AUTONOMIA: An Autonomic Computing Environment*

Xiangdong Dong, Salim Hariri, Lizhi Xue, Huoping Chen,
Ming Zhang, Sathija Pavuluri, Soujanya Rao
{nansheng, Hariri, lzxue, hpchen, mingz, pavuluri, soujanya}@ece.arizona.edu
Autonomic Computing Laboratory
Department of Electrical and Computer Engineering
University of Arizona
Tucson, AZ 85721
www.ece.arizona.edu/~hpdc

ABSTRACT

The proliferation of Internet technologies, services and devices, have made the current networked system designs, and management tools incapable of designing reliable, secure networked systems and services. In fact, we have reached a level of complexity, heterogeneity, and a rapid change rate that our information infrastructure is becoming unmanageable and insecure. This had led researchers to consider alternative designs and management techniques that are based on strategies used by biological systems to deal with complexity, heterogeneity and uncertainty. The approach is referred to as autonomic computing. An autonomic computing system is the system that has the capabilities of being self-defining, self-healing, self-configuring, self-optimizing, etc. In this paper, we present our approach to implement an autonomic computing infrastructure, Autonomia that provides dynamically programmable control and management services to support the development and deployment of smart (intelligent) applications. The AUTONOMIA environment provides the application developers with all the tools required to specify the appropriate control and management schemes to maintain any quality of service requirement or application attribute/functionality (e.g., performance, fault, security, etc.) and the core autonomic middleware services to maintain the autonomic requirements of a wide range of network applications and services. We have successfully implemented a proof-of-concept prototype system that can support the self-configuring, self-deploying and self-healing of any networked application.

1. Introduction

The wide deployment of Internet technology has resulted in exponential growth in Internet application services (e.g., content hosting for data with web-based access, shared payroll applications, firewall-based security services, email and shared file services). The management and control of these application services is a challenging research problem due to the huge amount of data that needs to be collected and coordinated, the heterogeneity and the independence of resources and components [10] required by these services and the fact that they run under different organizations and administration policies.

Further this problem is exacerbated with the proliferation of computer devices that has grown at exponential rates. In addition, demand is already outpacing supply when it comes to managing complex, and even simple computer systems. As access to information becomes omnipresent through PC's, hand-held and wireless devices, the stability of current infrastructure, systems, and data is at an increasingly greater risk to suffer outages and general disrepair. In the process, the systems have become increasingly difficult to use. This complexity has led to a situation where the cost to manage such systems is actually a lot more than the actual systems themselves [3]. Consequently, the growing complexity of the IT infrastructure threatens to

* The work presented here was supported by the National Science Foundation via grants numbers ACI 9984357 (CA-REERS), EIA 0103674 (NGS) and by DOE ASCI/ASAP (Caltech) via grant number PC295251.

undermine the very benefits information technology aims to provide.

A potential solution to these challenging research problems can be drawn from biological systems which have been very successful in controlling and managing complex, interactive, constrained systems. This solution approach is known as the Autonomic Computing that calls for designing distributed information systems that can automatically configure, deploy, secure, tolerate faults, optimize, and anticipate loads by themselves without the manual involvement of human administrators.

In this paper, we present an autonomic architecture to achieve automated control and management of networked applications and their infrastructure. We have successfully implemented a proof-of-concept prototype, referred to as AUTONOMIA, that implements two important properties of autonomic systems: self-configuring and self-healing.

The organization of this paper is as follows. In Section 2, we give a brief overview of autonomic computing and its main properties. In Section 3, we present a brief overview of related approaches and techniques. In Section 4, we give an overview of our approach to implement an autonomic computing system. In Section 5, we present in further detail our implementation approach and the technologies used to implement the proof-of-concept prototype "AUTONOMIA". In Section 6, we conclude the paper and discuss our future research activities.

2. Autonomic Computing: The next era of computing

IBM has recently launched a major research effort toward the development of autonomic computing systems and services [3]. The basic approach is to build computing systems that are capable of managing themselves; that can anticipate their workloads and adapt their resources to optimize their performance. This approach has been inspired by the human autonomic nervous system that has the ability to self-configure, self-tune and even repair themselves without any human conscience involvement. The concept of developing the next era of computing systems is driven by the convergence between the biological systems and the digital computing systems. To demonstrate the inefficiency of our approach to build the next generation of networked systems, just imagine if we apply our current design and management tools of distributed systems to implement the human central nervous system. That means, when a person jogs, he/sh needs to continuously monitor and adjust as appropriate the heart rate, the body temperature, blood pressure, send appropriate messages to selected areas of human body, etc. It is clear that it is almost impossible to do all these functions while a person is jogging; the person needs to focus on jogging rather than monitoring and adjusting a wide range of neurons and their systems.

Paul Horn [1] in his pioneering article on autonomic computing has identified a set of eight key elements or characteristics that form the criteria for a system to be classified as being an Autonomic Computing system. While the definition of autonomic computing will likely transform as contributing technologies mature, the following list suggests eight defining characteristics of an autonomic system: Self-Defining, Self-Protecting, Self-Optimizing, Self-Healing, Self-Configuring, Contextually Aware, Open and Anticipatory.

3. Related Works

There are several projects that are aimed at developing high performance programming models to develop large-scale distributed applications (Grid computing) that seamlessly allocate and obtain resources from the Grid infrastructure. Javelin of UCSB [11], Charlotte [22] of NYU, and Bayanihan [20] of MIT, allow clients to define components as Java Applets to be used in applications. Project Ninflet [12][23] intends to harness abundant idle computing power into a seamlessly integrated global as well as local parallel distributed computing environment. Another approach at University of Columbia aims at automating legacy systems and assembling autonomic systems of systems [4]. Other research projects like Recovery Oriented Computing (ROC) have emphasized the recovery and reliability of systems rather than performance [5][7], in which systems use excess computing, memory, storage, and other resources to improve the over-all system behavior and reliability.

In this paper, we present our approach to develop an autonomic computing infrastructure that supports complete control and management service in our prototype system, which we refer to as ***Autonomia***. ***Autonomia*** will essentially provide dynamic programmable control and management services to support the development and deployment of smart applications, automate performance and fault tolerant support for smart applications, provide automated deployment, registration, discovery of components, allow automated configuration of applications and system resources and finally provide a secure, open computing environment.

4. Autonomia Design Approach

The objective of this project is to automate deployment of mobile agents that have self-manageable attributes. The architecture of Autonomia is based on two previous projects: Adaptive Distributed Virtual Computing Environment (ADViCE) and CATALINA – A Proactive Application Control and Management System [6][27].

The Autonomia environment provides application developers with all the tools required to specify the appropriate control and management schemes, deploy and configure the required software and hardware resources, run application, provide on-line monitoring and management to maintain desired autonomicity. The architecture of Autonomia is shown in Figure 1. The main modules include Application Management Editor (AME), Application Delegated Manager, Autonomic Middleware Services (AMS), Java Space and Java Message Service. In this subsection, we will briefly discuss the functionality of the AME and the AMS. The other modules will be discussed further in the implementation section.

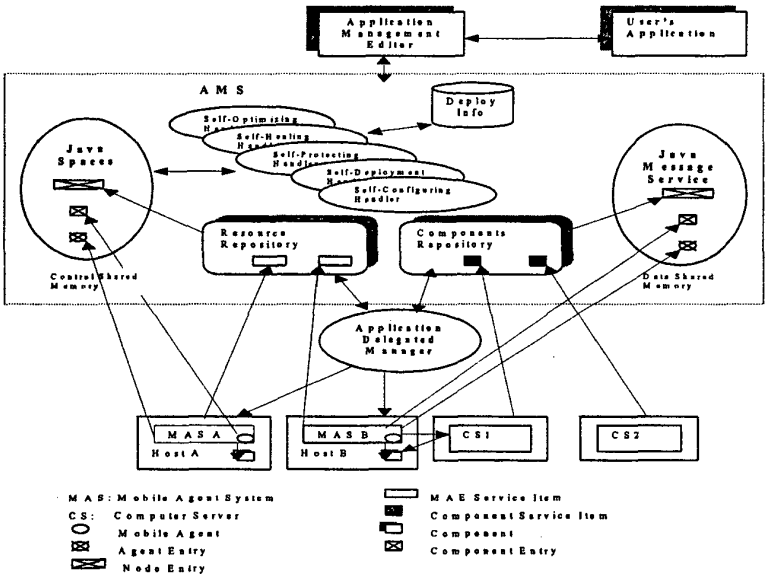

Figure 1: AUTONOMIA system architecture

4.1. Application Management Editor (AME)

It provides application developers with the services required for specifying an application's autonomic requirements (e.g. self-optimizing and self-healing) and also specifies the appropriate autonomic schemes to maintain the application requirements. The main functions of the editor are controlling the application editor workplace and storing the application management requirements in the component repository.

4.2. Autonomic Middleware Service (AMS)

Once the application management requirements are defined using the AME, the next step is to utilize the AMS services to build the appropriate application execution environment that can dynamically control the allocated resources to maintain the application requirements during application execution.

In what follows, we highlight how this architecture can achieve self-healing and optimizing properties for a given networked system or application.

Our methodology to achieve self-control and management of any functionality or property of an application service is based on three procedures: *Monitoring*, *Analysis and Verification*, and *Adaptation* procedures

1. Self Healing

For each fault type (system, component or agent), we will have a software agent (fault handler) that is responsible for executing the procedures. During the monitoring phase, the appropriate fault handler focuses on detecting faults once they occur. For example, for component fault detection, an agent continuously monitors the execution of the component and its consumption of CPU time. Once the component stops its execution, its execution status will be detected by the fault handler, which executes the next procedure (Analysis and

Verification). Then the self-healing handler will analyze and verify it in order to identify the fault type and what it is required to recover from that fault. Once that is done, the fault handler, will select and run the appropriate recovery procedure (Adaptation procedure). In case of host failure, the fault handler will consult with the Application Delegated Manger (ADM) to identify another machine to run all the affected components by the host failure.

2. Self Optimizing Handler

In a similar approach as in self-healing, there will be a software agent that we refer to as the Self-Optimizing Handler that is responsible to optimize the application as well as system performance at runtime. The self-optimization handler selects the appropriate mechanism to optimize application performance (by migrating application components, re-mapping the application components to resources, etc.), change the overall resource allocation and load balancing, just to name a few.

5. Autonomia Implementation Approach

In this section we discuss in detail our implementation of the main modules of Autonomia and focus on the implementation of the self-healing functionality of the system.

5.1. Mobile Agent System

The mobile agent system (MAS) for Autonomia is designed to provide mobile agents a uniform execution environment independent of the underlying hardware architecture and operating system. It provides functions to receive agents, start execution of agents, monitor execution state of agents, and transfer agents from host to host. It also provides facilities to enable Autonomia to keep track of the joining and withdrawal operations of hosts in the environment. In the mobile agent system, we define a component - Agent Transport Protocol, to act as the access point of the MAS. It is in charge of receiving agents from other remote hosts, either agent enabled or non-agent systems, and sending agents to other agent-enabled hosts.

Our mobile agent system is based on Java/Jini technologies. The appropriateness of using Jini technologies was discussed in [15][17]. The component Agent Transport Protocol (ATP) is implemented as a Jini service, which implements the interface AgentProtocolInterface that defines the behaviors of the protocol. During the installation of a mobile agent system, its Agent Transport Protocol is published with a Jini Lookup Service – Resource Repository (RR), through which the clients can query the host. To publish the service, a Jini service item of the Agent Transport Protocol is created. After registration, the proxy of the Agent Transport Protocol service as well as the host information is uploaded to RR. Meanwhile, a unique service ID is assigned to the Agent Transport Protocol service. We use this ID as the mobile agent system identifier. A Jini Lease Renewal Manager is used in the mobile agent system, which keeps renewing the lease of the registration with the RR. When a client wants to send agents to a host, it queries the Resource Repository for the proxy of the mobile agent system on that host. In our implementation, the proxy of the Agent Transport Protocol service is a Java RMI stub. Our implementation of the Mobile Agent System uses multithread programming. For each agent, MAS spawns a new thread to activate it. Thus the MAS has control of these threads and is capable of detecting their status by polling.

5.2. Application Management Editor (AME)

A user can develop an application by selecting components from a well-defined library or libraries of components that are registered in the Component Repository (see Figure 1), and how the components are interconnected; the application development involves constructing application flow graphs from the components registered in the component repository. In addition to developing the application flow graph, the AME enables the user to specify the management requirements for each component that are needed to control and manage the required autonomic properties for that component.

For each application to be registered in Autonomia environment, an Application Service Template (AST) is created and stored in the component repository. The AST provides a uniform format for representing all the management and control attributes associated with that application using an application template class. The control and management information is stored in the application template class and consists of name, description, dependencies (properties of every application template object), attributes (lists all application-specific properties), fault tolerance, security, monitoring techniques, and monitoring parameters. The AST is described using the Extensible Makeup Language (XML).

5.3. Autonomic Middleware Service

The AMS provides the core autonomic management services required to achieve an autonomic computing environment. The services offered by the AMS are implemented using mobile agents. The main services

offered by the AMS are Component Repository, Resource Repository, JavaSpaces, JMS, and the Fault and Security Performance Handlers. Furthermore, each AMS service can be implemented using one or more algorithms that can be selected dynamically at runtime based on the current application state as well as the system state.

5.3.1. Component Repository (CR). Component Repository is a Jini Lookup Service named "Components". It contains a collection of components that are currently available to the users to develop their applications. A component registers with the CR if it wants to provide its service to an application. The registration information cannot stay in the CR infinitely. A lease indicating the duration of the component registration is returned to the component provider when the component registers with the CR. The component provider needs to renew the registration before its lease expires if it wants to continue to provide its component service. A component will be eliminated from the CR if it stops renewing the lease.

In our implementation, components are implemented as Jini Services. For each component, there is a corresponding service wrapper class that is responsible for publishing the component as a Jini service. The components developed by different component providers can be published in our environment at any time and plugged in anywhere. They can be deployed to the appropriate hosts automatically without pre-installing.

5.3.2. Resource Repository (RR). Resource Repository is a Jini Lookup Service, named "Resources". It keeps track of all host registrations that are currently registered in the environment. The registration is made for a specific time period and a lease is returned to the host. Resources can be added or discovered in a similar approach to the component publishing and discovering techniques that were discussed in the previous subsection.

5.3.3. Control and Data Message Center. There are two message centers: Control and Management Message Center and Data Message Center, which were implemented using JavaSpaces and Java Message Service (JMS) respectively.

5.4. Application Delegated Manager

The Application Delegated Manager plays the role of a broker between components and resources. In addition, the ADM will have the responsibility of supervising the application execution at runtime to make sure we achieve all the required autonomic properties identified in the application service template as discussed in the Application Management Editor subsection. The main functions of ADM can be outlined as follows:

- Maintain an up-to-date list of resources as well as available components that can participate in executing a given application.
- Select the set of resources that can meet the autonomic requirements of an application.
- Dispatch the appropriate number of mobile agents to deploy, configure, perform on-line monitoring, analysis, and adaptation when the application cannot meet its requirements.

5.5. Fault handler

There are three types of fault handlers: System Fault Handler, Component Fault Handler and Agent Fault Handler. They deal with system fault, component fault and agent faults, respectively. These handlers are created by the ADM and they share the component deployment information. The ADM registers with the Coordinate Space (CS) that it is interested in the presence of the fault entries in the Coordinate Space. The fault handlers are designated as event listeners once the fault entries appear. In the next subsections, we discuss the fault-detection and recovery schemes adopted to achieve self-healing. We also show the dumpscreens of our proof-of-concept prototype implemented based on Autonomia architecture.

5.5.1. Fault Detection and Recovery Scheme. This implementation has the capability to handle three different kinds of faults defined as Component fault, Agent System fault and Node fault. In our environment, the ADM creates a Node Fault Handler, Component Fault Handler, and Agent Fault Handler, which register with the JavaSpaces their interest in the presence of the node fault entry, the component fault entry and the agent fault entry, respectively. The JavaSpaces will notify the appropriate fault handler when a fault entry is written in the space. Then, the fault handler will spawn a thread to recover that fault using checkpointing/migrating mechanism [25].

In Autonomia, whenever a fault is detected, the corresponding fault entry is created and is then written to the JavaSpaces. The lease of staying in the JavaSpaces for the entry is renewed by a global Lease Renewal Manager service. Any fault entry should stay in the JavaSpaces until a fault handler takes it out and deals with the fault even after their creators have crashed.

5.5.2 Component fault detection and recovery. Here we will focus on component fault detection and recovery mechanisms to illustrate the key aspect of our implementation of the self-healing property.

Our recovery scheme is based on rolling back the application execution to a previous error-free state that we refer to as a checkpoint state. In our implementation, the checkpoint data is written to JavaSpaces as a checkpoint

entry object. When a component is failed and needs to be migrated to another host. The mobile agent will not only deploy the component to the selected host, but it also downloads its related checkpoint data entry from the JavaSpaces. To enable components to migrate and resume execution on heterogeneous computing resources, we store the checkpoint data as a set of parameters. For any component to resume its execution, all what it needs are the checkpoint parameters and the executable file for the selected computing platform. Consequently, if a component needs to resume its execution on another machine, its proxy will fetch the checkpoint entry from the JavaSpaces and the checkpoint parameters in order to resume its execution from the latest checkpoint.

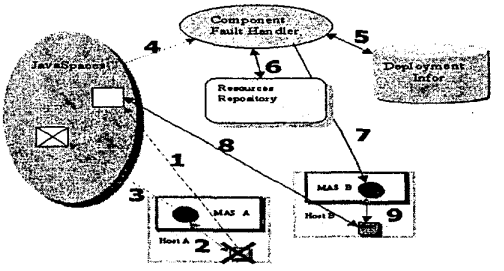

Figure 2: Component Fault Recovery

```
Create an empty vector;
boolean continueflag = true;
Do {
        Create a new transaction;
        Try to take out a component fault entry from JavaSpace under the transaction;
        if (the fault entry exists) {
                Create a new agent and set old component to it;
                for each available host {
                        successStatus = deploy agent on the host
                        if ( successStatus is true) transaction.commit(); break;
                }
                if ( successStatus is false) { add the transaction to the vector; }
        }
        else continueflag = false;
} while ( continueflag )
for each transaction in vector { trasaction.abort(); }
```

Figure 3: Component Fault Recovery Algorithm

Figure 2 shows the main steps involved in detecting and recovery from a component failure. The component periodically writes the check pointing parameters to the JavaSpaces (step 1 of Figure 2). When a component fails, the agent detects its failure by intercepting the return value from the component execution and then reports a component fault entry to the JavaSpaces (step 3 of Figure 2). The registration of a fault entry will then trigger an event message that is sent to the appropriate component fault handler (step 4 of Figure 2). The fault handler will attempt to process all the component fault entries currently stored in the JavaSpaces. To guarantee an atomic processing of the recovery procedure, it is executed as a transaction. Under the recovery transaction shown in Figure 3, the component fault handler takes out one component fault entry at a time and then creates a new mobile agent with the same agent ***ID*** in the fault entry. In steps 5 and 6, the fault handler searches the Deploy information and Resource Repository for a suitable host other than the old one and gets the appropriate proxy of the mobile agent system for the selected host. In Step 7, the agent is dispatched to the selected host. In Steps 8 and 9, the MAS on the new host will read the checkpoint information from the JavaSpaces and resumes the component execution. If the dispatch fails on the selected host, the handler interacts with all other available hosts until the recovery transaction is successful. Then the recovery transaction commits and the component fault entry is removed from the JavaSpaces. The new location of the component will be recorded in the deployment information table. If no host can accept the agent, the transaction is aborted, and

the component fault entry is left in the JavaSpaces. The self-healing procedures associated with host and agent failures follow similar steps to those shown in Figure 2.

In what follows, we show a few dump screens from our current implementation prototype of Autonomia system. In the execution environment, there are four workstations, three of them called *Catalina1, Catalina2 and Catalina3* running WindowsXP, and the fourth one called *Nimue* running RedHat Linux 7. The Agent System browser in Figure 4 shows that there are two mobile agent systems, ***alpha*** and ***beta***, that are currently running and monitoring the execution of the components on ***Catalina1*** and ***Nimue***, respectively. In this experiment, we show how Autonomia can achieve self-healing for component running a matrix multiplication (***MM***). The MM component is running on *Catalina 2*. All the component executables are stored in the Code base repository that is running an HTTP server to process client requests to download the appropriate code to any machine. The Jini services, Lookup Service, JavaSpaces, Transaction Manager Service, as well as the ADM are running on *Catalina 3*.

Then, the proxy of the ***MM*** component that is running on *Catalina1* multiplies two matrices. The on-line monitoring of that component execution is performed by the mobile agent system ***alpha***, which writes the checkpoint parameters to JavaSpaces after the multiplication of a given set of rows as shown in Figure 5. In Figure 5, we show the partial multiplication results after completing two rows of multiplications. Let us assume that after two rows of multiplication, *Catalina1* failed. The System Fault Handler will detect the system failure and it migrates the ***MM*** component to the MAS *beta* where it resumes its execution at *Nimue* (Linux) from that of the failed host (WindowsXP). Figure 6 shows that the matrix multiplication restarts on *Nimue* from where it stopped due to the host failure; i.e. it restarts execution of the matrix multiplication from row 2 rather than starting from the beginning.

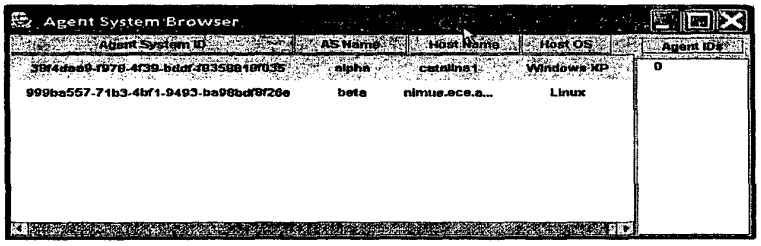

Figure 4: The Autonomia Agent System Browser.

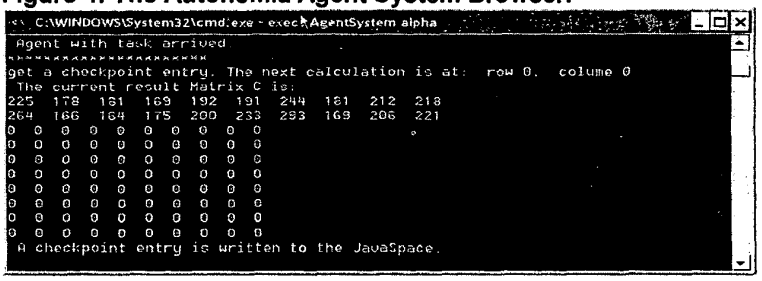

Figure 5: Mobile agent store the results after multiplying two rows.

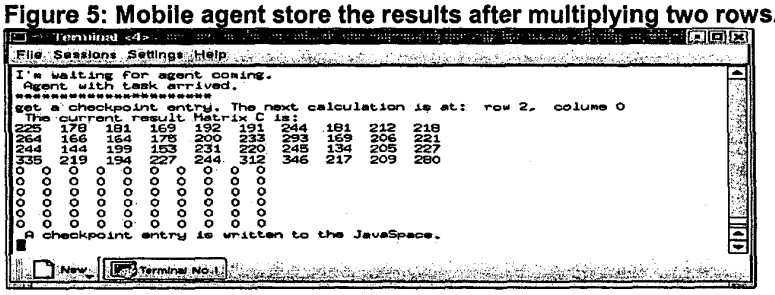

Figure 6: MM proxy resumes component execution from row 2.

6. Conclusions and Related Works

In this paper, we presented a novel architecture to implement an autonomic computing environment (Autonomia). Our implementation approach and services will make the control and management of large-scale parallel and distributed applications autonomic. We discussed in detailed our implementation approach in general to Autonomia and showed how we can achieve the self-healing when components, agents, and or systems fail. We are currently implementing other autonomic attributes to make our environment self-optimizing and self-protecting.

7. References

[1] Paul Horn, Autonomic Computing: IBM's perspective on the State of Information Technology http://researchweb.watson.ibm.com/autonomic/

[2] Autonomic Distributed Computing in Scientific Applications. International Workshop on Future Directions in Distributed Computing. 3-7 June 2002, Bertinoro, Italy.

[3] http://www.research.ibm.com/autonomic.

[4] Gail Kaiser, Phil Gross, Gaurav Kc, Janak Parekh, Giuseppe Valetto: An Approach to Autonomizing Legacy Systems; IBM Almaden Institute Symposium, 4/2002

[5] David Patterson, Aaron Brown, et al, Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies; Computer Science Technical Report UCB//CSD-02-1175, U.C. Berkeley March 15, 2002

[6] Salim Hariri, C.S. Raghavendra, Yonhee Kim, Rinda P. Nellipudi, et al; CATALINA: A Smart Application Control and Management.Active Middleware Services Conference, 2000.

[7] Brown, A. and D. A. Patterson. Embracing Failure: A Case for Recovery-Oriented Computing (ROC). 2001 High Performance Transaction Processing Symposium, Asilomar, CA, October 2001.

[8] S. Hariri, Y. Kim, M. Djunaedi. Design and Analysis of a Proactive Application Management System. Proc. of NOMS2000; April 2000.

[9] R. Koo, S. Toueg. Checkpointing and Recovery-Rollback for Distributed Systems. IEEE Transactions on Software Engineering; Vol. SE-13, No. 1; pp. 23-31; 1987.

[10] M. A. Iverson, F. Ozguner, L. C. Potter. Statistical Prediction of Task Execution Times through Analytic Benchmarking for Scheduling in a Heterogeneous Environment. Eighth Heterogeneous Computing Workshop (HCW'99).

[11] Michael O. Neary, Bernd O. Christiansen, Peter Cappello, and Klaus E. Schauser: Javelin: Parallel Computing on the Internet. Future Generation Computer Systems. October 1999.

[12] Hiromitsu Takagi, S. Matsuoka, H. Nakada, et al, Ninflet: A Migratable Parallel Objects Framework using Java, In proc. of the ACM 1998 Workshop on Java for High Performance Network Computing, 1998.

[13] Michael O. Neary, Alan Phipps, Steven Richman, and Peter Cappello, Javelin 2.0: Java-Based Parallel Computing on the Internet, In Euro-Par 2000.

[14] Adam J. Ferrari, Process Introspection: A Checkpoint Mechanism for High Performance Heterogeneous Distributed Systems. Technical Report CS-96-15, Department of Computer Science, University of Virginia, Charlottesville, VA, October 10, 1996.

[15] Nathalie Furmento, Anthony Mayer, Stephen McGough, et al, Optimisation of Task-based Applications within a Grid Environment, SuperComuting 2001

[16] R. Armstrong, D. Gannon, A. Geist, et al, Toward a Common Component Architecture for High-Performance Scientific Computing. In Proc. of the 8th High Performance Distributed Computing, 1999.

[17] K.A.Hawick and H.A.James, Dynamic Cluster Configuration and Management using JavaSpaces, 2001 IEEE International Conference on Cluster Computing, 2001.

[18] http://java.sun.com/products/jms/

[19]http://java.sun.com/products/jini/

[20] Luis F. G. Sarmenta, Satoshi Hirano, Bayanihan: Building and Studying Web-Based Volunteer
Computing Systems Using Java.

[21] F.Cristian, Understanding Fault Tolerant Distributed System, Communication on ACM, vol34, 1991.

[22] A. Baratloo, M. Karaul, Z. Kedem, P. Wyckoff, Charlotte: Metacomputing on the Web. In proc. Of the
9th International Conference on Parallel and Distributed Computing Systems, 1996.

[23] Hiromitsu Takagi, S. Matsuoka, H. Nakada, et al, Ninflet: A Migratable Parallel Objects Framework

[24] Michael O. Neary, Alan Phipps, Steven Richman, and Peter Cappello, Javelin 2.0: Java-Based Parallel Computing on the Internet, In Euro-Par 2000

[25] A.Beitz, S.Kent, and P.Roe. Optimizing Heterogeneous Component Migration in the Gardens Virtual Cluster Computer. In Heterogeneous Computing Workshop, May 2000.

[26] A. Baratloo, M. Karaul, H. Karl, Zvi M. Kedem. An Infrastructure for Network Computing with Java Applets. In Proceedings of ACM workshop on Java for High Performance Network Computing, February 1998.

[27] H. Topcuoglu, S. hariri, D. Kim, Y. Kim, X. Bing, B. Ye, I. Ra, J. Valente, The Design and Evaluation of a Virtual Distributed Computing Environment", The Journal of Networks, Software Tools and Applications(Cluster Computing), 1998.