QoS-Aware Component Architecture Support for Grid

Sharath Babu Musunoori, Frank Eliassen, Richard Staehli
Simula Research Laboratory
P. O. Box 134, 1325 Lysaker, Norway
E-mail: {sharath, frank, richard}@simula.no

Abstract

Grid is a research area, where many research communities have been working and significant results are produced in design and implementation of different computational, data-intensive, and service-oriented grid systems. However, most promising issues such as component-based application design and QoS management have received little attention. Specially in case of real-time grid QoS-sensitive applications, it is difficult to specify and satisfy QoS requirements. In this position paper, we propose QoS-aware component based architecture for building grid applications. Our QoS-aware component based approach will simplify and improve the flexibility in grid application design and development, while providing a higher level of abstraction.

1. Introduction

The importance of component technology is well understood [19], since it provides greater flexibility and simplicity at all levels of software development process. Such a component based approach is the basis for reusable software and plug-in mechanisms. For example, some popular component based architectures EJB, COM+ and CCM have been enjoying the features of componentization. However these current component based architectures have well known limitations for real-time and multimedia applications [1]. In addition, these component-based architectures do not serve grid computing environment, since their platform services are closed and inflexible (tightly coupled), and tolerance for imprecision is not balanced against real-time performance. In more simple words, in grid infrastructure (environment), it is usual that resources including computational and communicational resources may join and leave very often. Therefore it is necessary to avoid dependencies among different entities while making the overall application capable of handling such a dynamic nature of grid environment (infrastructure) efficiently.

The major objective of grid middleware is to provide a unified programming model for grid application development by solving largely heterogeneous distributed problems [20] [17] [16] [15] [7]. (more to add to point out the importance of QoS, component technology ....)

A widely adopted Open Grid Services Architecture (OGSA) provides a web-service-based development environment for developing grid applications. Although OGSA relies on Internet standards while making the platform services open for loose coupling, doesn’t support any QoS issues such as QoS specification, management, and is limited [21] in performance optimization. Our QuA architecture presented in this paper provides QoS support along with intelligent service planning, to make real-time grid application development process simple and flexible. We believe that the platform-managed QoS is the only solution to provide QoS management for component-based applications, independent of functional aspect of grid application development. The following are three initial factors of the QuA architecture considered to be important for grid application development process:

- **QoS-sensitive application**: is an application that will commonly perform unacceptably if platform resources are scare or if the deployment is not carefully configured and tuned for the anticipated load. A QoS sensitive application can provide greater utility by sacrificing quality in one dimension for better quality in another.

- **Safe deployment property**: of a component architecture guarantees that any component can be reused requiring its declared functionality, components deployed on any sufficiently provisioned implementation of the component architecture platform will function correctly.

- **Separation of concerns**: Separation of quality concerns from application development. That is,
QoS concerns of a grid application is separated from the functional aspect of grid application programming model.

The rest of this paper gives an outline on different characteristics of a component based architecture and its architectural support for grid middleware along with some essential component based grid middleware issues. Section 2 characterizes QoS dimensions of grid middleware, section 3 outlines the QuA core architecture. Some architectural requirements to support grid middleware are discussed in section 4. Section 5 address related work on QoS management and architectural frameworks with respect to grid. Finally, our conclusions on current QuA work and future work on investigation and validation of QuA architecture for grid, is discussed in section 6.

2. QoS dimensions in grid

QoS is one of the central concerns of grids, and a subset of QoS properties have to be achieved for any grid application based on its context. In more general words, QoS dimensions in grid architecture are the network and computational requirements essential for the acceptance of application execution, performance. Several categories of QoS properties including their parameter are presented in [9] [14]. One or more QoS parameters represent a QoS dimension of application performance guarantee. The resource pool of a grid computing environment changes along with the time. In addition, non-uniform nature of resources performance makes it clear that application should include knowledge to understand the network behaviour, resource requirements in order to configure a grid application service based on available resources while satisfying the QoS specification.

In the QuA architecture, our initial QoS-aware component architecture considers two abstract QoS dimensions: timeliness and accuracy. The timeliness dimension referred to a break through time (maximum acceptable time) in which a task is completed or an event occurs. The accuracy of a QoS-sensitive application can be defined by an error between estimated and actual service values. For example, the performance of a scientific computational grid application is acceptable if it executes a service in some milliseconds of time with an acceptable error rate. From computational and performance point of view, the two QoS dimensions are well enough to demonstrate and are applicable to all applications. In addition we may add new dimensions based on the application domain or context. For example, the additional quality dimension temporal resolution, adds extra value to the real-time content based video analysis applications [5].

2.1. QoS trading

In general distributed computational environment, applications might need more time to execute a service or task with an acceptable accuracy, sometimes it may require unacceptable period of time to achieve an acceptable accuracy. Therefore its important to have a trading technique among different QoS dimensions while considering resource availability, to achieve an optimal application performance. In this respect, our reasoning is to support by the QoS management, as a set of functions that operate on domain dependent QoS characteristics to achieve a specified QoS level. Here functions typically include QoS monitoring, QoS maintenance, QoS degradation, QoS signalling and QoS scalability. Different QoS dimensions must be balanced or prioritized to achieve desired application performance. Such process is called QoS trading. In other words, the performance of a QoS-sensitive grid application is characterized by these QoS dimensions.

3. QuA architecture for grid

The success of various existing component models have proved the importance of the component architectures as they simplify the complex distributed application development process. Some well known characteristics of component technologies are reusability, encapsulation, and an existing component can be replaced with an upgraded one without effecting the application model. The component technology and QoS service management are the key concerns of a grid service architecture [20] [18] [7].

The QuA architecture subsumes required level of all essential component properties. An additional important factor we consider is the safe deployment property of QoS-sensitive application. That is, a component reused for its functionality in any application, will function correctly provided that the entire component life cycle is supported by a component platform. In grid context, we reuse the term QoS-sensitive application as grid QoS-sensitive application to refer to a grid application that performs unacceptably when platform resources are scarce or when the deployment is not carefully configured or tuned (scheduled) according to the anticipated load. Some initial research work addressing how can the QuA architecture satisfy the safe-deployment property of the QoS-sensitive application when resources are sufficient, is presented in [1] and
is supported by a platform-based QoS management approach. From this, applications and components are written independent of physical resources and platform service implementations. The key to QuA platform managed QoS is the ability to compose and create services by specifying their logical types and QoS requirements without specifying how these properties are achieved. Therefore application level QoS requirements like accuracy and timeliness refer to the logical properties of the component interfaces.

The QuA architecture is supported by a open reflective distributed QuA platform. The QuA platform is a logical object (QuA component) implemented by a set of distributed capsules. A capsule is a QuA object representing a local QuA runtime platform services such as service instantiation and binding, execution of local service objects, and communication support between QuA objects. An open reflective model [a right reference to ORM???] used in QuA platform implementation, allows all the platform and its services be to configured to suit application and infrastructure requirements. The QuA platform is represented by a singleton object and, it can be seen as a distributed virtual machine supporting service execution and communication between QuA components. Applications built on QuA platform, are composed of components or instances manufactured from component templates. In general, an X software component is an immutable binary value associated with a platform of type X, acts as a component template and guarantees the behaviour of the objects manufactured from it when participating in a composition with other components within the platform type X. From [1], we believe that the above definition is stronger and more clear than szyperski’s often cited definition [19].

To build grid QoS-sensitive applications, an application developer is allowed to interact with the QuA platform. An initial application object provides a service request as input to the QuA local platform or capsule on which the application has been instantiated. A special service context for each service request, is maintained by the QuA platform. The service context maintains meta information about the service. A service request may include a service specification, quality specification, and input guarantee. The service specification is a composition description which may include new or existing QuA objects, references between them, and exported interfaces. Quality specification includes QoS description that the service needs to be satisfied. This QoS description might be in some form of quality constraints or utility functions. Finally the QuA platform (schedule or) implements and execute the application while satisfying the QoS requirements specified within quality specification. Its also the QuA platform’s responsibility to monitor and reconfigure the application according to the runtime QoS constraints. Such a responsible task will be done by a service planner, a QuA component employed by the QuA platform. The QuA service planner [1] is responsible for discovering resources and implementation alternatives, and for planning the optimal configuration of two to satisfy functional and QoS requirements. This complex task has no implementation that is both general and optimal. Current version of QuA platform implemented in both Java and Smalltalk has a default basic service planner, which comes with every QuA capsule. The basic service planner has no knowledge of QoS requirements and trading, instead it can only provide basic services like component type resolution, instantiation and binding using service of the local repository and capsule.

3.1. QuA service planning

Service planner is the key component of the QuA architecture, it can be replaced with a more specialized service planner as a plug-in for building more efficient QoS sensitive applications. The QuA Service planner is responsible for service creation while satisfying the QoS requirements specified in a service context. In simple words, QuA service planner reads the service request specifying what is required without saying how to implement, and returns a service capability in the form of a task graph or a service plan. QuA Service planning is similar to the scheduling and application tuning (configuration) process defined as a significant research challenges in the distributed grid computing environment [20]. Several details like application characteristics, resource availability and size of input and output data, available network bandwidth and latency, careful selection of type, and number of processors required, are necessary for application scheduling, tuning according to available resources. Such a scheduling process is one of the objectives of QuA service planning. A QuA-based grid application is an instance of a QuA service plan. Therefore resultant service plan acts as a blueprint of a grid service providing a best performance. However to determine such a service plan in the form of a blueprint for a defined service, is a complex task. In order to simplify it, we divide the activities involved in hierarchical and recursive QuA service planning process into four different levels: resource, QoS, error, and service levels. These levels in service planning work space are shown as a black diagram in the fig. 3.1.
Figure 1. A black diagram of service planning activity.

A service composition at service level, is described as a component graph. In the graph an atomic service is equivalent to a component and a sub-service to a sub-composition of components. A sub-composition may either be a grouping of components, where all components are accessible from outside the sub-composition, or a component that encapsulates the sub-composition. The foremost is how sub-compositions are represented in existing component technologies, and the latter one can find in research component models and middleware [22]. Alternatives in the component graph are compared via the utility functions.

The error level contains error models, that gives expected quality reduction (error) for each component type in the component graph. Error model used in the architecture is based on existing QoS semantics work [2] and an error model for multimedia streaming applications [23].

The QoS level defines QoS dimensions relevant for all component types in the component graph. Values per dimension is governed by resource availability for one or more resource types. The QoS dimensions is integrated into the error models.

The resource level defines computer and network resource types. Availability of resources gives the minimum and maximum values for each QoS dimension. Resource availability is input data to the error model.

In applications like real-time multimedia with QoS as objective, maintaining different acceptable levels of service is necessary to produce good service. This leads to other related issues like QoS trading and resource trading [10]. The complexity of the formulated problem on multiple QoS dimensions and multiple resources with respect to a single application [9], is applicable at resource, QoS level, and partially at error level of service planning (fig.3.1).

A service composition at service level contains one or more atomic entities (i.e. components) or even compound entities (sub-composition of components), creating a component graph. As stated in [1], a service planner assess the different alternatives in the component graph and select the appropriate service composition alternatives that satisfy the QoS requirements, while keeping a fair allocation of resources. A service graph has \( i \) levels, where the lowest level is atomic service level. At each given level the number of alternatives, \( N \) is recursive, except for the lowest level, where number of alternatives equal to the number of selection choices \( m \). For each parent node, the number of alternatives is given by following equation.

\[
N(S_i) = \begin{cases} 
\prod_{j=1}^{i} N(S_{ij}) & \text{if } i \text{ is leaf level} \\
 m & \text{if } i \text{ is leaf level} 
\end{cases}
\]

To handle this complexity, at run-time, a hierarchical approach is considered suitable, since it divides the complexity into sub tasks. ( where \( n \) is number of sub-nodes of a service at level \( i \) )

From the above general service planning activities, it is important to analyze and generalize both utility function and error model for different grid application contexts. Apart from the semantics or parameters used for determining an optimal service plan, the error model [1] may have one or more error functions for determining, allocating the error limits to each service and its sub services. These functions are defined as error budget and error allocator functions. The other important driving factor of error model is error prediction function. The error prediction function estimates the error for each service output value for the sub services involved in the service specification hierarchy. The error model computes the error for each service plan based on an error estimation criteria like service lifetime which satisfies the QoS: timeliness property of QuA. The utility function is used to compare the estimated quality of each acceptable alternative implementation to obtain best application performance. Therefore, the QuA service planner must be able to estimate the service capability or utility value of service implementation and this process should be done on partial information because of the propagation delay in distributed environment.

Such a service capability estimation for each implementation alternative can be done in different ways and initially are categorized into non-predictive and predictive approaches [24]. Non-predictive approach uses
heuristics or probability distribution models based on current information, whereas predictive approach uses heuristics or machine learning (or even pricing model in large scale distributed grid systems) based on current and past information.

The above process is a part of QuA service planning towards building performance-efficient applications. It is expected to include more issues such as monitoring of application performance and output quality, adaptation to the changes in resource availability, reconfiguration support for atomic transition of application execution according to the QoS management policy. Whenever the QuA monitoring service identifies any decrease in application performance, QuA service planner calls for reconfiguration procedure similar to the service planning described above.

4. Requirements

In addition to the current QuA prototype, QuA architecture has to be extended for more additional characteristics and properties. In this section we present fundamental requirements to realize an efficient component based grid architecture. However our intention of stating the following requirements is not to research each of them, but only to validate the QuA architecture in grid environment. (i.e. other generic problems of grid environment such as scalability are left for open research area).

QoS trading: When multiple quality dimensions are brought into the equation to satisfy the service performance needs, trading mechanisms between the quality dimensions becomes necessary to meet performance objectives of a grid application service.

Service discovery: Discovery of services within QuA distributed platform might become complex. An efficient service discovery technique is necessary to handle large number of distributed service types or instances. That is, an active registry service is required to index and maintain a logical level service information along with their references.

QoS mapping: The logical level QoS requirements specified within the service request at application level must be transformed into the system level QoS parameters. That is performance requirements given by the user into system level performance requirements.

Service composition techniques: In component based programming models, each component is viewed as a service (provides a one or more services). When one or more components are composed in a hierarchical format, the efficient composition technique is necessary to make any service composition work as a single component service.

Grid programming complexity: Programming a grid application might need to consider many issues such as a resource model for resource and QoS management, inter platform concerns, and other communication and performance problems. In order to make them simple, a special component based architecture is required to provide a higher level abstraction from both low-level system and application details. This will reduce the complexity of developing grid applications.

Heterogeneity: Multiple computational resources in grid might be connected by heterogeneous entities of grid network infrastructure. In order to integrate such different heterogeneous grid entities, a platform based component grid architecture is the promising approach to provide homogeneous access. In QuA, a distributed QuA platform acts as common platform to solve the problem of heterogeneity.

Reconfiguration: In case of real time application running in a grid environment, it is essential to make sure that a reconfiguration is not the same as terminating a service and instantiating a new one according to the available resources. Instead, there should be an atomic transition from old service contract to the new one.

Fault tolerance: In generic software systems, it is observed that the entire system crashes when any small system entity fails. In some sense, this complexity is also applicable to grid computing environment. That is, a noted failure of a sub service may lead to performance degradation of the entire application [fault tolerance].

Scheduling algorithms: Although all other requirements are necessary to shape a component-based grid architecture, application scheduling is very important deciding factor of the whole application performance, and it is quite difficult to generalize and adopt one specific scheduling algorithm for multiple realtime grid application contexts. In this regard QuA grid service planner must be able to adopt multiple techniques to deliver best (optimal) application performance since changes in resource availability and application demands need multiple component (service) support as an active plug-in rather than programming. For example, QuA service planning may need multiple scheduling algorithms.

5. Related work

In this section, we provide a brief overview of work related to the component based grid architectures, and respective grid middleware approaches like scheduling and resource management. The most prominent existing component architectures we have evaluated as a part of our QuA architecture development are EJB, CCA, .NET and CCM. Apart from the ca-
pability and scope of these component models to support grid concept, Common Component Architecture (CCA) [TowardsCCA] is the first component based architecture designed for high-performance scientific computing, and later evolved into XCAT [XCAT2.0] a component-based programming model for building grid applications. XCAT includes an integration framework to coordinate with other widely accepted OGSI-based web service components. Current version of XCAT only describe component-level support for grids but not any of the challenges discussed in the grid area[Blueprint][DCRICG]]. A five layered grid architecture presented in [Anatomy] include several different protocols representing different perspectives of the grid. That is a set of resource and collective protocols defined at the neck of hour glass grid architecture opens up for a broad way to build variety of grid applications. In addition, WSDL-based grid service creation and management as OGSA service models are presented in [Physiology][GSDSI]. Grid services and their access bindings are defined in a sub services description language(WSDL), a language, platform independent standard. A recursive service implementation approach used for lower-level service subservices of a service composition is similar to our service planning hierarchy in QuA service planning(section 3.1). Beside some QoS properties like security and reliability, OGSA do not address QoS specification and management support for service implementation. Especially in case of QoS-sensitive applications like video conferencing applications, trading mechanisms between the QOS dimensions(for example timeliness, sampling rate), is essential to guarantee the specified level of quality of a delivered service.

In support to the grid application development, a fault tolerance model including classification and analysis of faults, is presented in [FaultToleranceInGrid].

||SOA||
//ChenLee - how resource allocation is optimized
//Taxonomy. Blueprint- how different scheduling algorithms are used for application tuning
//Janathan Walpole -
//Janyvind Aagedal -
//QoS adaptation in real-time systems -
//Method for service composition and analysis -

6. Summary and future work

Our future work include resource discovery, service planning (scheduling), reconfiguration of grid services at different level of service hierarchy,

References


