

# A Proteus-Mediated Communications Substrate for LegionCCA and XCAT-C++

Deger Cenk Erdil, Kenneth Chiu, Madhusudhan Govindaraju, Michael J. Lewis  
Department of Computer Science  
State University of New York, Binghamton NY 13902  
{erdil, kchiu, mgovinda, mlewis}@cs.binghamton.edu

## Abstract

*Component frameworks, including those supporting the Common Component Architecture (CCA), comprise the software infrastructure necessary for disjoint components to interact and coordinate to accomplish an application's task. 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. This paper describes one solution to this problem, namely the use of a common, multiprotocol messaging layer within two disjoint framework implementations. The Proteus multi-protocol communication layer will allow the Legion-CCA and XCAT-C++ CCA framework implementations to support component-based applications to span the two distributed frameworks. This paper describes working implementations of Proteus within both Legion and XCAT-C++, and outlines plans for full integration and communication interoperability.<sup>1</sup>*

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

## 1 Introduction

The Common Component Architecture (CCA) [1] specification prescribes how independent software components can be combined to comprise a scientific application. CCA indicates that all exported component functions must be exposed in the form of *provides ports* and that the functions called by each component are defined in that component's list of

*uses ports*. Knowing all the function signatures implemented in, and called by, each component, allows application development tools to combine the components dynamically into applications. Component frameworks implement the software infrastructure necessary for components to interact and communicate, and CCA prescribes the interface (called the *Services* interface), through which components interact with the framework. This interface, and others that describe other functionality in the specification, is expressed in the form of a Scientific Interface Definition Language (SIDL) interface, which explicitly supports data types most prevalent in high performance computing.

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 [5] language interoperability tool 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. That is, without specifying the wire format for inter-component calls in a distributed CCA framework, two different implementations of the CCA specification cannot communicate without a sepa-

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

rate mechanism to address interoperability. This paper focuses on this particular problem.

We describe one approach to distributed CCA framework communication interoperability. In particular, we use a single multi-protocol communication library called Proteus, and incorporate it into two distinct distributed framework implementations of the CCA specification: LegionCCA and XCAT-C++ [4]. 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++.

The remainder of this paper proceeds as follows. Section 2 presents a high level overview of the three main parts of the project we describe, (1) the Proteus multiprotocol communication library, (2) Legion and LegionCCA, object based grid computing software and a layer on top that serves as a CCA framework implementation, and (3) XCAT-C++, a C++ Web services-based CCA framework implementation. Section 3 then discusses, in separate subsections, the integration of Proteus into LegionCCA, the use of Proteus in XCAT-C++, and the planned interoperability that Proteus will enable between the LegionCCA and XCAT-C++ component frameworks. Section 4 concludes with a summary and directions for future work.

## 2 Background

This section describes briefly the Proteus multiprotocol communication library, the Legion object-based grid computing system and its support for CCA, and the XCAT-C++ Web services-based CCA framework. Other papers describe this work in considerably more detail.

### 2.1 Proteus

Proteus [2] is a multiprotocol library that mediates between the protocol implementation, known as a protocol provider, and the application. Applications are written to the Proteus API instead of the protocol-specific API. Protocol providers can be implemented by wrapping existing protocol implementations.

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.

A separate API is used by protocol providers to interface with Proteus. By wrapping an existing protocol implemen-

tation in such that it conforms to this API, protocols can be added to Proteus.

### 2.2 Legion and LegionCCA

The Legion grid architecture [6] represents each grid component with a Legion object. 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.

**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.

### 2.3 XCAT-C++

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. We highlight key aspects of XCAT-C++ as follows.

**Orchestration of Components:** XCAT-C++ provides a simple yet powerful Python scripting interface to create and connect component instances into a distributed application on a set of remote hosts. The SWIG [7] library provides the translation of calls between Python and the C++ libraries of XCAT-C++. The scripting programming interface allows users to dynamically manage components on the fly, without the need for re-compilation of component code.

**Remote Component Instantiation Mechanism:** The responsibility of component creation and connection belongs to the BuilderService implementation. The XCAT-C++ BuilderService expects the remote creation environment to be specified as a list of name-value pairs. This list includes the preferred redirections for *stdout* and *stderr*, for example. Access to a remote resource is based on the SSH protocol. We are currently adding support for the grid Resource Allocation Manager [3]. Upon successful instantiation, the new component sends its `ComponentID` to the BuilderService that created it, which is subsequently returned to the user.

## 3 Integration

### 3.1 LegionCCA with Proteus

The integration of Proteus into Legion takes advantage of the Legion runtime library's inherent configurability and customizability [8]. 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 reimplements 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 interac-

tion 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.

### 3.2 XCAT-C++ with Proteus

**Communication Substrate:** XCAT-C++ is layered on top of the Proteus multi-protocol library. The framework makes calls on 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.

**ComponentID:** 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 unmarshalls it into a C++ object whenever a handle to the ComponentID needs to be provided to the user.

**Web Services:** 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.

### 3.3 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. The endpoints are serialized into a string format before being sent on on-the-wire. These serialized forms are converted back into framework-specific representations by using the Proteus bindings in the framework.

Further work is necessary to define exactly what invocations will be used to connect and create components. One further advantage of using Proteus as the communications substrate is the ability to communicate with other services using SOAP. Further work is necessary to map between CCA concepts and SOAP concepts, and to define mechanisms 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?

## 4 Summary and Future Work

We present the motivation for using a multi-protocol library to facilitate communication interoperability between disjoint and distributed CCA frameworks. We discuss the design and implementation of LegionCCA and XCAT-C++, emphasizing how *Proteus* has been incorporated in each framework. Finally, we briefly describe how Proteus can mediate between the two frameworks and allow composition of distributed applications that span LegionCCA and XCAT-C++ frameworks. 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 Com-

mon Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, August 1999.

[2] K. Chiu, M. Govindaraju, and D. Gannon. The Proteus Multiprotocol Library. In *Proceedings of Supercomputing 2002*, November 2002.

[3] Globus Alliance. Grid Resource Allocation and Management. <http://www-unix.globus.org/toolkit/docs/3.2/gram/ws/>.

[4] 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.

[5] 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.

[6] 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.

[7] SWIG. Simplified Wrapper and Interface Generator. <http://www.swig.org>.

[8] 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.