Enabling Autonomic, Self-managing Grid Applications *

Z. Li, H. Liu, and M. Parashar
The Applied Software Systems Laboratory
Dept of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ, 08854, USA
Email:{zhljenny,marialiu,parashar}@caip.rutgers.edu

1 Introduction

The emergence of pervasive wide-area distributed computing environments, such as pervasive information systems and computational Grid, has enabled a new generation of applications that are based on seamless access, aggregation and interaction. For example, it is possible to conceive a new generation of scientific and engineering simulations of complex physical phenomena that symbiotically and opportunistically combine computations, experiments, observations, and real-time data, and can provide important insights into complex systems such as interacting black holes and neutron stars, formations of galaxies, and subsurface flows in oil reservoirs and aquifers etc. Other examples include pervasive applications that leverage the pervasive information Grid to continuously manage, adapt, and optimize our living context , crisis management applications that use pervasive conventional and unconventional information for crisis prevention and response, medical applications that use in-vivo and in-vitro sensors and actuators for patient management, and business applications that use anytime-anywhere information access to optimize profits.

However, the underlying Grid computing environment is inherently large, complex, heterogeneous and dynamic, globally aggregating large numbers of independent computing and communication resources, data stores and sensor networks. Furthermore, these emerging applications are similarly complex and highly dynamic in their behaviors and interactions. Together, these characteristics result in application development, configuration and management complexities that break current paradigms based on passive components and static compositions. Clearly, there is a need for a fundamental change in how these applications are developed and managed. This has led researchers to consider alternative programming paradigms and management techniques that are based on strategies used by biological systems to deal with complexity, dynamism, heterogeneity and uncertainty. The approach, referred to as autonomic computing, aims at realizing computing systems and applications capable of managing themselves with minimum human intervention.

An autonomic self-managing application can be viewed as a collection of autonomic components that can manage their internal behaviors and their relationships and interactions with other components and the system using high-level policies. Achieving autonomic self-managing behaviors requires programming and middleware support for context and self awareness, knowledge and context based analysis and planning, and plan selection and execution.

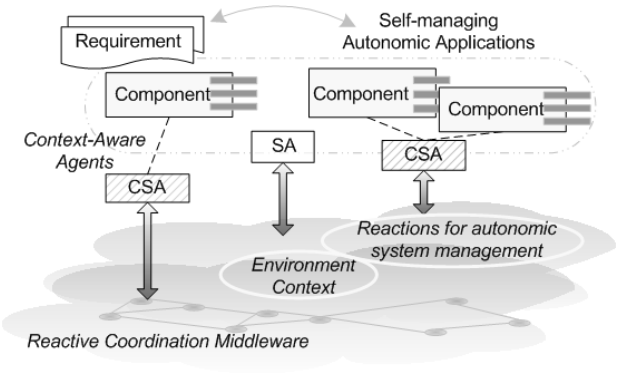

Figure 1. A conceptual overview of system architecture.

In this paper, we present an environment that supports the development of self-managed autonomic components, dynamic and opportunistic composition of these components using high-level policies to realize autonomic applications, and provides runtime services for policy definition, deployment and execution. A conceptual overview of the environment is illustrated in Figure 1. It builds on the following basic concepts: (1) a separation of computations from coordination and interactions; (2) a separation of

*The research presented in this paper is supported in part by the National Science Foundation via grants numbers ACI 9984357 (CAREERS), EIA 0103674 (NGS), EIA-0120934 (ITR), ANI-0335244 (NRT), CNS-0305495 (NGS) and by DOE ASCI/ASAP (Caltech) via grant number 82-1052856.

non-functional aspects (e.g. resource requirements, performance) from functional behaviors, and (3) a separation of policy and mechanism where policies are used to orchestrate a repertoire of mechanisms to achieve context-aware adaptive runtime computational behaviors and interaction and coordination relationships based on functional, performance, and QoS requirements.

The environment consists of the Accord programming framework and the Rudder agent-based middleware infrastructure. These systems are key parts of Project AutoMate [1] aimed at investigating fundamental issues in enabling autonomic applications. The rest of this paper is organized as follows. Section 2 describes the Accord framework. Section 3 presents an overview of the design and architecture of Rudder. Section 4 illustrates the operation of the system using an autonomic oil reservoir optimization application. Section 5 presents some concluding remarks.

2 Accord, A Programming Framework for Autonomic Applications

Accord programming framework [4] consists of 4 concepts. The first is an application context that defines a common semantic basis for components and the application. The second is the definition of autonomic components as the basic building blocks for autonomic application. The next is the definition of rules and mechanisms for the management and dynamic composition of autonomic components. And the final is rule enforcement to enable autonomic application behaviors.

Application context: Autonomic components should agree on common semantics for defining and describing application namespaces, and component interfaces, sensors and actuators. Using such a common context allows definition of rules for autonomic management of components and dynamic composition and interactions between the components. In Accord, functional and non-functional aspects of components are described using an XML-based language.

Autonomic Component: An autonomic component is the fundamental building block for autonomic application. It extends the traditional definition of components to define a self-contained modular software unit of composition with specified interfaces and explicit context dependencies. Additionally, an autonomic component encapsulates rules, constraints and mechanisms for self-management and dynamically interacts with other components and the system. An autonomic component shown in Figure 2 is defined by 3 ports:

- A *functional port* that defines the functional behaviors provided and used by the component;
- A *control port* that defines a set of sensors and actuators exported by the component for external monitoring and control, and a constraint set that defines the access to and operation of the sensors/actuators based on state, context and/or high-level access policies;
- An *operational port* that defines the interfaces to formulate, inject and manage the rules which are used to manage the component runtime behaviors.

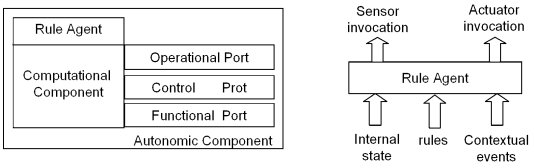

Figure 2. An autonomic component.

These aspects enhance component interfaces to export information and policies about their behaviors, resource requirements, performance, interactivity and adaptability to system and application dynamics. An embedded rule agent monitors the component's state and controls the execution of rules. Rule agents cooperate across application compositions to fulfill overall application objectives.

Rule Definition: Rules incorporate high-level guidance and practical human knowledge in the form of an IF-THEN expression. The condition part of a rule is a logical combination of component/environment sensors and events. The action part of a rule consists of a sequence of component/system sensor/actuator invocations. A rule fires when its condition expression evaluates to be true and the corresponding actions are executed. Two class of rules are defined: (1) *Behavioral rules* that control the runtime functional behaviors of an autonomic component (e.g., the dynamic selection of algorithms, data representation, input/output format used by the component). (2) *Interaction rules* that control the interactions between components, between components and their environment, and the coordination within an autonomic application (e.g., communication mechanism, composition and coordination of the components).

Rule Enforcement: Rules are injected into components at run time and enable autonomic application self-managing behavior. Behavioral rules are executed by a rule agent embedded within a single component without affecting other components. Interaction rules define interactions among components. For each interaction pattern, a set of interaction rules are defined and dynamically injected into the interacting components. The coordinated execution of these rules result in the realization of interaction and coordination behaviors between the components.

3 Rudder, An Agent-based Middleware Infrastructure

Rudder [3] is an agent-based middleware infrastructure for autonomic Grid applications. The goal of Rudder is to provide the core capabilities for supporting autonomic compositions, adaptations, and optimizations. Specifically, Rudder employs context-aware software agents and a decentralized tuple space coordination model to enable context and self awareness, application monitoring and analysis, and policy definition and its distributed execution. Rudder effectively supports the Accord programming framework and enables self-managing autonomic applications. The overall architecture of Rudder is shown in Figure 1. It builds on two concepts:

- Context-aware agents can control, compose and manage autonomic components, monitor and analyze system runtime state, sense changes in environment and application requirements, dynamically define and enforce rules to locally enable component self-managing behaviors.
- A robust decentralized reactive tuple space can scalably and reliably support distributed agent coordination. It provides mechanisms for deploying and routing rules, decomposing and distributing them to relevant agents, and enabling self-managing applications by coordinating rule execution.

Agent framework: The Rudder agent framework consists of three types of peer agent: Component Agent (CA), System Agent(SA), and Composition Agent(CSA). Component agents and system agents exist as system services, while composition agents are transient and are generated to satisfy specific application requirements. Component agents define interaction rules to specify component interaction/communication behaviors and mechanisms (e.g., synchronous/asynchronous, broadcast, or multi-cast). They are integrated with component rule agents to provide components with uniform access to middleware services, control their functional and interaction behaviors and manage their life cycles (e.g., dynamically configure and control the data exchange between components using interaction rules). System agents monitor, schedule and adaptively optimize physical resource utilization, such as CPU and disk. They are embedded within Grid resource units (e.g., computer, cluster, data archive). Agents exist at different levels of the system and represent their collective behaviors.

Compositions agents enable dynamic composition of Accord autonomic components by defining and executing workflow-selection and component-selection rules. Workflow-selection rules are used to select appropriate composition plans to enact, which are then inserted as reactive tuples into the tuple space. Component-selection rules are used to semantically discover, and select registered components, allowing tasks to optimize the execution and tolerate some failure in components, connections, and hosts. Composition agents negotiate to decide interaction patterns for a specific application workflow and coordinate with the associated component agents to execute the interaction-rules at runtime. This enables autonomic applications to dynamically change flows, components and component interactions to address application and system dynamics and uncertainty.

Reactive tuple space: The Rudder decentralized reactive tuple space provides the coordination service for distributed agents, and mechanisms for rule definition, deployment and enforcement. Rudder extends the traditional tuple space with a distributed, guaranteed and flexible content-based matching engine and reactive semantics to enable global coordination in dynamic and ad hoc agent communities. Runtime adaptive polices defined by the context-aware agents (e.g., composition rules from a dynamically selected workflow plan or rules for dynamic resource reallocation) can be inserted and executed using reactive tuples, to achieve coordinated application execution and optimized computational resource allocation and utilization.

The Rudder tuple space builds on a resilient self-organizing peer-to-peer content-based overlay, and supports the definition and execution of coordination policies through programmable reactive behaviors. These behaviors are dynamically defined using stateful reactive tuples. A reactive tuple consists of three parts: *Condition* associates reactions to triggering events. *Reaction* specifies the computation associated with the tuple's reactive behavior. *Guard* defines the execution semantics of the reactive behavior (e.g., immediately and once). Polices and constraints dynamically defined by administrators or agents can be triggered and executed to satisfy specific application requirements (e.g., prevent undesired malicious operations to defend system integrity).

The Rudder middleware infrastructure employs autonomous agent based management and the reactive tuple space coordination model to enable self-managing autonomic applications. This architecture can effectively support the autonomic behaviors as follows: **Self-configuration** is enabled through dynamic discovery and composition of new components and component reconfiguration at run time; **Self-optimization** is enabled through dynamic switching of workflows and components using composition rules, balancing of workload and resource utilization, and definition of component interaction patterns; **Self-healing** is enabled by restarting or replacing failed components; **Self-protection** is enabled through reactions defined to defend the system integrity from undesired operations of malicious agents, thus preventing the loss of data, tasks or services.

4 Autonomic Oil Reservoir Optimization: An Illustrative Application

The operation of the environment supporting autonomic self-managing applications presented in this paper is illustrated using an Autonomic Oil Reservoir Optimization application (AORO) [5] (see Figure 3). The goal of AORO is to maximize revenue from an oil field by optimizing the placement and configurations of oil wells.

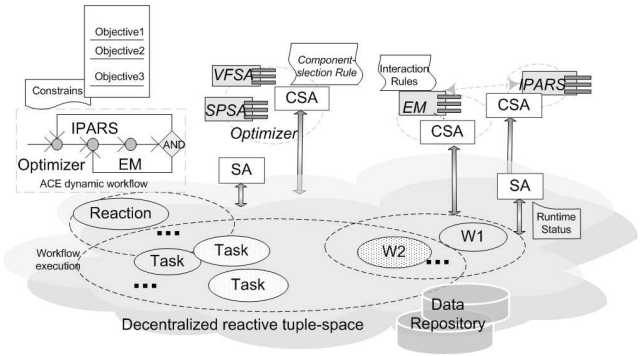

Figure 3. Autonomic composition and execution of AORO.

In this application, an autonomic composition engine (ACE) [2] generates application workflows to satisfy application objectives. The dynamically defined workflows include: (1) the optimization service provides the IPARS reservoir simulator with an initial guess of well parameters based on the configuration of the oil field; (2) IPARS uses the well parameters along with current market parameters to periodically compute the current revenue using an Economic Model (EM) service; and (3) IPARS iteratively interacts with the optimization service to optimize well parameters for maximum profit.

Based on these workflows, three composition agents are instantiated for the EM, Optimizer, and IPARS respectively. The CSAs dynamically discover the appropriate autonomic components with desired functionality and cost/performance property using the AutoMate discovery service [1], and configures these components using interaction rules. The CSAs then coordinate with the CAs associated with these components using the decentralized tuple-space to accomplish the oil reservoir optimization process. According to the interaction rules, component agents will dynamically establish interaction relationships among these components and establish communication relationships with different semantics. During execution, autonomic component are supported by system agents, which monitor and manage system operation.

Autonomic optimization in the application is achieved via the autonomous behaviors of the agents. The agents adaptively configure and compose components, tune system parameters to achieve application goals. Each CA monitors and manages the execution of its component, while the CSAs proactively search for available components and resources to satisfy current application objectives. The use of redundant or replicated components also allows tasks to tolerate some failure in components, connections, and hosts. For example, the Optimizer CSA selects the different optimization components, currently VSFA and SPFA, and configures them to optimize the application according to the current objectives of the application. Similarly, the SAs monitor the runtime utilization of the resource and dynamically balance workload.

5 Conclusion

The autonomic Grid application environment presented in this paper is based on fundamental innovations in the formulation, deployment, execution and optimization of Grid autonomic applications. Contributions include (1) the Accord programming framework that enables the definition of autonomic components capable of managing their runtime behaviors based on the current application states and requirements as well as environment context, and (2) Rudder, a middleware infrastructure that provides coordination service to support autonomic composition, interaction, and management of the components. A key component of Rudder is a decentralized reactive tuple space that enables scalable and reliable global coordination in Grid environments. Prototype implementations of Accord and Rudder are underway as parts of Project AutoMate.

References

[1] M. Agarwal and et all. Automate: Enabling autonomic applications on the grid. In *Proceedings of the 5th Annual International Active Middleware Services Workshop*, Seattle, WA, 2003.

[2] M. Agarwal and M. Parashar. Enabling autonomic compositions in grid environments. In *Proceedings of the 4th International Workshop on Grid Computing*, Phoenix, AZ, 2003.

[3] Z. Li and M. Parashar. Rudder: A rule-based multi-agent infrastructure for supporting autonomic grid applications. In *Proceedings of the International Conference on Autonomic Computing*, New York, NY, 2004.

[4] H. Liu and M. Parashar. A component based programming framework for autonomic applications. In *Proceedings of the International Conference on Autonomic Computing*, New York, NY, 2004.

[5] V. Matossian and M. Parashar. Autonomic optimization of an oil reservoir using decentralized services. In *Proceedings of the 1st International Workshop on Heterogeneous and Adaptive Computing*, Seattle, WA, 2003.