Self-Deployment and Self-Configuration of Network Centric Service

Huoping Chen, Salim Hariri, Byoung Kim, Ming Zhang, Yeliang Zhang, Bithika Kharghoria
Autonomic Computing Laboratory
Department of Electrical and Computer Engineering
University of Arizona
1230 E. Speedway, Tucson, AZ 85721
{hpchen, hariri, byoung, mingz, zhang, bithika_k}@ece.arizona.edu

Manish Parashar
Applied Software Systems Laboratory
Department of Electrical and Computer Engineering, Rutgers University
94 Brett Road, Piscataway, NJ 08854
parashar@caip.rutgers.edu

Abstract

With the rapid increase in complexity of software systems and applications, it becomes necessary 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.

In this paper, we present policy engine based approach to achieve self-deployment and self-configuration in our Autonomia Environment. Through decomposition and expressing the deployment and configuration operations of network centric applications into policies, Autonomia makes the deployment and configuration automated.

We describe an distributed application example of Linear Equation Solver on Globus environment and demonstrate that the application can be deployed autonomically and reconfigured on the fly without humane intervention by using policy-based Self-Deployment and Self-Configuration Services.

1. Introduction

The control and management of large-scale network centric distributed applications over a heterogeneous and dynamic execution environment present challenging research problems. These challenges are caused due to the complexity of application interactions that involve a wide range of heterogeneous resources and networks. We have reached a complexity and size that make our current network management tools incapable of controlling and managing such systems and their applications.

A possible solution to these challenging research problems can be inherited from biological system (autonomic nervous systems), which has been very successful in controlling and managing complex, interactive, and constrained systems. We refer to this approach by autonomic computing [1] that calls for designing distributed systems that can automatically configure, deploy, secure, tolerate faults, optimize, and anticipate loads by themselves without the manual involvement of human administrators.

There are several projects [2,3,4,5] that are aimed at developing self-managing system. [2] shows building autonomic computing systems based on ontological component models and a controller synthesis algorithm. Software Architecture-based Adaptation framework is described in [3]. IBM Neptune project [4] proposed a dynamic resource allocation and planning system for a Cluster Computing Utility. Other research projects like Recovery Oriented Computing (ROC) have emphasized the recovery and reliability of systems rather than performance [5], 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 policy engine based approach to achieve self-deployment and self-configuration for network centric applications. We have successfully implemented a proof-of-concept prototype that implements important properties of autonomic systems: self-configuring, self-deploying, self-optimizing and self-healing based on policy engine.

The remaining sections of the paper are organized as follows. Section 2 introduces our implementation architecture of Autonemia in briefly. In section 3,
detailed design and implementation of self-deployment and self-configuration based on policy engine are described. Some experimental results are given in section 4 and the paper is concluded in section 5.

2. Autonomia: An Autonomic control and management environment

The Autonomia environment [6] provides application developers with all the tools required to specify the appropriate control and management schemes, the services deploy and configure the required software and hardware resources, run application, provide on-line monitoring and management to maintain the desired autonomic attributes of applications as well as system services. The architecture of Autonomia is shown in Figure 1. The main modules include Application Management Editor (AME), Autonomic Middleware Services (AMS) and Application Delegated Manager (ADM). The AME enables users to develop applications with the ability to specify the control and management policies associated with each application task. The AMS provides common middleware services and tools needed by applications and systems to automate operation. The ADM main focus is on setting up the application execution environment and then maintaining its requirements at runtime. In what follows, we describe it in briefly.

2.1 Application Management Editor (AME)

The AME is a graphical user interface for developing an application with pre-developed components and specifying management requirements to configure and manage the execution of the application. The main functions include controlling the application editor workplace, composing the components, and storing the application management requirements in the Application Service Template in the Application Information Knowledge (AIK) repository.

2.2 Autonomic Middleware Service (AMS)

AMS can be viewed as an operating system to provide applications with all the services and tools required to achieve the desired autonomic requirements. The main modules are:

- **Component Repository (CR)**
  Component Repository contains a collection of components that are currently available for the users to develop their applications.

- **Resource Repository (RR)**

  Resource Repository is the collection system resources, which can be added or discovered in a similar approach to the component publishing and discovering techniques.

  - **Application Information and Knowledge (AIK) Repository**
  AIK repository is to store application information status, knowledge about optimal management strategies that have proven to be successful and effective, as well as pre-specified management policies.

  - **Event Server**
  Event server receives status of any monitored attributes and then notifies the corresponding engines that subscribe to these events once they become true.

  - **Policy Engine and Autonomic Services based on it**
  Policy engine is to parse the pre-specified rule and formulate action plan. We will explore it in detail at section 4.

2.3 Monitoring Service

There are two kinds of monitoring services: Resource Monitoring Service and Component Monitoring Service. They will be described in Section 3.2.1 and 3.2.2

2.4 Application Delegated Manger (ADM)

ADM is to allocate the appropriate resources to run the application and maintain the application requirements at runtime. It communicates with MAS.

2.5 Mobile Agent System (MAS)

The Mobile Agent System is designed to provide mobile agents [10] a uniform execution environment independent of the underlying hardware architecture and operating system.
3. Design of Self-Deployment and Self-Configuration Service

Figure 1 shows the main modules of our architecture to implement self-deployment and self-configuration. Our approach is based on two engines: Policy Engine and Action Engine. In this paper, we will focus on the policy engine and how self-deployment and self-configuration achieved. See [6] for detailed description of Action Engine, which is composed by Application Delegated Manager and Mobile Agent System.

We will describe the policy engine in detail in section 3.1. Section 3.2 introduces the monitoring services. We will investigate how all modules interact to achieve self-deployment and self-configuration in section 3.3 and section 3.4.

3.1 Policy Engine

Policy engine is to evaluate the conditions pre-specified and formulate one or more actions to be executed when the conditions are satisfied. The policy is in (IF conditions, THEN actions) format, for example, “IF CPU load is HIGH or response time is HIGH or memory consumption is HIGH, then stop the component C1 on Linux platform”. It is the first policy shown in Figure 2. The second policy specifies “IF component C2 and component C3 are all finished, THEN launch C4 on a Windows XP platform”. Finally, the third one expresses “IF component C4 is terminated anomaly OR response time is HIGH, THEN reallocate the component on server1”.

<table>
<thead>
<tr>
<th></th>
<th>IF CPU load is HIGH or response time is HIGH or memory consumption is HIGH, THEN stop the component C1 on Linux platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>IF component C2 and component C3 are all finished, THEN launch C4 on a Windows XP platform</td>
</tr>
<tr>
<td>3</td>
<td>IF component C4 is terminated anomaly OR response time is HIGH, THEN reallocate the component on server1</td>
</tr>
</tbody>
</table>

Figure 2: Policies expressions

3.1.1 Design of Policy Engine

Policy Engine receives the events from Event Server, compares the monitored data with the policy conditions, determines if the conditions are satisfied and formulates the action plan to be taken. Currently policy engine supports five types of events: self-deployment, self-configuration, self-optimization, self-healing and self-protection. Each different event contains event type, status, state and performance metrics data. For example, a deployment event uses the following parameters:

- Engine Type
- State of Component
- Component Name (Execution Time etc.)
- Host Name (OS, Memory, CPU data etc.)

Policy Engine supports 1-to-1, many-to-1, 1-to-many and many-to-many relationships between conditions and actions. It allows different patterns [11] such as AND, OR and XOR between multiple conditions. For the multiple actions, it supports only AND operation now.

Each policy is composed by three parts: conditions, logic pattern and actions, for example, the policy statement “IF CPU load is HIGH or response time is HIGH or memory consumption is HIGH, THEN stop the component C1 on Linux platform” can be parsed into Three conditions:

- CPU load is HIGH
- response time is HIGH
- memory consumption is HIGH

One Logic Pattern: OR
One Action: Stopping C1 on Linux platform

In addition to supporting the rule expression format, Autonomia provides a graphic interface (AME) to specify control and management policies which will translate the policy information into the Autonomia format and store them into database when users save the diagram.

Figure 3 shows how policy engine parse the policy, evaluate the conditions and formulate the actions.

Policy engine waits for the events from event server. If event comes, it will analyze the event and acquire the event type, machine and component name, monitored attributes and real monitored data. When finishing the analysis, policy engine will look for the pre-specified polices which have the same attributes.

Policy engine can get the pattern for all conditions from the policy. It then compares all the conditions of the policies with the data from events with the threshold specified in policies and evaluates if the conditions are triggered under the specified pattern.

If the conditions are satisfied, the policy engine will choose actions to be taken and attributes required from action table and then assign necessary resource for it. After validating the resource allocation (for example server, memory requirements etc.), the action request is sent to MAS through ADM.
Loop{
  Wait for Event;
  If (event comes){
    Parse Event to get event type, attributes and monitored data
    Find the corresponding policies by the attributes
    Get conditions’ pattern (OR, AND etc.) and Check If the logic is satisfied
    If the conditions are satisfied between the condition thresholds and real monitored data{
      Formulate action plan and get the action ID from the policies
      Read Action plan and attributes by Action ID
      Check the required attributes
      Allocate the necessary resource
      Send Action Request to Mobile Agent System through ADM
    }
  }
}

Figure 3: Policy Engine Algorithm

3.1.2 Data Model of Policy Engine

Autonomia creates a data model to implement policy engine shown in Figure 4. It is composed of 4 tables: Policy, Action_Desc, SysInfo and Component. The **policy** table stores the rules which contain conditions attributes and desirable action IDs.

The **Action_Desc** table stores the detailed information for a specific action ID which referred by Policy table. Each action ID may be composed of no, single or multiple activities which identified as activity ID.

The **SysInfo** table record the system level resource status data such as host name, IP address, physical disk, CPU utilization percentage etc, which is used as Resource Repository.

The **Component** table is used as Component Repository to store the status of component, performance metrics data, and the code base location.

One data model example will be described in section 3.3.

Figure 4: Policy Engine Data Model

3.2 Monitoring Services

3.2.1 Resource Monitoring Service (RMS)

The RMS is to provide the workload information and performance metrics data at system level. The monitored information includes three main parts: CPU/memory information, disk IO information and network information. The CPU information will basically provide CPU utilization information and total and free physical memory as well as total and free swap file size. The disk information includes the total bytes read and written to all attached disks on the host. The network information will give the statistics of the each network interface such as total packets in and out the interface. From this information, the total workload could then be estimated on each host for determining the load balance or other technical strategy. If system metric thresholds specified in policies are violated, the service generates an event and sends it to Autonomia Event Server.

3.2.2 Component Interface and Component Monitoring Service (CMS)

We define a general interface for the component that will be running on Autonomia Environment. It’s consisted of a set of get and set functions. Get functions are used by Monitoring service and set functions are used by component itself or Actuator (MAS). Through the interface, the component can expose its status data of execution time, current state.

The monitoring service collects the status and state information of components, execution time data, then analyzes its behavior. When state change happens (for example, component change from active to finish) or anomalies happens, it will generate events and notify the event server.
3.3 Implementation of Self-Deployment through policy engine

3.3.1 Decomposition of application

Self-Deployment can be achieved by decomposing an application into some interdependent components and expressing the dependencies in Autonomia policy format.

One dependency between A and B can be expressed as:

IF state of component A is changed from active to finish on machine M1, THEN launch Component B on Machine M2 with OS Windows XP.

The conditions are state change of component A, and the action is to launch component B. The required attributes for component B are: machine name is M2 and OS is Windows.

![Figure 5: Application Dependencies](image)

For example, Figure 5 shows another example of an application which is composed of four components C1, C2, C3 and C4, and there are some dependencies among them and each has some required attributes.

C4 depends on the following 2 components: C2 and C3. C2 and C3 depend on C1. There are also some attributes requirements: C1 and C4 are to be started on machine Host1, C2 needs running on machine Host 2 while C3 needs Host3.

1. IF initial,
   THEN launch C1 on Host1 with OS Windows XP.
2. IF C1 finishes,
   THEN launch C2 on Host2 with OS Windows XP AND C3 on Host3 with OS Linux.
3. IF C2 finishes AND C3 finishes,
   THEN launch C4 on Host1 with OS Windows XP.

![Figure 6: Policy expressions for the self-deployment example](image)

Their dependencies can be expressed as three policies as shown in Figure 6. Policy 1 specifies that under initial condition, launch Component C1 on machine Host1. The second policy indicates that when C1 finishes, C2 starts on Host2 and C3 on Host3. Finally, the third policy shows that when C2 and C3 are all done, C4 starts on Host1 again.

Figure 7 shows the data model implementation of the policies in Figure 6.

![Figure 7: Data Model for self-deployment example](image)

For policy_id 3 in Policy table, there are two conditions: conditionID 1 and 2. ConditionID 1 is defined as Component C2 finishing (value 2) on Host2, and conditionID 2 is defined as Component C3 finishing on Host3. When both these two conditions are satisfied, action plan is formulated with Action ID 3. For actionID 3 in Action_Desc table, it shows that the action plan is to launch component C4 on Host1 (activityType means the action type, here 1 is to start a new component).
3.3.2 Self-Deployment Algorithm

When all the dependencies and required attributes are expressed in policies and stored into AIK, Autonomia will take the following steps to achieve self-deployment as shown in Figure 8:

1. **Step 1**: Application dependencies and required attributes are specified by user
2. **Step 2**: Policy Engine reads the policies from AIK repository
3. **Step 3**: Initialize self-deployment service
4. **Step 4**: Self-deployment checks the attributes component required by getting data from Resource Repository (RR) and Component Repository (CR) according to the policies
5. **Step 5**: Policy engine formulates action plan and sends the action requirement to ADM
6. **Step 6**: ADM checks the available agent and asks MAS to take the action on local node
7. **Step 7**: MAS initializes the local monitoring services
8. **Step 8**: 8.1 Monitoring services store the normal status data into AIK repository
   8.2 When state change of one component happens, send an event to event server
9. **Step 9**: Event server notifies policy engine, then continues from Step 3

![Figure 8: Self-Deployment Algorithm](image)

### 3.4 Implementation of Self-Configuration through policy engine

Self-Configuration is achieved by expressing configuration conditions and actions into policies. The configuration policy can be expressed like:

**IF threshold is violated or status change of host OR component, THEN reallocate the resource or component.**

The conditions are threshold violation or status change of the components or hosts, and the action is to reallocate system resource or reallocate components.

For example, the management for the crash of component1 can be expressed in Autonomia policy format in “**IF component1 is crashed on machine catalina2, THEN resume its execution on machine catalina3**”. The self-configuration service will resume its execution on machine catalina3 when policy engine receives an event that Component1 terminates abnormally on machine catalina2.

To achieve self-configuration, Autonomia takes similar steps as it does for self-deployment.

### 4. Experiment

In this section, we will demonstrate the execution of the Linear Equation Solver (LES) application on Autonomia environment and show how it achieves the self configuration and self deployment objectives. The problem size for this experiment is 1024*1024 and it runs over ten node high-performance cluster at the University of Arizona. The Globus 2.0 [12] and MPICH-G2 [13] are also used for this experiment.

#### 4.1 self configuration and self deployment of LES application

In this experiment we used LES application as a running example to demonstrate how Autonomia self configuration and self deployment services autonomically deploy and reconfigure the application.

The LES workflow and the control and management properties can be specified by AME as shown in Figure 9.

In this application, the problem is to find the solution vector x in an equation Ax = B, where A is a known N*N matrix and B is a known vector. The Linear Equation Solver can be solved based on Gauss-Jordan elimination, Gaussian Elimination with back-substitution, or LU Decomposition. With LU Decomposition, any matrix can be decomposed into the product of a lower triangular matrix L and upper triangular matrix U. Once LU Decomposition is solved, the solution vector is derived. Based on the
workflow, the order of execution is Level 1, Level 2, Level 3, and Level 4. The Autonomia provides a user friendly interface that allows a novice to make an experiment with and evaluate each Autonomia task. Using Autonomia prototype, once a task (e.g. LES) is registered in the Autonomia component repository, the user can use that task or any other repository tasks by just clicking on the task name in the Application Management Editor window. Once the task is selected, the user can specify the desirable configuration to run the selected task as already explained. Selecting the Autonomia task and specifying how it will be implemented can be done in a few minutes. Once that is done, AME will parse the diagrams and generate the policies for self-deployment, self-configuration etc. When starting the LES application, its execution status can be visualized immediately.

![Figure 9: Workflow, Control and Management Properties of Linear Equation Solver Application](image)

The dependency relationships among components and the control and management attributes and policies of each component is depicted in Figure 9. LES application is composed of five components and there are some dependencies among them and each has some required attributes. MULT2 depends on MULT1. MULT1 depends on the following two components: INV1 and INV2. INV1 and INV2 depend on LU.

There are also some attributes requirements: The machine name on which each component is started, how many numbers of nodes for each component, and the input/output files.

The component dependencies and attribute requirements will be expressed as Autonomia policies shown in Figure 10.

```
1  IF initial, THEN launch LU on 4 nodes from grid1.ece.arizona.edu with INPUT FILE A
2  IF LU finishes, THEN launch INV1 on 10 nodes from grid1.ece.arizona.edu AND launch INV2 on 10 nodes from grid1.ece.arizona.edu
3  IF INV1 finishes, THEN launch MULT1 on 8 nodes from grid1.ece.arizona.edu with INPUT FILE B
4  IF MULT1 finishes AND INV2 finishes, THEN launch MULT2 on 4 nodes from grid1.ece.arizona.edu with OUTPUT FILE X
```

![Figure 10: LES Self-Deployment Policies](image)

As shown in Figure 9, the configuration policy: “IF MULT1 is crashed on machine grid1.ece.arizona.edu, THEN resume its execution on second machine grid2.ece.arizona.edu” specifies that when MULT1 crashes, the execution of MULT1 will be migrated to grid2.ece.arizona.edu.

When all those information are saved, AME will parse the diagram and store the policies, knowledge about actions into AIK repository.

4.2 Results

Figure 11 shows the visualization of execution status monitoring in LES task at each level. We observed that the application workflow is exactly the same as we specified in Figure 9. Through the policy engine, our Autonomia environment makes the deployment and configuration of LES application autonomic without any humane intervention.
5. Conclusion

We have presented the detail design and implementation of self-deployment and self-configuration services through policy engine in Autonomia environment. Our implementation approach will lead to automated control and management of a wide range of network centric applications and services. We have implemented a proof-of-concept prototype of Autonomia, built up self-deployment and self-configuration services based on policy engine. The experiment has demonstrated successfully that it can be implemented in grid computing environment and other distributed computing environment. Currently, we are incorporating self-optimization into our environment and conducting experimental tests on Globus and CCA environment.

References

[13]. http://www3.niu.edu/mpi/
[19]. Michael O. Neary, Alan Phipps, Steven Richman, and Peter Cappello, Javelin 2.0: Java-Based Parallel Computing on the Internet, In Euro-Par 2000
[20]. Gail Kaiser, Phil Gross, Gaurav Kc, Janak Parekh, Giuseppe Valetto: An Approach to Autonomizing Legacy Systems; IBM Almaden Institute Symposium, 4/