

Exploring the Design Space for CCA Framework Interoperability Approaches

Michael J. Lewis, Madhusudhan Govindaraju, and Kenneth Chiu
Department of Computer Science, State University of New York (SUNY) at Binghamton
{mlewis, mgovinda, kchiu}@cs.binghamton.edu

Abstract

One important challenge to building and deploying high performance scientific applications in grid environments is providing a software development model that abstracts the complexity of the environment and simplifies the programmer's task, allowing her to focus on the details of her particular application. Component frameworks, including those that support the Common Component Architecture (CCA), represent a promising approach to addressing this challenge, one that is being realized, for example, in our LegionCCA and XCAT-C++ frameworks. The next step beyond building independent individual frameworks is making them interoperate. Component-based applications should be able to transparently span multiple disjoint component frameworks with low overhead as compared to the same applications running within a single framework. Interoperable frameworks enable applications to take advantage of more resources, and to better match constituent parts to the underlying resources that best support them. This paper identifies five underlying component framework interoperability requirements, and three general approaches to addressing them. We then discuss how the approaches can be applied to meet the requirements, and address the advantages, issues, and implications of doing so. This effectively defines a design space for framework interoperability approaches.¹

Key Words: *Components, grid computing, interoperability, Web services, SOAP*

1 Introduction

A distributed grid-based component framework provides interfaces and mechanisms for composing grid applications from component parts, allowing programmers to build some components, and to select and add other components into the same application. Component frameworks are important tools for harnessing the potential of grid computing environments because they abstract some of the complexity of the grid software development process, and allow programmers to take advantage of extant software that can help solve many of the same challenges that they face for their application.

The Common Component Architecture (CCA) [1] is one component model that is specifically designed for high-performance scientific applications. Programmers specify their components using the Scientific Interface Definition Language (SIDL) [3], build the back end implementations that realize the interfaces they specify, and compile them into reusable CCA components. These components are registered with running frameworks that then make them available to applications, and define how they can be composed. The CCA specification describes how these components should be named (using component identifiers), how their functionality should be described (through *uses* and *provides* ports), and how they can be composed using specific creation and connection interfaces.

Different CCA framework implementations harness the computational potential of different environments. Sequential frameworks allow CCA components to run within a single address space, parallel frameworks compose them into applications that can take advantage of parallel machines, and distributed frameworks allow the components to run in separate address spaces that span multiple machines. Clearly, each of these environments has different advantages and disadvantages, due to the nature of their computation/communication tradeoff, and to the specific constituent parts they contain. For example, one environment might contain access to a special purpose scientific instrument, such as an X-ray diffractometer used for crystallography, that is only available through its framework. Furthermore, some components are designed to run better in some environments than others, for example because of their computational granularity.

For these reasons, it is important for multiple component frameworks to coexist and interoperate, so that applications can benefit from the advantages of multiple different computational environments through the component frameworks that represent them.

¹This research is supported by NSF Career Award ACI-0133838, NSF Award ANI-0330568, NSF Award DBI-0446298, and DOE Grant DE-FG02-02ER25526.

This paper describes issues related to component framework interoperability, focusing specifically on grid environments and the CCA component model. The ultimate interoperability goal is for component-based applications to transparently span multiple disjoint component frameworks, with overhead no greater than the same applications running within a single framework.

Frameworks must meet several *underlying interoperability requirements*, including (1) description and specification, (2) communication, (3) naming interoperability (for both components and interfaces/ports), (4) discovery, and (5) creation. We develop these requirements in more detail in Section 4, focusing on how they map and translate into the CCA specification.

A variety of approaches can help satisfy these above requirements. Ultimately, they can be classified into the following fundamental *approaches for interoperability*:

- *standards* specify interfaces and protocols that all implementations must obey,
- *adaptors* translate and map constituent parts of one framework into those of another, and
- *proxies* help span frameworks by providing representatives of one framework in another.

It is worth noting that standards are the ultimate interoperability solution, but that they are not always possible or desirable, because they do not support existing systems that were not built to the standards, and because they may preclude some optimized implementations. Various social, political, and economic factors may also prevent a standard from being adopted by all implementations. Although we identify only three basic approaches, where and how they are employed and implemented influences the overall interoperability approach and its characteristics. Each has its own performance and functionality characteristics and tradeoffs. In this paper we describe a well-defined approach and complete design space toward interoperability of distributed grid-based component frameworks. Section 2 describes related work, and Section 3 and 4 expand on the approaches and requirements identified above.

2 Related Work

The CORBA Internet Inter-ORB Protocol (IIOP) specifies how CORBA objects created in different CORBA Object Request Broker (ORB) implementations can interoperate. The early CORBA specifications did not define the wire protocol, leaving it up to the vendors. CORBA 2.0 defined IIOP to provide interoperability between the vendors. The Global Grid Forum (GGF) and other organizations have developed numerous standards for Grid interoperability, including the Open Grid Services Architecture (OGSA) [2]. Web services has also been a focus of much recent activity. Web services is seen as a promising area of distributed systems that many hope will see the same widespread adoption as HTML and HTTP. Web services include a wide range of specifications. Of special significance are: XML as a basic representation for structured data, XML Schema as a type system for XML, SOAP for messaging, WSDL for the description of operations that can be performed between Web service entities, and UDDI for registration and discovery of Web services.

3 Interoperability Approaches

Framework details should not be exposed to programmers, who should not have to concern themselves with which framework houses each component; how to create, instantiate, and deploy components across frameworks; nor the efficiency considerations of application objects' spanning frameworks. Clearly, this goal is not completely realizable in all cases, as it implies transparency, generality, and efficiency, goals which are often mutually conflicting. However, the goal is useful in driving the interoperability design.

3.1 Approaches

The three primary approaches to interoperability are *standards*, *proxies*, and *adaptors*. Specifying standards and requiring all implementations to adhere to them works best in theory. However, several considerations limit the practical effectiveness of standards. First, existing systems and environments must be "retrofit" to meet standards, and this often requires fundamental changes to core aspects of a system. Sometimes this approach is simply not possible (because the standard is so fundamentally different from the system), sometimes it leads to increasingly complex software, and sometimes it results in unacceptable performance. Often, the best approach to meeting standards is complete reimplementations. Second, the standardization process itself, with input from multiple factions with many and varied requirements and intended uses, often takes more time than even the implementation of the standards. By the time standards are complete, they may be obsolete and require change. To alleviate this problem, some things can be left unspecified—but that undermines interoperability, which is the presumed intent for standardization in the first place.

The problems with standards make other complementary solutions necessary. In particular, we describe the related (but slightly different) approaches of adaptors and proxies. The purpose of an adaptor is to convert the representation of some entity (e.g., a message, a description, or the name of a component or function) in one system into a corresponding representation that has meaning

in another. This conversion can be done either at the “sender” or “receiver” side. It is often desirable to have all conversions be done within one of the systems, thereby allowing the other to remain unchanged. Proxies serve as representatives of one environment within another. Generally, a proxy must be able to exist, communicate, and interact in both environments. Interoperability and communication is achieved by having the members of one environment interact with the proxy using its language, protocols, data format, etc., and letting the proxy translate the communication into a separate conversation with the appropriate components in the other system. Proxies are attractive because they isolate the change to a subset of objects in the system, rather than requiring that every object be equipped with interoperability functionality.

The difference between proxies and adaptors, at least as we use the terms and apply them to component framework interoperability, is primarily in their scale. Adaptors are generally assumed to run in the address space of other larger objects, and handle small, fast conversions. Proxies run in their own address space and are charged with more difficult mapping or communication problems.

It is our belief that no one approach—standards, proxies, or adaptors—is best in all cases, and that the three can be used in conjunction with one another because they are complementary. We describe how each of the approaches can be applied to the specific requirements for component framework interoperability below.

4 Interoperability Requirements

As mentioned in the introduction, we identify five different requirements for component framework interoperability, which are described separately below.

4.1 Description and Specification

Before building components, programmers must have a way of describing their interfaces and specifying functionality. Specification and description interoperability is achieved when the descriptions of resources and members in one system somehow have meaning in the other. How this is achieved depends on the description usage. For example, Web services require WSDL documents describing resources to be available at runtime to potential clients. Therefore, service callers must be able to utilize WSDL contents to create invocations, or translate the WSDL into a specification that can be used. On the other hand, some interface definition languages are intended to be used only statically. In this case, interoperability requires a language mapping, from the specification to an implementation language that can be used to build the service, object, or component in that system.

CCA includes a specification that dictates the use of, for example, ports, component identifiers, builder services, etc. The implementation of the constituent parts, however, is left to individual framework developers. CCA is a good example, therefore, of a specification that provides a template or basis for interoperability; but two equally valid and compliant CCA framework implementations may not be interoperable. For example, the contents of a component identifier and of the communication protocol used for inter-component calls can be very different for different frameworks. This flexibility enables efficient implementations for different environments (when components reside in the same address space, local function calls can be used, for example), but provides an incomplete interoperability solution.

4.2 Communication

Before two components can communicate, there must be some agreement on the protocols they use. Usually communication is layered, so not only do they need to agree on the lower-level protocols such as TCP/IP, but they also need to agree on the higher-level semantics of the communications. A number of standards are available for RPC-like interactions. Rather than forcing the use of one protocol, it is desirable to have a multiprotocol approach with SOAP as the designated language to serve as the common, base protocol. The Proteus multiprotocol library can be used to interpose between the actual providers of RPC and the framework implementation. Proteus can mediate at a relatively high-level, which allows an existing protocol implementation to be wrapped as a Proteus protocol provider, and added to a framework without changing existing framework code.

4.3 Naming

Applications that run within the same process can use local references or pointers to refer to ports and components within that process. For the distributed case, specialized handles need to be designed that can serve as “global pointers” to ports and components residing in remote address spaces. Often these handles are designed as global references: objects that function as proxies for remote objects. Distributed frameworks use various mechanisms to create a global namespace to allow names and references of the various entities to be understood across processes running in different physical address spaces. As a result, when data types are received

on the wire, the framework is able to unmarshal it to an object representation that is specific to the internal implementation in that process. The design and implementation of a global namespace is dependent on the internal architecture of each framework.

To facilitate interoperability between different frameworks, and thus allow applications to span various frameworks, it is important to provide “name-mapping” for entities in different frameworks. There are three different possible approaches to achieve this goal: (1) Distributed CCA frameworks could use a common format for all names and data types that are sent over the wire for communication with components in remote address spaces. For example, all remote references could be serialized as a WSDL document, and all data types could be serialized in XML, via the SOAP protocol; (2) References in one format could be converted to another using well-known registries, but then we would require some common way to access the registry. Such a scheme is complex, and it merely exchanges the issue of defining a common format for the issue of defining a common reference conversion mechanism; and (3) Proxies/adaptors for all pairs of distributed CCA frameworks could convert data types and names from the format of one framework to the other.

4.4 Discovery

To use and compose components into applications that span frameworks, the existence of the components must first be discovered. The resource discovery problem *within* a single grid-based framework is difficult, and generally includes information dissemination, query processing, and repository maintenance. For full interoperability of discovery mechanisms, the names and properties of all components that exist in one framework must be discoverable from the other. One approach would be to make the dissemination and query forwarding protocols span the two grid frameworks. Scalable resource discovery for wide-area grids, however, is far from a solved problem, so the potential effectiveness of extending them across grids is unclear and currently unrealistic.

We therefore focus on exposing individual information repositories, containing descriptors about components, from one framework to the other. This approach requires that the interface of a component repository in one framework have meaning in the other. Standards can be used to represent the component information the same way in each framework. Examples of standard repositories include UDDI and CORBA’s Interface Repository. Because the most important information about components includes names, of both components and ports, this solution is fully viable only if naming interoperability is also achieved through standards.

4.5 Creation

The CCA specification requires that the Builder Service API should be used to create CCA compliant components. Each framework is unique in the kind of environment it needs to instantiate a component. The Builder Service encapsulates the component instantiation mechanism, thus shielding the component developers from the low-level, implementation-specific details of the instantiation mechanisms. The Builder Service allows creating instances of components from a set of environment name-value pairs. For interoperability, it is essential to agree on a common format for specification of the environment characteristics that can be used by all frameworks.

The Component identifier is specific to the framework in which the component was created. For applications to span frameworks, it is critical to facilitate the creation of components on different frameworks. An example of the standardization approach for interoperability would be the use of Grid Service Handles (GSH) and Grid Service References (GSR) as defined in the OGSA specification [2]. An immutable string could be returned whenever a component is created and serve as the GSH. This GSH can be mapped to a GSR, represented by a WSDL document. The mapping from a GSH to a GSR could be provided by well known CCA Mapping registries.

Another approach is to build adaptors for every pair of distributed CCA frameworks. These adaptors would convert the component identifier formats of the framework where the component was created to the framework that initiated the instantiation call. With the proxies approach, each framework would include proxies for the frameworks in which it is interested in creating components. This would require changes to the design and implementation of the instantiating framework, as opposed to the use of adaptors, where the two frameworks do not need to be modified.

5 Summary and Future Work

This paper defines the structure of a solution space for making distributed component frameworks interoperate. We identify five different interoperability requirements—specification, communication, naming, discovery, and creation interoperability. We describe three well-known techniques—specifying standards and implementing adaptors and proxies—and argue that applying these three approaches in specific ways to meet the five requirements defines a taxonomy of interoperability solution strategies.

References

- [1] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, August 1999.
- [2] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer* 35(6), 2002.
- [3] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing Language Dependencies from a Scientific Software Library. In *Proceedings of 10th SIAM Conference on Parallel Processing, Portsmouth, VA*, March 12-14, 2001.