An Autonomic Service Architecture for Self-Managing Grid Applications

Hua Liu, Viraj Bhat, Manish Parashar, and Scott Klasky

Abstract—**The scale, heterogeneity and dynamism of Grid applications and environments require Grid applications to be self-managing or autonomic. This paper presents the Accord autonomic services architecture that addresses this requirement. Accord enables service and application behaviors and their interactions to be dynamically specified and adapted using high-level rules, based on current application requirements, state and execution context. The design, implementation and evaluation of Accord are presented. An autonomic data streaming application is used to illustrate the self-managing behaviors enabled by Accord.**

Index Terms—**Grid programming system, autonomic computing, service based architecture, adaptive data streaming.**

I. Introduction

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. The key characteristics of Grid execution environments and applications include: (1) Heterogeneity: Both Grid environments and applications aggregate multiple independent, diverse and geographically distributed elements and resources; (2) Dynamism: Grid environments are continuously changing during the lifetime of an application. Applications similarly have dynamic runtime behaviors including the organization and interactions of its elements; (3) Uncertainty: Uncertainty in Grid environments is caused by multiple factors, including dynamism that introduces unpredictable and changing behaviors, failures that have an increasing probability of occurrences as system/application scales increase, and incomplete knowledge of global state, which is intrinsic to large distributed environments; (4) Security: A key attribute of Grids is secure resource sharing across organization boundaries, which makes security a critical challenge [1].

The characteristics listed above require that Grid applications must be able to detect and dynamically respond during execution to changes in both, the state of execution environment and the state and requirements of the application [1]. This requirement suggests that (1) Grid applications should be composed from discrete, self-managing elements (components/services), which incorporate separate specifications for functional, non-functional and interaction/coordination behaviors; (2) The specifications of computational (functional) behaviors, interaction and coordination behaviors, and non-functional behaviors (e.g. performance, fault detection and recovery, etc.) should be separated so that their combinations are compose-able; and (3) policy should be separated from mechanisms and used to orchestrate a repertoire of mechanisms to achieve context-aware adaptive runtime behaviors. Given these features, 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 application state and the execution environment.

This paper presents the Accord autonomic services architecture that addresses these requirements and enables self-managing Grid applications. Accord extends the service-based Grid programming paradigm to relax static (defined at the time of instantiation) application requirements and system/application behaviors and allows them to be dynamically specified using high-level rules. Further, it enables the behaviors of services and applications to be sensitive to the dynamic state of the system and the changing requirements of the application, and to adapt to these changes at runtime. This is achieved by extending Grid services to include the specifications of policies (in the form of high-level rules) and mechanisms for self-management, and providing a decentralized runtime infrastructure for consistently and efficiently enforcing these policies to

Manuscript received June, 2005. 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 and by a subcontract from the Princeton Plasma Physics Laboratory.

Hua Liu (e-mail: marialiu@caip.rutgers.edu), TASSL, Dept. of Electrical and Computer Engineering, Rutgers Univ., Piscataway, NJ 08854. Viraj Bhat (e-mail: virajb@caip.rutgers.edu), TASSL, Dept. of Electrical and Computer Engineering, Rutgers Univ., Piscataway, NJ 08854. Contact author: Manish Parashar (e-mail: parashar@caip.rutgers.edu), TASSL, Dept. of Electrical and Computer Engineering, Rutgers Univ., Piscataway, NJ 08854. Phone: (732) 445-5388. Fax: (732) 445-0593. Scott Klasky (email: sklasky@pppl.gov), Princeton Plasma Physics Laboratory, Princeton University, NJ.

enable autonomic self-managing functional, interaction, and composition behaviors based on current requirements, state and execution context. The design and implementation of Accord is presented. Accord is part of Project AutoMate [2], which provides required middleware services.

This paper also describes the use of Accord to enable the adaptive transfer of multi-terabyte data from live simulations running on supercomputers at NERSC and ORNL to local visualization and analysis clusters at PPPL while minimizing overheads to the simulation.

The rest of the paper is organized as follows. Section II describes the design and implementation of the Accord autonomic service architecture. Section III illustrates self-managing behaviors enabled by Accord using the autonomic data streaming application. Section IV discusses related work. Section V presents a conclusion.

II. ACCORD AUTONOMIC SERVICES ARCHITECTURE

Accord defines conceptual, implementation and enforcement models for utilizing human knowledge (in the form of rules) to guide the execution and adaptation of services. This is achieved by adapting the behaviors of individual services and their interactions (communication/ coordination) to changing application requirements/state and execution environments based on dynamically defined rules.

A. Definition of Autonomic Services

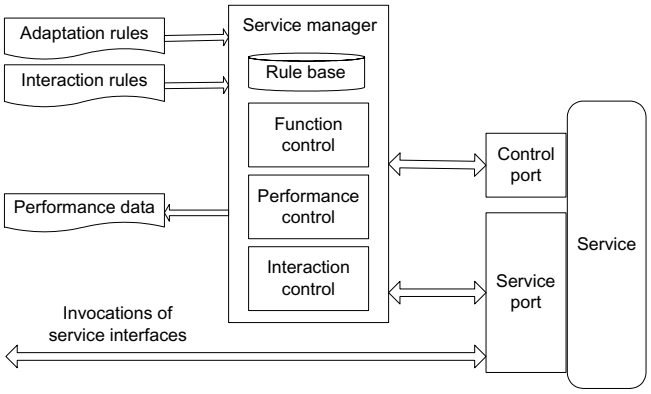

Figure1. An autonomic service in Accord.

An autonomic service (see Figure 1) extends a Grid service with a control port for external monitoring and steering, and a service manager that monitors and controls the runtime behaviors of the managed service according to changing requirements and state of applications as well as their execution environment based on user-defined rules.

The control port consists of *sensors* that enable the state of the service to be queried, and *actuators* that enable the behaviors of the service to be modified. The control port and service port are used by the service manager to control the functions, performance, and interactions of the managed service. The control port is described using WSDL and may be a part of the general service description, or may be a separate document to control access to it. An example of the control port is shown in Figure 5. Rules are simple *if-condition-then-action* statements described using XML and include service adaptation and service interaction rules. An example of a rule is shown in Figure 6.

B. The Runtime Infrastructure

The Accord runtime infrastructure (shown in Figure 2) consists of a user/developer portal, peer service and application composition/coordination managers, the autonomic services, and a decentralized rule enforcement engine. An application composition manager decomposes incoming application workflows (defined by the user or a workflow engine) into interaction rules for individual services, and forwards these rules to corresponding service managers. Service managers execute these rules to establish interaction relationships among services by negotiating communication protocols and mechanisms and dynamically constructing coordination relationships in a distributed and decentralized manner.

Application managers also forward incoming adaptation rules to appropriate service managers. Service managers execute these rules to adapt the functional behaviors of the managed services, and evaluate and tune their performance. These adaptations are realized by invoking appropriate control (sensors, actuators) and functional interfaces.

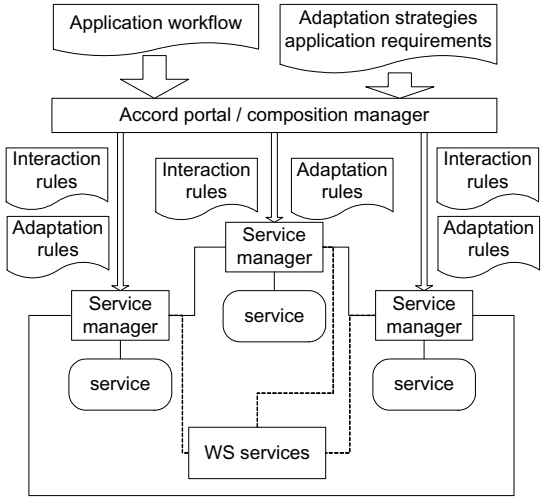

Figure 2. Accord runtime infrastructure. Solid lines indicate interactions among services and dotted lines represent invocation of WS instances providing supporting services such as naming and discovery.

Adaptation rules are typically used to adapt the behaviors of individual services and do not change their functionalities (described in service ports as contracts), and are therefore transparent to other services. This localized adaptation simplifies the specification and execution of adaptation rules by limiting the conditions monitored and actions performed to within individual services.

Interaction rules are used to adapt service interactions, for example communication paradigms and/or coordination relationships. When local optimization of individual services cannot satisfy the global objectives, interaction rules are

used to modify the application composition.

Rules are evaluated and executed by service managers as shown in Figure 3. In the figure, the *condition* part of the sample rule consists of three triggers belonging to service A and B, and the *action* part has two actions that invoke the actuators exposed by service A and C. Triggers are injected into corresponding service managers A and B, and their results are collected by the service manager A. Service manager A evaluates the condition, invokes *actuator1* and notifies service manager C to invoke *actuator2*.

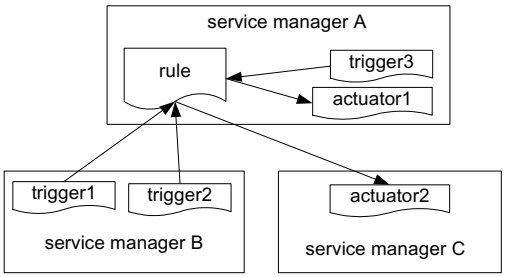

Figure 3. Execution of a sample rule.

Rule execution at the service managers consists of three phases: condition inquiry, condition evaluation and conflict resolution, and batch action invocation. During condition inquiry, the service managers query the sensors used by the rules in parallel, assimilates their current values, and fire corresponding triggers.

During the next phase, condition evaluations for all the rules are performed in parallel. Rule conflicts are detected during this phase when the same actuator is invoked with different values. These conflicts are resolved by relaxing the rule condition, using user-defined strategies, until the actuator-actuator conflict is resolved. If the conflicts are not resolved, errors are reported to users. If interacting services try to use different communication/coordination paradigms as a result of their independent adaptation behaviors, for example, one of the interacting services try to use RPC and the other proposes to use notification, the services negotiate with each other to resolve the conflict [3].

After rule conflict resolution, the actions are executed in parallel. Note that the rule execution model presented here focuses on correct and efficient execution of rules, providing mechanisms to detect and resolve conflicts at runtime. However, correctness of rules and conflict resolution strategies are the responsibilities of the users.

C. Autonomic Service Adaptation and Composition

Autonomic service adaptations, defined by adaptation rules, include modification of service parameters and dynamic selection of algorithms and implementations to optimize and tune service performance, meet QoS requirements, correct detected errors, avoid or recover from failures, and/or to protect the service. These adaptations are local to individual services and independent of and transparent to other services.

Dynamic and autonomic compositions are enabled in Accord using combinations of interaction and adaptation rules. Composition consists of defining the organization of services and the interactions among them [3]. The service organization describes a collection of services that are functionally compose-able, determined semantically (e.g., using OWL [4]) or syntactically using WSDL [5]. Interactions among services define the coordination between services and the communication paradigm used, e.g., message passing, RPC/RMI, or shared spaces.

Once a workflow has been generated (e.g., using the mechanism in [6]), and the services have been discovered (using middleware services), the Accord composition manager decomposes the workflow into interaction rules. This decomposition process consists of mapping workflow patterns [7] in the workflow into corresponding rule templates [3]. Accord provides templates for basic communication paradigms such as notification, publisher/subscriber, rendezvous, shared spaces and RPC/RMI, and control structures such as sequence, AND-split, XOR-split, OR-split, AND-join, XOR-join, and OR-join. More complex interaction and coordination structures (e.g., loops) can be constructed from these basic patterns.

The interaction rules are then injected into corresponding service managers, which execute the rules to establish communication and coordination relationships among involved services. Note that there is no centrally controlled orchestration. While the interaction rules are defined by the composition manager, the actual interactions are established by service managers in a decentralized and parallel manner. Also note that the communication paradigms and coordination relationships among the interacting autonomic services can be dynamically changed according to current application state and execution context by replacing/changing the related interaction rules.

The two adaptation approaches, adaptation within individual services and dynamic composition of services, can be used separately or in combination to enable the autonomic self-configuring, self-optimizing and self-healing behaviors of services and applications [3].

D. Implementation Overview

The current prototype implementation of the Accord autonomic services architecture extends the Apache Axis [8] Toolkit and is being integrated with the Globus toolkit GT4 [9]. In our current implementation, both control ports and service ports are implemented as WSDL documents. Service ports are used by interacting services in the normal way, and control ports are used by managers to manage the services. The publication/subscription structure is used for interactions between managers. Each manager maintains a subscription table consisting of triggers of interest, and publishes trigger information to subscribing managers as XML messages.

Further, the prototype uses middleware services provided

by AutoMate [2] to enable (1) content-based routing/discovery, associative messaging, and a decentralized reactive tuple space for interaction/coordination among service managers, and (2) context-based access control for service authorization and authentication. An experimental evaluation of the Accord prototype and its overheads are presented in [3].

III. Autonomic Data Streaming Using Accord

A. Application Setup

This section illustrates the self-managing behaviors enabled by the Accord autonomic service architecture using an autonomic data streaming service. The overall application is presented in Figure 4. The application consists of the G.T.C. fusion simulation that runs for days on a parallel supercomputer at NERSC (CA) and generates multi-terabytes of data. This data is analyzed and visualized live, while the simulation is running, at PPPL (NJ). The data also has to be archived either at PPPL (NJ) or ORNL (TN). Data streaming techniques from a large number of processors have been shown to be more beneficial for such a runtime analysis than writing data to the disk [10]. The goal of the autonomic data steaming service is to stream data from the live simulation to support remote runtime analysis and visualization at PPPL while minimizing overheads on the simulation, adapting to network conditions, and eliminating loss of data. The application workflow consists of following five core services:

1. The **Simulation Service** (**SS**) executes in parallel on 6K processors of Seaborg an IBM SP machine at NERSC and generates data at regular intervals that has to be transferred at runtime for analysis and visualization at PPPL, and archived at data stores at PPPL or ORNL.
2. The **Data Analysis Service** (**DAS**) runs on a 32 node cluster located at PPPL. This service analyzes and visualizes the steamed data.
3. The **Data Storage Service** (**DSS**) archives the streamed data using the Logistical Networking backbone [11], which builds a Data Grid of storage services located at ORNL and PPPL.
4. The **Autonomic Data Streaming Service (ADSS)** is constructed using the Accord autonomic services architecture and manages the streaming of data from SS (at NERSC) to DAS (at PPPL) and DSS (at PPPL/ORNL). It is a composite service composed of two services:
   a. The **Buffer Manager Service (BMS)** manages the buffers allocated by the service based on the rate and volume of data generated by the simulation and determines the granularity of blocks used for data transfer.
   b. **Data Transfer Service (DTS)** manages the transfer of blocks of data from the buffers to remote services for analysis and visualization at PPPL, and archiving at PPPL or ORNL. The data transfer service uses the IBP [12] protocol to transfer data.

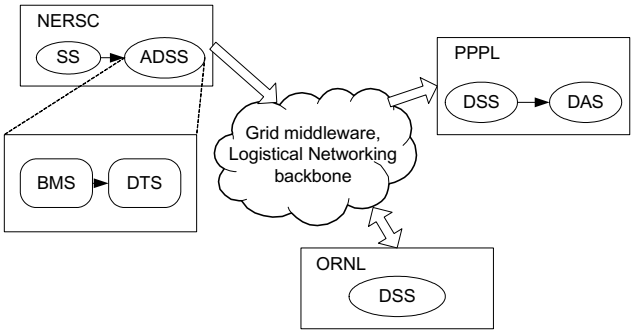

Figure 4. The autonomic data streaming application.

As mentioned above, the objective of ADSS is to minimize overheads of data transfer on the simulation, adapt the transfer to network conditions, and ensure that there is no loss of data. Three self-managing scenarios using ADSS are described below.

B. Self-Managing Scenarios

Scenario 1: Self-optimizing behavior of BMS.

This scenario illustrates the self-optimizing behavior of the BMS. BMS selects the appropriate blocking technique, orders blocks in the buffer and optimizes the size of the buffer(s) used to ensure low latency high performance steaming and minimize the impact on the execution of the simulation. The adaptations are based on the current state of the simulation and more specifically the following three runtime parameters. (1) The data generation rate, which is the amount of data generated per iteration divided by the time required for the iteration, and can vary from 1 to 400 Mbps depending on the domain decomposition and the type of analysis to be performed. (2) The network connectivity and the network transfer rate. The latter is limited by the 100 Mbps link between NERC and PPPL. (3) The nature of data being generated in the simulation, e.g., parameters, 2D surface data or 3D volume data. BMS provides three algorithms:

- Uniform Buffer Management: This algorithm divides the data into blocks of fixed sizes, which are then transmitted by the DTS. This static algorithm is more suited for the simulations generating data at a small or medium rate (50Mbps). Using smaller block sizes have significant advantages at the receiving end as less time is required for decoding the data and processing it for analysis and visualization.
- Aggregate Buffer Management: This algorithm aggregates blocks across iterations and the DTS transmits these aggregated blocks. This algorithm is suited for high data generation rates, i.e., between 60-400 Mbps.
- Priority Buffer Management: This algorithms orders data blocks in the buffer based on the nature of the data.

For example, 2D data blocks containing visualization or simulation parameters are given higher priority as compared to 3D raw volume data. To enable adaptations, the BMS exports two sensors, “DataGenerationRate” and “DataType”, and one actuator, “BlockingAlgorithm” as part of its control port shown in Figure 5. This document describes the name, type, message format and protocol details for each sensor/actuator. Further, the BMS self-optimization behavior is governed by the rule shown in Figure 6, which states that if the data generation rate is greater than the peak network transfer rate (i.e., 100 Mps), the aggregate buffer management is used otherwise the uniform buffer management algorithm is used.

```
<controlPort name=”BMS_controlPort” service=”BufferManagerService”>
  <types>
    <sensor name=”DataGenerationRate”>
      <element name=”DataGenerationRateReq” type=”string”/>
      <element name=”DataGenerationRateResp” type=”double”/>
    </sensor>
    <sensor name=”DataType”>
      <element name=”DataTypeReq” type=”string”/>
      <element name=”DataTypeResp” type=”string”/>
    </sensor>
    <actuator = name=”BlockingAlgorithm”>
      <element name=”BlockingAlgorithmReq” type=”string”/>
    </actuator>
  </types>

  <message name=”GetDataGenerationRateIn”>
    <part name=”body” element=”DataGenerationRateReq”/>
  </message>
  <message name=”GetDataGenerationRateOut”>
    <part name=”body” element=”DataGenerationRateResp”/>
  </message>
  <message name=”GetDataTypeIn”>
    <part name=”body” element=”DataTypeReq”/>
  </message>
  <message name=”GetDataTypeOut”>
    <part name=”body” element=”DataTypeResp”/>
  </message>
  <message name=”SetBlockingAlgorithm”>
    <part name=”body” element=”BlockingAlgorithmReq”/>
  </message>

  <portType name=”BMSControlPortType”>
    <operation name=”SensorDataGenerationRate”>
      <input message=”tns:GetDataGenerationRateIn”/>
      <output message=”tns:GetDataGenerationRateOut”/>
    </operation>
    <operation name=”SensorDataType”>
      <input message=”tns:GetDataTypeIn”/>
      <output message=”tns:GetDataTypeOut”/>
    </operation>
    <operation name=”ActuatorBlockingAlgorithm”>
      <input message=”tns:SetBlockingAlgorithm”/>
    </operation>
  </portType>
</controlPort>
```

Figure 5. The control port for BMS.

The resulting adaptation behavior is plotted in Figure 7. The figure shows that BMS switches to aggregate buffer management during simulation time intervals 75 sec to 150 sec and 175 sec to 250 sec, as the simulation data generation rate peaks to 100Mbps and 120 Mbps during these intervals. The aggregation is an average of 7 blocks. Once the data generation rate falls to 50Mbps, BMS switches back to the uniform buffer management scheme, and constantly sends 3 blocks of data on the network. Figure 7 (b) plots the percentage overhead on simulation execution with and without autonomic management. Overhead is computed as the absolute difference between the time required to generate data without the ADSS service and the time required to stream the data using ADSS service.

```
<rule name=”BlockingRule" attribute=”active”>
  <trigger name=”2D” sensor=”DataType” op=”EQ” value=”2D” type=”string”/>
  <trigger name=”DGR” sensor=”DataGenerationRate” op=”GT”
value=peakRate  type=”float”/>

  <when>
    <and>
      <operand trigger=”2D”/>
      <operand trigger=”DGR”/>
    </and>
  </when>
  <do>
    <action actuator=”BlockingAlgorithm”>
      <input value=”priorityAggregation” type=”string”/>
    </action>
  </do>

  <when>
    <and>
      <operand trigger=”2D”/>
        <not>
          <operand trigger=”DGR”/>
        </not>
    </and>
  </when>
  <do>
    <action actuator=”BlockingAlgorithm”>
      <input value=”priority” type=”string”/>
    </action>
  </do>

  <when>
    <and>
      <operand trigger=”DGR”/>
      <not>
        <operand trigger=”2D”/>
      </not>
    </and>
  </when>
  <do>
    <action actuator=”BlockingAlgorithm”>
      <input value=”aggregate” type=”string”/>
    </action>
  </do>

  <else>
    <action actuator=”BlockingAlgorithm”>
      <input value=”uniform” type=”string”/>
    </action>
  </else>
</rule>
```

Figure 6. The adaptation rule for BMS.

The plot shows that the BMS switches from uniform buffer management to aggregate buffer management at data generation rates of around 80-90 Mbps. This increases the overhead slightly, however the overheads remains less than 5%. Without autonomic management, the overheads increase to about 10% for higher data rates as BMS continues to use uniform buffer management.

When the simulation service generates 2D visualization data in addition to 3D data, the priority buffer management algorithm is triggered. The 2D data blocks are given higher priority and are moved to the head to data transmission queue. As a result, transmission of the 2D data is expedited with almost no impact to the 3D data.

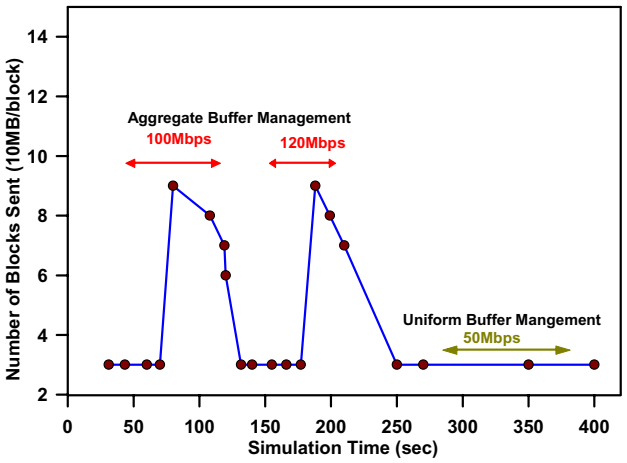

Figure 7(a). Self-optimization behaviors of the Buffer Management Service (BMS) – BMS switches between uniform blocking and aggregate blocking algorithms based on application data generation rates, network transfer rates and the nature of data generated.

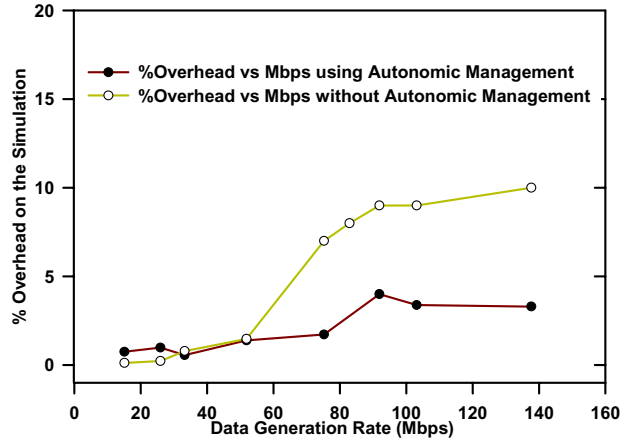

Figure 7(b). Percentage overhead on simulation execution with and without autonomic management.

```
<rule name="SplitRule" attribute="active">
  <trigger name="SmallNTR" sensor="NetworkTransferRate"
           op="LT" value=lowerthreshold type="float"/>
  <trigger name="LargeNTR" sensor="NetworkTransferRate"
           op="GT" value=upperthreshold type="float"/>
  <trigger name="ADSSNum" sensor="NumOfADSS" op="LT"
           value=num type="integer"/>

  <when>
    <and>
      <operand trigger="SmallNTR"/>
      <operand trigger="ADSSNum"/>
    </and>
  </when>
  <do>
    <action actuator="Accord:NewInstances">
      <input value="BMS" type="service"/>
    </action>
    <action actuator="Accord:LoadRules">
      <input value="BMS" type="service"/>
      <input value="BMSRuleName" type="string"/>
    </action>
    <action actuator="Accord:NewInstances">
      <input value="DTS" type="service"/>
    </action>
    <action actuator="Accord:LoadRules">
      <input value="DTS" type="service"/>
      <input value="DTSRuleName" type="string"/>
    </action>
  </do>

  <when>
    <operand trigger="LargeNTR"/>
  </when>
  <do>
    <action actuator="Accord:GetInstances">
      <input value="BMS" type="service"/>
      <output value="BMSInstanceList" type="serviceInstanceList"/>
    </action>
    <action actuator="Accord:DelInstances">
      <input value="BMSInstanceList" type="serviceInstanceList"/>
      <input value="number" type="integer"/>
    </action>
  </do>
</rule>
```

Figure 8. The adaptation rule for ADSS.

Scenario 2: Self-configuring/self-optimizing behavior of ADSS.

The effectiveness of the data transfer between the simulation service at NERSC and the analysis/visualization service at PPPL depends on the network transfer rate, which depends on data generation rates and/or network conditions. Falling network transfer rates can lead to buffer overflows and require the simulation to be throttled to avoid data loss. One option to maintain data throughputs is to use multiple data streams. Of course, this option requires multiple buffers and hence uses more of the available memory. Implementing this option requires the creation of multiple instances of ADSS. In this scenario, ADSS monitors the effective network transfer rate, and when this rate dips below a threshold, the service causes another instance of the ADSS to be created and incorporated into the workflow. Note that the maximum number of ADSS instances possible is predefined. Similarly, if the effective data transfer rate is above a threshold, the number of ADSS instances is decreased to reduce memory overheads. The upper and lower thresholds have been determined using experiments in [10].

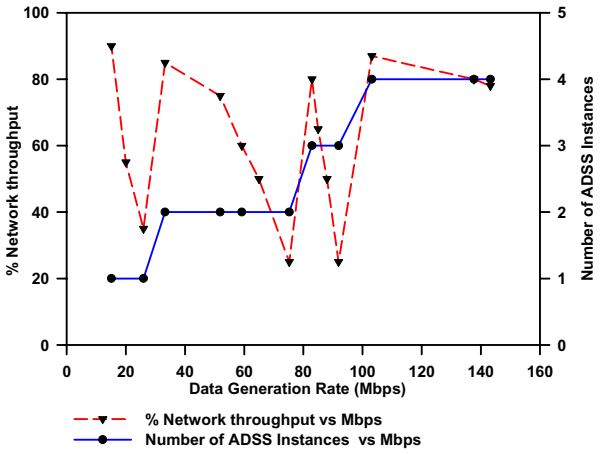

Figure 9. Effect of creating new instances of the ADSS service when the %Network Throughput dips to below the user defined 50% threshold.

The self-configuration behavior of ADSS is governed by the rule shown in Figure 8. When the network transfer rate is below a pre-defined threshold, ADSS will use Accord to create new instances of ADSS including BMS and DTS and

load corresponding rules into the new BMS and DTS instances to enable interactions between them. When the network transfer rate is above a pre-defined threshold, ADSS obtains the list of exiting ADSS instances using the Accord runtime, and deletes a pre-defined number of instances.

The resulting behaviors are plotted in Figure 9. This figure plots the percentage of network throughput, which is the difference between the current network transfer rate and the maximum network rate between PPPL and NERSC, i.e., 100 Mbps. The figure shows that the number of ADSS instances first increases as the network throughput dips below the 50% threshold (corresponding to data generation rates of around 25 Mbps in the plot), as defined by the rule in Figure 8. This causes the network throughput to increase to above 80%. Even more instances of ADSS services are created at data generation rates of around 40 Mbps and the network throughput once again jumps to around 80Mbps. The ADSS instances increase until the limit of 4 is reached.

Scenario 3: Self-healing behavior of ADSS

This scenario addresses data loss in the cases of extreme network congestion or network failures. These cases cannot be addressed using simple buffer management or replication. One option in these cases to avoid loss of data is to write data locally at NERSC rather than streaming. However, this data will not be available for analysis and visualization until the simulation complete, which could be days. Writing data to the disk also causes significant overheads to the simulation [10]. ADSS addresses these cases by temporarily or permanently switching the streaming of the data to the DSS at ORNL instead of PPPL. NERSC and ORNL are connected by a low latency [13] link which has a lower probability of being saturated. The data can be later transmitted from ORNL to PPPL. Congestion is detected by observing the buffer - when the buffer is filled to a capacity, the ADSS switches subsequent streaming to ORNL, and when the buffer is no longer saturated, switches the steaming back to PPPL. If the service observes that buffer is being continuously saturated, it infers that there is a network failure and permanently switches the streaming to ORNL. In this case, the blocks already in the PPPL buffer are transferred to the ORNL queue. The rule specifying this self-management behavior is listed in Figure 10.

The resulting self-healing behavior is plotted in Figure 11. The figure shows that as the ADSS buffer(s) get saturated, the data streaming switches to the DSS at ORNL, and when the buffer occupancy falls below 20% it switches back to PPPL. Note that while the data blocks are written to ORNL, data blocks already queued for transmission to PPPL continue to be streamed. The figure also shows that, at simulation time 1500 (X axis), the PPPL buffers once again get saturated and the streaming switches to ORNL. If this persists, the steaming would be permanently switched to ORNL.

```
<rule name="TransferRule" attribute="active">
  <trigger name="transferFailed" sensor="DataTransfer"
           op="EQ" value="0" type="integer"/>
  <trigger name="transferSwitch" sensor="NumOfSwitches"
           op="LT" value=switchThreshold type="integer"/>

<when>
  <and>
    <operand trigger="transferFailed"/>
    <operand trigger="transferSwitch"/>
  </and>
</when>

<do>
  <action actuator="TransferAlgorithm">
    <input value="remote" type="string"/>
  </action>
</do>

<when>
  <not>
    <operand trigger="transferSwitch"/>
  </not>
<do>
  <action actuator="TransferAlgorithm">
    <input value="remote" type="string"/>
  </action>
  <action actuator="Accord:SetRuleAttribute">
    <input value="TransferRule" type="string"/>
    <input value="inactive" type="string"/>
  </action>
</rule>
```

Figure 10. The interaction/adaptation rule for ADSS.

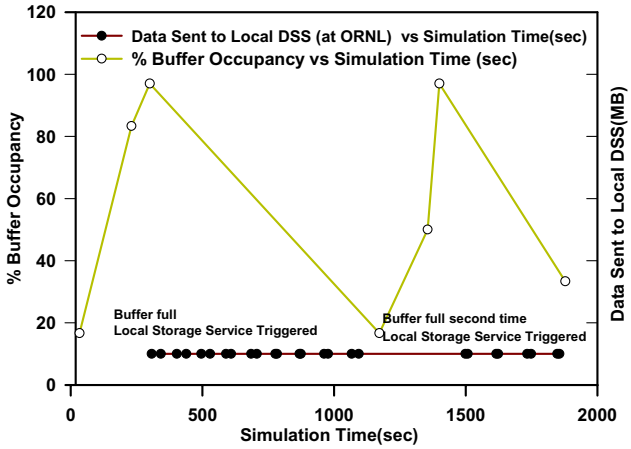

Figure 11 Effect of switching from the DSS at PPPL to the DSS ORNL in response to network congestion and/or failure.

IV. Related Work

Management behaviors tend to be application-specific. For example the selection of parameters or algorithms and the usage of specific communication and coordination paradigms are determined by application requirements, formulations and inputs, and its execution context. However, the approaches and mechanisms to support runtime self-management/adaptation are independent of applications.

Existing systems that support autonomic management can be classified based on the way in which the management behaviors are specified and enforced. Management behaviors can be either statically or dynamically specified. Several existing systems use templates [14] or adaptation classes [15], special scripts [16] or languages [17] to statically specify management behaviors a priori. However, a key drawback of these static approaches is that all the possible adaptation must be known a priori and coded into

the application. If new adaptations are required or if application requirements change, the application code has to be modified and the application possibly re-compiled. Accord enables adaptive management behaviors, in the form of rules, to be dynamically added, removed and modified at runtime. Other systems that support dynamic specification of adaptations include [18] and [19].

Different approaches have also been proposed to enforce management behaviors. The mobile agent approach presents power and flexibility in specification and deployment [20], but requires virtual machines or milieus to support the execution of mobile agents and may lead to possible security problems such as masquerading, denial of service, unauthorized access, eavesdropping, alteration, repudiation, etc. [21]. Code instrumentation [22], superimposition [23], and wrapping [24] have also been used to enforce management behaviors. When source code is not accessible, filters [25] and proxies [26] can be interposed between services to manipulate the interacting messages or re-direct messages to different services, to introduce dynamic adaptations into the execution of the application.

V. Conclusion

This paper presented the Accord services architecture for self-managing Grid applications. Accord enables the development of autonomic services and the formulation of autonomic applications as the dynamic composition of autonomic services, where the runtime computational behavior of the services as well as their compositions and interactions can be managed at runtime using dynamically injected rules. As a result, applications are capable of adapting their runtime behaviors to deal with the dynamism and uncertainty of Grids and Grid applications. An autonomic data streaming application is used to illustrate the self-managing behaviors enabled by Accord.

As platforms change and applications evolve, adaptation rules may need to be changed and thresholds may need to be modified. In the current Accord prototype, rule maintenance is manual, however, autonomic methods for deriving thresholds [27] is under investigation.

References

[1] M. Parashar and J.C. Browne, Conceptual and Implementation Models for the Grid, *Proceedings of the IEEE, Special Issue on Grid Computing*, IEEE Press, Vol. 93, No. 3, pp 653 – 668, March 2005.

[2] 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, 2006.

[3] H. Liu, A Programming System for Autonomic Self-Managing Applications, *PhD thesis*, Rutgers University, October 2005.

[4] OWL Web Ontology Language Overview, http://www.w3.org/TR/2004/REC-owl-features-20040210/, 2004.

[5] Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl, 2001.

[6] M. Agarwal and M. Parashar, Enabling Autonomic Compositions in Grid Environments, *Proceedings of the 4th International Workshop on Grid Computing (Grid 2003),* Phoenix, AZ, USA, IEEE Computer Society Press, pp 34 - 41, November 2003.

[7] W. M. P.Van Der Aalst, A. H. M. Ter Hofstede, B Kiepuszewski and A P. Barros, Workflow patterns, *In distributed and parallel databases*, 14(3), pages 5-51, 2003.

[8] Axis, http://ws.apache.org/axis/.

[9] Globus Toolkit, http://www.globus.org/toolkit/.

[10] V. Bhat, S. Klasky, S. Atchley, M. Beck, D. McCune and M. Parashar, High Performance Threaded Data Streaming for Large Scale Simulations. *Proceedings of the 5th International Workshop on Grid Computing (Grid 2004),* Pittsburgh, PA, USA, IEEE Computer Society Press, pp 243-250, November 8, 2004.

[11] J.S. Plank and M. Beck, The Logistical Computing Stack -- A Design For Wide-Area, Scalable, Uninterruptible Computing, *DNS: 2002 Dependable Systems and Networks, Workshop on Scalable, Uninterruptible Computing*, Bethesda, Maryland, USA, June, 2002.

[12] J. S. Plank, M. Beck, W. R. Elwasif, T. Moore, M. Swany and R. Wolski, The Internet Backplane Protocol: Storage in the Network, *NetStore99: The Network Storage Symposium*, Seattle, WA, USA, 1999.

[13] Energy Sciences Network, http://www.es.net/.

[14] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J Schopf, G. Shao, S. Smallen, N. Spring, A. Su and D. Zagorodnov, Adaptive Computing on the Grid Using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, April 2003.

[15] P. Boinot, R. Marlet, J. Noy, G. Muller and C. Cosell, Declarative Approach for Designing and Developing Adaptive Components. *Proceedings of 15th IEEE International Conference on Automated Software Engineering*, pages 111–119, 2000.

[16] D. Beazley and P. Lomdahl, Controlling the data glut in Large-Scale Molecular-Dynamics Simulations. *Computers in Physics*, 11(3), 1997.

[17] G. Duzan, J. Loyall and R. Schantz, Building adaptive distributed applications with middleware and aspects. *In the 3rd International Conference on Aspect-oriented Software Development*, pages 66–73, Lancaster, UK, 2004.

[18] S. M. Sadjadi and P. K. McKinley, Transparent Self-Optimization in Existing CORBA Applications. *In the first international conference on autonomic computing*, NYC, NY, USA, 2004.

[19] C. Ururahy, N. Rodriguez and R. Ierusalimschy, Alua: Flexibility for parallel programming. *Computer Languages*, 28(2), 2002.

[20] B. Kohn, E. Kraemer, D. Hart and D. Miller, An Agent-based Approach to Dynamic Monitoring and Steering of Distributed Computations. *In International Association of Science and Technology for Development (IASTED)*, Las Vegas, Nevada, 2000.

[21] S. Fischmeister, Mobile code paradigms. http://www.softwareresearch.net/site/teaching/WS0203/PDFdocs.2002.

[22] S. Parker and C. Johnson, An Integrated Problem Solving Environment: The SCIRun Computational Steering Environment. *In HICCS-31*, 1998.

[23] J. Bosch, Superimposition: A Component Adaptation Technique, *Information and Software Technology*, 1999.

[24] E. Truyen, W. Joosen, P. Verbaeten and B. N. Jorgensen, On Interaction Refinement in Middleware. *In the 5th International Workshop on Component-Oriented Programming*, 2000.

[25] S.R. Ponnekanti and A. Fox, SWORD: A Developer Toolkit for Web Service Composition. *In Proceedings International WWW Conference(11)*, Honolulu, Hawaii, USA, 2002.

[26] J. Ray, N. Trebon, R. C. Armstrong, S. Shende and A. Malony, Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment: A Case Study. *In the 18th International Parallel and Distributed Processing Symposium (IPDPS04)*, Santa Fe, NM, USA, 2004.

[27] K. Appleby and G. Goldszmidt, Using automatically derived load thresholds to manage compute resources on-demand, *In the 9th IFIP/IEEE International Symposium on Integrated Network Management*, Nice-Acropolis, Exhibition Hall, Nice, France, 15-19 May 2005.