

Design and Implementation Issues for Distributed CCA Framework Interoperability

Madhusudhan Govindaraju, Michael J. Lewis, Kenneth Chiu
Grid Computing Research Lab,
Department of Computer Science,
State University of New York (SUNY) at Binghamton, NY, 13902
{mgovinda, mlewis, kchiu}@cs.binghamton.edu

Abstract

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. The CCA specification does not prescribe a wire format for inter-component calls in distributed frameworks, thereby promoting considerable flexibility and customization for the framework developer. This approach thus requires an additional specific strategy outside of the CCA to support interoperability between distributed frameworks. Mandating one common wire format, however, risks choosing the wrong format. We discuss in detail 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. We then address the communication interoperability in detail via a single multi-protocol communication library called Proteus, and discuss how we have incorporated it into two distinct distributed framework implementations of the CCA specification: LegionCCA and XCAT-C++.

Key Words: *Components, Common Component Architecture (CCA), interoperability, Distributed Frameworks, Grid Computing. Web services.*

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) [17], 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 con-

tain 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.

At the highest level, CCA leverages the efficacy of component based software development, and applies it to high performance scientific applications. CCA will enable independently developed application components to work together in the same application, thereby promoting code reuse. To meet this goal, what CCA does *not* specify is as important as what it does specify. For example, CCA does not specify any of the following:

Implementation language: Components can be written in any supported implementation language. The Babel language interoperability tool [6, 8, 17] handles cross-language calls within a single address space.

Framework characteristics: CCA can be, and has been, implemented for components that are assumed to share a single address space, for components that will run in a tightly coupled application on a parallel machine, and for components that will run within distributed or even grid-like environments.

Wire format: Components invoke port functions by sending messages over a network; the wire format for the call is left to the framework developer.

The approach of focusing on a small core set of requirements for CCA components increases flexibility and enables high performance implementations. The approach, however, comes at the expense of interoperability. 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 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, which are described in Section 3.

None of these individual components or approaches is new. We extend our previous work in this area [9, 19] in this paper by defining the space of potential component interoperability solutions and describing in detail how the three approaches can be utilized to meet the five fundamental interoperability requirements. To address the interoperability problem in more detail, this paper proceeds in Section 5 to describe in detail one approach to distributed CCA framework *communication* interoperability, as an example of a concrete interoperability approach that goes beyond specifying standards. In particular, we use a single multi-protocol communication library called Proteus [5], and incorporate it into two distinct distributed framework implementations of the CCA specification: LegionCCA [12] and XCAT-C++ [14, 13]. We have incorporated Proteus into both frameworks separately; Proteus can be used as the communication layer for both LegionCCA and XCAT-C++. The next step is to use Proteus for inter-framework communication, thereby enabling CCA applications whose components span LegionCCA and XCAT-C++.

2 Background

This section describes background necessary for understanding the constituent parts of our solutions. In particular, we briefly introduce CCA, Legion, XCAT, and Proteus, and give pointers and references to more information about each of the projects.

2.1 CCA

The Common Component Architecture (CCA) specification is designed to provide a high-performance plug-and-play environment for large scale scientific applications. It facilitates

the import of existing scientific code into a CCA compliant framework by imposing only a small number of requirements on component developers. The CCA model consists of two fundamental entities: *components* and *frameworks*. Components encapsulate pieces of code that implement a certain functionality, and are the units of software that can be composed to build applications. The framework provides the required milieu in which components run and interact with one another.

The component specification of CCA is expressed as a set of interfaces that define the required behavior for component-to-component and component-to-framework interaction. However, the specification does not mandate nor suggest how the framework itself is constructed. This approach explicitly allows research groups to experiment with different ideas and to develop specialized frameworks. For example, it is possible to develop frameworks that are tailored for wide-area networks and also those that are highly specialized for single-process interactions.

The most important CCA concepts are: (1) *ports*, to define interfaces through which components communicate with one another; (2) *services objects*, to manage a component's ports; (3) *component identifiers*, to serve as opaque handles for components in the framework, and (4) *builder service*, to create, destroy, and connect an application's components. A more in depth discussion of these concepts is beyond the scope of this paper, but can be found elsewhere [1, 2, 3].

2.2 Legion

The Legion grid architecture [15, 18] represents each grid component with a Legion object. Each Legion object belongs to a class and each class is itself a Legion object. Much of the Legion object model's power comes from the role of classes, which are user-level objects that are enabled with system-level responsibility. Every Legion object is defined and managed by its class object, which serves as a manager and a policy maker that can create new instances, activate and deactivate them, and provide bindings for client objects. In particular, class objects are required to implement `createInstance()`, `activateInstance()`, and `deactivateInstances()` functions (among others), but how these functions are carried out is left unspecified.

Legion objects are independent of one another, which means that they are disjoint in address-space and communicate with one another via remote method calls using a Legion-specific method invocation protocol. This protocol is based on packaging invocations within a transmissible data structure called a program graph, and delivering the program graph to the individual components of a computation. The current implementation allows the selection of either TCP or UDP sockets as underlying data delivery mechanisms; the Legion 1.8 implementation does not support SOAP.

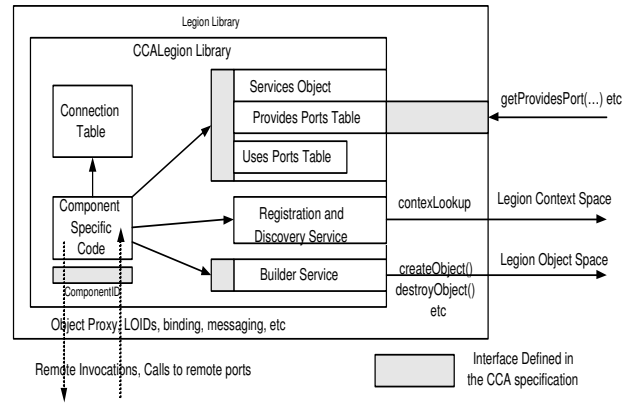


Figure 1. The anatomy of a Legion CCA component: Each CCA component is a Legion object which links both the Legion library and the CCALegion library. The CCA Legion library exposes the standard CCA specification interface to component code, and implements the functionality primarily by utilizing the facilities of the Legion library.

2.2.1 LegionCCA

In general, the LegionCCA approach is to model CCA components as Legion objects. This allows LegionCCA to make use of Legion's existing services and objects, including the following: component creation by class objects and host objects, component discovery through Legion name bindings and context space, and provides port descriptions through Legion object interfaces. Legion objects that serve as CCA components are linked against a *LegionCCA library*, thereby giving them the added functionality to run within the CCA framework atop Legion. This library supports the API that is defined in the CCA Specification, thereby making its services and functionality available to applications programmers from within the components they build. To this end, the LegionCCA library contains implementations of important CCA types, including `ComponentID` and `Ports`, a `Services Object` and a `BuilderService` for the component, and a connection table that describes how the component is attached to others in the framework. The LegionCCA approach is further described in [12]. The anatomy of a Legion CCA component is depicted in Figure 1.

2.3 XCAT

XCAT is a CCA compliant framework, implemented in both C++ and Java, that is based on Web Services standards. Communication between components in XCAT-Java is facilitated by the XSOAP toolkit [16], which is an RMI system that uses SOAP as on-the-wire protocol. XCAT-Java uses

Globus GRAM (via Java CoG) for authenticated job creation in remote resources. XCAT-C++ uses the Proteus library for remote communication.

The XCAT-C++ framework is designed to allow each component to serve both as a CCA compliant component and as a service that is accessible to Web service clients. The modular architecture allows applications programmers to plug in custom implementations that are best suited for their specific application requirements.

2.3.1 Mapping CCA Concepts in XCAT-C++

We list a few important CCA concepts and describe how they are designed and implemented in XCAT-C++.

- **Services Object:** Each CCA component contains a *Services* object that is responsible for managing the component's ports, including the ones that are dynamically added during the execution of the distributed application. Application developers can retrieve handles to a component's ports or just inspect its current state via the standard API of the *Services* object. In XCAT-C++, the *Services* object is designed to encapsulate the framework specific bindings for the *provides* and *uses* ports. Whenever a *uses* port is requested, a pointer to a local object is returned, while for a *provides* port a global (serializable) reference is returned that can be sent to components in remote address spaces. The serializable form of the reference contains information necessary to communicate with the provides port from any component. This information includes details such as host name, port number, communication protocol and a globally unique ID for the provides port.
- **ComponentID:** The CCA specification states that each component should design the `ComponentID` as a opaque handle, but does not require any standard format. The motivation for this approach is to allow each framework to design the handle according to its application requirements. In XCAT-C++, the handle has been designed as an object that is serialized to a string format whenever it is transported to another component. The idea is for the remote handle to be compatible with emerging standards in Grid Web services, which have adopted the Web Services Description Language (WSDL) [7] document to represent distributed services. A WSDL document is an XML document that is commonly stored in a string format. This design also allows an XCAT-C++ *ComponentID* to be used for component assembly via work-flow engines [11].
- **Builder Service:** The CCA *Builder Service* presents a standard API for all components to instantiate, connect and disconnect other components. Once a component

has been instantiated, the service returns a *ComponentID* to the new component. This *ComponentID* can then be used to directly communicate with the component. In XCAT-C++, the builder service instantiates new components from a set of name-value pairs that encapsulate the remote environment details such as command line arguments, executable location, target machine name, and creation protocol. Currently, XCAT-C++ supports the use of SSH and we are testing the incorporation of Globus GRAM [22] for authenticated launch of components on Grid resources.

- **Component Communication:** In distributed scientific computing component systems, components are instantiated on remote machines and wired together dynamically with running components. As a result the choice of protocol depends on dynamically changing factors including the data type and size that needs to be transferred, security policies, and the list of common protocols supported by a pair of interacting components. Also, a component is typically connected to several components at any given time, each connection probably optimized for a different protocol. To address these requirements, XCAT-C++ uses the Proteus multi-protocol library, described in Section 2.4.

2.4 Proteus

Proteus [5] is a multiprotocol library for distributed communications. It mediates between protocol implementations, known as protocol providers, and the application by providing a uniform API to both applications and protocols. Protocol providers can be implemented by wrapping existing protocol implementations with an adaptation layer. This layer converts Proteus invocations into a form suitable for the protocol implementation. An application can then switch between different protocols at run-time through an interceptor-like facility.

Addresses in Proteus are simple aggregations of the addresses for each protocol provider. At invocation time, a user-supplied interceptor object makes the actual selection of which provider to use. The interceptor object may use whatever information it has available, such as recent performance history, or the results of a prior, application-defined negotiation phase.

We describe some important goals of Proteus.

- **Specify Mechanism, not policy:** Proteus serves as a flexible middle-ware layer to enable communication to switch to optimized protocol implementations on a per-call basis. It provides an API to application developers for protocol selection, negotiation and message routing.
- **Modules for High Performance:** To enable scientific applications, Proteus includes features such as *wormhole*

routing and *matter*. In wormhole routing, a message is divided into a sequence of (fixed size) data units called flits. As the header flit moves, the remaining flits follow in a pipeline fashion. As opposed to the store-and-forward policy, worm-hole routing allows parts of a message to be forwarded to the next node even before the entire message has been received. The overlapping of transmitting and receiving of data sets, when done for fixed sized chunks tuned for each system, can also maximize the benefits of cache hits. Matter (described in detail in Section 3.2 of [5]) is an object in protocol-specific form. This object is not deserialized explicitly when used by intermediaries, and allows for the object to be routed by just examining the contents in the header of the object.

- Allow different invocation models: The primary mode of communication available in Proteus is a sequence of asynchronous one-way messages, as most communication modes can be built on top of this model. This allows applications to use various programming paradigms for invoking remote operations. For example, it is possible for Proteus enabled applications to interoperate with SOAP-based Web services as well as high-performance scientific computing applications that use binary communication protocols.

2.4.1 Message Model

Messages in Proteus are exchanged between *nodes*. These nodes function as endpoints and provide a binding context for reception and transmission of message parameters. The main building block of a message is a *vobject*. Vobjects are similar to value types in CORBA and serializable objects in Java RMI. Each *message type* is named by a string. The format of the string is dependent on the application. For example, C++ and XML based applications can derive namespace information from the format of the string. At the protocol provider (implementation) end, a message is eventually converted to the on-the-wire syntax specific to the implementation protocol.

A message consists of a set of named parts. Each part is either *matter* or a *vobject*. This allows a message's information content to be divided according to the likely role of each part. For example, an intermediary can examine just those parts that provide routing information.

3 Interoperability Approaches

Ideally, multiple different component frameworks would be able to support applications that comprise many components, some of which run in one framework, and some in others. 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.

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 reimplementation. 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. Furthermore, the result of the standardization process may not completely meet the requirements of all potential users, because it is stretched in so many different directions. 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.

Proxies and adaptors are examples of the adage that all computer science problems can be solved with another level of indirection. 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.

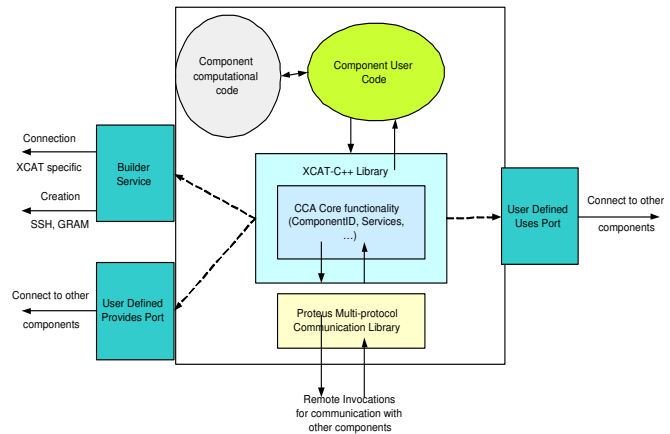


Figure 2. The figure shows the architecture of an XCAT component. The dotted lines indicate that even though the ports reside within the XCAT framework, they appear to users as the connection points/interfaces of the component. Communication for ports is handled via the Proteus multiprotocol library.

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. For example, agreeing to use ASCII over TCP/IP will allow some communication to occur, but interoperability will fail unless there is agreement that to connect to components, we send “connect” instead of .”CONNECT.”

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. It also allows protocols to be chosen dynamically, so once initial communication has been established with the common protocol, the components can switch to more efficient protocols.

Figure 2 shows the architecture of an XCAT component that uses Proteus for all its communication needs. A component user has access to the XCAT framework via the CCA specification. To send and receive data, as required by the

computational code that the component user is running, the user makes calls on ports in the framework. XCAT framework redirects these calls via its internal API with the Proteus multiprotocol library. The Component identifier resides with the XCAT framework, but is implemented as a proxy within the Proteus library. XCAT provides API to the users to switch between the different protocols support by Proteus.

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 them 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.

In the CCA world, applications are composed by connecting *uses* and *provides* ports of components. Each port is given a name, which uniquely identifies the port within the component. The string names of these ports, and Component identifiers of the components they belong to, are passed to the Builder Service to establish the connection. The Builder service retrieves a reference to the *provides* port and places it in an internal table of the *uses* port. This allows all calls on the *uses* port to be directed to the *provides* port via the stored reference. If the components belong to two different frameworks, then a mechanism is required to translate names of ports and format of the Component identifiers from one framework-specific format to another. 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:

- 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.
- 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.

- 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.

Another solution strategy is to maintain proxy servers that serve as front ends to information repositories in remote frameworks. The proxies translate queries in one framework into corresponding ones that have meaning within the other. Ideally the originating query would be expressed in a way that has meaning in another. This is a difficult problem in general, one that is analogous to multi-database research, whose literature includes examples of standards (in the form of federated databases and common high-level query languages), adaptors (query and schema translators and proxies (front-end database transaction managers that route and map incoming requests to back-end databases)).

4.5 Creation

The CCA specification states 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. Examples of environment variables include *stdin*, *stdout*, *stderr*, *LD_LIBRARY_PATH*, location of executable on target machine and location of input files. For interoperability, it is essential to agree on a common format for specification of the environment characteristics that can be used by all frameworks.

Once a component is instantiated, the Builder Service and the new component need to interact through Remote Procedure Calls (RPC) so that the Component identifier of the instantiated component can be returned. So, even though two frameworks may agree on the same RPC middleware for communication, they still need to agree on the exact RPC calls, format of the representation of parameters on the wire, and RPC endpoint(s) necessary to create a component.

The Component identifier is specific to the framework in which the component was created. 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 [10]. A GSR is a precise description of how to contact and communicate with a service instance in a distributed location. GSRs can be complete WSDL descriptions of a service instance. A GSH is an immutable name for a service. This two level naming scheme is needed as a GSR may change over time as a service is moved or upgraded. It is possible for a GSH may be bound to different GSRs over time, but the GSH can always be resolved to the latest version of the service instance. 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 Proteus for Standards-Based Communication Interoperability

5.1 LegionCCA with Proteus

The integration of Proteus into Legion takes advantage of the Legion runtime library's inherent configurability and customizability [21]. Section 2.2 above mentions that Legion

can be configured to use either of two “data delivery layers”, one based on TCP sockets, and one based on UDP. This is possible because sending a Legion message to invoke a Legion function is accomplished through an internal event mechanism. When compiler-generated application code is ready to invoke a function, rather than calling directly to some library routine to implement the send, a “MethodInvoke” event is raised within the address space of the object. This event is captured by whatever handlers are configured at runtime to capture it. TCP- and UDP-based handlers come with the default Legion library. Importantly, the Legion library allows the set of handlers to be dynamically altered at runtime. Thus, the first step toward using Proteus-based communication in LegionCCA is to define and register a set of handlers for all relevant communication events, including *MethodInvoke*, *MessageSend*, *MessageReceive*, and *MethodReceive*. In this way, the Proteus communication library is injected into the path of outgoing and incoming messages, allowing it to replace the default functionality of the TCP or UDP sockets based communication, with Proteus-specific functionality.

Once control is passed to the communication library, Proteus writes all information necessary for a Legion method invocation into a buffer to send to a Proteus-enabled Legion server object (component). This information need not be in a pre-specified format, because like CCA, Legion does not prescribe a specific wire format, and Proteus is currently assumed to run in both client and server components. To capture the relevant information, Proteus-based LegionCCA re-implements the *LegionProgramGraph* internal data structure. *LegionProgramGraph* provides an interface to Legion-targeting compilers so that they may build up data-flow based object interactions. The simplest such interaction is a client-server request/response; more complex interactions can be built to forward results of a function call onto other objects, instead of transferring them directly back to the caller. *LegionProteusProgramGraph* inherits from *LegionProgramGraph*, keeping the same interface and overloading only the implementation of the class's *execute()* method, which is used to make the call once the function name, the server address, and the call's parameters have been added. This re-implementation targets Proteus rather than UDP or TCP sockets, by using a Proteus endpoint and other related communication scheme elements. The Proteus layer selects a communication protocol provider (e.g. XSOAP, binary, CORBA, etc.) and initiates the Proteus invocation on the Proteus-enabled server object. The Proteus layer on the receiving end transforms this invocation request into a Legion message first and then into a Legion method, within the event handling sequence. Thus, *LegionProteusProgramGraph* provides a software module within each CCA component. This component encapsulates the Proteus-specific communication behind a low-level interface at a point where all necessary

communication information (function name and arguments, object name and address, etc.) is available. Because the interface to the LegionProgramGraph remains the same, the impact on Legion-targeting compilers is small.

5.2 XCAT-C++ with Proteus

XCAT-C++ is layered on top of the Proteus multi-protocol library. The framework makes calls to the Proteus API whenever data types need to be exchanged between remote components. The details of mapping remote entities to Proteus concepts is hidden from the user. However, the cost of this layering between user code and Proteus is limited to a couple of virtual function calls. XCAT-C++ allows users to dynamically load a communication module for each component. This module however needs to conform to the Proteus *provider* API.

In [13] we report the performance overhead imposed by the XCAT-C++ component layering on data types commonly used by scientific applications. In the performance study, the high performance streaming XBS parser [4] from the Proteus library was used for communication between XCAT-C++ components. The performance results showed that for all data sizes, of arrays of doubles, the throughput obtained by XCAT-C++ components closely matched that of Proteus-XBS. Performance analysis concluded that the overhead imposed by the component layering in XCAT-C++ is restricted to just a few virtual method calls, and the implementation ensures that all inter-component calls are transferred to the Proteus communication module without buffer copying.

Each ComponentID in the XCAT-C++ framework is mapped to a Proteus endpoint. This endpoint is converted to a string format before it is sent on the wire. The framework unmarshals it into a C++ object whenever a handle to the ComponentID needs to be provided to the user. Our current work involves using a WSDL document to represent this endpoint in a string format.

Proteus provides a SOAP implementation as part of its multi-protocol suite. Currently available tools for XCAT-Proteus include a code generator that uses a WSDL description of a Proteus endpoint to hide SOAP specific details in stubs and skeletons. Interaction between Web services based clients and XCAT-C++ components occurs via the mechanisms provided in Proteus. We are working in incorporating a flexible code generator tool [20] to generate the appropriate stubs and skeleton code for a wide variety of communication protocols that will be specified in the WSDL document for an XCAT provides port.

5.3 Proteus Mediated Communication

As described above, both LegionCCA and XCAT-C++ have hooks in their frameworks to invoke the Proteus API

for communication with remote objects. The next step is to use this common layer to enable interoperability between the two frameworks. Since SOAP is the de facto standard for distributed systems, we will first use the *uses* port in LegionCCA or XCAT-C++ components to initiate communication with the remote entity and negotiate the use of a common efficient protocol. This step is important to provide application users with the option of interacting with Web services, if necessary. If the two communicating entities are CCA components in XCAT-C++ or LegionCCA, Proteus switches to a high performance streaming binary protocol. As a result, mediation by Proteus, ensures that the same wire-protocol is used by both the *uses* and *provides* components.

Component connection and creation by LegionCCA in the XCAT-C++ framework (and vice versa) is handled by converting remote entities such as ComponentIDs and *provides* ports to Proteus endpoints. Further work is necessary to define exactly what invocations will be used to connect and create components. An advantage of using Proteus as the communications substrate is the ability to communicate with other services using SOAP. We are currently working on defining a mapping between CCA and SOAP concepts, such that a CCA component can control SOAP features that have no analogue in the CCA. For example, should a SOAP service look like a “virtual” CCA component to a normal CCA component?

6 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. We discuss specifically how these approaches combine to contribute to the design of our interoperability efforts for two CCA component frameworks, LegionCCA and XCAT-C++.

Future work will involve the design and development of other aspects of interoperability, including naming, description, specification, and discovery.

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 Computing*, August 1999.

- [2] D. E. Bernholdt, W. R. Elwasif, and J. A. Kohl. Communication Infrastructure in High-Performance Component-Based Scientific Computing. volume 2474 of *Lecture Notes in Computer Science*, pages 260–270. Springer, September 2002.
- [3] CCA Forum. Common Component Architecture Forum, visited September, 2003. <http://www.ccaforum.org>.
- [4] K. Chiu. XBS: A streaming binary serializer for high performance computing. In *Proceedings of the High Performance Computing Symposium 2004*, 2004.
- [5] K. Chiu, M. Govindaraju, and D. Gannon. The Proteus Multiprotocol Library. In *Proceedings of Supercomputing 2002*, November 2002.
- [6] A. Cleary, S. Kohn, S. Smith, and B. Smolinski. Language Interoperability Mechanisms for High-Performance Applications. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-Operable Scientific and Engineering Computing*, 1998.
- [7] E. Christensen, et. al. Web Services Description Language (WSDL) 1.1, March 2001. <http://www.w3.org/TR/wsdl>.
- [8] N. Elliot, S. Kohn, and B. Smolinski. Language Interoperability for High-Performance Parallel Scientific Components. In *International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE 1999)*, San Francisco, CA, September 29 - October 2nd.
- [9] D. C. Erdil, K. Chiu, M. Govindaraju, and M. J. Lewis. A Proteus-Mediated Communications Substrate for Legion-CCA and XCAT-C++. *Workshop on Component Models and Frameworks in High Performance Computing (CompFrame 2005)*, July 2005.
- [10] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer* 35(6), 2002.
- [11] D. Gannon, R. Ananthakrishnan, S. Krishnan, M. Govindaraju, L. Ramakrishnan, and A. Slominski. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 9, Grid Web Services and Application Factories. Wiley, 2003.
- [12] M. Govindaraju, H. Bari, and M. J. Lewis. Design of Distributed Component Frameworks for Computational Grids. *To appear in International Conference on Communications in Computation*, June 2004.
- [13] M. Govindaraju, M. R. Head, and K. Chiu. Xcat-c++: Design and performance of a distributed cca framework. *The 12th Annual IEEE International Conference on High Performance Computing (HiPC) 2005*, pages 270–279, Goa, India, December 18-21.
- [14] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. Merging the CCA Component Model with the OGSF Framework. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 12-15, 2003, Tokyo, Japan.
- [15] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Legion: An operating system for wide-area computing. *IEEE Computer*, 32((5)), 1999.
- [16] Indiana University, Extreme! Computing Lab. Grid Web Services. <http://www.extreme.indiana.edu/xgws/>.
- [17] 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.
- [18] M. Lewis, A. Ferrari, M. Humphrey, J. Karpovich, M. Morgan, A. Natrajan, A. Nguyen-Tuong, G. Wasson, and A. Grimshaw. Support for extensibility and site autonomy in the legion grid system object model. *Journal of Parallel and Distributed Computing*, 63:(525–538), 2003.
- [19] M. J. Lewis, M. Govindaraju, and K. Chiu. Exploring the Design Space for CCA Framework Interoperability Approaches. *Workshop on Component Models and Frameworks in High Performance Computing (CompFrame 2005)*, Atlanta GA, June 2005.
- [20] Madhusudhan Govindaraju. XML Schemas Based Universal Code Generation Framework for Distributed CCA Applications. in *GECO-COMPFRAME06: Workshop HPC Grid programming Environments and COmponents and Component and Framework Technology in High-Performance and Scientific Computing (at HPDC-15)*, Paris, France, June 2006.
- [21] C. Viles, M. Lewis, A. Ferrari, A. Nguyen-Tuong, and A. Grimshaw. Enabling flexibility in the legion run-time library. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '97)*, pages (265–274), July 1997.
- [22] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. In *Concurrency and Computation: Practice and Experience*, vol. 13, no. 8-9, pp. 643-662, 2001.