Enabling Autonomic Grid Applications: Dynamic Composition, Coordination and Interaction*

Zhen Li and Manish Parashar

The Applied Software Systems Laboratory,
Rutgers University, Piscataway NJ 08904, USA
{zhljenny, parashar}@caip.rutgers.edu

Abstract. The increasing complexity, heterogeneity and dynamism of networks, systems and applications have made our computational and information infrastructure brittle, unmanageable and insecure. This has necessitated the investigation of an alternate paradigm for system and application design, which is based on strategies used by biological systems to deal with similar challenges of complexity, heterogeneity, and uncertainty, i.e. autonomic computing. Project AutoMate investigates conceptual models and implementation architectures to enable the development and execution of self-managing applications. It supports the definition of autonomic elements, the development of autonomic applications as the dynamic and opportunistic composition of these autonomic elements, and the policy, content and context driven execution and management of these applications. This paper introduces AutoMate architecture and describes the Rudder coordination framework and its use in enabling autonomic behaviors.

1 Introduction

The emergence of wide-area distributed and decentralized "Grid" environments, such as pervasive information systems, peer-to-peer systems, and distributed computational infrastructures, has enabled a new generation of applications that are based on seamless access, aggregation and interactions. 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, scientific and engineering simulations of complex physical phenomena that symbiotically and opportunistically combine computations, experiments, observations, and real-time data to provide important insights into complex systems, and business applications that use anytime-anywhere information access to optimize profits.

However, these emerging Grid computing environments are inherently large, heterogeneous and dynamic, globally aggregating large numbers of independent computing and communication resources, data stores and sensor networks. Further, emerging

* The research presented in this paper is supported in part by the National Science Foundation via grants numbers ACI 9984357, EIA 0103674, EIA 0120934, ANI 0335244, CNS 0305495, CNS 0426354 and IIS 0430826.

J.-P. Banâtre et al. (Eds.): UPP 2004, LNCS 3566, pp. 260–275, 2005.

Grid applications are similarly large and highly dynamic in their behaviors and interactions. Together, these characteristics result in application development, configuration and management complexities and uncertainties that break current paradigms based on passive elements and static compositions and interactions. 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 [8], aims at realizing computing systems and applications capable of managing themselves with minimal human intervention.

Enabling autonomic systems and applications presents many conceptual and implementation challenges, primarily due to the highly dynamic, context and content-dependent behaviors. A key challenge is supporting coordination in a robust and scalable manner. Coordination is the *management of runtime dependencies and interactions among the elements in the system*. In case of autonomic systems/applications, these dependencies and interactions can be complex and various (e.g. peer-to-peer, client-server, producer-consumer, collaborative, at-most/at-least/exactly, etc.), and both, the coordinated entities and the nature of the relationships and interactions between them can be ad hoc and opportunistic.

Project AutoMate investigates autonomic solutions to deal with the challenges of complexity, dynamism, heterogeneity and uncertainty in Grid environments. The overall goal of Project AutoMate is to develop conceptual models and implementation architectures that can enable the development and execution of such self-managing Grid applications. These include programming models, frameworks and middleware services that support definition of autonomic elements, the development of autonomic applications as the dynamic and opportunistic composition of these autonomic elements, and the policy, content and context driven execution and management of these applications. This paper introduces AutoMate and its key components. Specifically, this paper focuses on the design and implementation of the Rudder coordination framework. Rudder provides software agents that enable application/system self-managing behaviors, and a fully decentralized coordination middleware that enables flexible and scalable interaction and coordination among agents and autonomic elements. The operation of AutoMate and Rudder is illustrated using an autonomic oil reservoir optimization application that is enabled by the framework.

The rest of this paper is organized as follows. Section 2 outlines the challenges and requirements of pervasive Grid systems and applications. Section 3 introduces Project AutoMate, presents its overall architecture and describes its key components. Section 4 presents the describes the design, implementation and evaluation of the Rudder coordination framework, including the Rudder agent framework and the COMET coordination middleware. Section 5 presents the autonomic oil reservoir application enabled by AutoMate and Rudder. Section 6 presents a conclusion.

2 Enabling Grid Applications – Challenges and Requirements

The goal of the Grid concept is to enable a new generation of applications combining intellectual and physical resources that span many disciplines and organizations,

providing vastly more effective solutions to important scientific, engineering, business and government problems. These new applications must be built on seamless and secure discovery, access to, and interactions among resources, services, and applications owned by many different organizations.

Attaining these goals requires implementation and conceptual models. Implementation models address the virtualization of organizations which leads to Grids, the creation and management of virtual organizations as goal-driven compositions of organizations, and the instantiation of virtual machines as the execution environment for an application. Conceptual models define abstract machines that support programming models and systems to enable application development. Grid software systems typically provide capabilities for: (i) creating a transient "virtual organization" or virtual resource configuration, (ii) creating virtual machines composed from the resource configuration of the virtual organization (iii) creating application programs to execute on the virtual machines, and (iv) executing and managing application execution. Most Grid software systems implicitly or explicitly incorporate a programming model, which in turn assumes an underlying abstract machine with specific execution behaviors including assumptions about reliability, failure modes, etc. As a result, failure to realize these assumptions by the implementation models will result in brittle applications. The stronger the assumptions made, the greater the requirements for the Grid infrastructure to realize these assumptions and consequently its resulting complexity. In this section we first highlight the characteristics and challenges of Grid environments, and outline key requirements for programming Grid applications. We then introduce self-managing Grid applications that can address these challenges and requirements.

2.1 Characteristics of Grid Execution Environments and Applications

Key characteristics of Grid execution environments and applications include:

Heterogeneity: Grid environments aggregate large numbers of independent and geographically distributed computational and information resources, including supercomputers, workstation-clusters, network elements, data-storages, sensors, services, and Internet networks. Similarly, applications typically combine multiple independent and distributed software elements such as components, services, real-time data, experiments and data sources.

Dynamism: The Grid computation, communication and information environment is continuously changing during the lifetime of an application. This includes the availability and state of resources, services and data. Applications similarly have dynamic runtime behaviors in that the organization and interactions of the components/services can change.

Uncertainty: Uncertainty in Grid environment is caused by multiple factors, including (1) dynamism, which introduces unpredictable and changing behaviors that can only be detected and resolved at runtime, (2) failures, which have an increasing probability of occurrence and frequencies as system/application scales increase; and (3) incomplete knowledge of global system state, which is intrinsic to large decentralized and asynchronous distributed environments.

Security: A key attribute of Grids is flexible and secure hardware/software resource sharing across organization boundaries, which makes security (authentication, authorization and access control) and trust critical challenges in these environments.

2.2 Requirements for Programming Systems and Middleware Services

The characteristics outlined above require that Grid programming systems and middleware services must be able to specify and support applications that can detect and dynamically respond to the changes in the runtime environment and application states. This requirement suggests that (1) Grid applications should be formulated from discrete composable elements, which incorporate separate specifications for all of functional, non-functional, and interaction and coordination behaviors; (2) The interface definitions of these elements should be separated from their implementations to enable heterogeneous elements to interact and to enable dynamic selection of elements; (3) Specifications of composition, coordination and interaction should be separated from computation behaviors, and may be dynamically specified and implemented.

Given these requirements, a Grid application requiring a given set of computational behaviors may be integrated with different interaction and coordination models or languages (and vice versa) and different specifications for non-functional behaviors such as fault recovery and QoS to address the dynamism and heterogeneity of the application and the underlying environments.

2.3 Self-managing Applications on the Grid

As outlined above, the inherent scale, complexity, heterogeneity, and dynamism of emerging Grid environments and applications result in significant programming and runtime management challenges, which break current approaches. This is primarily because the programming models and the abstract machine underlying these models make strong assumptions about common knowledge, static behaviors and system guarantees that cannot be realized by Grid virtual machines and, which are not true for Grid applications. Addressing these challenges requires redefining Grid programming frameworks and middleware services to address the separations outlined above. Specifically, it requires (1) static (defined at the time of instantiation) application requirements and system and application behaviors to be relaxed, (2) the behaviors of elements and applications to be sensitive to the dynamic state of the system and the changing requirements of the application and be able to adapt to these changes at runtime, (3) required common knowledge be expressed semantically (ontology and taxonomy) rather than in terms of names, addresses and identifiers, and (4) the core enabling middleware services (e.g., discovery, messaging) be driven by such a semantic knowledge. In the rest of this paper we describe Project AutoMate, which attempts to address these challenges by enabling autonomic self-managing Grid applications.

3 Project AutoMate: Enabling Self-managing Grid Applications

Project AutoMate [17, 16] investigates autonomic computing approaches to realize systems and applications that are capable of managing (i.e., configuring, adapting, optimiz-

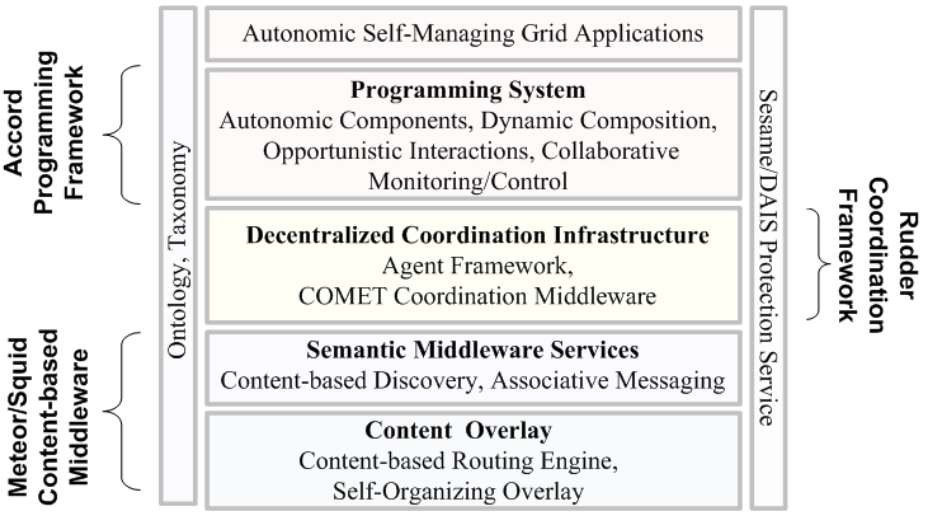


Fig. 1. A schematic overview of AutoMate

ing, protecting, healing) themselves. The overall goal is to investigate the conceptual models and implementation architectures that can enable the development and execution of such self-managing Grid applications. Specifically, it investigates programming frameworks and middleware services that support the development of autonomic applications as the dynamic and opportunistic composition of autonomic elements, and the execution and management of these applications.

A schematic overview of AutoMate is presented in Figure 1. Components of AutoMate include the Accord [10, 11] programming system, the Rudder [9] decentralized coordination framework and agent-based deductive engine, which is the focus of this paper, and the Meteor [7, 6] content-based middleware providing support for content-based routing, discovery and associative messaging. Project AutoMate additionally includes the Sesame [21] context-based access control infrastructure, the DAIS [20] cooperative-protection services and the Discover collaboratory [4, 12, 13] services for collaborative monitoring, interaction and control, which are not described here.

The Accord programming system [10, 11] extends existing programming systems to enable autonomic element definitions, self-managing Grid application formulation and development. Specifically it extends the entities and composition rules defined by the underlying programming model to enable computational and composition/interaction behaviors to be defined at runtime using high-level rules. *Autonomic Elements* in Accord extend programming elements (i.e., objects, components, services) to define a self-contained modular software unit with specified interfaces and explicit context dependencies. Additionally, an autonomic element encapsulates rules, constraints and mechanisms for self-management, and can dynamically interact with other elements and the system.

Each autonomic element is associated with an element manager (possibly embedded) that is delegated to manage its execution. The element manager monitors the state

of the element and its context, and controls the execution of rules. Rules incorporate high-level guidance and practical human knowledge. *Behavioral rules* control the runtime functional behaviors of an autonomic element (e.g., the dynamic selection of algorithms, data representation, input/output format used by the element), while *Interaction rules* control the interactions between elements, between elements and their environment, and the coordination within an autonomic application (e.g., communication mechanism, composition and coordination of the elements).

Meteor [7, 6] is a scalable content-based middleware infrastructure that provides services for content routing, discovery, and associative interactions. The Meteor stack consists of 3 key components: (1) a self-organizing overlay, (2) a content-based routing engine and discovery service (Squid), and (3) the Associative Rendezvous Messaging Substrate (ARMS). The Meteor overlay is composed of Rendezvous Peer (RP) nodes, which may be any node on the Grid (e.g., gateways, access points, message relay nodes, servers or end-user computers). RP nodes can join or leave the overlay network at any time. The overlay topology is based on standard structured overlays. The content overlay provides a single operation, *lookup(identifier)*, which requires an exact identifier (e.g., name). Given an identifier, this operation locates the peer node where the content should be stored. Squid [18] is the Meteor content-based routing engine and decentralized information discovery service. It supports flexible content-based routing and complex queries containing partial keywords, wildcards, and ranges, and guarantees that all existing data elements that match a query will be found.

The ARMS layer [7] implements the Associative Rendezvous (AR) interaction paradigm. AR is a paradigm for content-based decoupled interactions with programmable reactive behaviors, and extends the conventional name/identifier-based rendezvous in two ways. First, it uses flexible combinations of keywords (i.e, keyword, partial keyword, wildcards and ranges) from a semantic information space, instead of opaque identifiers (names, addresses) that have to be globally known. Interactions are based on content described by these keywords. Second, it enables the reactive behaviors at the rendezvous points to be encapsulated within messages increasing flexibility and enabling multiple interaction semantics (e.g., broadcast multicast, notification, publisher/subscriber, mobility, etc.).

Rudder [9] is an agent-based decentralized coordination framework for enabling self-managing Grid applications, and provides the core capabilities for supporting autonomic compositions, adaptations, optimizations, and fault-tolerance. It enables composition, coordination and interaction behaviors to be separated from computational behaviors, and allows them to be semantically separately expressed and efficiently implemented. Rudder and its components are described in more detail in the following sections.

4 Rudder Coordination Framework

Rudder consists of two key components: an agent framework and the COMET coordination middleware. The agent framework provides protocols for coordination and cooperation to enable peer agents to individually and collectively achieve self-managing behaviors. COMET implements the coordination abstractions and mechanisms and provides a decentralized and associative shared coordination-space.

4.1 The Rudder Agent Framework

The Rudder agent framework is composed of a dynamic network of software agents existing at different levels, ranging from individual system/application elements to the overall system/application. These agents monitor the element states, manage the element behaviors and dependencies, coordinate element interactions, and cooperate to manage overall system application behaviors.

Agent Classification: The Rudder agent framework consists of three types of peer agents: Component Agent (CA), System Agent (SA), and Composition Agent (CSA). CAs and SAs are part of the system/application elements, while CSAs are transient and are generated to satisfy specific application requirements. CAs manage the computations performed locally within application elements and their interaction and communication behaviors and mechanisms. They are integrated with the Accord element managers. SAs are embedded within Grid resource units (e.g., compute resources, instrument, data store). CSAs enable dynamic composition of autonomic elements by defining and executing workflow-selection and element-selection rules. Workflow-selection rules are used to select appropriate composition plans to enact. Element-selection rules are used to semantically discover and select registered elements. CSAs negotiate to select interaction patterns for a specific application workflow, and coordinate with associated element agents to define and execute associated interaction rules at runtime. This enables autonomic applications to dynamically change flows, elements and element interactions to address application and system dynamics and uncertainty.

Agent Coordination Protocols: Rudder provides a set of common discovery and control protocols to all agents. Discovery protocols support the registering, unregistering, and discovery of system/application elements. Control protocols allow the agents to query element states, control their behaviors and orchestrate their interactions. These protocols include negotiation, notification, and mutual exclusion. The agent coordination protocols are scalably and robustly implemented in logically decentralized, physically distributed Grid environments using the abstractions provided by COMET, which are described below.

4.2 COMET Coordination Middleware

The overall goal of COMET is to enable scalable peer-to-peer content-based coordination in large-scale decentralized distributed environments. The COMET implements a global Linda-like shared-space [5], which is constructed from a globally known semantic multi-dimensional information space. The information space is defined by the ontology used by the coordinated entities, and is deterministically mapped, using a locality preserving mapping, to a dynamic set of peer nodes in the system. The resulting peer-to-peer information lookup system maintains content locality and guarantees that content-based information queries, using flexible content descriptors in the form of keywords, partial keywords and wildcards, are delivered with bounded costs.

The COMET Model. The COMET model consists of layered abstractions prompted by a fundamental separation of communication and coordination concerns.

The *communication abstraction* provides an associative communication service and guarantees that content-based information queries, specified using flexible content descriptors, are served with bounded costs. It supports content-based discovery, routing and messaging. This layer essentially maps the virtual information space in a deterministic way on to the dynamic set of currently available peer nodes in the system, while maintaining content locality. It thus manages system scale, heterogeneity and dynamism. The communication abstraction provides a single operator: **deliver** ($\mathcal{M}$). The message $\mathcal{M}$ consists of (1) a semantic selector that is flexibly defined using keywords from the information space, and specifies a region in this space, and (2) a payload consisting of the data and operation to be performed at the destination.

The *coordination abstraction* extends the traditional data-driven model with event-based reactivity to changes in system state and to data access operations. It defines a *reactive tuple*, which consists of 2 additional components: a *condition* that associates *reaction* to events, and a *guard* that specifies how and when the reaction will be executed (e.g., immediately, once). This abstraction provides the basic **Out**, **In**, and **Rd** primitives. These basic operations operate on regular as well as reactive tuples and retain the Linda semantics. The operations are directly implemented on the **deliver** operator provided by the communication abstraction.

Transient Spaces in COMET. Coordination middlewares based on the model outlined above are naturally suitable for context-transparent applications that are developed and executed without explicit knowledge of the system context. Furthermore, since the underlying implementation maintains content locality in the information space, it is both scalable and flexible. However, certain applications, e.g., mobile applications, require context locality to be maintained in addition to content locality, i.e., they impose requirements for context-awareness. The uniform operators provided by COMET do not distinguish between local and remote components of a space. While this is a convenient abstraction, it does not maintain context locality and may have a detrimental effect on system efficiency for these applications. To address this issue, COMET defines transient spaces that have a specific scope definition (e.g., within the same geographical region or the same physical subnet). The transient spaces have exactly the same structure and semantics as the original space, and can be dynamically created. An application can switch between spaces at runtime and can simultaneously use multiple spaces.

The COMET Design and Implementation. A schematic overview of the COMET system architecture is shown in Figure 2. The current prototype has been implemented on Project JXTA [2], a general-purpose peer-to-peer framework. The coordination space is provided as a JXTA peergroup service that can be concurrently exploited by multiple applications. The design and implementation of the COMET coordination and communication layers are described below.

Communication Layer: The communication layer of COMET is built on the Meteor messaging substrate[7], which provides scalable content-based routing and data delivery operations. Meteor consists of a structured self-organizing overlay and the Squid content-based routing engine.

Squid [18] provides a decentralized information discovery and associative messaging service. It uses a locality preserving and dimension reducing indexing scheme,

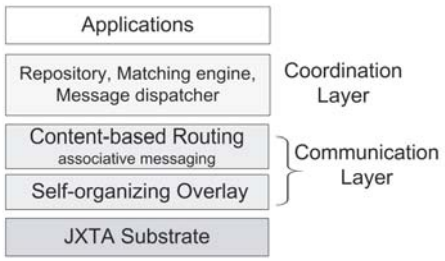


Fig. 2. A schematic overview of the COMET system architecture

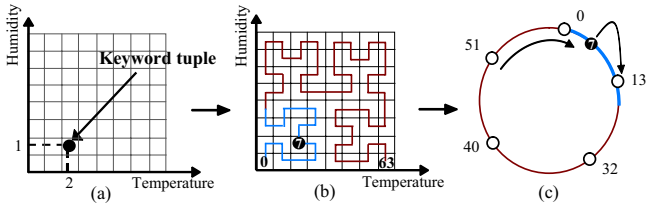


Fig. 3. Routing using a simple keyword tuple in Squid: (a) the simple keyword tuple (2, 1) is viewed as a point in a multi-dimensional space; (b) the keyword tuple is mapped to the index 7, using Hilbert SFC; (c) the data will be routed in the overlay (an overlay with 5 RP nodes and an identifier space from 0 to 2^6-1) at RP node 13, the successor of the index 7

based on the Hilbert Space Filling Curve (SFC), to effectively map a multi-dimensional information space to the peer identifier space and to the current peer nodes in the system. The peer nodes form a structured overlay. The resulting peer-to-peer information system supports flexible content-based routing and complex queries containing partial keywords, wildcards, and ranges, and guarantees that all existing data elements that match a query will be found. Keywords can be common words or values of globally defined attributes, and are defined by applications. In the case of COMET, these keywords are part of the common ontology used by the coordinating entities. The keywords form the multi-dimensional information space, i.e., keyword tuples represent points or regions in this space and the keywords are the coordinates. A *keyword tuple* in Squid is defined as a list of d keywords, wildcards and/or ranges, where d is the dimensionality of the keyword space. A keyword tuple only containing complete keywords is called *simple*, and a tuple containing partial keywords, wildcards and/or ranges is called *complex*.

Content-based routing in Squid is achieved as follows. SFCs are used to generate a 1-dimensional index space from the multi-dimensional keyword space. Further, using the SFC, a query consisting of a simple keyword tuple can be mapped to a point on the SFC. Similarly, any complex keyword tuple can be mapped to regions in the keyword space and to corresponding clusters (segments of the curve) in the SFC. The 1-dimensional index space generated from the entire information space is mapped onto the 1-dimensional identifier space used by the overlay network formed by the peer

nodes. As a result, using the SFC mapping any simple or complex keyword tuple can be located. Squid provides a simple abstraction to the layer above consisting of a single operation: **post**(*keyword tuple, data*), where *data* is the message payload provided by the messaging layer above. The routing for simple and complex keyword tuples is illustrated in Figures 3 and 4 respectively.

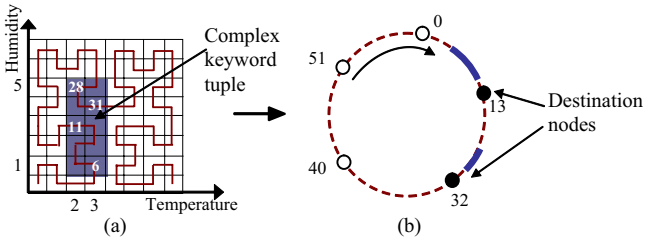


Fig. 4. Routing using a complex keyword tuple (2-3, 1-5): (a) the keyword tuple defines a rectangular region in the 2-dimensional keyword space consisting of 2 clusters (2 segments on the SFC curve); (b) the clusters (the solid part of the circle) correspond to destination RP nodes 13 and 32, which are routed to

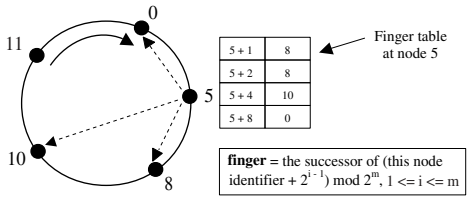


Fig. 5. Example of the Chord overlay network. Each node stores the keys that map to the segment of the curve between itself and the predecessor node

The Meteor content overlay is composed of peer nodes, which may be any node in the system (e.g., gateways, access points, message relay nodes, servers or end-user computers). The peer nodes can join or leave the network at any time. The overlay topology is based on standard structured overlays. The current implementation of Meteor uses the Chord [19] overlay network where peer nodes form a ring topology. Advantages of Chord include its guaranteed performance and logarithmic in number of messages. Every node in Chord is assigned a unique identifier and maintains a *finger table* for routing. The lookup algorithm in Chord enables the efficient data routing with O(log N) cost, where N is the number of nodes in the system. An example of a Chord overlay network with 5 nodes is shown in Figure 5. The Meteor overlay network layer provides a simple abstraction to the layers above, consisting of a single operation: **lookup**(*identifier*). Given an identifier, this operation locates the node that is responsible for it, i.e, the node with an identifier that is the closest identifier greater than or equal to the queried identifier.

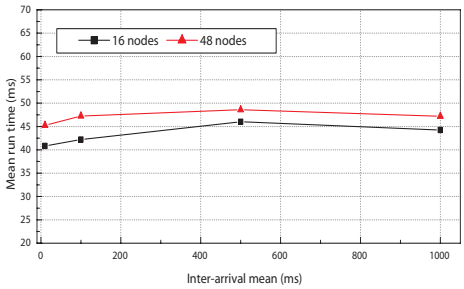


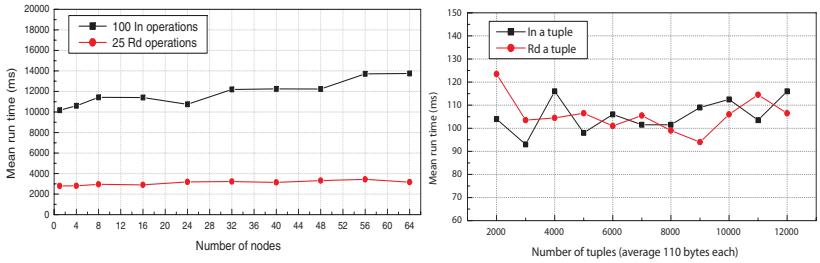
Fig. 6. Average round trip time for the *Out* operation for different mean inter-arrival time

Coordination Layer: The coordination layer implements the coordination abstraction and primitives. Its main components include a data repository for storing, pending requests, and retrieving tuples, a flexible matching engine, and a message dispatcher that interfaces with the communication layer to convert the coordination primitives to messaging operations and vice versa. Tuples and templates are represented as simple XML strings, as they provide small-sized flexible formats that are suitable for efficient information exchange in distributed heterogeneous environments.

The COMET **Out**, **Rd** and **In** operations are implemented using Squid routing. Using the tag and fields of a tuple, each tuple/template is associated with a sequence of keywords, which are then used to generate the *keyword tuple* required by the Squid *post* operator. It is assumed that all peer nodes agree on the structure and dimension of the information space used to define the keyword tuples.

Tuple distribution consists of the following steps: (1) Keywords are extracted from the tuple and used to create the keys for the Squid *post* operation. The payload of the message consists of the tuple and the coordination operation. (2) Squid uses the SFC mapping to identify the indices corresponding to the keyword tuple and the corresponding peer id(s). (3) The overlay *lookup* operator is used to route to the appropriate peer nodes. This operator maps the logical peer identifer to a JxtaId and sends the tuple using the JXTA Resolver Protocol. The *Out* operator only returns after receiving the *Resolver Query Response* from the destination to guarantee tuple delivery. In the case of *In* and *Rd* operations, the templates are routed in a similar manner. These two operations block until a matched tuple is returned by the destination in a peer-to-peer manner.

Experimental Evaluation of COMET. COMET has been deployed in a distributed network of 64 Linux-based computers in Rutgers University. Each node has an Intel(R) Pentium-4 1.70GHz CPU with 512MB RAM and is running Linux 2.4.20-8 (kernel version). Each machine serves as a peer node in COMET overlay. The experiments include measuring the average run time for each of the coordination primitives provided by COMET. For an *Out* operation, the measured time corresponds to the time interval between when the tuple is posted into the space and when the response from the destination is received. For a *In/Rd* operation, the measured time is the time interval between when the template is posted into the space and when the matched tuple is returned to the application assuming that a matched tuple exists in the space. This time includes the duration of template routing, repository matching and returning the matched tuple. The measurements use the native clocks of the peer nodes.



(a) Average time for 100 *In* and 25 *Rd* operations for increasing system sizes.

(b) Average time for *In* and *Rd* operations with increasing number of tuples. System size fixed at 4 nodes.

Fig. 7. Scalability and performance of *In* and *Rd* operations

Evaluation of the Out Operation. To evaluate the *Out* operation, regular XML tuples were used, which consist of randomly generated strings with fixed length. The average size of a tuple was 110 bytes. Furthermore, network traffic was modelled as poisson arrival *Out* operations with different inter-arrival mean time. Figure 6 shows the performance of the *Out* operations with inter-arrival mean time of 10ms, 100ms, 500ms, and 1000ms, and a system size of 16 and 48 peer nodes. The Y axis is the average run time. The figure shows that the *Out* operation is fairly independent of the traffic inter-arrival time and scales with system size at the order of O(logN) where N is the number of nodes in the system. The maximum average time of the 48 peer node system is approximately 47ms, which we believe is acceptable.

Evaluation of the In/Rd Operation. To study the behavior of the *In/Rd* operation, two experiments were conducted. The first experiment evaluated the average time required for data retrieval and extraction using *In* and *Rd* operations with different system sizes. The operation latency was measured for 25 *Rd* operations and 100 *In* operations. In this experiment we assumed that the tuples were previously stored into the space by *Out* operations. In the second experiment, the average time required for each single operation was measured for different numbers of tuples, with a fixed system size of 4 nodes. The lengths of the tuples are fixed at 110 bytes. The tuples were generated with random strings. The results are shown in Figure 7. The plots show that the *In/Rd* operations scale well with the number of nodes and their performance is largely independent of the number of tuples in the system. The average latency for *Rd/In* operations is approximately 105ms for experiments with numbers of tuples ranging from 2000 to 12000.

5 An Illustrative Example: Autonomic Oil Reservoir Optimization

One of the fundamental problems in oil reservoir production is determining the optimal locations of the oil production and injection wells. However, the selection of appropriate optimization algorithms, the runtime configuration and invocation of these

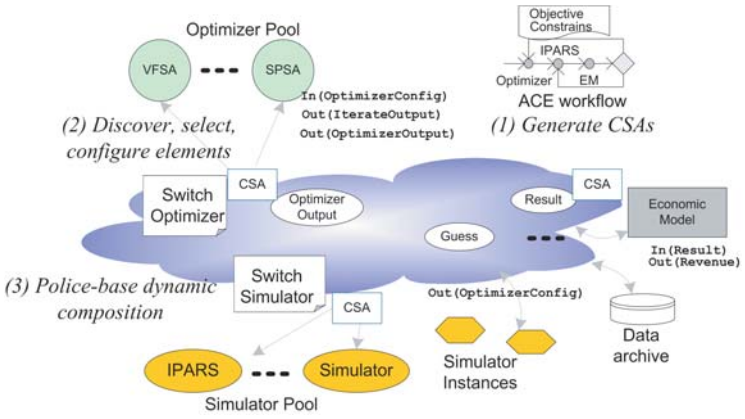


Fig. 8. Autonomic Oil Reservoir Optimization

algorithms and the dynamic optimization of the reservoir remain challenging problems. In this example we use AutoMate to support the autonomic compositions, interactions and adaptations to enable an autonomic self-optimizing reservoir application. The application consists of the following elements: (1) sophisticated reservoir simulation components (e.g. IPARS [1] factory) that encapsulate complex mathematical models of the physical interactions in the subsurface; (2) distributed data archives that store historical, experimental, and observed data; (3) sensors embedded in the instrumented oilfield providing real-time data about the current state of the oil field; (4) optimization services based on the Very Fast Simulated Annealing (VFSA) [14] and Simultaneous Perturbation Stochastic Approximation (SPSA) [15]; (5) the economic modeling service.

These elements need to dynamically discover one another and interact as peers to achieve the overall application objectives. First, the simulation components should dynamically obtain necessary resources, detect current resource state, and negotiate required qualities of service. Next, the simulation components must interact with one another, and with archived history and real-time sensor data, to enable a better characterization of the reservoir. Further, the reservoir simulation components interact with optimization services and with the data to optimize well configuration and operation, with weather services to control production, and with economic modelling service to detect current and predicted future oil prices so as to maximize the revenue from the production.

The operation of this application using AutoMate, and specially Rudder, is illustrated in Figure 8. The overall process is achieved by (1) generating composition agents based on application workflows, (2) agents discovering and composing the involved components to enable the oil reservoir management process, which includes monitoring oil production behaviors and detecting needs for optimization, and (3) agents using high-level policies to orchestrate interactions to optimize well operation and oil production.

First, the AutoMate composition engine (ACE) [3] generates the following workflows to satisfy the application objectives: (i) the optimization service provides the

IPARS reservoir simulator with an initial guess of well parameters based on the configuration of the oil field; (ii) IPARS uses the well parameters along with current market parameters to periodically compute the current revenue using an Economic Model (EM) service; and (iii) IPARS iteratively interacts with the optimization service to optimize well parameters for maximum profit. Based on above workflows, three CSAs are instantiated for the EM, Optimizer, and IPARS respectively. The CSAs dynamically discover the appropriate autonomic elements with desired functionality and cost/performance characteristics using the discovery protocol, and configure the workflows using interaction rules. The CAs use the interaction rules to dynamically establish interaction relationships among the elements and using appropriate communication mechanisms. The CSAs then coordinate with the CAs using the decentralized tuple-space.

Application self-management and self-optimization behaviors are achieved via the police-based autonomic behaviors of the agents. Each CA monitors and manages the execution of its element, while the CSAs discover and compose elements and resources to satisfy current application objectives. For example, the choice of optimization algorithm depends on the size and nature of the reservoir. In case of reservoirs with many randomly distributed maxima and minima, the VFSA algorithm can be employed during the initial optimization phase. Once convergence slows down, VFSA can be replaced by SPSA, which is suited for larger reservoirs with relatively smooth characteristics. Using these policies, the Optimizer CSA selects the appropriate optimization service, and configures it to optimize the application according to the current objectives of the application. Similarly, the SAs monitor and manage the runtime utilization of the resource and dynamically balance workload.

6 Conclusion

In this paper, we introduced Project AutoMate and described Rudder, its coordination framework. Project AutoMate investigates solutions that are based on the strategies used by biological systems to deal with challenges of complexity, dynamism, heterogeneity and uncertainty. This approach, referred to as autonomic computing, aims at realizing systems and applications that are capable of managing (i.e., configuring, adapting, optimizing, protecting, healing) themselves. The overall goal of Project AutoMate is to investigate conceptual models and implementation architectures that can enable the development and execution of such self-managing Grid applications. Specifically, it investigates programming models, frameworks and middleware services that support the definition of autonomic elements, the development of autonomic applications. The Rudder coordination framework consists of an agent framework and the COMET coordination middleware and enables dynamic discovery, composition and the policy, content and context driven definition, execution and management of these applications. The design, implementation and evaluation of Rudder was presented. The operation of AutoMate and Rudder was illustrated using an autonomic self-optimization oil reservoir application.

References

1. IPARS: Integrated Parallel Reservoir Simulator, The University of Texas at Austin, http://www.ices.utexas.edu/CSM.
2. "Project JXTA", http://www.jxta.org.
3. M. Agarwal and M. Parashar, "Enabling Autonomic Compositions in Grid Environments," in *Proceedings of 4th International Workshop on Grid Computing (Grid 2003)*, IEEE Computer Society Press 2003, 34 - 41.
4. V. Bhat and M. Parashar, "Discover Middleware Substrate for Integrating Services on the Grid," in *Proceedings of 10th International Conference on High Performance Computing (HiPC 2003)*, Springer-Verlag, December 2003, 373-382.
5. D. Gelernter, "Generative communication in Linda", *ACM Trans. Program. Lang. System*, ACM Press, 7(1) 1985, 80–112.
6. N. Jiang and M. Parashar, "Enabling Applications in Sensor-Based Pervasive Environments," in *Proceedings of BROADNETS 2004: Workshop on Broadband Advanced Sensor Networks (BaseNets 2004)*, San Jose, CA, USA October 25, 2004.
7. N. Jiang, C. Schmidt, V. Matossian and M. Parashar, "Content-based Middleware for Decoupled Interactions in Pervasive Environments", Rutgers University, Wireless Information Network Laboratory (WINLAB), Piscataway, NJ, USA, 2004.
8. J. Kephart, M. Parashar, V. Sunderam and R. Das, eds., *Proceedings of the First International Conference on Autonomic Computing*, IEEE Computer Society Press, 2004.
9. Z. Li and M. Parashar, "Rudder: A Rule-based Multi-agent Infrastructure for Supporting Autonomic Grid Applications," in *Proceedings of 1st IEEE International Conference on Autonomic Computing (ICAC-04)*, May 2004, 10 -17.
10. H. Liu and M. Parashar, "Accord: A Programming Framework for Autonomic Applications," *IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Engineering Autonomic Systems*, Editors: R. Sterritt and T. Bapty, IEEE Press, to appear.
11. H. Liu, M. Parashar and S. Hariri, "A Component-based Programming Framework for Autonomic Applications," in *Proceedings of 1st IEEE International Conference on Autonomic Computing (ICAC-04)*, IEEE Computer Society Press 2004, 278 - 279.
12. V. Mann, V. Matossian, R. Muralidhar and M. Parashar, "DISCOVER: An Environment for Web-based Interaction and Steering of High-Performance Scientific Applications", *Concurrency and Computation: Practice and Experience*, 13(8-9), 2001, 737-754.
13. V. Mann and M. Parashar, "Engineering an Interoperable Computational Collaboratory on the Grid", *Concurrency and Computation: Practice and Experience, Special Issue on Grid Computing Environments*, 14(13-15), 2002, 1569-1593.
14. V. Matossian, V. Bhat, M. Parashar, M. Peszynska, M. Sen, P. Stoffa and M. F. Wheeler, "Autonomic Oil Reservoir Optimization on the Grid", *Concurrency and Computation: Practice and Experience, John Wiley and Sons*, John Wiley and Sons, Vol. 17, Issue 1, pp. 1 - 26, 2005.
15. V. Matossian, M. Parashar, W. Bangerth, H. Klie and M. F. Wheeler, "An Autonomic Reservoir Framework for the Stochastic Optimization of Well Placement", *Cluster Computing: The Journal of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing, Kluwer Academic Press*, March 2004.
16. M. Parashar, Z. Li, H. Liu, C. Schmidt, V. Matossian and N. Jiang, "Enabling Autonomic Applications: Models and Infrastructure," in *Proceedings of European Commission - US National Science Foundation Strategic Research Workshop on Unconventional Programming Paradigms: Challenges, Visions and Research Issues for New Programming Paradigms*, Spring Verlag 2004.

17. M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang and S. Hariri, "AutoMate: Enabling Autonomic Grid Applications," *Cluster Computing: The Journal of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing*, Kluwer Academic Publishers, November 2003.
18. C. Schmidt and M. Parashar, "Enabling Flexible Queries with Guarantees in P2P Systems", *IEEE Internet Computing*, 8(3), May-June 2004, 19 - 26.
19. I. Stoica, R. Morris, D. Karger, F. Kaashoek and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *Proceedings of ACM SIGCOMM*, August 2001.
20. G. Zhang and M. Parashar, "Cooperative Defense against Network Attacks,", in *Proceedings of the 3rd International Workshop on Security In Information Systems (WOSIS 2005), 7th International Conference on Enterprise Information Systems (ICEIS 2005)*, Miami, FL, USA, May 2005.
21. G. Zhang and M. Parashar, "Dynamic Context-aware Access Control for Grid Applications," in *Proceedings of 4th International Workshop on Grid Computing (Grid 2003)*, IEEE Computer Society Press 2003, 101 - 108.