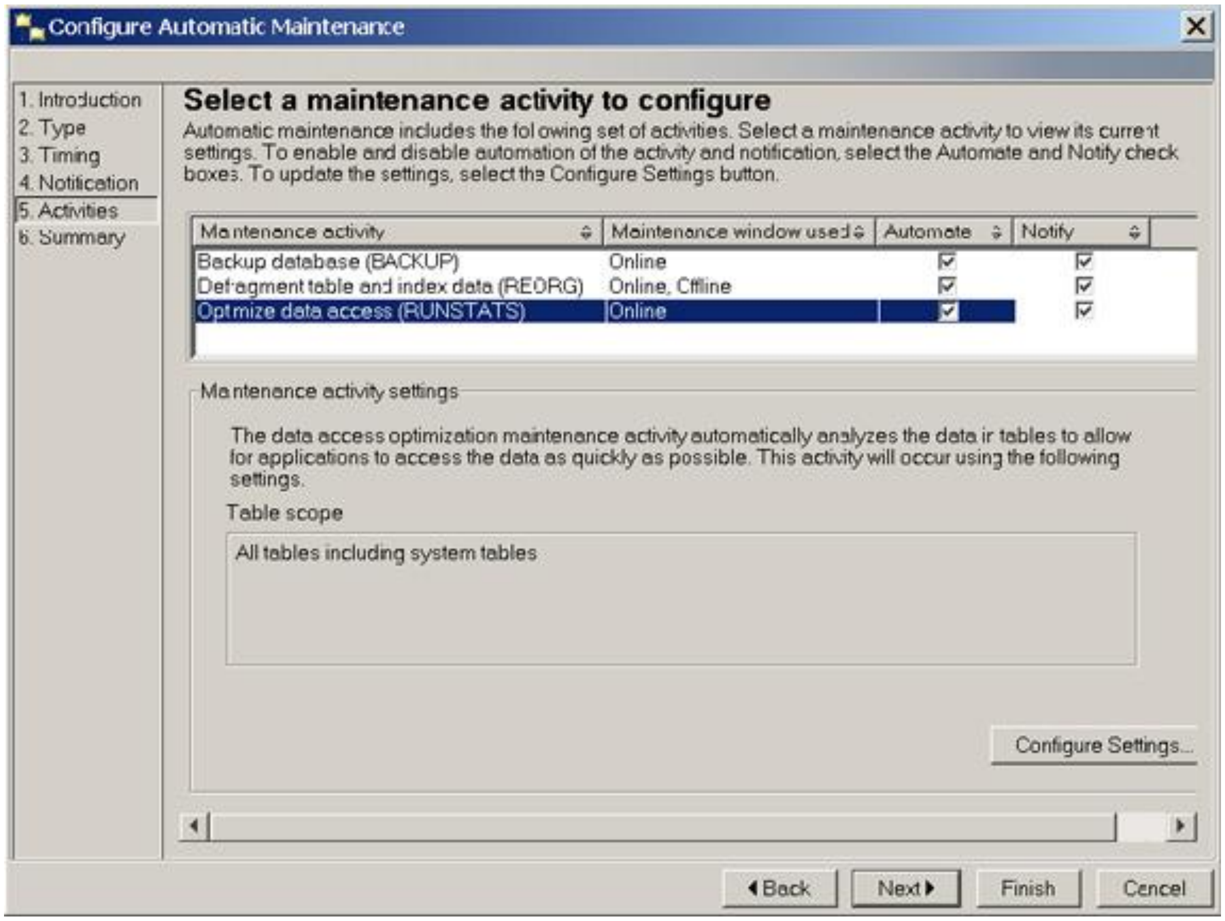
2.5.1 DB2

IBM is the main user of Autonomic Computing principles. This is exemplified in the enterprise database suite, DB2. DB2 version 9 has a number of autonomic features that are designed to make database administration almost effortless, such as:

- self-tuning memory management
- auto-configuration enabled by default
- automatic database maintenance
- automatic storage management

Figure 2.6 below shows a screenshot of the automatic maintenance feature: [1]

[1] http://www.ibm.com/developerworks/data/library/techarticle/dm-0606ahuja2/



Select a maintenance activity to configure

Automatic maintenance includes the following set of activities. Select a maintenance activity to view its current settings. To enable and disable automation of the activity and notification, select the Automate and Notify check boxes. To update the settings, select the Configure Settings button.

Maintenance activity	Maintenance window used	Automate	Notify
Backup database (BACKUP)	Online	✓	✓
Defragment table and index data (REORG)	Online, Offline	✓	✓
Optimize data access (RUNSTATS)	Online	✓	✓

Maintenance activity settings

The data access optimization maintenance activity automatically analyzes the data in tables to allow for applications to access the data as quickly as possible. This activity will occur using the following settings.

Table scope

All tables including system tables

Figure 2.6 –Configuring Automatic Maintenance in DB2 (screenshot)

DB2 9 utilizes a feature called self tuning memory manager (STMM), which is shown below in Figure 2.7. This feature eases the task of memory configuration (often a burden on database administrators). It does this by automatically setting optimal values for most of the memory configuration parameters, including package cache, buffer pools, locking memory, sort heap, and the total database shared memory.[2]

[2] http://www.ibm.com/developerworks/data/library/techarticle/dm-0709saraswatipura/

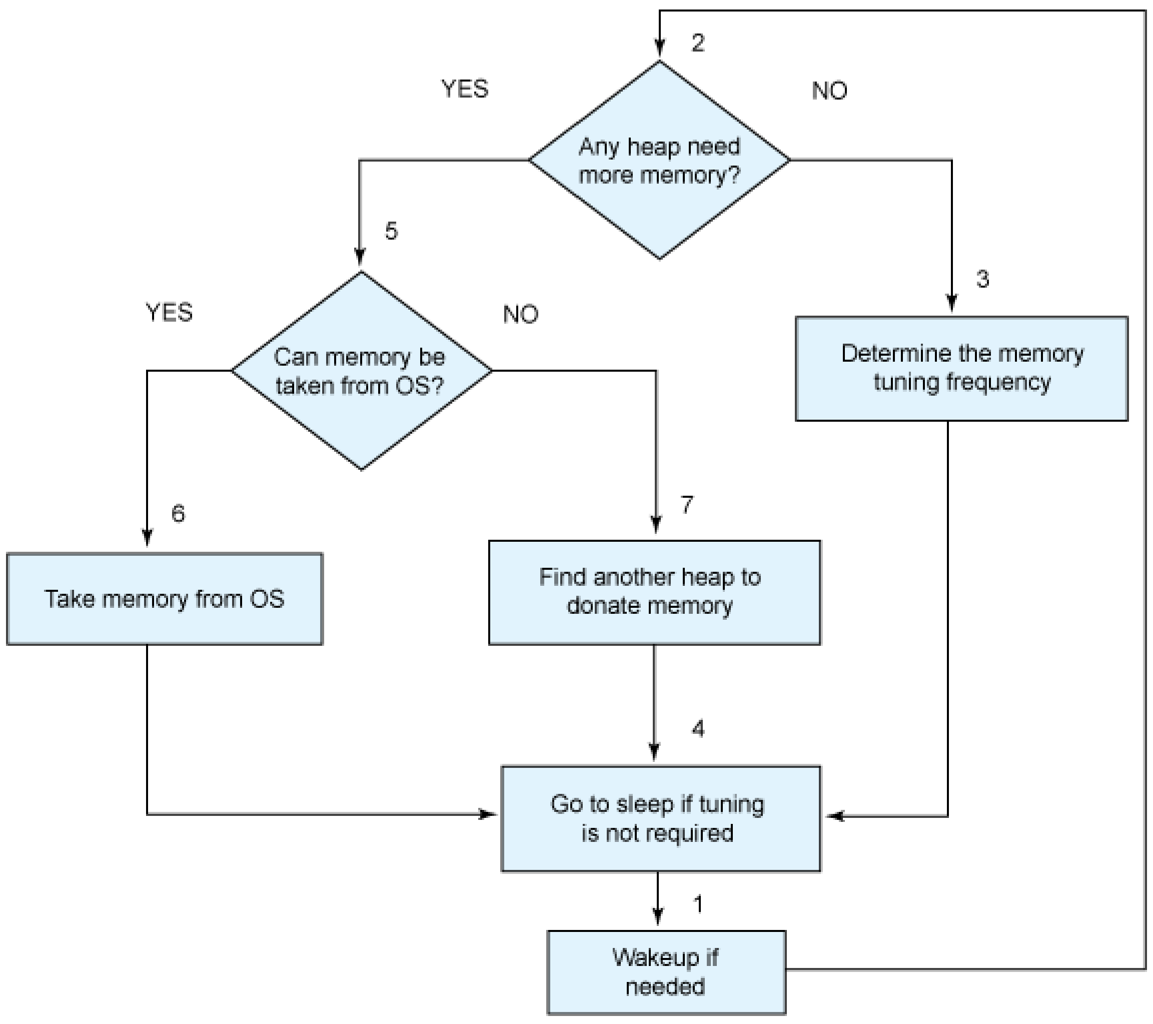

2
YES
NO
Any heap need more memory?
5
3
YES
NO
Can memory be taken from OS?
Determine the memory tuning frequency
6
7
Take memory from OS
Find another heap to donate memory
4
Go to sleep if tuning is not required
1
Wakeup if needed

Figure 2.7 - The Self-Tuning Memory Manager (Saraswatipura)

2.5.2 Oracle

In Oracle SQL versions 11g and up, there are a number of tools that can aid in implementing an autonomic system. One example is using REDO logs, which are binary files that store change vectors. The changes are written to the redo logs in a circular fashion. Most enterprises operate in a mode called ARCHIVELOGMODE. This means that as the instance switches from one redo log to another, the previous log is written out to an archive log file.

The following is an example of how we can use Oracle's memory management functions:

```
ALTER SYSTEM SET log_archive_dest_1='LOCATION=[/LOCATION]'
SCOPE=both;
```

If we want to simulate a series of log switches, we may use the following command several times:

```
ALTER SYSTEM SWITCH LOGFILE;
```

In general, Oracle Database provides a built-in infrastructure for defining maintenance activities. There is powerful scheduling functionality introduced with Oracle Database 11g Schedule to run tasks such as the following:

- gathering optimizer statistics in order to update the system catalog with information on the data in indexes and tables
- creating statistical profiles for optimal statistics collection (based on table analysis and query feedback)
- Performing database backups
- Reorganizing fragmented tables

By default the maintenance window begins at 10:00 PM every night, and lasts until 6:00 AM the next morning and throughout the weekend. All the maintenance window attributes are customizable (start/end time, frequency, days of the week, etc).

This allows it to adapt to specific scheduling needs. The tasks of automated optimizer statistics collections, automatic SQL tuning, and reorganizing fragmented tables are all built into Oracle Database 11g and run periodically.

Oracle - Resource Control

One powerful aspect of running a task in a maintenance window is the ability to control the system resources and limit resource use in favor of more business-critical activities. By using bandwidth quotas, the Oracle Resource Manager optimizes resource utilization globally and allows maintenance tasks to use the available resources without affecting higher priority activities.

Oracle - Space Reorganization

Tables in a database become fragmented, and therefore reorganization tasks are run to increase object access performance and optimize space usage. Oracle 11g is capable of scheduling reorganization jobs online. Also, it provides a set of options like moving tables, adding/dropping partitions, changing structures, etc.

Oracle - Automation of SQL Tuning

Oracle provides a SQL Tuning Advisor, which analyzes problematic SQL statements and makes specific recommendations to tune it. The core of this technology is the Automatic Tuning Optimizer (ATO). ATO is an improved version of Oracle's query optimizer that runs the actual analyses on behalf of the SQL Tuning Advisor. It uses dynamic sampling and partial execution methods to verify its own estimates of selectivity, cost, and cardinality. It also uses the past execution history of the SQL statement in order to determine optimal settings for the optimizer. Recommendations generated by the ATO include creation of new indexes, updating of optimizer statistics, optimizing of the SQL statement design, and creation of a SQL Profile (a database object in Oracle 11g offering a way to tune SQL statements). Then it collects information on predicate selectivity, skews, and data correlations for the specific SQL statement. These are then used by the query optimizer to produce an ideal execution plan (“Oracle" 10).

Table 2 - Oracle Database’s AC Abilities

Self-Configuring	**Self-Healing**	**Self-Optimizing**	**Self-Protecting**
Resource Control Space Reorganization	n/a	ATO	n/a

2.5.3 Monit

Monit is a utility available on UNIX systems, which is used to manage and monitor the operating system. It is an open source program.

Monit can exhibit proactive capabilities. For example, if `sendmail` is not running, Monit can restart sendmail automatically. Also, if Apache is using excessive resources then Monit can stop/restart Apache, and send the user an alert. Monit also can monitor specific process characteristics (i.e. how much memory a process is using, CPU usage, and Load Average).[3]

[3] http://mmonit.com/monit/

Monit is also used to monitor files, directories, and file systems on `localhost`. This is useful for checking issues such as timestamp changes, size changes, or checksum changes. Security issues can also be monitored – you can check that md5 or sha1 checksum of files do not change.

Below in Figure 2.8 is a graph generated by Monit, which shows CPU and memory usage, along with other information :

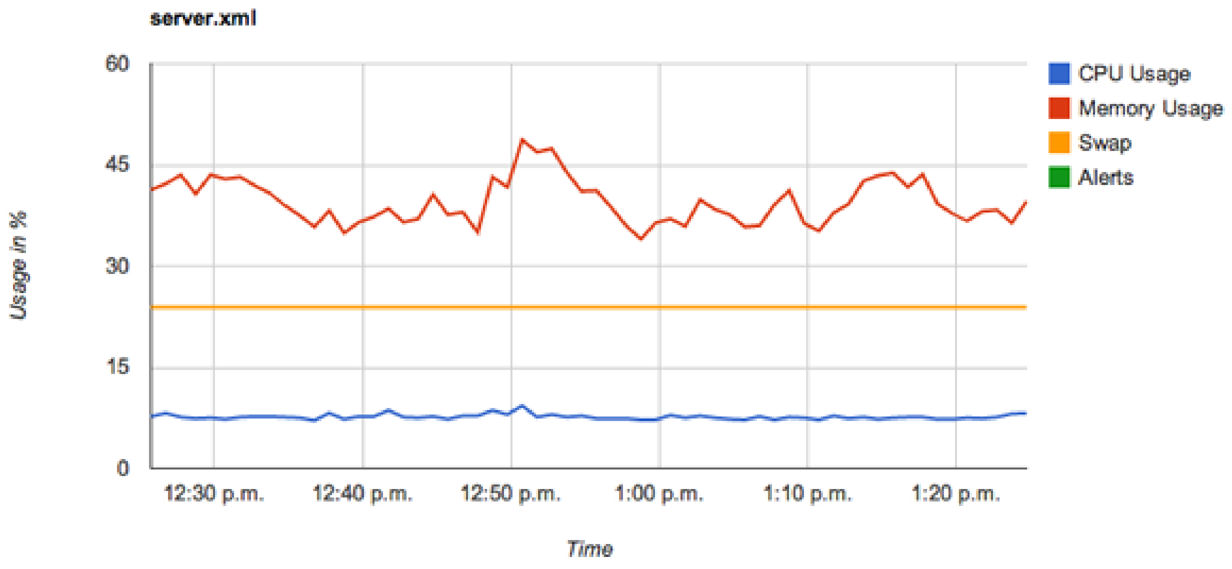


Figure 2.8 - Monit ("Danschultzer/monit-graph")

Table 3 - Monit's AC Abilities

Self-Configuring	**Self-Healing**	**Self-Optimizing**	**Self-Protecting**
Applicable	n/a	Applicable	n/a

2.5.4 MapReduce

MapReduce has gained huge popularity, mainly due to its adoption by Google.[4] MapReduce borrows from the autonomic computing genre in that it utilizes distributed computing. MapReduce is used to process and generate large data sets, via a distributed and parallel algorithm on a cluster.[5]

MapReduce is useful in a large range of applications, including distributed sorting, distributed pattern-based searching, web link-graph reversal, Singular Value Decomposition, web access log stats, inverted index construction, machine learning, document clustering, and statistical machine translation. Furthermore, the MapReduce model has been ported to several computing environments such as multi-core and many-core systems, desktop grids, volunteer computing environments, mobile environments, and dynamic cloud environments.

Google, for example, used MapReduce to completely regenerate its index of the Internet. This replaced old ad hoc programs that updated the index and ran analyses.

A distributed file system is used to store stable inputs and outputs for MapReduce. The transient data is stored on local disk, and fetched by the reducers.

[4] http://research.google.com/archive/mapreduce.html
[5] http://en.wikipedia.org/wiki/MapReduce

In Figure 2.9 below is a high-level diagram of MapReduce, which is an algorithm used to divide a computation among many computers for parallel processing:

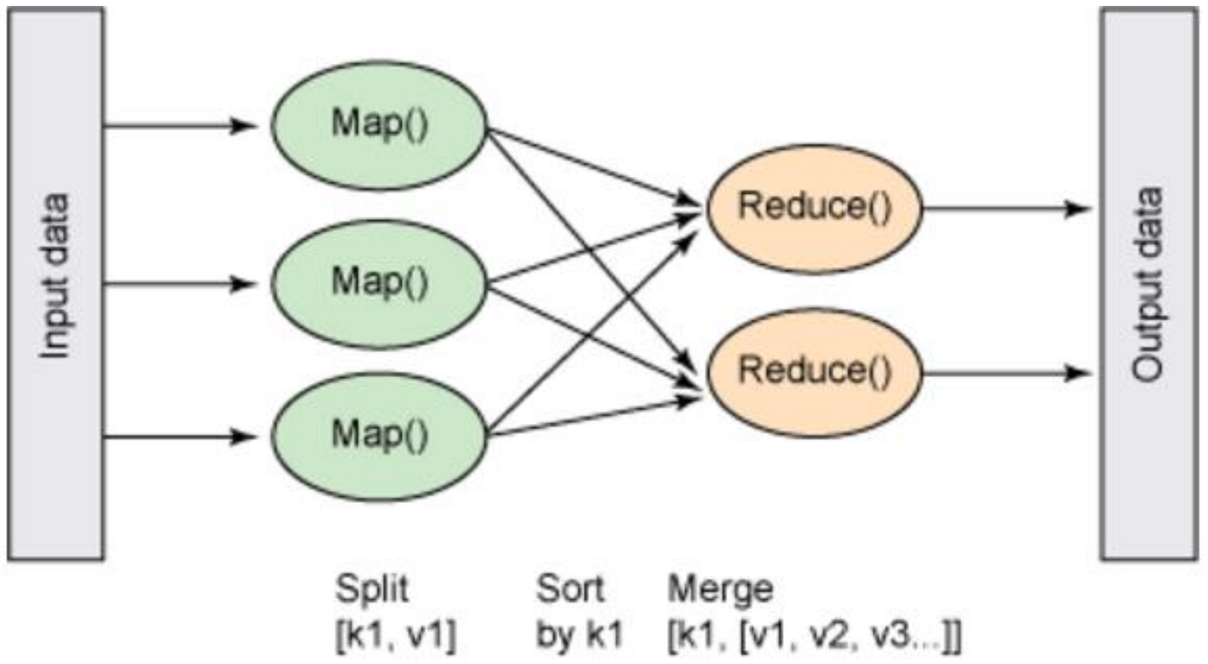


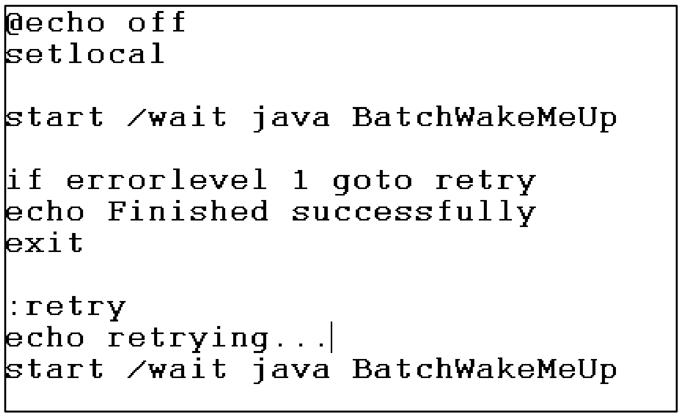
Figure 2.9 - MapReduce ("Deploy")

Table 4 - MapReduce AC Abilities

Self-Configuring	**Self-Healing**	**Self-Optimizing**	**Self-Protecting**
Applicable	n/a	Applicable	n/a

2.7.5 Watchdog Scripts

One way that programmers often implement failsafe abilities is via "watchdog scripts." A watchdog script is simply a system script that is triggered after a specific action on the program being monitored. The following is a simple example of a Windows watchdog script, which utilizes a Windows batch file:



```
@echo off
setlocal

start /wait java BatchWakeMeUp

if errorlevel 1 goto retry
echo Finished successfully
exit

:retry
echo retrying...
start /wait java BatchWakeMeUp
```

Figure 2.10 – Watchdog Script (Windows Batch File)

A watchdog script is especially useful for system administrators, who often need to keep track of runaway processes. These scripts are popular in Unix and Linux system administration.

2.7.6 Apache Commons Daemon

One way autonomic computing behavior can be implemented using Java is to use an Apache utility called Daemon.[6]

Apache Daemon has two parts: a native library written in C which interfaces with the operating system, and the library that provides the Daemon API (written in Java). [7]

[6] http://stackoverflow.com/a/30020831/763029

Depending on which operating system we are using, an application can be run either as a service (Windows) or a daemon (Linux). On Windows, we may use ProcRun.exe to run the Java application as a service. And on Unix, we may use JSvc for running the Java application.

Table 5 - Apache Common's AC Abilities

Self-Configuring	**Self-Healing**	**Self-Optimizing**	**Self-Protecting**
n/a	Applicable through ProcRun.exe on Windows, or JSvc on Unix	n/a	n/a

2.7.7 daemontools

Specific to the UNIX operating system, there is a very useful utility called daemontools. This is a collection of tools for managing UNIX services, and among them there is a program called `supervise.` `supervise` starts and monitors a service. It is run as follows:

```
supervise  s
```

When run, it switches to the directory named `s` and starts `./run`. It restarts `./run` if `./run` exists, and also maintains status information (in a binary format) inside the directory `s/supervise` (which must be writable to supervise). This status

[7] http://en.wikipedia.org/wiki/Commons_Daemon

information can be read by `svstat`. To check whether supervise is running successfully, one may use the `svok` tool.[8]

2.8 Code Adaptation

At the heart of autonomic computing is the concept of software adaptation. Self-management cannot be achieved without the ability to modify the behavior and structure of a system. Software adaptation requires changing low-level code, which is often intricate and complicated. Side effects could be unforeseen. Since the code that is to be modified has been run, the computation state has to be preserved. Software adaptation can be achieved either at the operating system levels or at the specific programming levels (Lalanda 2014).

2.8.1 Operating System-Level Adaptation

Figure 2.11 below demonstrates an OS-Level Autonomic Manager:

[8] http://cr.yp.to/daemontools/supervise.html

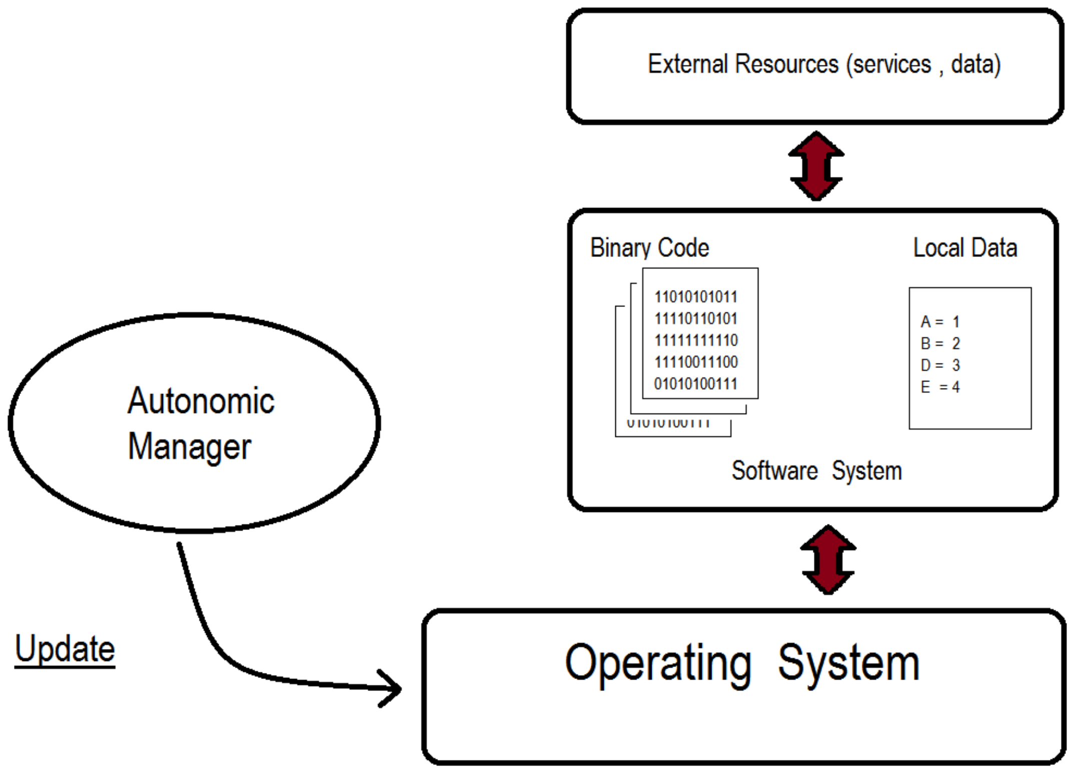


Figure 2.11 – OS-Level Autonomic Manager (Parashar)

Dynamic adaptation at the operating system level is not intrusive because the internals of the programs are not changed. They change the resources and services provided by the operating system.

Much research has been conducted in order to allow the dynamic integration of services and resources. For example, most operating systems are capable of integrating new resources in real-time without interruption of services. This is the case of the Universal Plug and Play (UPnP) standard, originally developed by Microsoft (Lalanda 2014).

Operating systems allow the dynamic deployment of new services. Deployment includes activities such as installation, activation, and deactivation. On Linux and many Unix systems, in order to install a new shell command one simply copies the executable (binary file/script) into the ‘/bin’ directory. The service is "launched" afterward by typing its name, and it then becomes available to the OS computing environment. In a sense, operating systems have an infrastructure that appears autonomic, in order to dynamically adapt software systems. But new code is not finely integrated into the existing code. Rather, it is packaged and deployed as a stand-alone service in the OS file. Some running code can then call this new service (Lalanda 2014).

In spite of the aforementioned ability to dynamically integrated new code, building a dynamic application on top of an OS is very complex. It is often based on ad hoc mechanisms set up by the application itself. The example of the shell command that we mentioned above is based on the introspection of directories specified in a global environment variable. Also, it is difficult to create an infrastructure for the interception and redirection of messages that are exchanged between two internal structures of the application code. Thus, OS-based techniques remain impractical due to excessive complexity (Lalanda 2014).

2.8.2 Application Program-Level Adaptation

Dynamic adaptation is at the core of many programming languages, and a number of techniques have been applied to allow code evolution at runtime. Dynamic linking is one of them. In the Smalltalk programming language, code is dynamically typed and reflective. In JavaScript programming variables are weakly typed and dynamic. In the Erlang language developers can dynamically load new code and explicitly manage code replacement (Lalanda 2014).

These are powerful techniques which can be used by autonomic managers to bring about dynamic code adaptation. However, these techniques are very hard to master and control. Excessive use of programming-level techniques results in buggy code, which is very hard to test and maintain. Programming-level techniques have a strong impact on the code itself. For instance, in order to dynamically load C libraries, extension points (variation points) need to be introduced. This results in complex code. Variation points cannot be introduced everywhere in the code because it quickly becomes unmaintainable (Lalanda 2014).

The C modules that can be linked dynamically are packaged into specific libraries. The implementation of these libraries depends on the OS. These specific libraries are called shared libraries (`.so`) in Linux and dynamic-link libraries (`.dll`) in Windows. For example, the following code shows how to load a library with the `dlopen()` function, how to get a shared library symbol address, and finally how to unload the shared library with the `dlclose()` function:

```
void *handle;
int  *iptr, (*fptr)(int);

/* loading expected library */
handle = dlopen("/usr/home/me/mylib.so", RTLD_LOCAL | RTLD_LAZY);

/* Getting the address << my_function >> and << my_data >> */
*(void **)(&fptr) = dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_data");
/* call to << my_function >> with << my_data >> as param */
(*fptr)(*iptr);
```

Dynamism in C is unwieldy and complex. Furthermore, C has little supporting mechanisms that limit programming errors. Dynamic libraries get loaded with no verification about the correctness of the new code, such as a typing system. Issues like dangling pointers can arise freely, as there is no preventive verification when a module is unloaded (Lalanda 2014).

2.8.3 Program-Level Adaptation in Light of the Java Language

In Java, source code is always transformed into byte code to be executed by a virtual machine. A virtual machine is an abstraction layer that isolates applications from computer specifics (operating system, physical hardware architecture). This abstraction layer can minimize the effort involved in porting software between different systems (Lalanda 2014).

In Java there is no static linking. Loading a class is performed on demand when the class is needed for execution. The action of loading classes in a virtual machine at runtime is done by a specific entity called a class loader. The job of the class loader is to resolve external references. To do this it needs to locate libraries that contain the appropriate classes in the system resources, and then load them into the virtual machine. Many class loaders can be used in the same virtual machine, and their use is based on the following rules:

- Every class loader (except the initial one, called a bootstrap) has a parent
- Each class loader delegates the task of class loading to its parent before doing so itself

By default, a Java virtual machine has three hierarchical class loaders:

- The initial class loader whose job is to load standard Java classes (`rt.jar`)
- The extension class loader which loads classes of the extension directory (`jre/lib/ext`)
- The application class loader, which loads archives defined by the CLASSPATH

More class loaders may be added to load specific aspects in a systematic way, and each class loader will then have its own name scope. This allows the loading of two implementations of the same class as long as they are loaded by two different class loaders. Therefore, two versions of a class may be used by different parts of a system, which provides flexibility. In addition, backtracking to an earlier state is made possible. However, the class loader concept is not easy for programmers to master. This results in buggy situations where unexpected classes are used in a program (Lalanda 2014).

Unlike the C approach, verification is done before loading a Java library. Type system compatibility is checked. The following example code shows how to dynamically load a Java class:

```
Class type = ClassLoader.getSystemClassLoader().loadClass(name);
Class type = this.getClass().getClassLoader().loadClass(name);
Object obj = type.newInstance();
```

Dynamic code loading is an essential feature that allows the introduction of new code in the scope (namespace) of some code already running without interruption. But the ability to integrate new code is not well supported. Unloading code is necessary when one wants to replace a class. However, a class loader cannot unload a class. Unloading a class needs unloading the class loader itself.

In this case, it is necessary to be able to deploy, load, and instantiate the new structure. Finally, the structure to be replaced is then required to vanish. It has been unloaded from the virtual memory, with clients of the old structure being rerouted into the new one (Lalanda 2014).

2.9 Autonomic Computing and its Relation to Service-Oriented Architecture (SOA)

Service-Oriented Architecture provides an architectural framework where software components communicate with other components using an agreed-upon protocol. It provides an easy way to build up complex applications from many different sources. SOA may also be used to implement AC features.

2.9.1 Service-Oriented Architecture

Many data centers, including Google and eBay, organize their systems using Service Oriented Architecture (SOA). A collection of servers, applications, and data at a site is called a *farm*, while a collection of farms is termed a *geoplex.*

A service is either cloned or partitioned. Cloning means that the data is copied onto a collection of nodes. Each node can provide its own storage (inefficient if many nodes) or it may use a shared disk or disk array. The collection of clones is called a Reliable Array of Cloned Services, or RACS. We may then partition the data among a collection of nodes. These partitions may be replicated onto a few new nodes, which then form a pack. The set of nodes that provide a packed-partitioned service is called a Reliable Array of Partitioned Services (or RAPS).

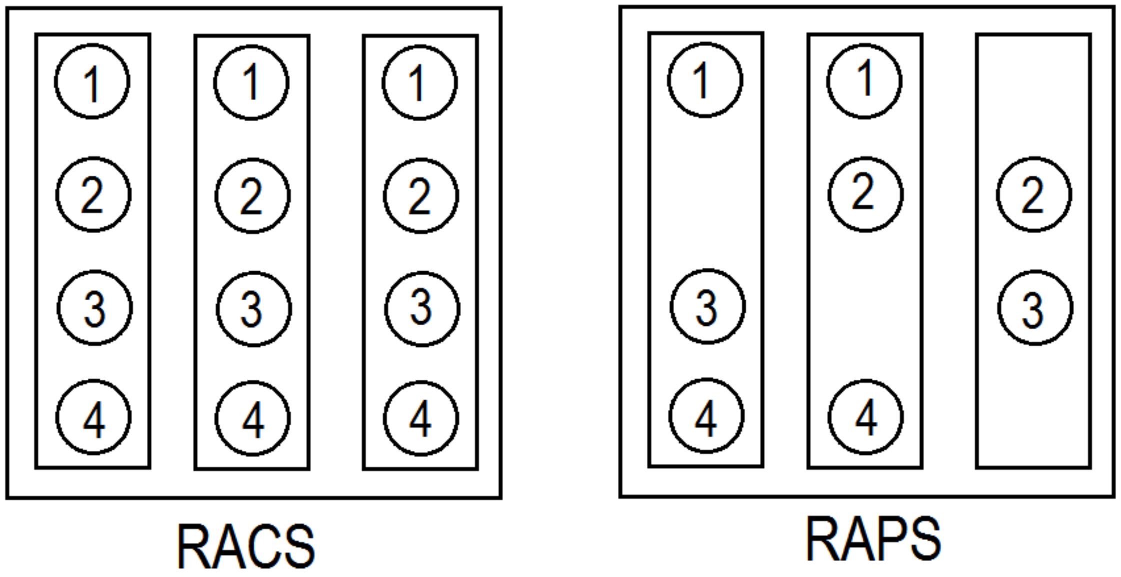


Figure 2.12 - RACS and RAPS architectures (Parashar)

A RAPS is functionally superior to a RACS, but a RACS is easier to build and maintain. Therefore, many data centers try to maximize the use of the RACS design. RACS are good for read-only, stateless services (often found at the front end of a data center), while RAPS are better for update-heavy states (as found in storage back-end). Figure 2.12 above illustrates both architectures.

Most data centers apply SOA principles. All these systems have developed in an organic way, beginning with a few machines in a single room. The configuration and management of these systems is a daunting task, and problems that result in black-outs need to be resolved within seconds. Thus, automation is a highly desired quality (van Renesse 2007).

In modern SOA technology, each service has its own management console. However, the deployed services depend on one another. If one service is problematic, it may be due to a failure in another service. Therefore, obtaining a global view and quickly finding the source of the problem is necessary. When each service supports for self-management and self-configuration, a global diagnosis could be possible (van Renesse 2007).

A Scalable Monitoring and Control Infrastructure (SMCI) can be used to solve these problems. It can monitor arbitrary sensor data available in the system, and has a global presence. Following this, the SMCI can give a global view by installing an appropriate aggregation query. Not only is this useful to a system administrator, but such data can also be fed back into the system to automate the control of resource allocation or even drive actuators (rendering the system self-managing and self-configuring) (van Renesse 2007).

2.9.2 Autonomic Multimodal Interaction Model Utilizing Service-Oriented Architecture in a Pervasive Environment

There is a wealth of research on the SOA-approach to autonomic computing for pervasive systems. One approach is described by Avouac et al (Avouac 2011).

Pervasive computing systems typically consist of multiple devices and software entities that may interact with each other. Many types of devices can be made available for different purposes – interacting with the real environment, providing display and control services to users, or exposing data and application interfaces to other devices.

The challenge of pervasive computing lies in providing coherent pervasive environments. These environments offer useful applications and services in a heterogeneous setting with varied and distributed dynamic devices and software services, that communicate via various protocols (Avouac 2011).

Advances in SOA have improved the outlook of pervasive computing. Many smart devices today are exposed as services, and their capabilities are described and dynamically published by service providers, and are chosen and called by service consumers at runtime (Avouac 2011).

SOA allows ample flexibility in operations and functions which results in powerful solutions that are hard to manage. To meet the challenges in management a “Dynamic Multimodality” software framework is proposed. In this framework the whole multimodal processing system is generated and maintained at runtime by an autonomous manager. The processing system is modular and is made of service-oriented components. This framework is shown below in Figure 2.13:

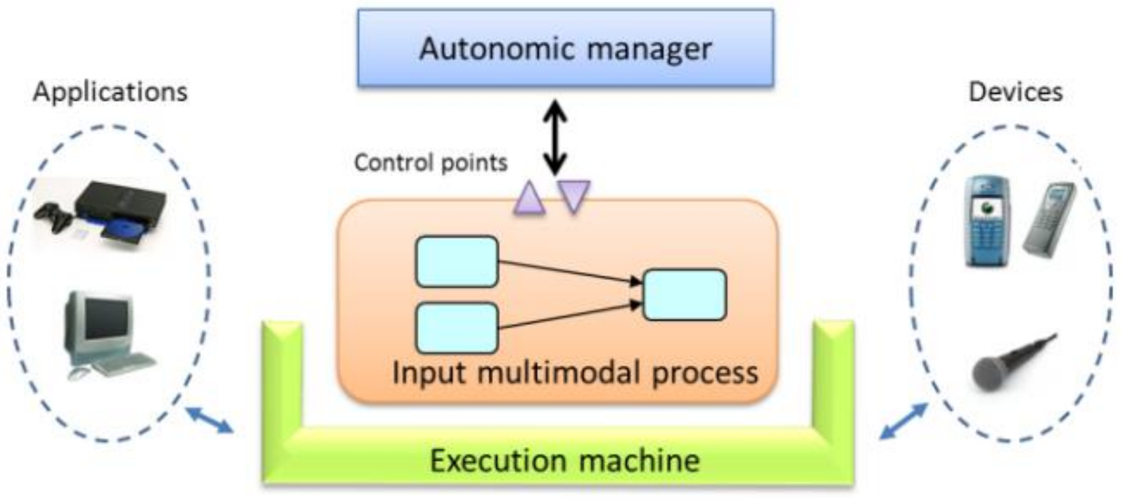


Figure 2.13 – The Multimodal Model (Avouac 2011)

The purpose is to clearly separate the multimodal processes, the input interaction devices (which are volatile and may leave/join the environment), and the applications that may appear and disappear dynamically (Avouac 2011).

2.9.3 Service Configuration and Composition Design Pattern for Autonomic Computing Systems

Mannava and Ramesh (Mannava and Ramesh, 2012 B) proposed a software architecture, which uses a service configuration and composition design pattern for the dynamic configuration and composition of communication services. It does this by satisfying the self-configuration and self-composition characteristics of autonomic computing systems, which software designers and/or programmers can utilize to drive their work (Mannava 2012).

The aforementioned architecture is divided into four modules: the monitoring module, the decision-making module, and the Self-Optimization and Self-Configuration modules (Mannava 2012).

There are a number of design patterns used in this architecture. The following is a list of these design patterns :

- **Observer**: A One-to-Many dependency between objects, so that when an object changes state, all of its dependents will be notified
- **Cased based reasoning**: this design pattern separates the decision-making logic from the functional logic

- **Row Data Gateway:** Borrowing from database theory, enfold the data structures and their database access code within row data gateways whose internal structure models a database record (which offers a representation-independent data access interface to clients)
- **Adaptation Detector:** Interpret monitoring data and decide when an adaptation is required
- **Event Notifier:** Enable components to react to the occurrence of particular events in other components without knowledge of each other (while allowing dynamic participation of components and introduction of new events)
- **Strategy**: Family of encapsulated algorithms that can be interchanged. Strategy allows the algorithm to vary based on the client
- **Master-Slave**: used when implementing an encapsulated implementation. We may need to provide fault tolerance, increased performance, or result accuracy for a component implementation
- **Thread per connection**: a single thread is created for each process. This design pattern is usually used for the client-server model
- **Server reconfiguration**: This pattern is used to reconfigure an application structured as a server-client architecture. Components can be removed from the server architecture through the Component Removal Pattern.

In real software systems, pattern composition has been known to be a challenge to applying design patterns. The authors, Mannava et al, utilized a number of design patterns including web services, which is defined based on three roles: service provider, service registry, and service requester (Mannava 2012).

Chapter 3. Applied Technology and Techniques

In this chapter, we will explore technologies and techniques that are used in the quest of making systems demonstrate aspects of autonomic computing functionality. We will look firstly at Java Management Extentions (JMX), which is a powerful means of enabling autonomic functionality. Afterward, we will delve into the topic of checkpoints. Checkpoints are used, upon system failure, to bring a system back to a reasonable state whereby the user can continue working without undue inconvenience. Checkpoints are a key feature of almost all complex systems (operating systems, major office applications, database management systems, etc).

3.1 Java Management extensions (JMX)

Java Management Extensions was added to Java in the Java 2 Platform, Standard Edition (J2SE) 5.0 release. JMX technology is a standardized way to manage resources dynamically (devices, services, and/or applications) using Java. JMX can monitor and manage the Java Virtual Machine (JVM). Furthermore, JMX is extensively used in Java Enterprise Edition application servers, along with other middleware, for administrative purposes.

In order to use the JMX technology, resources are paired with Java objects known as Managed Beans (`MBeans`). These `MBeans` are registered in a JMX server known as an `MBean` server, that behaves as a management agent defined by the JMX specifications. An `MBean` server and a set of services for handling the `MBeans` make up a JMX agent. Following this pattern, properly configured JMX agents control resources and then make them available to remote management apps ("Instrumenting").

JMX technology defines standard connectors (known as JMX connectors) with which a management application can manage resources transparently, regardless of communication protocol used ("Lesson: Overview"). Figure 3.1 below shows a sampling of some of the more commonly used JMX interfaces, along with a sample method for each.

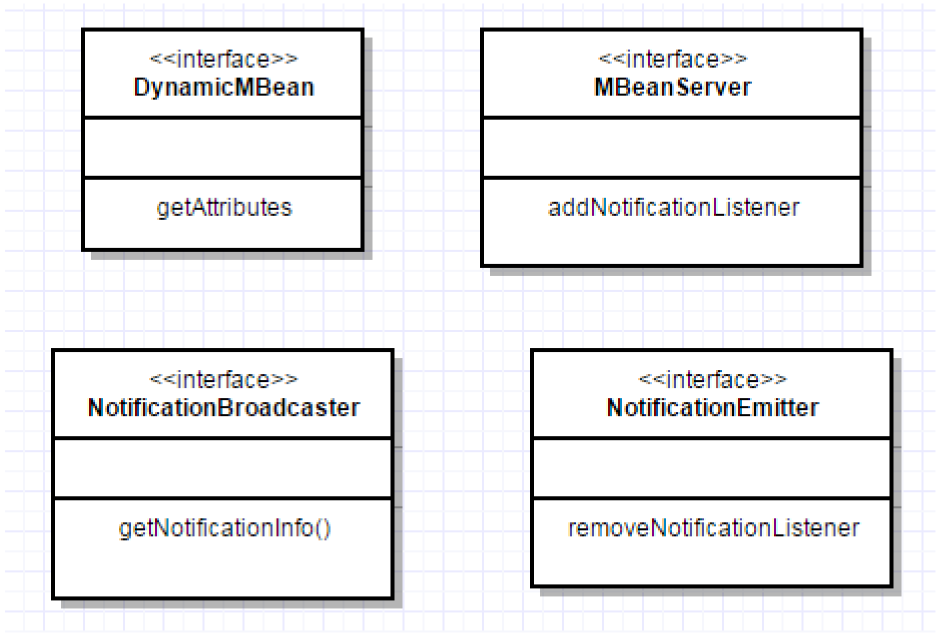


Figure 3.1 Commonly used JMX interfaces ("Online")

For Java software, at the lowest layer the Java Virtual Machine is the most essential resource. The JMX technology can be used to manage the Java Virtual Machine (JVM). To monitor and manage different aspects of the JVM, the JVM includes a platform `MBean` server and special `MXBeans` that are used by management applications.

MXBeans are provided with the Java SE platform to manage the JVM and other parts of the Java Runtime Environment (JRE). Different parts of the JVM functionality, such as a class-loading system and garbage collector, are encapsulated by a platform MXBean. These MXBeans are displayed and interacted with by using a monitoring and management tool that complies with the JMX specification. The Java SE platform's JConsole graphical user interface is one of such monitoring and management tools. Java SE provides the platform MBean server that allows the platform MXBeans to be registered. The platform MBean server can also be used for custom MBeans. Further, all application and platform MBeans that are registered in a connected JMX agent such as the platform MBean server can be accessed through the MBeans tab of the JConsole.

3.1.2 The Roles of JMX in Autonomic Computing

JMX facilitates the self-configuration properties of the Autonomic Computing paradigm. For instance, using MXBeans one can manage and configure Java's garbage collection abilities in real-time. More importantly, JMX facilitates the self-monitoring capability with the uniformed model MBeans and MXBeans ("Instrumenting"). All the self-CHOP properties can be implemented in corresponding MBean client components.

3.1.3 JMX for Autonomic Logging

JMX can be used to change log levels dynamically, according to the state of a process (in-process, waiting, finished, etc), which is a fulfillment of self-optimization (Sharif). This is done with the built-in LoggingMXBean along with enhancements in two aspects. In the subject processes, an enhanced MBean that uses the standard Runnable and Callable interfaces is deployed to track tasks as they go through their lifecycle. In the logging component, which is the client of the enhanced `MBean`, we can set a desired logging level for the reported process state. Or we can fine-tune logging in powerful ways - one way is to log the level after `N` number of logs at a certain level (for instance, change from INFO to WARN upon tracking 5 INFO log messages).

3.1.4 JMX for Self-Healing with Respect to Memory Usage

JMX also has powerful memory management abilities through its JConsole tool. As shown below in Figure 3.2, JConsole dynamically displays the memory usage (Heap Memory and Non-Heap Memory Usage). In this case, JConsole is the client of the built-in platform `MBean` that tracks memory usage.

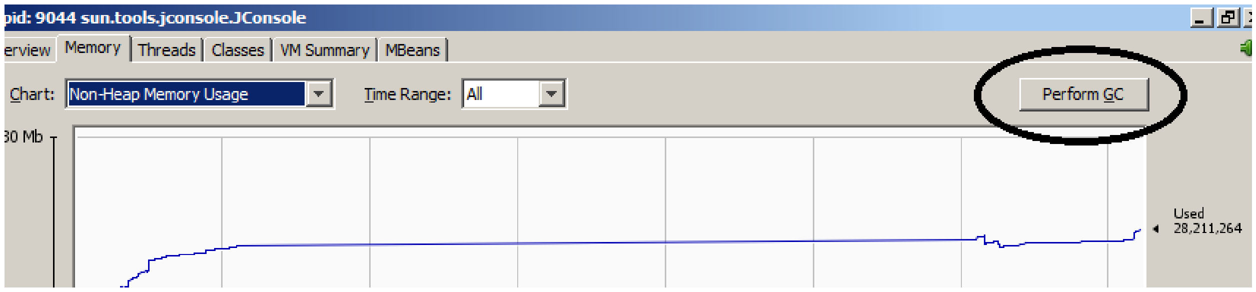


Chart: Non-Heap Memory Usage

Time Range: All

Figure 3.2 - JConsole Screenshot

The standard memory `MBean` has four attributes :

`HeapMemoryUsage` - read-only attribute describing the current heap memory usage

`NonHeapMemoryUsage` - read-only attribute describing non-heap memory usage

`ObjectPendingFinalizationCount` - read-only attribute describing the number of objects pending for finalization

`Verbose` – boolean attribute describing the garbage collector's verbose tracing setting. This may be dynamically set

Details about the MBean interface are defined in the `java.lang.management.MemoryMXBean` specification. A particularly noteworthy class in regards to autonomic computing is the `MemoryPoolMXBean`, which contains a collection usage threshold which can be queried with the `isCollectionUsageThresholdExceeded()` function.

It should be noted, though, that not all garbage-collected memory pools choose to support the collection usage threshold. In this case we should first call the `isCollectionUsageThresholdSupported()` method to check whether the collection usage threshold method is supported.

By replacing JConsole with a customized client of the same platform `MBean`, we will be able to achieve self-healing when heap memory usage exceeds a threshold. At such a moment, the customized client component can restart the same Java program in a new JVM with larger heap memory claim.

3.2 Logging and Checkpointing as a Means of Facilitating AC Features

Logging is also another technology that facilitates autonomic computing, and has existed since the early days of information technology. It was developed for system recovery and auditing. Among the most popular logging frameworks is Apache Log4J (now Apache Logback). The author utilized this framework extensively in logging experiments. By using Log4j, we can have logging at runtime, without modifying the application binary. This allows us to have a much clearer picture of system failures. Logging via Apache Log4j or Logback is very flexible, and output can be directed to an `OutputStream`, a file, a `java.io.Writer`, a remote Unix Syslog daemon, a remote Log4j server, or many other output targets ("Apache Log4j 1.2").

In regards to checkpointing tools, one such technology is called the Berkeley Lab Checkpoint/Restart (BLCR). This technology, developed for Linux, is a hybrid kernel/user implementation of checkpoint/restart. The goal is to provide a robust implementation that provides checkpoints for a wide range of applications, without requiring any changes to be made to the application code. Checkpointing is needed to develop robust distributed computing applications, because it enables us to roll back to a correct logical state. BLCR focuses on checkpointing parallel applications that communicate through the Message Passing Interface (MPI). BLCR also focuses on compatibility with the SciDAC Scalable Systems Software ISIC software suite ("Berkeley").

BLCR is divided into four main areas:

- Checkpoint/Restart for Linux (CR)
- MPI Libraries that can be Checkpointed
- Resource Management Interface to Checkpoint/Restart
- Development of Process Management Interfaces

Chapter 4. Experiments on JMX-Based Autonomic Features

In the autonomic architectural model, MAPE-K, monitoring is the first necessary component in any implementation of autonomic computing (Lalanda 2014). The powerful, extensive monitoring capabilities provided by JMX have proven effective in high-end, complex enterprise software products such as Oracle databases for over a decade. However, they have not been explored in computer science education due to overwhelming complexity.

Based on our understanding of autonomic computing through the survey in Chapter 2 of this work, we have decided to develop an API that supports learning and implementing autonomic features for programming projects in software courses. This could not be possible without the clearly documented and readily available JMX, as explained in Section 3.1. Meanwhile, our development is needed in assisting computer science students' education in autonomic computing. Even though multiple high-quality tutorials on JMX such as (Oracle 2015) have explained the model of the JMX framework, they all share the limit of JMX itself – monitoring only.

Nearly every example of a JMX application delivers the results in the `JConsole`, the powerful graphical user interface illustrating monitoring data from `MBeans` (an essential building block of the JMX framework). We have constructed a series of `MBean` classes and a set of corresponding listener classes. These `MBean` and listener classes have the capabilities of monitoring sockets, I/O operation, logging, checkpointing and memory usage. To shield complexity further, a class `ServerSideACManager` is implemented using our `MBean` and listener classes. On one hand, all the components including the server-side manager component can be directly utilized in software to achieve a number of features of autonomic computing. On the other hand, the students can learn from the working examples how to incorporate autonomic actions with the MBean listeners, and how to integrate multiple MBean listeners for self-healing actions.

In the aforementioned API for autonomic computing, there are 12 classes and 5 interfaces. All of them are packed in package autonomic_util. Figure 3.3 below presents a diagram of the interfaces within the package, along with two standard JMX classes that are extended (the class NotificationBroadcasterSupport and interface NotificationListener).

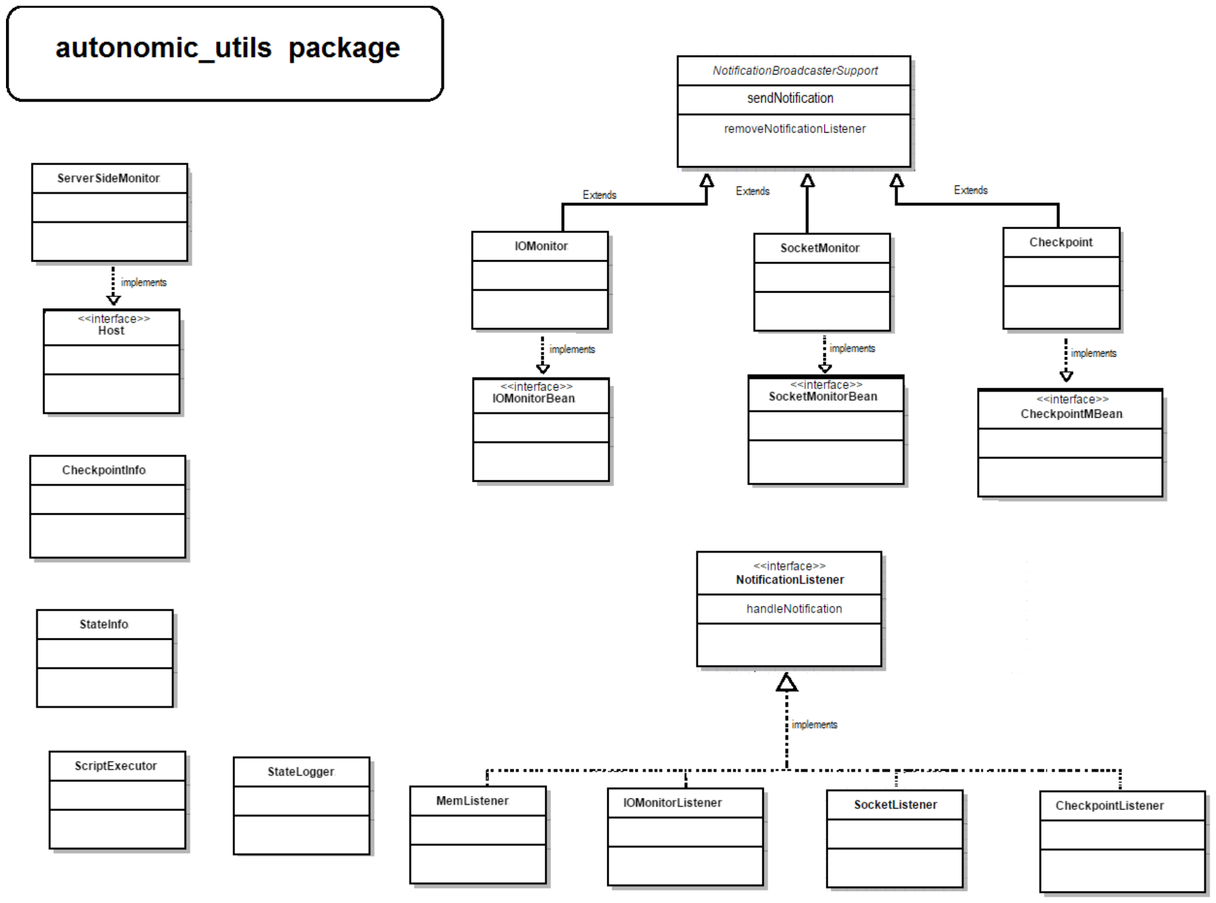


Figure 3.3 - The autonomic_util package class diagram ("Online")

To demonstrate the applications of package `autonomic_util`, four distributed programs are instrumented using the MBeans, and enhanced with listeners and other classes. These programs are (1) a pair of programs having conversation by sockets, (2) recoverability by logging, (3) distributed matrix multiplication by pipelines, and (4) a virtual classroom, demonstrating communication with students via sockets.

4.1 The JMX-based API Supporting Autonomic Features

In package `autonomic_util`, four `MBeans` are implemented, which are specified by three interfaces - `CheckpointMBean`, `IOMonitorMBean` and `SocketMonitorMBean`.

```
public interface CheckpointMBean {
	public void checkpointIntMatrix(int[][] m, int seqNum);
	public int[][] recoverIntMatrix(int seqNum);
	public void checkpointIntVector(int[] v, int seqNum);
	public int[] recoverIntVector(int seqNum);
}
```

Figure 4.1 Interface CheckpointMBean

The checkpoint MBean is specified above in Figure 4.1. For the purpose of

```
public  class Checkpoint extends NotificationBroadcasterSupport
                                        implements CheckpointMBean {
     …
     public void checkpointIntMatrix(int[][] arr, int seqNum) {
          // Serialize an int[][]
          try {
               chkpFileName = chkpFileNamePrefix+seqNum+".dat";
               ObjectOutputStream out = new ObjectOutputStream(
                           new FileOutputStream(chkpFileName));
               out.writeInt(seqNum);  out.writeObject(arr);
               out.flush();  out.close();
               notifyCheckpointDone();
          } catch (IOException ioe) { … }
     }

     public int[][] recoverIntMatrix(int seqNum) {
          int[][] arr = null;
          try {  // Deserialize the int[][]
               ObjectInputStream in = new ObjectInputStream(
                           new FileInputStream(chkpFileNamePrefix+seqNum+".dat"));
               int fileSeqNum = in.readInt();
               arr = (int[][]) in.readObject();
               in.close();
          } catch (ClassNotFoundException cfe) { … }
          return arr;
     }
```

Fig. 4.2 part of class Checkpoint

illustration, only creating and recovering integer matrices and vectors are included. Since this package is to support distributed computation, checkpoints have to support recovery across multiple computers. Therefore, the parameter `seqNum` is necessary to indicate the sequence numbers of the checkpoint files. The `recoverIntMatrix` (and `recoverIntVector`) takes one parameter, `seqNum`. It is expected to receive a correct seqNum value from the controller agent, the `ServerSideMonitor` object. The actual `MBean` class is class `Checkpoint` which contains the actual action of making a checkpoint, such as the method shown above in Figure 4.2.

Not only does class Checkpoint implement interface CheckpointMBean, it also extends class NotificationBroadcasterSupport. In doing so, class Checkpoint inherits the capabilities of broadcasting notifications to every registered listener. However, method notifyCheckpointDone has to be overridden as shown in Figure 4.3. Among many types, AttributeChangeNotification is chosen because it can carry many parameters.

```
public void notifyCheckpointDone () {
   Notification n = new AttributeChangeNotification(this, seqNum,
   System.currentTimeMillis(), callerClass, myID+"", chkpFileName, null, ip);
   sendNotification(n);
```

Fig. 4.3 Method notifyCheckpointDone in class Checkpoint

The MBean objects are embedded in the subject software components that are to be monitored. In the case of Checkpoint, this MBean is to notify the listener that a checkpoint is made. The primary purpose of Checkpoint is to preserve progress of computation and minimize loss caused by adversaries. If everything is normal, checkpoints are just some overhead. If in a distributed computation a program crashes, the coordinator of the computation has to determine the correct checkpoint in each of the participating computer nodes, and rollback to the latest consistent state. This capability is implemented in the ServerSideACManager object residing at a coordinator node, which utilizes listener CheckpointListener.

The last action of method checkpointIntMatrix is to invoke notifyAttributeChangeDone(). As shown in Figure 4.3, this method just packs many data items into an object notification, including the sequence number of the checkpoint file, the application program name (`callerClass`), the computer node identifier, the checkpoint file name, and the IP address of the hosting computer. Then the `sendNotification` method is invoked. This method is provided as a Java built-in capability of the abstract class `NotificationBroadcasterSupport`, from which our customized `MBean Checkpoint` extends. Upon invocation of `sendNotification`, the `MBean` agent server provided by the JMX framework delivers the notification to every registered `CheckpointListener` object.

The reception of notifications by an `MBean` listener is carried out by the `handleNotification` method of a listener class. Such a method in `CheckpointListener` is shown in Figure 4.4.

Corresponding to method `notifyCheckpointDone` in Figure 4.3, method `handleNotification` in Figure 4.4 processes the same notification object. Therefore, the same set of data items are unpacked here. Typically, the `notificationListener` is to be used by the controller or the coordinator of a distributed computation. For simplicity, we assume that TCP/IP is the communication protocol. The order of the messages from the same source is preserved when they arrive at each destination node. Therefore, the latest message always arrives last. Under such a condition, we only need to record the information of the checkpoint in the latest notification from each source node. In an object `CheckpointListener`, a `HashMap` list `cpInfo` of size K keeps the information from the latest notification from each source, where K is the number of resource nodes.

Every `Checkpoint MBean` object has a sequence counter, `seqNum`, which is initialized to zero. Every participating node in the same distributed computation has the same checkpoint interval. A checkpoint carries a value of the current `seqNum`. When a checkpoint is completed, `seqNum` increases by one. In case a rollback is needed, the coordinator can determine the latest consistent checkpoint set based on the `seqNum` value carried in the latest notification of each node. Let `seqNum[i]` represent the `seqNum` value in the latest notification from node *i*.

```
C = MIN{ seqNum[i] ) where i = 0, 1, 2, …, K-1.
```

`C` is the latest consistent checkpoint that every node should rollback to. `C` will not change unless a notification of a new checkpoint is received. Therefore, `C` is computed at the end of method `handleNotification`. It is stored in the variable `watermark`.

```
public void handleNotification(Notification notification, Object handback) {
   if (notification instanceof AttributeChangeNotification) {
      AttributeChangeNotification acn
            = (AttributeChangeNotification) notification;
      String className = acn.getSource().getClass().getName();
      if (className == "Checkpoint") {
         Integer id = Integer.parseInt(acn.getAttributeName());
         String hostClass = acn.getMessage();
         String fileName = acn.getAttributeType();
         int seqNum = (int)acn.getSequenceNumber();
         String ip = (String)acn.getNewValue();
         CheckpointInfo info
                  = new CheckpointInfo(id, hostClass, fileName, ip, seqNum);
         cpInfo.put(id, info);
         // update watermark
         ArrayList<CheckpointInfo> infoList
                        = new ArrayList<CheckpointInfo>(cpInfo.values());
         if (cpInfo.size() >= numNodes) {
            watermark = Collections.min(infoList).getSeqNum();
}  }  }  }
```

Figure 4.4 Method handleNotification in class CheckpointListener

The interfaces of the other two customized MBeans, IOMonitorMBean and SocketMonitorMBean, are shown in Figures 4.5 and 4.6.

```
public interface IOMonitorBean {
	public void notifyIOException(int progNum);
}
```

Figure 4.5 Interface IOMonitorMBean

```
public interface SocketMonitorMBean {
	public Socket reconnect();
	public Socket getSocket();
	public void setSocket(Socket socket);
	public void notifySocketException();
	public void closeSocket();
}
```

Figure 4.6 Interface SocketMonitorMBean

The actual MBean SocketMonitor has method notifySocketException as shown in Figure 4.7. This method is designed for the program that holds the monitored socket to inform its counterpart program when a socket exception is beyond local handling. Therefore, this method is typically used in catch-blocks. The code for the notifySocketException method is shown in Figure 4.7. The corresponding class SocketListener has method handleNotification, shown in Figure 4.8.

```
public void notifySocketException() {
   String localIp = subjectSoc.getLocalAddress().toString();
   String localPort = subjectSoc.getLocalPort()+"";
   Notification n = new AttributeChangeNotification(this,
            sequenceNumber++, System.currentTimeMillis(), callerClass,
            serverIp, serverPort+"", localIp, localPort);
   sendNotification(n);
}
```

Figure 4.7 Method notifySocketException in class SocketMonitor

```
public void handleNotification(Notification notification, Object handback) {
   if (notification instanceof AttributeChangeNotification) {
      AttributeChangeNotification acn =
                              (AttributeChangeNotification) notification;
      String className = acn.getSource().getClass().getName();
      if (className == "SocketMonitor") {
         String nodeIp = (String)acn.getOldValue();
         String script = iplist.getProperty(nodeIp);
         ScriptExecutor.execScript(script);
}  }  }
```

Figure 4.8 handleNotification method in class SocketListener

The code for methods `notifyIOException` in class `IOMonitor` and `handleNotification` in class `IOMonitorListener` are equally short. None of them have more than ten lines.

Since extensive monitoring capabilities of memory are provided by Java's built-in platform `MXBeans`, autonomic features such as self-configuration can be implemented in a customized listener class. A simple example, `MemListener`, is illustrated in Figure 4.9. When a `MemListener` object receives a notification from Java runtime's platform `MXBean`, either Java's heap memory became tight or it is out of memory . Given the prior knowledge regarding the maximum number of megabytes (`maxmem`) that a Java VM can have on a node (i.e, a computer), this notification handler will enlarge the Java heap size configuration setting if the current size is less than `maxmem`.

```
public void handleNotification(Notification notification, Object handback) {
   String notifType = notification.getType();
   if (notifType.equals(MemoryNotificationInfo.MEMORY_THRESHOLD_EXCEEDED)
      || notifType.equals(
            MemoryNotificationInfo.MEMORY_COLLECTION_THRESHOLD_EXCEEDED)) {
         // retrieve the memory notification information
      CompositeData cd = (CompositeData) notification.getUserData();
      MemoryNotificationInfo info = MemoryNotificationInfo.from(cd);
      long xmCurrent = info.getUsage().getMax() / (1024 * 1024);
      if (xmCurrent > max_xmx) {
         System.out.println("WARNING: Memory usage exceeded threashold at "
                                    + notification.getSource());
      } else {
         ScriptExecutor.execScript("java -Xmx" + max_xmx + "m "
                        + this.getClass().getName(), null);
}  }  }
```

Figure 4.9 Method handleNotification in class MemListener

Finally, `ServerSideManager` is an active class to be executed as an independent thread, as shown in Figure 4.10. It utilizes `MemListener`, `CheckpointListener`, and `IOMonitorListener` objects to assist the coordinator of a distributed computation.

The run method is used to configure the `MBean` agent server, and register each listener object at the remote nodes through JMX's built-in `MBean` agent server. The method `rollback` (shown in Figure 4.11) demonstrates how to manage all the participating nodes and assist them to rollback to the latest consistent checkpoint, recover their state to that point, and automatically continue computation.

```java
public void run() {
   // Manage the MemoryBean MBeans
   MemListener memListener = new MemListener(max_xmx);
   MemoryMXBean mbean = ManagementFactory.getMemoryMXBean();
   // Manage checkpointing
   chkpListener = new CheckpointListener(numNodes);
   NotificationEmitter emitter = (NotificationEmitter) mbean;
   emitter.addNotificationListener(memListener, null, null);
   // Manage the IOMonitor MBeans
   try {
      // register SocketMoniter MBean
      // Construct the ObjectName for the SocketMonitor MBean
      ObjectName mbeanName = new ObjectName("appmonitor_jmx:type=IOMonitor");
      // Create a dedicated proxy for the MBean instead of going directly
      // through the MBean server connection
      IOMonitorBean mbeanProxy = JMX.newMBeanProxy(mbsc, mbeanName,
      ``                  IOMonitorBean.class, true);
      // Add notification listener on IOMonitor MBean
      mbsc.addNotificationListener(mbeanName, ioListener, null, null);
      // register memory MBean
      mbeanName = new ObjectName("java.lang.management:type=MemoryMXBean");
      // Add notification listener on MemListener MBean
      mbsc.addNotificationListener(mbeanName, memListener, null, null);
      // register checkpoint MBean
      mbeanName = new ObjectName("appmonitor_jmx:type=Checkpoint");
      // Add notification listener on Checkpoint MBean
      mbsc.addNotificationListener(mbeanName, chkpListener, null, null);
      // keep alive
      nap.wait();
   } catch (Exception e) {
      System.err.println("Exception happened in connecting MBean SocketMonitor.");
         e.printStackTrace();
      }
   }
```

Figure 4.10 ServerSideManager is an active class to be executed as an independent thread

```java
int watermark = chkpListener.getWatermark();
   HashMap<Integer, CheckpointInfo> chkpList = chkpListener.getCpInfo();
   for (Map.Entry<Integer, CheckpointInfo> entry : chkpList.entrySet()) {
      int nodeId = entry.getKey();
      String hostClass = entry.getValue().getClassName();
      String fileName = entry.getValue().getFileName();
      String ip = entry.getValue().getIp();
      String cmd = "java "+hostClass+" "+fileName+" "+watermark+".dat";
      ScriptExecutor.execScript(cmd, ip);
   }
}
```

Figure 4.11 Rollback function in class ServerSideMonitor

4.2 Conversation by Sockets

In this section, a simple example using Java sockets is used to illustrate the use of `SocketMonitor MBeans`. In this example, classes `FixedMessageSequenceServer` (`Server` for short) and `FixedMessageSequenceClient` (`Client` for short) interact with

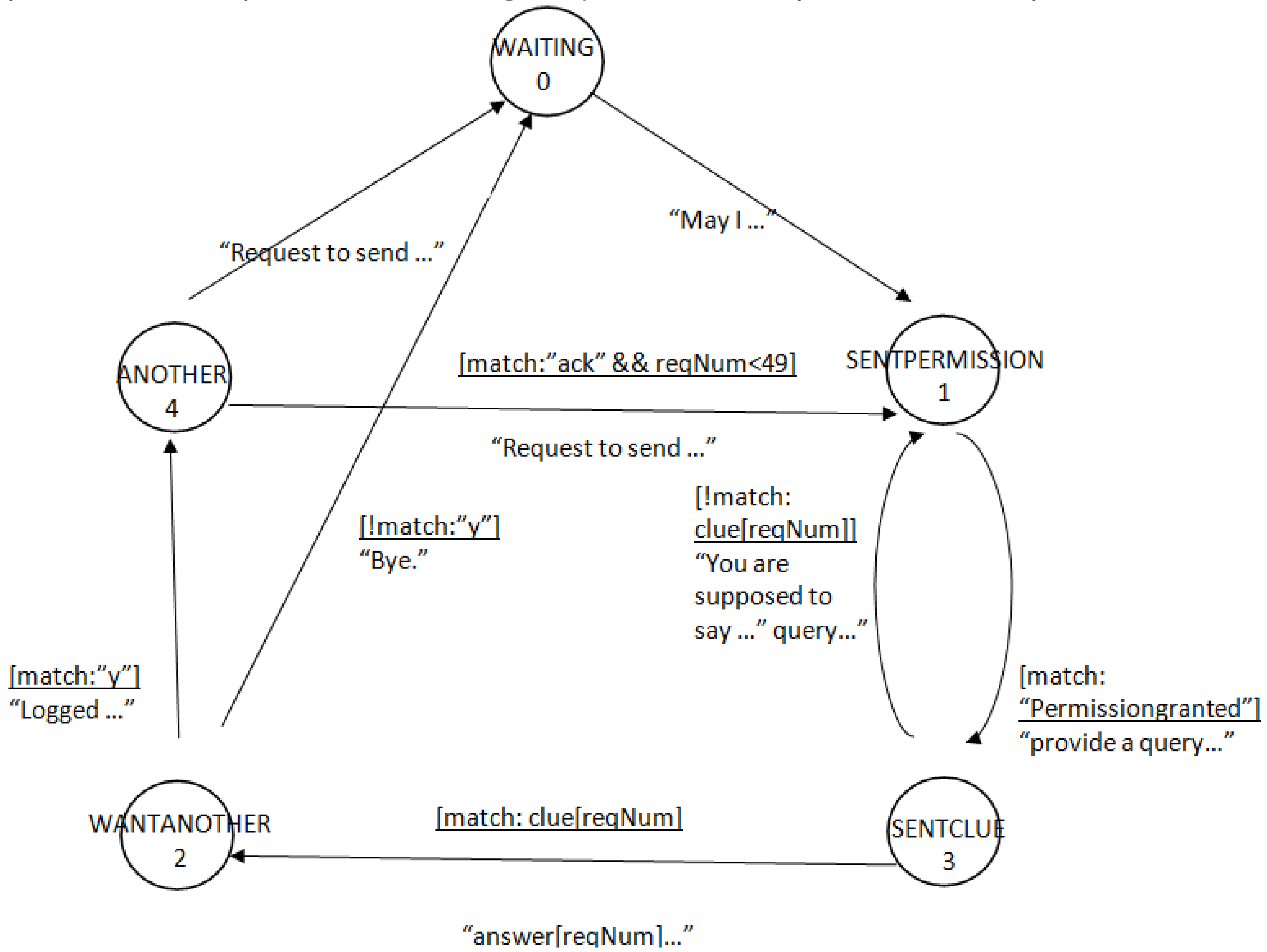


Figure 4.12 State Diagram of FixedMessageSequenceProtocol

messages according to a predefined script. The server controls the conversation using a protocol enforced by an object of `FixedMessageSequenceProtocol`. The client simply sends its stored sentences in a fixed order. This example represents interactions between stateful objects. The conversation protocol enforced by `FixedMessageSequenceProtocol` is shown in Figure 4.12. The output sentences are shown below the arrows, and the conditions are shown in square brackets (`[` and `]`) above the arrows.

The state of the object of `FixedMessageSequenceProtocol` contains two variables, `reqNum` and the current state, which is also the state of program `FixedMessageSequenceServer`. The state of program `FixedMessageSequenceClient` is just `reqNum`. To make these two programs recoverable, we have instrumented each of them in two places. First, after a message is sent, the program's state is logged in a local file using an object of `StateLogger`. Second, in the catch-statements information about the state is obtained from the last line of the local log file, then the program is restarted with the state set to the breaking point. In the socket's `IOException` catch-block, a limited number (3) of retrials are allowed. Once the limit is exceeded, a `SocketException` is thrown and the program restarts. Figure 4.13 shows the instrumentation part of the `FixedMessageSequenceServer` program in boldfaced-font. The `StateLoggerListener` object in each program's counterpart will instruct its host program to react.

```
        try {
           while (inputLine != null) {
              try {
                     inputLine = in.readLine();
                     System.out.println("Client said: " + inputLine); "
                     outputLine = fmsp.processInput(inputLine);
                     out.println(outputLine);
                     System.out.println("I said: " + outputLine);
                     if (outputLine.equals("Bye.")) {
                            in.close(); out.close(); socket.close();
                            break;
                     }
                     Thread.sleep(DELAYSECS);
                     // logging state
                     Attributes states = new Attributes(2);
                     states.put(new Attributes.Name("requestNum"),
                                   fmsp.getRequestNum()+"");
                     states.put(new Attributes.Name("state"),
                                   fmsp.getState()+"");
                     logger.logState(states);
              } catch (InterruptedException ie) {
                     ie.printStackTrace();
              } catch (IOException ioe) {
                     System.out.println("Server Warning: … " + ++warnNum);
                     if (warnNum >= WARNINGLIMIT) {
                            System.out.println("I/O exceptions exceed limit.");
                            ioe.printStackTrace();
                            throw new SocketException();
           }  }      }
        } catch (SocketException so) {
           logger.close();
           StateInfo states = logger.getLatestState();
           String cmd = "java " + this.getClass().getName() + …
           ScriptExecutor.execScript("ClientBatch.bat");
        }
```

Figure 4.13 Instrumentation in FixedMessageSequenceServer

In Figure 4.13, the four statements for “logging state” seem cumbersome because the chosen data structure to wrap a program’s state is `java.util.jar.Attributes`. Before logging, the user’s program has to put the name and value of every state variable into an `Attributes` object. Thus, the number of statements will be `V+2`, where `V` is the number of state variables in that program. The

`Attributes` class is chosen for its flexibility. In general, the state of a program is made of many variables.

4.3 Distributed Matrix Multiplication by Pipelines

Various distributed matrix multiplication algorithms have long been the classic examples of distributed programming. It's not only every engineering problem using a linear model that needs matrix multiplication, but modern Internet search also relies on matrix multiplication for computing metrics such as the PageRank algorithm. In this section, a unique pipeline algorithm for matrix multiplication (Andrews) is chosen as an example to show the use of the `IOMonitor` `MBeans`, `Checkpoint` `MBeans`, and the `ServerSideACManager` object from our API.

4.3.1 Matrix Multiplication Pipelines

For simplicity, two `n × n` matrices **a** and **b** are considered. We want to compute `c = a × b`. In this algorithm, initially row `i` in **a** is shifted toward the left `i` columns as shown in Figure 4.14; column `j` in **b** is shifted upwards `j` rows as shown in Figure 4.14, where `i` and `j` take the range of 1 to n.

a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}

a

$\mathbf{a_{12}}$	$\mathbf{a_{13}}$	$\mathbf{a_{14}}$	$\mathbf{a_{11}}$
$\mathbf{a_{23}}$	$\mathbf{a_{24}}$	$\mathbf{a_{21}}$	$\mathbf{a_{22}}$
$\mathbf{a_{34}}$	$\mathbf{a_{31}}$	$\mathbf{a_{32}}$	$\mathbf{a_{33}}$
$\mathbf{a_{41}}$	$\mathbf{a_{42}}$	$\mathbf{a_{43}}$	$\mathbf{a_{44}}$

a’

Figure 4.14 Initializing matrix a

b_{11}	b_{12}	b_{13}	b_{14}
b_{21}	b_{22}	b_{23}	b_{24}
b_{31}	b_{32}	b_{33}	b_{34}
b_{41}	b_{42}	b_{43}	b_{44}

b

$\mathbf{b_{21}}$	$\mathbf{b_{32}}$	$\mathbf{b_{43}}$	$\mathbf{b_{14}}$
$\mathbf{b_{31}}$	$\mathbf{b_{42}}$	$\mathbf{b_{13}}$	$\mathbf{b_{24}}$
$\mathbf{b_{41}}$	$\mathbf{b_{12}}$	$\mathbf{b_{23}}$	$\mathbf{b_{34}}$
$\mathbf{b_{11}}$	$\mathbf{b_{22}}$	$\mathbf{b_{33}}$	$\mathbf{b_{44}}$

b’

Figure 4.15 Initializing matrix b

The product of the numbers in the corresponding cells of **a’** and **b’** is one term of every cell of matrix **c**, as shown in Figure 4.16.

$a_{12} \times b_{21}$	$a_{13} \times b_{32}$	$a_{14} \times b_{43}$	$a_{11} \times b_{14}$
$a_{23} \times b_{31}$	$a_{24} \times b_{42}$	$a_{21} \times b_{13}$	$a_{22} \times b_{24}$
$a_{34} \times b_{41}$	$a_{31} \times b_{12}$	$a_{32} \times b_{23}$	$a_{33} \times b_{34}$
$a_{41} \times b_{11}$	$a_{42} \times b_{22}$	$a_{43} \times b_{33}$	$a_{44} \times b_{44}$

Figure 4.16 Computing a term in matrix c

Then, the entire matrix **a'** is shifted leftward one column and becomes **a''**; the entire matrix **b'** is shifted upward one row and becomes **b''**. By multiplying the numbers in the corresponding cells of **a''** and **b''** we obtain another term for every cell in matrix **c**.

a_{13}	a_{14}	a_{11}	a_{12}
a_{24}	a_{21}	a_{22}	a_{23}
a_{31}	a_{32}	a_{33}	a_{34}
a_{42}	a_{43}	a_{44}	a_{41}

a''

b_{31}	b_{42}	b_{13}	b_{24}
b_{41}	b_{12}	b_{23}	b_{34}
b_{11}	b_{22}	b_{33}	b_{44}
b_{21}	b_{32}	b_{43}	b_{14}

b''

$a_{13} \times b_{31}$	**$a_{14} \times b_{42}$**	**$a_{11} \times b_{13}$**	**$a_{12} \times b_{24}$**
$a_{24} \times b_{41}$	**$a_{21} \times b_{12}$**	**$a_{22} \times b_{23}$**	**$a_{23} \times b_{34}$**
$a_{31} \times b_{11}$	**$a_{32} \times b_{22}$**	**$a_{33} \times b_{33}$**	**$a_{34} \times b_{44}$**
$a_{42} \times b_{21}$	**$a_{43} \times b_{32}$**	**$a_{44} \times b_{43}$**	**$a_{41} \times b_{14}$**

c

Figure 4.17 The second round of matrix multiplication

Such multiply-then-shift operations continue for 4 rounds, whereby matrix c will accumulate all four terms of products. That is,

$$c_{ij} = a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + a_{i3} \times b_{3j} + a_{i4} \times b_{4j}$$

To the extreme, every row forms a circular pipeline; so does every column. By using $2n$ pipelines the algorithm can be literally implemented in super-computing hardware. Practically, the matrices can be divided into grids of any size, for example p rows and q columns. p or q can be as small as 1. Then $p \times q$ nodes are needed, which form $p + q$ pipelines. Compared to the algorithm for matrix multiplication that is used by MapReduce, the pipeline algorithm saves the time of hashing and distributing elements that occurs with the MapReduce framework (Leskovec).

4.3.2 Instrumentation in the Pipeline Program for Autonomic Features

In the implementation of the matrix multiplication pipeline algorithm, there are two classes - `Coordinator` and `Worker`. The duties of `Coordinator` include

- configuring the grid by informing every `Worker` node about the IP addresses and port numbers of its left and upper neighbor nodes
- initializing matrices `a` and `b` according to the shifting rules of the pipeline algorithm
- distributing blocks to every node,
- collecting the blocks of the resulting matrix

An object of `ServerSideManager` is instantiated in `Coordinator`. Its capabilities were described in Section 4.1. In this matrix multiplication pipeline program, `ServerSideManager` listens to every `Checkpoint MBean` and updates the information of the latest checkpoint in each node. `ServerSideManager` also listens to the `IOMonitor MBean` in every node. The reaction of an I/O Exception from a node is to restart the `Worker` program at every node from the latest consistent checkpoint. In addition, `Coordinator` also deploys one `IOMonitor MBean` locally. This is to monitor the final communication phase – collecting the final result from every node. In the catch-statement of the final receiving loop, method `notifyIOException` of the only `IOMonitor MBean` in `Coordinator` looks as below.

```
iom.notifyIOException(Integer.MAX_VALUE);
```

The actual parameter should have been a checkpoint sequence number. However, `Coordinator` does not actually make any checkpoint. Therefore, `Integer.MAX_VALUE` is fed into this method call, which will avoid any interference with the computation of the watermark value among the `Worker` nodes.

The actual computation carried out by each `Worker` node is to calculate the multiplication of two numbers *m* times. Afterward, matrix **a** is shifted *m* times (leftward) and matrix **b** is shifted *m* times (upward), where $m \times g = n$, and where *g* is the grid dimension, and *m* is the number of rows and columns in each grid. For simplicity, we assume every grid is of the same size. The subject `Worker` class has 332 rows. The desired autonomic feature is to recover automatically from the latest checkpoint. To fulfill this, two `MBeans` from package `autonomic_util` are instantiated. They are objects of `Checkpoint` and `IOMonitor`.

```
cp = new Checkpoint(nodeId, checkptFilePrefix, seqNum,
          this.getClass().getName(), null);
iom = new IOMonitor(nodeId, null, this.getClass().getName());
```

The `Checkpoint MBean` needs information about the `Worker` object's node id, the prefix of checkpoint filenames, the checkpoint sequence number, and the program name (the class name of the hosting object). We choose to make a checkpoint for every *m* multiply-then-shift operations. Therefore, each node will make *g* checkpoints in a normal computation. Four lines of code are added (as shown below) at the end of the iterations for completing *m* rows of **a** and *m* rows of **b**.

```
// making checkpoint
cp.checkpointIntMatrix(a, seqNum);
cp.checkpointIntMatrix(b, seqNum);
cp.checkpointIntMatrix(c, seqNum);
seqNum++;
```

In the `Worker` class, the sockets are hidden in helper classes. This is because the actual required objects for shifting rows and columns across nodes are the two `DataOutputStream` objects, which send the top row to the upper neighbor and the leftmost column to the left neighbor. Furthermore, two `DataInputStream` objects are needed to receive the values into the bottom row from the lower neighbor node, and to receive the values into the rightmost column from the right neighbor node. In this case, we have realized in addition to the `SocketMBean` for each socket, an MBean bounding to all the input and output operations is needed, which is the IOMonitor MBean. In the catch statement in each of incoming communications, method `notifyIOException` of the `IOMonitor MBean iom` is invoked. For example, below are the operations for receiving the bottom row.

```
// receive the bottom row
for (int i = 0; i < width; i++) {
   try {
      tempIn[i] = disBottom.readInt();
   } catch (IOException ioe) {
      System.out.println("error in receiving from bottom, col=" + i);
      ioe.printStackTrace();
      iom.notifyIOException(iterations);     // call for rollback
}  }
```

Chapter 5. Discussion and Conclusion

Automation has always been the prime motivation behind computers themselves. From mapping to calculating taxes to running enterprise websites, automation is a huge source of profit for companies. Yet, only recently have we witnessed that we could benefit from automating the management of computers. IBM's manifesto points out that there are still great challenges ahead (“IBM” 1):

"To really benefit IT customers, autonomic computing will need to deliver measurable advantage and opportunity by improving interaction with IT systems and the quality of the information they provide, and enabling e-sourcing to be adopted as the future of IT services delivery."

In recent years, the emphasis of autonomic computing has been focused on the data center, and has even touched the physical infrastructure. As Kephart states : “In terms of emphasis, it is disappointing but understandable that the preponderance of work in the field continues to focus on self-optimization, with self-healing and self-configuration receiving far less attention” (Kephart 2011). As technology continues to advance, we will witness ever-increasing demands of autonomic computing. It is only natural that the ongoing evolution of computing will include great advancements in autonomic computing.

In this thesis, we have provided a survey of autonomic computing - both historically and as it currently exists. The original IBM Autonomic Computing toolkit, as well a European follow-up to it, were touched upon. We also discussed the Java

Management Extensions technology, and how it provides monitoring that can be leveraged to build applications displaying autonomic computing features.

Furthermore, by using JMX, we have implemented an API (package autonomic_utils) that supports learning and implementing autonomic features for programming projects. These AC features include socket management, I/O exception handling, logging, checkpointing, and recovery of distributed computation. This API has been applied to the implementation of AC features in two distributed Java programs, conversation by sockets and distributed matrix-multiplication by pipelines.

In the experiments, we have not only realized the powerful capabilities of Java that are unknown to many Java educators, we also illustrated the feasibility of learning and practicing autonomic computing as early as in senior computer science courses.

References

ACE Autonomic Toolkit. http://acetoolkit.sourceforge.net/. 4/20/2016

Alaya, Mahdi Ben, Salma Matoussi, Thierry Monteil, and Khalil Drira. "Autonomic Computing System for Self-management of Machine-to-machine Networks." Proceedings of the 2012 International Workshop on Self-Aware Internet of Things - Self-IoT '12 (2012): n. pag. Web.

"An architectural blueprint for autonomic computing." IBM. 2005. 13 Jun. 2015 <http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf>.

Andrews, Gregory R. Foundations of Multithreaded, Parallel, and Distributed Programming. Reading, MA: Addison-Wesley, 2000. Print.

"Apache Log4j 1.2 -." Apache Log4j 1.2 -. N.p., n.d. Web. 27 Mar. 2016. <https://logging.apache.org/log4j/1.2/>.

Avouac, Pierre-Alain, Philippe Lalanda, and Laurence Nigay. "Service-Oriented Autonomic Multimodal Interaction in a Pervasive Environment." Proceedings of the 13th International Conference on Multimodal Interfaces - ICMI '11 (2011): n. pag. Web.

Benko, Borbala. CASCADAS Deliverable D7.2: Electronic learning material. 2008.

"Berkeley Lab Checkpoint/Restart (BLCR) for LINUX." Berkeley Lab Checkpoint/Restart (BLCR). N.p., n.d. Web. 27 Mar. 2016. <http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/>.

"Danschultzer/monit-graph." GitHub. N.p., n.d. Web. 07 Apr. 2016. <https://github.com/danschultzer/monit-graph>.

"Deploy an OpenStack Private Cloud to a Hadoop MapReduce Environment." Deploy an OpenStack Private Cloud to a Hadoop MapReduce Environment. IBM, n.d. Web. 07 Apr. 2016. <http://www.ibm.com/developerworks/cloud/library/cl-openstack-deployhadoop/>.

Duc, Bao Le, Pierre Châtel, Nicolas Rivierre, Jacques Malenfant, Philippe Collet, and Isis Truck. "Non-functional Data Collection for Adaptive Business Processes and Decision Making." Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing - MWSOC '09 (2009): n. pag. Web.

"Dynamic Java Log Levels with JMX/LoggingMXBean, JConsole, VisualVM, and Groovy."JavaWorld. JavaWorld, n.d. Web. 14 Mar. 2016. <http://www.javaworld.com/article/2073316/dynamic-java-log-levels-with-jmx-loggingmxbean--jconsole--visualvm--and-groovy.html>.

Ganek, Alan. "Overview of Autonomic Computing: Origins, Evolution, Direction." *Autonomic Computing: Concepts, Infrastructure, and Applications* . Ed. Manish Parashar, Salim Hariri. Boca Raton: CRC Press, 2007. 3-18.

Ghosh, Sukumar. Distributed Systems: An Algorithmic Approach. Boca Raton: Chapman & Hall/CRC, 2007. Print.

Gusworld Article Archive. Autonomic Transmission. <http://www.gusworld.com.au/writing/auton.htm>

Huebscher, Markus C., and Julie A. Mccann. "A Survey of Autonomic Computing—Degrees, Models, and Applications." CSUR ACM Comput. Surv. ACM Computing Surveys 40.3 (2008): 1-28. Web.

IBM Research | Autonomic Computing. <http://www.research.ibm.com/autonomic/> . 4/20/2016.

IBM Research | Autonomic Computing | Glossary. <http://www.research.ibm.com/autonomic/glossary.html> . 4/20/2016.

IBM Research Autonomic Computing Overview Frequently Asked Questions. <http://www.research.ibm.com/autonomic/overview/faqs.html>. 4/20/2016

"Instrumenting Your Resources for JMX Technology." Instrumenting Your Resources for JMX Technology. Oracle, 02 Aug. 2008. Web. 11 Apr. 2016. <http://docs.oracle.com/javase/8/docs/technotes/guides/jmx/overview/instrumentation.html>.

International Business Machines. Autonomic Computing: IBM's Perspective on the State of Information Technology. Armonk: International Business Machines, 2001. Print.

Jacob, Bart, Richard Lanyon-Hogg, Devaprasad Nadgir, Amr Yassin. *A Practical Guide to the IBM Autonomic Computing Toolkit*. N.p: IBM, 2004. Print.

Jasnowski, Mike. JMX Programming. New York, NY: Wiley Pub., 2002. Print.

Kephart, J.O.; Chess, D.M. (2003), "The vision of autonomic computing", *Computer* **36**: 41-52, doi:10.1109/MC.2003.1160055

Kephart, Jeffrey O. "Autonomic Computing: The First Decade." Proceedings of the 8th ACM International Conference on Autonomic Computing - ICAC '11 (2011): n. pag. Web.

Kephart, Jeffrey O. "Autonomic Computing." Proceedings of the 8th ACM International Conference on Autonomic Computing - ICAC '11 (2011): n. pag. Web.

Kephart, J.O. "Research Challenges of Autonomic Computing: An Industry Perspective." International Conference on Autonomic Computing, 2004. Proceedings. (2004): n. pag. Web.

Kephart, Jeffrey O. "Engineering Decentralized Autonomic Computing Systems." Proceeding of the Second International Workshop on Self-Organizing Architectures - SOAR '10 (2010): n. pag. Web.

Kreger, Heather, Ward Harold, and Leigh Williamson. Java and JMX: Building Manageable Systems. Boston: Addison-Wesley, 2003. Print.

Lalanda, Philippe, Julie McCann, and Ada Diaconescu. Autonomic Computing: Principles, Design and Implementation. London: Springer, 2014. Print.

Leskovec, Jurij, Anand Rajaraman, and Jeffrey D. Ullman. Mining of Massive Datasets / Jure Leskovec, Anand Rajaraman, Jeffrey David Ullman, Standford University. N.p.: n.p., n.d. Print.

"Lesson: Overview of the JMX Technology." (The Java Tutorials Java Management Extensions (JMX)). Oracle, n.d. Web. 14 Mar. 2016. <https://docs.oracle.com/javase/tutorial/jmx/overview/index.html>.

Maggio, Martina, Henry Hoffmann, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. "Decision Making in Autonomic Computing Systems." Proceedings of the 8th ACM International Conference on Autonomic Computing - ICAC '11 (2011): n. pag. Web.

Mannava, Vishnuvardhan, and T. Ramesh. "Multimodal Pattern-Oriented Software Architecture for Self-Optimization and Self-Configuration in Autonomic Computing System Using Multi Objective Evolutionary Algorithms." Proceedings of the International Conference on Advances in Computing, Communications and Informatics - ICACCI '12 (2012): n. pag. Web.

Mannava, Vishnuvardhan, and T. Ramesh. "A Service Configuration and Composition Design Pattern for Autonomic Computing Systems Using Service Oriented Architecture." Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology - CCSEIT '12 (2012): n. pag. Web.

"MemoryMXBean (Java Platform SE 7)." MemoryMXBean (Java Platform SE 7). Oracle, n.d. Web. 14 Mar. 2016. <https://docs.oracle.com/javase/7/docs/api/java/lang/management/MemoryMXBean.html>.

"MemoryPoolMXBean (Java Platform SE 7)." MemoryPoolMXBean (Java Platform SE 7). Oracle, n.d. Web. 14 Mar. 2016. <https://docs.oracle.com/javase/7/docs/api/java/lang/management/MemoryPoolMXBean.html>.

"morphogenesis." Dictionary.com Unabridged. Random House, Inc. 25 May. 2015. <Dictionary.com http://dictionary.reference.com/browse/morphogenesis>.

"Oracle Database 11g vs IBM DB2 UDB V9.7 Manageability Overview." Oracle.com. Dec. 2009. http://www.oracle.com/technetwork/database/manageability/oracle-11g-vs-db2-9-7-manageability-131878.pdf

"Online Diagram Creation Tool - Sign up for Free Trial | Gliffy.com." Online Diagram Creation Tool - Sign up for Free Trial | Gliffy.com. N.p., n.d. Web. 07 Apr. 2016. <https://www.gliffy.com/home/>.

Parashar, Manish. "Autonomic Computing: A System-Wide Perspective." *Autonomic Computing: Concepts, Infrastructure, and Applications*. Ed. Manish Parashar, Salim Hariri. Boca Raton: CRC Press, 2007. 49-70. Print.

Sharif, Fahd. "Fahd.blog." : Change Logging Levels Using JMX [Howto]. Blogspot, n.d. Web. 15 Mar. 2016. <http://fahdshariff.blogspot.com/2008/06/change-logging-levels-using-jmx-howto.html>.

Saraswatipura, Mohankumar. "Understanding the Advantages of DB2 9 Autonomic Computing Features." DeveloperWorks. IBM, n.d. Web. 06 Apr. 2016. <http://www.ibm.com/developerworks/data/library/techarticle/dm-0709saraswatipura/>.

Tretola, Giancarlo, and Eugenio Zimeo. "Autonomic Internet-Scale Workflows." Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond - MONA '10 (2010): n. pag. Web.

Zhang, Li, and Danilo Ardagna. "SLA Based Profit Optimization in Autonomic Computing Systems." Proceedings of the 2nd International Conference on Service Oriented Computing - ICSOC '04 (2004): n. pag. Web.

Unknown. "A Metamodel for the Runtime Architecture of an Interactive System." SIGCHI Bull. ACM SIGCHI Bulletin 24.1 (1992): 32-37. Web.

"Using JConsole to Monitor Applications." Using JConsole to Monitor Applications. Oracle, n.d. Web. 14 Mar. 2016. <http://www.oracle.com/technetwork/articles/java/jconsole-1564139.html>.

van Renesse, Robbert, and Birman, Kenneth P. "Autonomic Computing: A System-Wide Perspective." *Autonomic Computing: Concepts, Infrastructure, and Applications*. Ed. Manish Parashar, Salim Hariri. Boca Raton: CRC Press, 2007. 35-47. Print.

Vasilakos, Athanasios V., Parashar, Manish, Karnouskos, Stamatis, Pedrycz, Witold. Autonomic Communication. Springer Science. New York: 2009.

Vita

The author was born in New Orleans, Louisiana in 1985. He graduated from Grace King High School in 2003 (Metairie, Louisiana), and received his Bachelor of Science in Computer Science from The University of New Orleans in 2010. He then joined the Computer Science graduate program at UNO in 2010, working under Dr. Shengru Tu in the area of autonomic computing. The author is currently employed by IBM.