Self-adaptive Architectures for Autonomic Computational Science

Shantenu Jha[1], Manish Parashar[2], and Omer Rana[3]

[1] Center for Computation & Technology and Department of Computer Science, Louisiana State University, USA, and e-Science Institute, University of Edinburgh, UK
sjha@cct.lsu.edu
[2] Department of Electrical & Computer Engineering, Rutgers University, USA
parashar@rutgers.edu
[3] School of Computer Science, Cardiff University, UK
o.f.rana@cs.cf.ac.uk

Abstract. Self-adaptation enables a system to modify it's behaviour based on changes in its operating environment. Such a system must utilize monitoring information to determine how to respond either through a systems administrator or automatically (based on policies pre-defined by an administrator) to such changes. In computational science applications that utilize distributed infrastructure (such as Computational Grids and Clouds), dealing with heterogeneity and scale of the underlying infrastructure remains a challenge. Many applications that do adapt to changes in underlying operating environments often utilize ad hoc, application-specific approaches. The aim of this work is to generalize from existing examples, and thereby lay the foundation for a framework for Autonomic Computational Science (ACS). We use two existing applications – Ensemble Kalman Filtering and Coupled Fusion Simulation – to describe a conceptual framework for ACS, consisting of *mechanisms, strategies and objectives*, and demonstrate how these concepts can be used to more effectively realize pre-defined application objectives.

1 Introduction

Developing and deploying self-adaptive applications over distributed infrastructure provides an important research challenge for computational science. Significant recent investments in national and global cyberinfrastructure, such as the European EGEE/EGI, the US TeraGrid, the Open Science Grid and the UK National Grid Service, have the potential for *enabling* significant scientific insights and progress. The use of such infrastructure in *novel* ways has still not been achieved however, primarily because of the inability of applications that are deployed over such infrastructure to adapt to the underlying heterogeneity, fault management mechanisms, operation policies and configuration parameters associated with particular software tools and libraries. This problem is only compounded by new and more complex application formulations such as those

D. Weyns et al. (Eds.): SOAR 2009, LNCS 6090, pp. 177–197, 2010.

based on dynamic data. Tuning and adapting an application is often left to the skills of specialist developers, with limited time and motivation to learn the behaviour of yet another deployment environment. In applications where automation has been achieved, this generally involves understanding specific application behaviours, and in some instances, specialised capabilities offered by the underlying resources over which the application is to be executed. Generalising from such applications and developing a more generic *framework* has not been considered. We take an application-centric approach to better understand what such a framework should provide, focusing on how: (i) applications can be characterized, to enable comparison across different application classes – the basis of our previous work [17]; (ii) understanding tuning mechanisms and associated strategies that can be applied to particular application classes. The longer term objective of this work is to derive adaptation *patterns* that could be made available in a software library, and directly made use of when constructing distributed scientific applications.

It is useful to note that the development of self-adaptive systems has generally followed either: (i) a top-down approach, where overall system goals need to be achieved through the modification of interconnectivity or behaviour of system components – realized through a system manager; (ii) a bottom-up approach, where local behaviour of system components needs to be aggregated (without a centralized system manager) to generate some overall system behaviour. In our approach we are primarily focused on (i), as this relates closely with existing approaches within computational science. However, we also believe that approach (ii) could be used as an initial phase, whereby resource ensembles could be dynamically formed within Grid and Web-based communities, using self-organization approaches, and as discussed in Serugendo et al. [24]. Such an approach would enable application characteristics or resource characteristics to be used as an initial phase to cluster resources/applications prior to utilizing an autonomic deployment strategy.

2 A Conceptual Framework for Autonomic Computational Science

A conceptual framework to support autonomic computational science applications is presented in this section. We identify possible architectures, and relate the approaches discussed here to reflective middleware and control loop models. In the context of Grid computing environments considered in this work, the use of a shared, multi-tasking environment is assumed. An application (or user) in such an environment requests access to a pre-determined number of resources (CPUs, memory, etc), and it is the responsibility of the resource management system (generally a batch queuing system) to ensure that access to these resources is granted over the requested time interval. The resource manager does not (in most cases) provide any quality of service guarantees.

2.1 The Autonomic Computing Paradigm

The autonomic computing paradigm is modelled after the autonomic nervous system and enables changes in its essential variables (e.g., performance, fault, security, etc.) to trigger changes to the behavior of the computing system such that the system is brought back into equilibrium with respect to the environment [16]. Conceptually, an autonomic system requires: (a) sensor channels to sense the changes in the internal state of the system and the external environment in which the system is situated, and (b) motor channels to react to and counter the effects of the changes in the environment by changing the system and maintaining equilibrium. The changes sensed by the sensor channels have to be analyzed to determine if any of the essential variables have gone out of their viability limits. If so, it has to trigger some kind of planning to determine what changes to inject into the current behavior of the system such that it returns to the equilibrium state within the new environment. This planning requires knowledge to select the right behavior from a large set of possible behaviors to counter the change. Finally, the motor neurons execute the selected change. Sensing, Analyzing, Planning, Knowledge and Execution are thus the keywords used to identify an autonomic computing system. A common model based on these ideas was identified by IBM Research and defined as *MAPE* (Monitor-Analyze-Plan-Execute) [19]. There are, however, a number of other models for autonomic computing [23], [11] – in addition to work in the agent-based systems community that share commonalities with the ideas presented above. In what follows, we explore the applications of this paradigm to support computational science.

2.2 Conceptual Architectures for ACS

Looking at existing practices in computational science, two corresponding conceptual architectures can be observed, which are described below. These architectures are composed of the application, a resource manager that allocates, configures and tunes resources for the application, and an autonomic manager that performs the autonomic tuning of application and/or system parameters. Figure 1 illustrates the first conceptual architecture, where the application and resources are characterized using a number of *dynamically modifiable* parameters/variables that have an impact on the overall *observed behaviour* of the application. Each of these parameters has an associated range over which it can be modified, and these constraints are known *a priori*. The autonomic tuning engine alters these parameters based on some overall *required behaviour* (hereby referred to as the application objective) that has been defined by the user. Tuning in this case is achieved by taking into account, for example, (i) historical data about previous runs of the application on known resources, obtained using monitoring probes on resources; (ii) historical data about previous selected values of the tunable parameters; (iii) empirically derived models of application behavior; (iv) the specified tuning mechanism and strategy; etc.

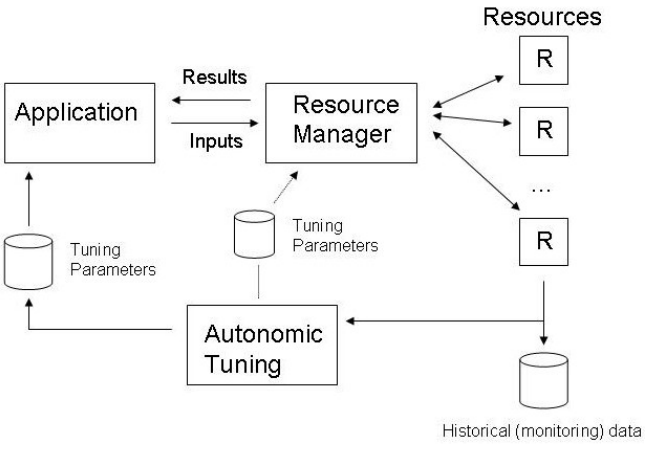

Fig. 1. Autonomic tuning *of* application and resource manager. *R* refers to a computational resource. parameters.

For example, an autonomic tuning mechanism in this architecture may involve changing the *size* of the application (for instance, the number of data partitions generated from a large data set, the number of tiles from an image, etc.), or the set of parameters over which execution is being requested. This tuning is used to make desired tradeoffs between quality of solution, resource requirements and execution time or to ensure that a particular Quality of Service (QoS) constraint, such as execution time, is satisfied.

A variant, also illustrated in Figure 1, involves updating the resource manager based on information about the current state of the application. As an example of such an approach, consider an application using dynamic structured adaptive mesh refinement (SAMR) [9] techniques on structured meshes/grids. Compared to numerical techniques based on static uniform discretization, SAMR methods employ locally optimal approximations and can yield highly advantageous ratios for cost/accuracy by adaptively concentrating computational effort and resources to regions with large local solution error at runtime. The adaptive nature and inherent space-time heterogeneity of these SAMR implementations lead to significant challenges in dynamic resource allocation, data-distribution, load balancing, and runtime management. Identifying how the underlying resource management infrastructure should adapt to changing SAMR requirements (and possibly vice versa) provides one example of the architecture in Figure 1.

Figure 2 illustrates another conceptual architecture, where the application is responsible for driving the tuning of parameters, and choosing a tuning strategy. The autonomic manager is now responsible for obtaining monitoring data from resource probes and the strategy specification (for one or more objectives to be realized) from the application. Tuning now involves choosing a resource management strategy that can satisfy the objectives identified by the application. This approach primarily relates to the *system-level* self-management described

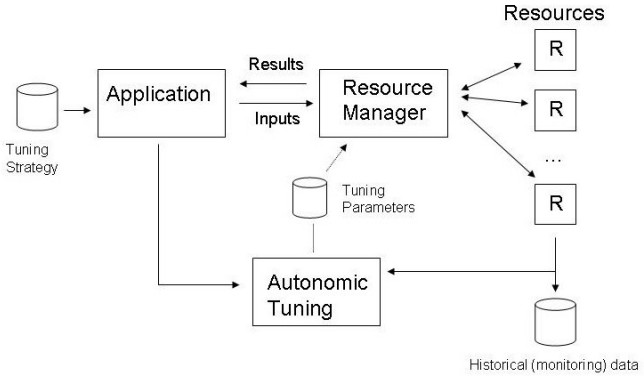

Fig. 2. Autonomic tuning *by* application. R refers to a computational resource.

in Section 1. An example of such an architectural approach is the use of resource reservation to achieve a particular QoS requirement. The G-QoSM framework [1] demonstrates the use of such an architecture, involving the use of a soft real-time scheduler (DSRT) along with a bandwidth broker to make resource reservation over local compute, disk and network capacity, in order to achieve particular application QoS constraints.

We reiterate that the conceptual architectures – tuning by and of applications, defined above are not exhaustive, but provide an initial formulation with a view towards understanding a set of applications that we discuss.

2.3 A Conceptual Framework

A conceptual framework for ACS can be developed based on the conceptual architectures discussed above (and illustrated in Figures 1 and 2), comprised of the following elements:

Application-level Objective (AO): An AO refers to an application requirement that has been identified by a user – somewhat similar to the idea of a "goal" in Andersson et al. [2]. Examples of AO include increase throughput, reduce task failure, balance load, etc. An AO needs to be communicated to the autonomic tuning component illustrated in Figures 1 and 2. The autonomic tuning component must then interact with a resource manager to achieve these objectives (where possible). There may be multiple AOs that need to be satisfied, and there may be relationships between them.

Mechanism: a mechanism refers to a particular action that can be used by the application or the resource manager (from Figures 1 and 2) to achieve an AO. A mechanism is triggered as a consequence of some detected event(s) in the application or it's environment over some pre-defined duration.

Hence, a mechanism m may be characterized in terms of: $\{m_i^e\}$ – the set of events that lead to the triggering (activation) of the mechanism; $\{m_i\}$ – the

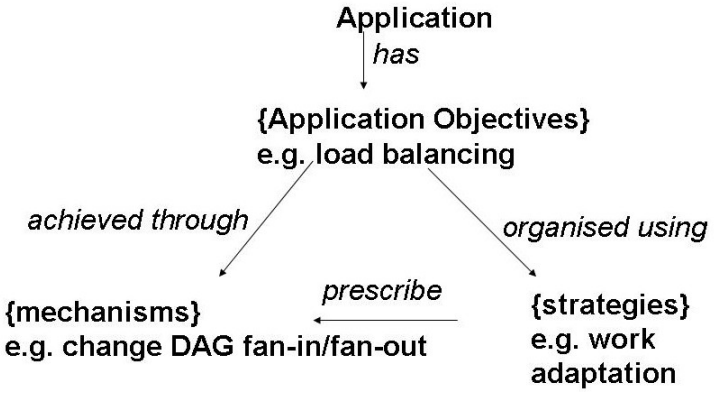

Fig. 3. Relationship between AO, mechanism and strategy. An Application can have multiple application objectives (indicated by {...} notation). Each objective can be achieved through a set of mechanisms based on strategies. The set and organization of mechanisms used to achieve an AO is prescribed by the specific strategy used.

set of data inputs to the mechanism; $\{m_o\}$ – the set of output data generated; and $\{m_o^e\}$ – the set of output events that are generated during the execution of the mechanism or after its completion. An example of a mechanism includes *file staging*. In this mechanism, $\{m_i\}$ corresponds to one or more file and resource references prior to the staging process has started, $\{m_o\}$ corresponds to the file references after staging has completed, $\{m_i^e\}$ refers to the input events that trigger the file staging to begin, and $\{m_o^e\}$ corresponds to output events that are generated once file staging is completed.

Comparing with the view of a mechanism in [2], where the action undertaken is classified as being: (i) structural or parametric (primarily parametric in this framework); (ii) system or human assisted; (iii) centralized or decentralized; (iv) local or global; (v) short, medium or long term; (vi) best effort or guaranteed; and (vii) event or time triggered. All of these classifications also apply in the context of the ACS presented here.

Strategy: One or more strategies may be used to accomplish a particular AO. A strategy is defined in terms of one or more partially-ordered mechanisms. A strategy may be specified manually (by a systems administrator, for instance) or constructed using an autonomic approach. The focus of this particular work is on the latter. A strategy is managed by the autonomic tuning component illustrated in Figures 1 and 2, and may be maintained as a collection of templates that are adapted depending on the application or the resource manager properties.

Note that given AO can be fulfilled by multiple strategies, and the same strategy can be used for different AO. For example, self-configuration can be used for both load-balancing as well as higher-throughput. Figure 3 illustrates the relationship between these concepts.

2.4 Relationship to Reflective Middleware

Reflective middleware [21] attempts to adapt the behaviour of an application depending on the deployment platform. One primary motivation for this work stems from the need to expose the dynamic state of the underlying middleware (and provide tuning capability) to the application, so that it can utilize more effective control strategies to adapt it's behaviour as the underlying deployment infrastructure changes (assuming that an instance of the same reflective middleware is available on these different infrastructures). Two key features enable this capability within the middleware: (i) "reflection" to enable the system to reason about and act upon itself, through a representation of its own behaviour, amenable to examination and change; (ii) "causal connectivity" to enable any changes made to the system's self-representation to impact it's actual state and behavior, and vice-versa. A component-based approach is generally adopted to enable different component instances to be chosen depending on the underlying deployment platform.

Reflective middleware relies on the existence of multiple component implementations that can be deployment over multiple platforms. In addition, such middleware relies on the ability of the underlying platform to expose it's state to the middleware, so that it can be managed externally. This is not always possible, as many scheduling engines, for instance, would not allow an application to alter job execution priorities on a given platform. From Section 2.2, in the conceptual architecture in figure 1, the autonomic tuning engine is considered to be external to the application. In figure 2, the application interacts with the tuning engine and not directly with the resource manager. Hence, behaviour or structural reflection capabilities could be used by the tuning engine, but not directly by the application. In many practical application deployments, it is unrealistic to assume the availability of specialist middleware on externally hosted platforms.

There are three key differences between the approach advocated here and concepts identified in reflective middleware efforts: (i) tuning of the application is *logically* external to the application and undertaken through a tuning engine; (ii) the tuning engine does not rely on the availability of specialist modules to be hosted/executed by the resource manager, instead relying on existing tuning parameters that are already made available by the resource manager; (iii) there is also generally a separation between adaptation policy and mechanisms – both of which can be dynamic, and combined in different ways to get different autonomic behaviors. In the reflective middleware this is not the case - policy and mechanisms are generally integrated in middleware design.

Andersson et al. [3] take a wider view and suggest how self-adaptation could be supported in software systems through reflection. They describe a self-adaptive software system as one that is able to "change it's behaviour by reflecting on itself". In this work, reflection is primarily associated with developing a meta-model of computation undertaken by a system. The meta-model is proposed as the basis for allowing a computational system to reason about and act upon itself. Hence, two types of activities are supported at such a meta-level,

"introspection" and "intercession". Introspection is the process of inspection and reasoning about the system, whereas intercession is the subsequent modification of the system's meta-model. Self-Representation, Reflective Computation and Separation of Concerns are used to characterize the properties of a reflective system. The overall framework provided in this work provides a useful basis for developing adaptive systems that can be driven through the development of a meta-model and a suitable representation of a domain-model. However, developing such a meta-model (or domain-model) is difficult in complex applications. In our work, we therefore do not require a meta-model to be created, instead relying on a tuner component that is able to observe the outputs only, and not the internal working of a system. In the same context, when considering the tuning *of* an application, the granularity of what can be modified is also limited in many realistic applications, and access to granularity at the level of classes, objects, methods and method calls (as advocated in [3]) is often not possible.

2.5 Relationship to Control Loop Models

It is useful to note that the MAPE architecture identified in section 2.1 shares similarities with architectures adopted in feedback control, such as Model Reference Adaptive Control (MRAC) [25] and Model Identification Adaptive Control (MIAC) [26]. MRAC relies on comparing existing system state with that of a known model, and using this response to drive a controller that manipulates the system, whereas MIAC relies on deriving known properties of the model from system function (observations and measurements). Both of these approaches have been found to be of most benefit when the system being controlled has limited capability, and where the control loop is explicit. In many computational science applications, control loops are often hidden / abstracted, or hard to identify in the same way as in the types of applications utilizing a *traditional* MRAC architecture. Where control loops do exist, either the construction of a model (to steer the controller) or the tunable parameters that can be externally modified are limited. It is also possible in distributed computational science application for multiple control loops to exist, one associated with each resource (or ensemble).

As outlined in [10] adaptation mechanisms may be external to an application and not hard-wired. Developing an autonomic tuner that is external to the application provides a separation between system capability and tuning strategies. In the conceptual architectures presented in section 2.2, we have identified an explicit control loop external to the application. It is also possible to use the catalogue of self-adaptive mechanisms derived from natural systems (as outlined in [10]), by combining a top down approach (as being advocated in this work) and top-down approaches based on 'emergent' coordination mechanisms.

Model-driven approaches in software engineering have also been proposed to maintain a link between high and low-level views of software. Such approaches advocate the inclusion of particular run-time configurable parameters that are added at design time to a model of the software being adapted. However, as Nierstrasz et al. [22] point out, certain types of anomalies arise only after the

software has been deployed (and cannot be known at design time) – thereby making it difficult to anticipate what and where to trace to observe the problematic behaviour. Nierstrasz et al. [22] propose the idea of a *model-centric* view that can take the context of deployment and execution into account, in order to control the scope of adaptations to be made to the software system. Our approach is closely aligned with this thinking, as we believe that the context is important in the types of tuning that can be supported. In the case of the application scenarios we describe, such context is based on the particular computing environment over which the application is deployed. A key difference from the work of Nierstrasz et al. [22] is that our adaptation is not at the same level of granularity – as [22] propose software modification at the level of source code and the dynamic addition of instrumentation code (through the use of an aspect-oriented approach). For many scientific applications, having access to source code is often not possible. Similarly, such applications may use external libraries (such as numeric libraries) that would be difficult to instrument directly. In our approach, therefore, identifying what needs to be monitored has to be specified beforehand – and we cannot overcome the constraints of anticipating where monitoring should take place.

3 Application Case Study

In Jha et al. [18] we provided a discussion of application vectors that may be used to characterize scientific applications. Every distributed application must, at a minimum, have mechanisms to address requirements for communication, coordination and execution, which form the vectors we use: Execution Unit, Communication (Data Exchange), Coordination, and Execution Environment. *Execution unit* refers to the set of pieces/components of the application that are distributed. *Communication (data exchange)* defines the data flow between the executions units. Data can be exchanged by messages (point-to-point, all-to-all, one-to-all, all-to-one, or group-to-group), files, stream (unicast or multicast), publish/subscribe, data reduction (a subset of messaging), or through shared data. *Coordination* describes how the interaction between execution units is managed (e.g. dataflow, control flow, SPMD (where the control flow in implicit in all copies of the single program), master-worker (tasks executed by workers are controlled by the master), or events (runtime events cause different execution units to become active.)) An *execution environment* captures what is needed for the application to run. It often includes the requirements for instantiating and starting the execution units (which is also referred to as *deployment*) and may include the requirements for transferring data between execution units as well as other runtime issues (e.g. dynamic process/task creation, workflow execution, file transfer, messaging (MPI), co-scheduling, data streaming, asynchronous data I/O, decoupled coordination support, dynamic resource and service discovery, decoupled coordination and messaging, decoupled data sharing, preemption, etc.).

Two applications are described in sections 3.1 and 3.2 which demonstrate how the conceptual architectures described in section 2.2 are utilized, and which make use of the application vectors described above.

3.1 Ensemble Kalman Filters

Ensemble Kalman filters (EnKF) are widely used in science and engineering [14]. EnKF are recursive filters that can be used to handle large, noisy data; the data can be the results and parameters of ensembles of models that are sent through the Kalman filter to obtain the true state of the data. EnKF-based History Matching for Reservoir simulations [13] is an interesting case of an application with irregular, hard-to-predict run time characteristics. The variation in model parameters often has a direct and sizable influence on the complexity of solving the underlying equations, thus varying the required runtime of different models (and consequently the availability of the results). Varying parameters sometimes also leads to varying systems of equations and entirely new scenarios. This increases the computational size requirements as well as memory requirements. For example as a consequence of the variation in size, the underlying matrix might become too large or even effectively lead to doubling the number of the system of equations, which could more than double the memory required to solve these system of equations. The forecast model needs to run to completion – which is defined as convergence to within a certain value. The run time of each model is unpredictable and uncorrelated with the run-time of models running on the same number of processors. At every stage, each model must converge, before the next stage can begin. Hence dynamically load-balancing to ensure that all models complete as close to each other as possible is the desired aim. The number of stages that will be required is not determined a priori. In the general case the number of jobs required varies between stages. Table 1 provides the tuning mechanisms used in the EnKF application.

We define T_c as the time in seconds it takes to complete a defined application workload – comprising of a fixed ensemble size (one hundred members) and number of stages (five iterations of ensemble runs followed by KF). Any consistent reduction in the total time to completion will eventually have a greater impact on larger runs with more stages. There are three main components that are necessary to consider in order to understand T_c . The first is the time that it takes to submit to the queuing system and time for file-transfers (in and out) – labelled as $t_{overhead}$, and which is typically small for these large, long-running

Table 1. Tuning Mechanisms in EnKF

Vectors	**Mechanisms**
Coordination	Centralized Data-store (SAGA) Pilot-Job (BigJob) abstraction
Communication	File staging, File indexing
Execution Environment	Centralized Scheduler Resource Selection/Management, Task re-execution, Task migration, Storage management, File caching, File distribution, Checkpointing

simulations. The second component is the time that the submitted jobs wait in queue for resources requested to become available – labelled as t_{wait}; the final component is the run-time that simulations actually take to complete – labelled as t_{run}. Thus, $T_c = t_{overhead} + t_{wait} + t_{run}$.

Experiments to determine T_c using different number of machines working concurrently towards a solution of the same application instance were performed; an increase in performance was measured by a reduced T_c for up to three machines. Although there were fluctuations in both the wait-time in queue and the time to complete the work-load, the fluctuations were dominated by the former. Therefore in an attempt to minimise wait-times in queue and thus to lower the overall T_c , this application attempts to launch jobs on multiple TeraGrid (TG) resources using a Batch Queue Predictor (BQP) [4] [8]– a tool available on a number TG resources that allows users to make bounded predictions about the time a job of a given size and duration will spend in the queue. The objective is to run a number of jobs corresponding to a different stage of EnKF execution. BQP-based prediction is given with a degree of confidence (probability) that the job will start before a certain deadline (i.e. the time in the queue) and quantile. Quantile value is a measure of repeatability; more precisely it is an indication of the probability that jobs of similar sizes and durations will have the same wait time. This information is vital when submitting jobs to various machines as the longer a job sits in the queue, the longer the delay for the entire stage. BQP provides the ability to predict, with given probability, which resource and when a job (with a specified number of processors and estimated run-time) is most likely to finish.

Figure 4 shows the results when using three TG machines: Ranger, QueenBee and Abe. Ranger is the flagship machine of the Texas Advanced Computing Centre; QueenBee (QB) the flagship machine of LONI and Abe a large machine at NCSA. This figure demonstrates how machine combinations can be used, with and without BQP. To ensure that each machine is used efficiently, it is necessary to undertake load balancing that takes account of the properties of each machine (such as queue times and predictions derived from BQP). This therefore becomes an application objective that could be achieved manually by an application scientist or supported through an autonomic strategy, as identified in Table 2. Hence, either the size of a job/task could be adapted based on what machines are available to run the application, or suitable resources could be discovered using queue prediction information derived from BQP. It is also possible to make use of BQP to autonomically chose resources for a particular job (based on resource properties), and to guide the selection of resource configuration on a predetermined machine. When using more than one machine, e.g., RQA-BQP, both the selection of the resources and the selection of resource configuration are variables. For RQA-BQP, it is possible that even though three resources are available, all jobs will be submitted to a single resource with much higher capacity or temporary lower load-factors (e.g, after a power-down/start-up). These results are further explained in [13].

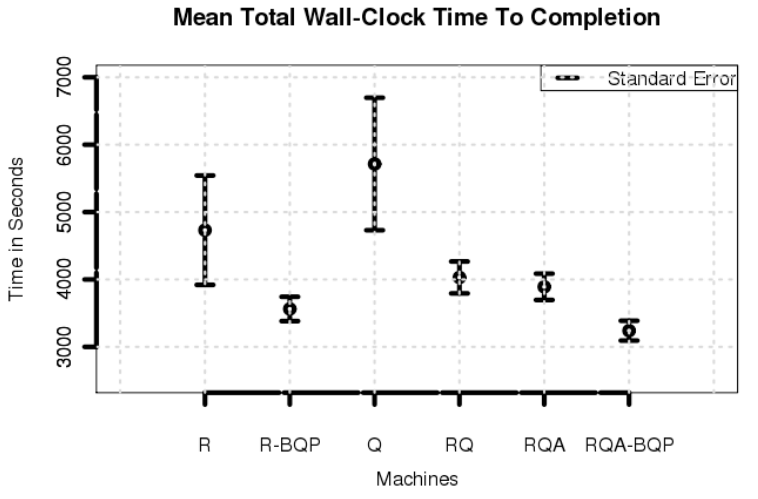

Fig. 4. Time to completion for different configurations. Left to Right: (i) Ranger (ii) Ranger when using BQP, (iii) QueenBee, (iii) Ranger and QueenBee, (iv) Ranger, QueenBee and Abe, (v) Ranger, QueenBee and Abe when using BQP. [12]

Table 2. EnKF application management through autonomic strategies

Application Objective	**Autonomic Strategy**
Load Balancing	**1. Adapt task mapping granularity based on system capabilities/state** File staging, File splitting/merging Task rescheduling, Task migration File distribution and caching, Storage Management **2. Resource Selection** Resource selection (using BQP), resource configuration update, Task rescheduling, Task migration File distribution and caching Storage Management
Scientific Fidelity	**Algorithmic Adaptivity** Change solvers -

This application therefore demonstrates that the use of queue information to guide the selection of resources and to configure resources, provides a better overall job execution performance. Rather than postponing such a decision until run-time, utilizing prior information to enable an autonomic manager to support resource selection can lead to better overall quality of service. This application

also demonstrates the use of the conceptual architecture in Figure 1, where each `R` utilizes a queuing system with BQP, and collects historical data. The `Autonomic Tuning` manager can then communicate with a `Resource Manager` to determine which resource to select. The EnKF application utilizing the resource manager does not need to be modified, as the tuning is undertaken external to the application.

3.2 Coupled Fusion Simulation

The DoE SciDAC CPES fusion simulation project [20] is developing an integrated, Grid-based, predictive plasma edge simulation capability to support next-generation burning plasma experiments, such as the International Thermonuclear Experimental Reactor (ITER). The typical application workflow for the project consists of coupled simulation codes, i.e., the edge turbulence particle-in-cell (PIC) code (GTC) and the microscopic MHD code (M3D), which run simultaneously on thousands of processors on separate HPC resources at supercomputing centers, requiring data to be streamed between these codes to achieve coupling. Furthermore, the data has to be processed en-route to the destination. For example, the data from the PIC codes has to be filtered through "noise detection" processes before it can be coupled with the MHD code. As a result, effective online management and transfer of the simulation data is a critical part for this project and is essential to the scientific discovery process.

As a result, a core requirement of these coupled fusion simulations is that the data produced by one simulation must be streamed live to the other for coupling, as well as, possibly to remote sites for online simulation monitoring and control, data analysis and visualization, online validation, archiving, etc. The fundamental objective being to efficiently and robustly stream data between live simulations or to remote applications so that it arrives at the destination just-in-time – if it arrives too early, times and resources will have to be wasted to buffer the data, and if it arrives too late, the application would waste resources waiting for the data to come in. A further objective is to opportunistically use in-transit resources to transform the data so that it is more suitable for consumption by the destination application, i.e., improve the quality of the data from the destination applications point of view. Key objectives/constraints for this application can be summarized as: (1) Enable high-throughput, low-latency data transfer to support near real-time access to the data; (2) Minimize overheads on the executing simulation; (3) Adapt to network conditions to maintain desired QoS – the network is a shared resource and the usage patterns typically vary constantly. (4) Handle network failures while eliminating loss of data – network failures usually lead to buffer overflows, and data has to be written to local disks to avoid loss, increasing the overhead on the simulation. (5) Effectively schedule and manage in-transit processing while satisfying the above requirements – this is particularly challenging due to the limited capabilities and resources and the dynamic capacities of the typically shared processing nodes.

These objectives can be effectively achieved using autonomic behaviors [5,6] based on a range of strategies and mechanisms. Autonomic behaviors in this case

Table 3. Tuning mechanisms in the Coupled Fusion Simulation application

Vectors	Mechanisms
Coordination	Peer-2-Peer interaction
Communication	Data Streaming, Events
Execution Environment	Storage Selection (local/remote), Resource Selection/Management, Task migration, Checkpointing Task execution (local/remote) Dynamic provisioning (provisioning of in-transit storage/processing nodes)

span (1) the application level, e.g., adapting solver behavior, adapting iteration count or using speculative computing by estimating the delayed data and rolling back if the estimation error exceeds a threshold; (2) the coordination level, e.g., adapting end-to-end and in-transit workflows as well as resources allocated to them; and (3) the data communication level, e.g., adaptive buffer management and adaptive data routing. Furthermore, this application also involves the use of a hybrid autonomic approach that combines policy-based autonomic management with model-based online control [7].

A conceptual overview of the overall architecture is presented in Figure 5. It consists of two key components. The first is an application-level autonomic data streaming service, which provides adaptive buffer management mechanisms and proactive QoS management strategies, based on online control and governed by user-defined polices, at application end-points. The second component operates at the in-transit level, and provides scheduling mechanisms and adaptive

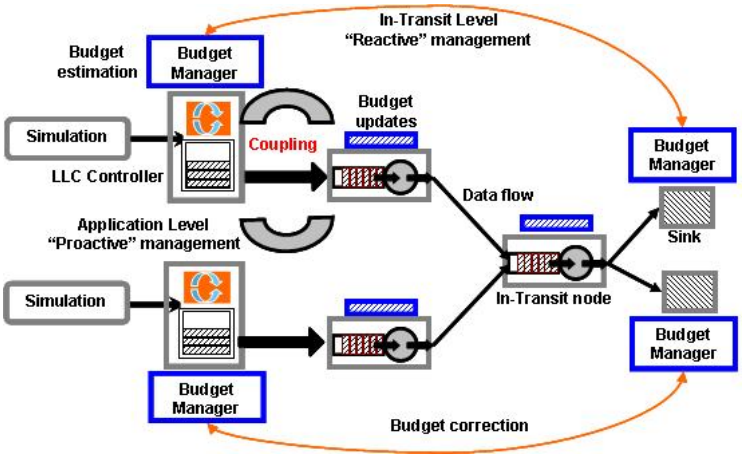

Fig. 5. Conceptual overview of the self-managing data streaming and in-transit processing service

runtime management strategies for in-transit data manipulation and transformation. These two components cooperate to address overall application constraints and QoS requirements.

Application level autonomic data streaming: The application level autonomic data streaming service combines model-based limited look-ahead controllers (LLC) and rule-based autonomic managers, with adaptive multi-threaded buffer management and data transport mechanisms at the application endpoints. The service manager monitors the state of the service and its execution context, collects and reports runtime information, and enforces the adaptation actions determined by its controller. Augmenting the manager with an LLC controller allows human defined adaptation polices, which may be error-prone and incomplete, to be combined with mathematically sound models and optimization techniques for more robust self-management. Specifically, the controller decides when and how to adapt the application behavior, and the service managers focus on enforcing these adaptations in a consistent and efficient manner.

In-transit data processing: The in-transit data manipulation framework consists of a dynamic overlay of in-transit nodes, which is on the path between the source and the destination. The in-transit node services run on commodity clusters (or possibly public clouds) with heterogeneous capabilities, interconnects and loads, and are shared between multiple scientific workflows. They perform simple operations such as processing, buffering, and forwarding. The processing performed by a node on a data item depends on the node's capacity and capability, and the amount of processing outstanding for the data item. The latter information is contained in the data item itself as metadata. Any processing that is outstanding when the data item reaches the sink will have to be performed at the sink. The processing capabilities themselves can be pre-staged at the in-transit nodes or can be dynamically deployed. Our initial explorations have assumed the former.

Cooperative self-management: Coupling application level and in-transit management: The application level and in-transit management can be coupled to achieve cooperative end-to-end self-management. Such a coupling has many benefits, especially in cases of congestion, which often occurs at a shared links in the data path between the sources and sink nodes. Without such a coupling, the application level controller would detect congestion by observing a decrease of the effective bandwidth, and in response, it would advise the service manager to reduce the amount of data sent on the network and increase the amount of data written to the local storage, to avoid data loss. While this would eventually reduce the congestion in the data path, it would require that the data blocks written to the local storage be separately transferred to and processed at the sink. By contrast, using coupling between the application and in-transit management levels, the in-transit node signals the application level controller at the source in response to the local congestion that it has detected by observing its buffer occupancy, and sends the information about its current buffer occupancy.

This allows the application level controller to detect congestion much earlier, rather than having to wait until the congestion propagates back to the source, and in response, it can increase its buffer size and buffer data items until congestion at the in-transit nodes is relieved. This, in turn, reduces the amount of data that has to be written to the local disk at the source and improves QoS at the sink.

Summary of results: An experimental evaluation demonstrating the feasibility and benefits of the concepts and the framework described above, as well as investigating issues such as robustness, stability and overheads are presented in [5,7,6]. These experiments are conducted using a data streaming service that is part for the CPES FSP workflow and streams data between applications running between Rutgers and PPPL in NJ, ORNL in TN, and NERSC in CA. The data transport service managed the transfer of blocks of data from applications buffers at NERSC to PPPL/ORNL or to local storage. Service adaptations included the creation of new instances of the streaming service when the network throughput dipped below a certain threshold. Furthermore, during network congestion, adaptations included changing the buffer management scheme used to ensure better network throughput and lower average buffer occupancy. In cases of extreme congestion and buffer overflows, the data was moved to local storage rather than over the network to prevent further congestions and buffer overflows and thus maximizes the amount of data reaching the sink. These adaptation resulted in an average buffer occupancy of around 25% when using a combination of model and rule based self-management. This leads to lower application overheads and this avoids writing data to shared storage. The percentage overhead due to the service at the application level was less than 5% of the computational time. We also performed experiments with in-transit data processing, which demonstrated processing at the in-transit nodes reduced buffering time from 40% to 2% (per data item) during network congestion. Using cooperative management, the & buffer occupancy further reduced by 20% and 25% fewer data items had to be diverted to local storage at end-points in spite of network congestions and high loads. Furthermore, the "quality" of data reaching "in-time" at the sink increased, effectively reducing the processing time at the sink by an average of 6 minutes per run.

This application demonstrates the use of the conceptual architecture in Figure 2, where the application can interact with the `Autonomic Tuning` engine to undertake model correction, modify solver behaviour or iteration count, for instance. In this architecture, the tuning engine also utilizes performance data to undertake buffer management and data routing.

4 Discussion and Analysis

In Sections 3.1 and 3.2, we identify how two scientific applications make use of the conceptual architectures illustrated in figures 1 and 2. In the first application scenario, a tuning engine is responsible for interacting with a resource manager

Table 4. Coupled fusion simulation application management using autonomic strategies

Application Objective	**Autonomic Strategy**
Maintain latency-sensitive data delivery	**Resource Management** adaptive data buffering (time, size), adaptive buffering strategy, adaptive data transmission & destination selection
Maximize data quality	**Resource Management** opportunistic in-transit processing adaptive in-transit buffering
Scientific Fidelity	**Algorithmic Adaptivity** in-time data coupling model correction using dynamic data solver adaptations -

to support resource selection, whereas in the second, the tuning engine can modify the application behaviour directly based on the observed outcome. In both instances, the tuning engine is external to the application and resource management components, thereby making it more general purpose and re-usable. In the first application scenario, we make use of a centralized tuning engine, while in the second application scenario, we have multiple coordinated tuning engines.

It is also important to emphasize that as an application can have multiple objectives, an application can operate in multiple *usage* modes. Hence, the examples of application scenario mappings to particular conceptual architectures presented above were selected to provide illustrative examples of *particular* usage. For example, an alternative version of the EnKF application (presented in Section 3.1) could utilize the conceptual architecture in figure 2. This involves combining individual jobs into *job aggregates* (batches) based on their particular properties. Hence, if the same application was to be deployed over a combination of the TeraGrid and commercial Cloud computing infrastructure, such as Amazon EC2 and S3 for instance, it would be possible to match jobs to resource characteristics. In this approach, computation intensive jobs could be mapped to the TeraGrid, whilst those that require a quick turn around may be mapped to EC2. As discussed in Ref. [15], in this scenario the autonomic tuning engine is responsible for modifying the behaviour of the application to aggregate computations based on an estimate of which resource type is: (i) currently available; (ii) most likely to respond within some time threshold; and/or (iii) within cost/allocation budgets. Such characterisation may also be used for capacity planning, for instance to determine the number of virtual machine/EC2 instances required for a particular group of jobs. This would be supported through an abstraction such as customized pilot-jobs, allowing a sufficiently large number of resources to be instantly available for executing jobs in the future, thereby minimising data transfer and queuing overheads. Utilizing job and resource properties in

this way would also lead to a complementary self-organizing architecture – i.e. one that is able to adapt application properties and job scheduling based on the characteristics of resources on offer. The two proposed architectures could also be integrated, so that the tuning *by* and *of* approaches could be applied during different periods of execution for the same application. Integrating these architectures also provided mulitple redundant pathways for adaptation. For example, in case of the coupled fusion simulations, tunning *by* the application is used to adapt buffer management strategies or in-transit processing based on data productions rates and tunning *of* the application is used to adjust data production rates to react to congestion or in-network loads. Also, increased buffer occupances due to link congestion can be tolerated by adapting in-transit paths, buffering strategies, and/or data production rates.

As illustrated in figure 3, an application objective is *organized using* an application tuning strategy. However, it is important to emphasize that when supporting autonomic tuning, it is necessary to chose tuning strategies that are able to satisfy multiple application objectives concurrently. For instance, in the case of the EnKF application, it is necessary to support *both* load balancing and scientific fidelity, within some pre-defined bounds. In future work we will discuss how the same application instance can use multiple pathways towards achieving multiple objectives.

5 Conclusion

The need for self-adaptation within computational science applications is outlined, along with two conceptual architectures that can be used to support such adaptation. The approach utilizes ideas from control theory and reflective middleware, although it differs from these approaches by considering practical concerns in realistic application deployments – where the application tuning mechanism needs to be separated from the application itself. Such an architectural approach also renders it more general purpose, and therefore re-usable. Two specific real world applications are used to demonstrate the use of the two conceptual architectures, outlining the basis for a framework for autonomic computational science – consisting of mechanisms, strategies and objectives. It is also important to emphasize that reflective middleware as well as the control models may be used as specific mechanisms to achieve autonomic behaviors.

In section 2.5 we identify how our approach aligns with that of Nierstrasz et al. [22], primarily considering a model-centric view that takes account of deployment and execution. The primary reason is the particular focus we adopt in section 2.2, which involves managing the execution of such an application on computational resources. However, the general autonomic computing concepts identified in section 2.1 are much broader in scope and a model-driven approach could also considered more generally. We believe such an approach would be useful to better understand how an application could be re-formulated, for instance, based on different scientific objectives (e.g. time to solution, error tolerance, accuracy etc); such criteria being mostly application domain specific.

Our motivation has come from existing work in executing computational science applications over distributed infrastructure. Using autonomic computing approaches to improve this execution has been the primary focus of the conceptual architectures outlined in section 2.2 and the conceptual framework in section 2.3. Developing an adaptive design methodology that extends these would be the next step in our work. Such a methodology would provide a set of stages that a designer of an application would need to follow to map application-level objectives to autonomic strategies and mechanisms. Furthermore, the conceptual architecture presented in figures 2 and 1 represents a centralized autonomic tuner – which may be co-located with the application. However, an implementation of such an architecture may have a tuning process that is distributed – for instance, each resource may itself use a tuning strategy for improving queuing times for particular types of jobs.

Acknowledgment

This paper is the outcome of the e-Science Institute sponsored Research Theme on Distributed Programming Abstractions. We would like to thank Murray Cole, Daniel Katz and Jon Weissman for being partners in the DPA expedition and having contributed immensely to our insight and understanding of distributed applications and systems, which form the basis of their application specifically to Autonomic Computational Science. We would also like to thank Hyunjoo Kim (Rutgers University), Viraj Bhat (Yahoo! Research) and Yaakoub El Khamra (TACS, Univ. of Texas Austin) for the two applications described in this paper.

References

1. Al-Ali, R.J., Amin, K., von Laszewski, G., Rana, O.F., Walker, D.W., Hategan, M., Zaluzec, N.J.: Analysis and Provision of QoS for Distributed Grid Applications. Journal of Grid Computing 2(2), 163–182 (2004)
2. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 27–47. Springer, Heidelberg (2009)
3. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Reflecting on self-adaptive software systems. In: Proceedings of Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Vancouver, BC, Canada. IEEE, Los Alamitos (2009)
4. Batch Queue Predictor, `http://nws.cs.ucsb.edu/ewiki/nws.php?id=Batch+Queue+Prediction` (last accessed: May 2010)
5. Bhat, V., Parashar, M., Khandekar, M., Kandasamy, N., Klasky, S.: A Self-Managing Wide-Area Data Streaming Service using Model-based Online Control. In: 7th IEEE International Conference on Grid Computing (Grid 2006), Barcelona, Spain, pp. 176–183. IEEE Computer Society, Los Alamitos (2006)

6. Bhat, V., Parashar, M., Klasky, S.: Experiments with In-Transit Processing for Data Intensive Grid workflows. In: 8th IEEE International Conference on Grid Computing (Grid 2007), Austin, TX, USA, pp. 193–200. IEEE Computer Society, Los Alamitos (2007)
7. Bhat, V., Parashar, M., Liu, H., Khandekar, M., Kandasamy, N., Abdelwahed, S.: Enabling Self-Managing Applications using Model-based Online Control Strategies. In: 3rd IEEE International Conference on Autonomic Computing, Dublin, Ireland, pp. 15–24 (2006)
8. Brevik, J., Nurmi, D., Wolski, R.: Predicting bounds on queuing delay for batch-scheduled parallel machines. In: Proc. ACM Principles and Practices of Parallel Programming (PPoPP), New York, NY (March 2006)
9. Chandra, S., Parashar, M.: Addressing Spatiotemporal and Computational Heterogeneity in Structured Adaptive Mesh Refinement. Journal of Computing and Visualization in Science 9(3), 145–163 (2006)
10. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
11. Dobson, S., Denazis, S.G., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM TAAS 1(2), 223–259 (2006)
12. El-Khamra, Y., Jha, S.: Developing autonomic distributed scientific applications: A case study from history matching using ensemble kalman-filters. In: GMAC 2009: Proceedings of the 6th International Conference on Grids Meets Autonomic Computing. ACM Press, New York (2009)
13. El-Khamra, Y., Jha, S.: Developing autonomic distributed scientific applications: a case study from history matching using ensemblekalman-filters. In: Proceedings of the 6th International Conference on Autonomic Computing (ICAC); Industry session on Grids meets Autonomic Computing, pp. 19–28. ACM, New York (2009)
14. Evensen, G.: Data Assimilation: The Ensemble Kalman Filter. Springer, New York (2006)
15. Kim, S.J.H., Khamra, Y., Parashar, M.: Autonomic approach to integrated hpc grid and cloud usage. Accepted for IEEE Conference on eScience 2009, Oxford (2009)
16. Hariri, S., Khargharia, B., Chen, H., Yang, J., Zhang, Y., Parashar, M., Liu, H.: The autonomic computing paradigm. Cluster Computing 9(1), 5–17 (2006)
17. Jha, S., Cole, M., Katz, D., Parashar, M., Rana, O., Weissman, J.: Abstractions for large-scale distributed applications and systems. ACM Computing Surveys (2009) (under review)
18. Jha, S., Parashar, M., Rana, O.: Investigating autonomic behaviours in grid-based computational science applications. In: GMAC 2009: Proceedings of the 6th International Conference on Grids Meets Autonomic Computing, pp. 29–38. ACM Press, New York (2009)
19. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
20. Klasky, S., Beck, M., Bhat, V., Feibush, E., Ludäscher, B., Parashar, M., Shoshani, A., Silver, D., Vouk, M.: Data management on the fusion computational pipeline. Journal of Physics: Conference Series 16, 510–520 (2005)
21. Kon, F., Costa, F., Campbell, R., Blair, G.: A Case for Reflective Middleware. Communications of the ACM 45(6), 33–38 (2002)

22. Nierstrasz, O., Denker, M., Renggli, L.: Model-centric, context-aware software adaptation. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 128–145. Springer, Heidelberg (2009)
23. Parashar, M.: Autonomic grid computing. In: Parashar, M., Hariri, S. (eds.) Autonomic Computing – Concepts, Requirements, Infrastructures. CRC Press, Boca Raton (2006)
24. Serugendo, G.D.M., Foukia, N., Hassas, S., Karageorgos, A., Mostefaoui, S.K., Rana, O.F., Ulieru, M., Valckenaers, P., Aart, C.: Self-organising applications: Paradigms and applications. In: Di Marzo Serugendo, G., Karageorgos, A., Rana, O.F., Zambonelli, F. (eds.) ESOA 2003. LNCS (LNAI), vol. 2977, Springer, Heidelberg (2004)
25. Sevcik, K.: Model reference adaptive control (mrac), `http://www.pages.drexel.edu/~kws23/tutorials/MRAC/MRAC.html` (last accessed: August 12, 2009)
26. Söderström, S.: Discrete-Time Stochastic Systems - Estimation and Control, 2nd edn. Springer, London (2002)