Design and Implementation Choices for Implementing Distributed CCA Frameworks

Rainer Schmidt*, Michael R. Head†, Madhusudhan Govindaraju†, Michael J. Lewis† and Siegfried Benkner*
*Institute of Scientific Computing, University of Vienna, Austria
†Grid Computing Research Laboratory (GCRL), State University of New York (SUNY) at Binghamton

Abstract— **The Common Component Architecture (CCA) specification is designed to provide a plug-and-play environment for scientists to manage the complexity of large-scale scientific simulations. The same specification is used for the implementation of sequential, parallel and distributed frameworks. The CCA specification places minimal requirements on the framework design, thus allowing various research groups to manage the complexity of the underlying run-time systems in ways that match the performance requirements of their target applications. In this paper we discuss the various design choices, constraints and complexities of implementing the CCA specification for high-performance distributed applications. In particular, we focus on the following CCA features: component instantiation and connections, port type representations, Builder Service design, choice of middleware, remote component communication, registration and discovery, client interface and QoS. We present a discussion on the design space of distributed CCA frameworks with specific examples from three concrete implementations: VGE-CCA, XCAT-C++ and LegionCCA.** [1]

I. Introduction

An important challenge in building and deploying high performance scientific applications is providing a software development model that abstracts the complexity of the run-time environment and simplifies the task for scientists, allowing them to focus just on the details of their particular application. The software engineering benefits of component based software have been widely described in the literature: they facilitate encapsulation, reuse of existing components, and the modular construction of programs, resulting in improved application productivity. Component based systems also foster code re-usability and provide high level abstractions to shield users from low level details. They provide a manageable unit for software testing, distribution and management, and reduce the complexity of building large scale scientific applications, which often require the integration of multiple numerical libraries into a single application. Component architectures are well-suited for scientists to build applications by composing existing software components that exploit specialized computing and storage resources. The plug-and-play characteristic of component architectures provides the ability to reuse components in multiple applications, and serve performance needs by allowing components to be swapped at run-time with others that meet the required Quality of Service (QoS) metrics.

A consortium of university and national laboratory researchers launched the "CCA Forum" [1] in 1998, to develop a Common Component Architecture (CCA) specification for large scale scientific computation. The DOE Office of Science recognized CCA as one of its top 10 science achievements of 2002 [2]. The CCA specification defines the roles and functionality of entities necessary for high performance component-based application development. The specification is designed from the perspective of the required behavior of software components. However, the design and implementation of the framework and implementation of various features is dependent on the target applications that each framework is designed for. In this paper we focus on the mapping of the CCA specification to distributed frameworks and discuss the design space for various implementation choices including: component instantiation and connections, port type representations, Builder Service design, choice of middleware, remote component communication, registration and discovery, client interface and QoS.

VGE-CCA [3] is a prototype implementation that provides CCA-based programming capabilities on top of the Vienna Grid Environment (VGE) [4]. VGE provides a Grid infrastructure for secure and automatic provision of compute-intensive applications running on parallel hardware over standard Web service technology. As a key feature, VGE supports negotiable QoS support for time-critical service provision, which has been utilized in the GEMSS [5] Project for the Grid provision of advanced medical simulation services [6].

XCAT-C++ [7] is an implementation of CCA for distributed scientific applications. It uses a high-performance multi-protocol library so that the most appropriate communication protocol is employed for each pair of interacting components. Scientific applications can dynamically switch to the most suitable communication protocol to maximize effective throughput. The component layering imposes minimal overhead and application components can achieve highly efficient throughput for the large data sets commonly used in scientific computing.

Legion [8] has its own protocols and formats for remote method invocation. Therefore, a CCA mapping to Legion uses Legion's method invocation, object identification, creation mechanisms, and other services. LegionCCA is evolving to include support for Web Services, in recognition of their emergence as Grid computing standards.

In earlier work, we discussed the characteristics of distributed frameworks with XCAT-Java [9] and an early version of LegionCCA as case studies [10]. We build on that work

[1]Supported in part by NSF grants IIS-0414981, CNS-0454298, ACI-0133838, and DOE Grant DE-FG02-02ER25526.

by focusing on the design space and implementation details of interesting distributed CCA features with the following as case studies: VGE-CCA, XCAT-C++ and latest version of LegionCCA.

The rest of the paper is organized as follows: we provide an overview of the CCA specification and its key concepts in Section II. Section III provides a brief introduction to the three distributed CCA frameworks: VGE-CCA, XCAT-C++ and LegionCCA. In Section IV, we present the various design features of distributed CCA frameworks and discuss them with reference to specific implementation details of the three frameworks. We discuss initial plans for an interoperability standard for distributed CCA in Section V. Finally, we present conclusions and pointers to future work in Section VI.

II. The Common Component Architecture

The Common Component Architecture (CCA) [1] specification is an initiative to develop a common architecture for building large-scale scientific applications. The CCA places minimal requirements on components to facilitate the integration of existing scientific libraries into a CCA framework and also to minimize the impact of the component layer on performance. The component specification of the CCA is expressed as a set of interfaces that precisely state the expected behavior for component-to-component and component-to-framework interaction. The specification does not mandate the use of any specific form of distributed or parallel technology as the underlying communication architecture, thereby ensuring that it does not preclude applicability to serial, parallel, distributed or Grid systems. This approach explicitly facilitates research groups to focus on utilizing the same high level component specification for a wide range of distributed and parallel applications including combustion research, computational chemistry, remote data visualization and global climate simulation [11]. The functionality provided by a component may be used in a wide variety of applications. Also, a number of different components can provide the same functionality by implementing the same set of component interfaces. Application scientists can thus choose from a palette of available components and mix, match and experiment to formulate effective solution strategies.

The CCA specification is specifically focused on the needs of high performance applications that include direct-connect communication between collocated components, performance engineering, use of parallel and distributed components, interoperability between different CCA frameworks, and support for languages commonly used by scientists such as Fortran. The CCA specification targets the needs of high performance applications and it is this focus on support for high-performance that differentiates CCA from other component models. The CCA promotes interoperability by requiring all components to define their interfaces via a Scientific Interface Definition Language (SIDL) [12]. The Babel toolkit [13] can be used to generate glue code from SIDL to many programming languages including C, C++, Java, Fortran and Python. SIDL has been specifically designed for high performance scientific applications. It explicitly supports complex numbers, dynamic multi-dimensional arrays, parallel attributes, and communication directives. Babel [13] provides a suite of tools to allow efficient interaction between two components developed in different programming languages, but that are co-located in the same process. The CCA Forum defines a component as a software unit that interacts with other components encapsulating well-defined functionality, or a set of functionalities. A component can consist of multiple processes (an MPI job, for example), or multiple components could all be running within a single process. Some fundamental CCA concepts include *ports*, *services objects*, *component identifiers*, and the *builder service*. We discuss these concepts in detail as well as their implementations in VGE-CCA, XCAT-C++, and LegionCCA, in Section III.

Communication between CCA components takes place via their ports, which follows a *uses/provides* design pattern. A *provides* port is the public interface implemented by a component. It can be referenced and used by other components. It can also be viewed as the set of services that are exported by the component. A *uses* port is a connection endpoint that represents the set of functions that it needs to call. Port descriptions for CCA components are provided using the SIDL specification. CCA applications are composed by connecting the *uses* port of one component to the *provides* port of one another. The mechanism by which calls are transferred from the *uses* port to the *provides* port of the connected component is handled differently by each underlying framework.

III. Example Distributed CCA Frameworks

A fundamental requirement for distributed frameworks, as opposed to all sequential and some parallel frameworks, is that calls between components must be supported across address spaces and machine boundaries. The framework must provide necessary hooks to enable the instantiation of remote components, efficient communication between components, management of component binaries in a heterogeneous environment, and tools to allow composition of various components into applications. Furthermore, registration and discovery of components is also required for components to dynamically search and connect to other components.

Distributed frameworks differ in the design and implementation of these features. In this section we provide an introduction to three distributed CCA frameworks and highlight their key features.

A. VGE-CCA

The VGE-CCA prototype framework implements the CCA specification on top of a Web services based Grid infrastructure. The Vienna Grid Environment (VGE) provides a generic service provision framework that encapsulates native HPC applications available on clusters or other parallel hardware. It offers a common set of operations for job execution, job monitoring, data staging, error recovery and application-level quality of service support.

The idea behind this work is to integrate Grid and Web services with a component model for remote, peer-based application composition, coordination, and execution. A key design goal is the preservation of the service-oriented architecture of the environment. The system therefore provides parts of the CCA framework functionality as common Web services to clients and components. The framework also serves as a component registry and maintains a proxy repository facilitating dynamic discover and access of Grid components. For remote communication the system uses SOAP-RPC via HTTP and utilizes SOAP attachments for large data transfers.

A significant design aspect of a Grid component framework is the way components are addressed. Services in a Grid may be entirely stateless (e.g. providing Certificate Revocation Lists) and addressed by a Web service endpoint or may provide access to a remote resource (e.g. using WSRF [14]) requiring additional addressing information. VGE services work based on a session mechanism that maps a conversational identifier to an individual client application.

To address time-critical Grid applications, such as medical simulations, VGE services provide application-level QoS support. Clients may dynamically negotiate various QoS guarantees (e.g. response time, price) in the form of Web Service Level Agreements (WSLA [15]) with VGE services. The VGE-CCA Builder service integrates this mechanism by supporting QoS related component attributes that have to be supplied by the user. These QoS descriptions include a descriptor containing meta-data about a specific service request (e.g. mesh size, accuracy) and a document specifying the required QoS constraints (e.g. response time). The framework utilizes a negotiation-broker service to locate and create a component of a certain application-level quality at run-time.

B. XCAT-C++

XCAT-C++ is an implementation of the CCA specification that is designed for distributed scientific applications. XCAT-C++'s design is modular, so that the capabilities of the system can be easily extended. This allows specialized components, developed by different institutions and stored in a common repository, to be seamlessly combined to form distributed applications that address domain specific needs. In previous work, we showed that instead of extending a single communications subsystem to handle the wide range of scientific computing requirements, a more effective solution is to use multiple communication protocols [16], [17]. When additional performance is needed, a multi-protocol approach allows a faster, more specialized protocol to be dynamically inserted to move data. XCAT-C++ uses a multi-protocol communication library so that the appropriate communication protocol is employed for each pair of interacting components. It also provides the capability at the application level to seamlessly and dynamically switch to a more suitable communication protocol to maximize effective throughput. The component encapsulation adds additional levels of indirection in the execution stack of every component call. However, the overhead due to the component layering is minimal (two virtual function calls) and does not impact the overall performance of the distributed application. Each XCAT-C++ component can interact with endpoints that are compliant with Grid Web services standards. XCAT-C++ has a flexible, extensible and powerful code generation toolkit that can generate the transport protocol specific code and shield away the complexity of the run-time specific details in *stubs* and *skeletons*.

C. LegionCCA

The LegionCCA approach to implementing the CCA specification over Legion, is to model CCA components as Legion objects. Each CCA object gets its own address space, name, and interface within Legion. Legion objects that serve as CCA components are linked against a *LegionCCA library*, which gives them the added functionality to run within the CCA framework atop Legion. This library implements 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. The LegionCCA library contains implementations of important CCA types, including ComponentID and Ports, a Services Object and a builder service for the component, and a connection table that describes how the component is attached to others in the framework. Behind these CCA-defined interfaces, LegionCCA uses the mechanisms enabled by the Legion run-time library (LRTL) to carry out their definition. For example, the LRTL

- implements a Legion Object Identifier (LOID) type, which is used as part of a CCA component ID to identify and find objects/components
- automatically binds named LOIDs to process addresses, and uses a sockets-based communication library to deliver messages to remote objects and services
- contains client-side stub routines that allow callers to invoke remote Legion services, including those for creating, discovering, and destroying objects (components)

This approach—implementing CCA components as Legion objects—achieves the benefits of exporting a standard component framework to applications developers, without having to reinvent and re-implement every Grid and middleware service for this framework to run in a Grid environment. Thus, the design and implementation choices described throughout the rest of this paper are heavily influenced by the mechanisms and solutions of the Legion version 1.8 implementation. More details about Legion can be found in other papers [8], [18].

IV. Design and Implementation

In this section, we present design choices and implementation aspects for distributed CCA framework implementations. The design space we identify, targets CCA implementations built upon Grid and Web service technologies. We present a case study using three concrete implementations, VGE-CCA, XCAT-C++ and LegionCCA. We examine the implementation of fundamental CCA concepts including component identification, ports, connections, and framework as well as features relevant for Grid computing including middleware support, remote job execution and client-sided programming interface.

A. Key Design Choices

1) ComponentID: The ComponentID is an opaque handle to a component that can be used to introspect the component and obtain information on the list of available ports and types. The ComponentID also plays an important role in component assembly. It is typically implemented as a global pointer to the *Services* object of a CCA component.

- VGE-CCA implements a one-to-one relationship between Web services and CCA components. Every Web service port is mapped to a CCA *provides* port, dependencies on other services are modeled as CCA *uses* ports. Services are remotely accessed using local proxy objects that encapsulate interaction details and SOAP messaging. The ComponentID is designed in a way that it can be used by a proxy to identify a particular service or an application provided by a Web service. The ComponentID comprises a Web service endpoint and a conversational identifier, if supported by the remote service. In its serialized form, the ComponentID is represented as a service endpoint optionally containing an URL encoded session identifier. The ComponentID provides sufficient information to detect co-location of components, which can be exploited using native method calls for inter-component communication instead of processing the whole SOAP protocol stack.
- In XCAT-C++, the ComponentID has been designed as an object with a remote interface. Since XCAT-C++ uses the *proteus* [16] library for communication, the ComponentID is represented as a string-encoded proteus endpoint when it is transmitted on the wire. The information in the endpoint that represents the ComponentID includes information on the host, port, communication protocol and a globally unique ID.
- Each LegionCCA ComponentID consists of three elements: *type*, *string name*, and *handle*. Legion LOID is used to serve as a unique identifier for the component in the Legion Grid. The string name is resolved within Legion's *context space*—a global namespace of user-defined strings—and bound to the LOID of the Legion object that implements the specified component. The Legion library creates a local proxy for each LOID. Methods invoked on this proxy are transported to the remote component by the Legion library.

2) CCA Ports: CCA components exchange data via their ports. The design and implementation of the port API, usually specified in SIDL, is different in each framework.

- The VGE-CCA *Services* object allows components to register descriptions of the port types, that they provide and use, with a common "framework Web service". A port description contains the ComponentID, port name, method signatures, and optionally a list of properties. Port objects can be used via the Builder service API to remotely interconnect components that exhibit a pair of complementary *uses* and *provides* ports (using a remote connection interface). Furthermore, a Web service proxy object can be retrieved from a *provides* port object to directly access a service.
- In the XCAT-C++ framework, ports are instantiated by *factories* that are specialized for each port type. These factories are generated during the code-generation phase for each component. Whenever a port is registered via the standard CCA API, a local object is created that is specialized for that port type. The stub-skeleton code automatically converts a *provides* port into its on-the-wire representation whenever it is passed as a parameter in a method call between components.
- LegionCCA manages its *uses* and *provides* ports via tables in the implementation of the Services object. The additions and deletions of ports to the component are recorded in these tables. Access to the *uses* port information is however restricted to local calls. Legion supports both function identifiers and interfaces. Legion objects have the capability to obtain information on the list of methods that are supported by any other object in the Legion Grid. The existing Legion C++ classes, LegionFunctionID and LegionInterface, are used by the Services object implementation to gain access to the Legion library for serialization of calls on the CCA ports.

3) BuilderService: The CCA specification states that the responsibility of the Builder Service includes creation, connection, disconnection, and destruction of components. The same Builder Service API is used by all sequential, distributed and parallel CCA frameworks. However, each framework implements these features in a unique way.

- Using the VGE-CCA Builder Service, components can be introspected to discover the ports they provide and use. The ports a Web service exposes are modeled as CCA *provides* ports and may be accessed by clients or other components via *uses* ports implemented as stubs/proxies. A component registers its ports with descriptors containing information required to discover and invoke its services (e.g. interface descriptions, associated properties, proxy class) with a common "framework Web service". The remote framework also maintains a proxy repository that is accessed by clients in order to generate appropriate *uses* ports.
- Each XCAT-C++ component has a BuilderService by default. XCAT-C++ has a services based architecture, and consequently each of the BuilderService methods can be implemented by a standard Grid/distributed service for creation, connection, disconnection and destruction. XCAT-C++ just wraps implementations of these services with CCA component layering. The creation service is based on "SSH", and current work is directed towards incorporating the GRAM [19] library for authenticated launch on Grid resources.
- LegionCCA maps all methods in the BuilderService API to corresponding calls in the Legion library. This is possible as Legion maps objects to components, and many of its features directly match those required by the CCA specification. The *createObject* and *destroyObject*

methods are used to create and destroy components respectively.

4) Component Creation and Connection: The requirement that distributed applications span multiple address spaces implies that frameworks must have the capability to instantiate components on remote machines. Each component may need many environment variables to be set to appropriate values for its execution, including JAVA_HOME and LD_LIBRARY_PATH. The framework also needs to be aware of the various architectures used in a heterogeneous environment to ensure that the right binary format is used on the component host machine. In some cases it is also required for the component binaries to be stored to, replicated, or fetched from a remote location.

A standard API for connecting ports is necessary to facilitate the assembly of components into a distributed application via standard builder tools including scripts, GUIs and portals. For a connection to take place between two components in a distributed framework, a remote reference to the *provides* port of the first component needs to be placed in the table associated with the *uses* port of the other component.

- VGE-CCA does not create new Web services by remotely starting a Web service container or by using a service factory as is common in OGSI implementations. Instead, the framework makes existing Grid/Web services accessible by means of the CCA Builder service. For stateless Web services, *create* generates a local component representation that refers to a service matching the component description. VGE services provide access to remote applications by additionally incorporating an session identifier that is mapped to particular client requests. The creation of a VGE component may coincide with a QoS negotiation resulting in WSLA contract and advance resource reservation. This mechanism allows the component to provide certain (application level) quality guarantees.
  In VGE-CCA, applications are constructed from abstract component descriptions that are mapped to available resources at run-time. Components can therefore be connected by their *provides* and *uses* ports. Connections are accessed within the components through corresponding *uses* ports. If a component connection is time/state dependent it may be required that the interaction is controlled by the client application. A component may therefore make a connection endpoint remotely accessible over a *provides* port (e.g. data push).
  If a client invokes the *connect()* method of a remote component, an interface descriptor identifying the corresponding *uses* and *provides* ports as well as the target ComponentID is passed and stored at the source component's connection table. Ports are distinguished logically by their interface type but can be cast and connected to any port with a similar method signature. This allows a component to register multiple *uses* ports of the same type (e.g. a database port) that can be connected to different service ports. A component may also simply use the local Builder service to create and access remote components in a transparent manner.
- In XCAT-C++, the “Creation” and “Connection” services are implemented as two separate modules and the BuilderService provides a common interface for these services. When the BuilderService is used to connect two components, XCAT-C++ uses non-CCA calls between components to retrieve the specific *provides* port and place it in the internal table of the component with the *uses* port. These calls are made using the communication API provided by the proteus library, and the details of these calls are transparent to the user. Unlike, LegionCCA, XCAT-C++ does not maintain a central table to track all the connections in the application. Current work in XCAT-C++ is focused on incorporating an event system to keep track of the state of creation, connection and deletion operations.
- LegionCCA objects export their interfaces to other objects through explicit object-mandatory functions that are guaranteed to be in the interface of all Legion objects. That is, any object or service running within Legion can call the getInterface() function on any other object. This facility means that half of the necessary introspection functionality—the half for *provides* ports—has direct built-in support from Legion. The CCALegion library needs to explicitly maintain *Uses* ports information, however, because the functions that a Legion object calls are not readily available to other objects at run time.
  A connection is established by retrieving a handle for the remote *provides* port and placing it in the appropriate table entry of the corresponding *uses* port in the other component. Each component has a table that contains information about the connections of all its *provides* and *uses* ports.

5) Communicating with Remote Components:

- VGE has been mainly implemented in Java and relies on standard Web Service technologies such as WSDL, SOAP/HTTP, and WS-Security. It utilizes the open-source frameworks, Tomcat and Axis, for service hosting and deployment. For large file transfers VGE utilizes SOAP attachments. The VGE-CCA client API provides mechanisms to facilitate dynamic retrieval of Web service proxies from a remote repository. The component framework currently uses custom SOAP serializers, based on Java serialization and Base64 encoding, to transfer CCA specific data types (e.g. port, services) over the network. XML-based serialization mechanisms for these data types will be added in future versions.
- Due to the diverse communication characteristics of distributed scientific applications, XCAT-C++ uses the proteus multi-protocol library as the communication substrate in the framework. Proteus currently has support for two protocols: (1) XBS [20], an efficient streaming binary protocol; and (2) XSOAP, a C++-based implementation

of the SOAP specification. Communication between two XCAT-C++ components can dynamically switch on a per-call basis. Communication modules, adhering to the proteus API, can also be dynamically loaded. Current work is focused on using the proteus communication library as the basis for interoperable communication between different CCA frameworks [21], [22].

- In practice, communication between Legion objects, and therefore LegionCCA components, is carried out over a data delivery layer based on Unix sockets. This layer itself is a replaceable feature of the Legion library; implementations for TCP-based and UDP-based communication could be replaced by any other mechanism that delivers data from one process that represents a Legion object, to another. The particular data delivery layer in use must match with the object identifier type that is used for the low-level name of a running Legion object. Sockets-based object identifiers contain pairs consisting of IP addresses and port numbers. Since this object id type can itself be replaced by a user-defined type (containing a URL, for example), and since the global name of the object is a location-independent LOID that is mapped down to the object address, the Unix sockets data delivery layer could be replaced.

6) Implementation of the Client API:

- VGE-CCA provides a client-side Java API that implements the CCA Builder service concepts. Components may be described and created based on the ComponentID or an abstract component description. Applicable services are located and selected at run-time using the component framework service. A component description comprises the set of required interfaces the component contains as part of the CCA specification. Furthermore, the component may be associated with meta-data attributes assigned to predefined constants (e.g. service name, application class, or QoS attributes). A component instance can be introspected on the ports it provides as well as on associated meta information. Connecting pairs of compliant *uses*/*provides* ports allows establishment of peer connections between the services. Clients may directly access component ports to control time and state dependent interactions (e.g. retrieve application status, stage output data). A mechanism is therefore provided to generate a proxy object of a desired interface type based on a *provides* port. At run-time, the client environment invokes a service using a proxy class associated with the *provides* port. In case of an invocation failure or if the required libraries are not accessible locally, the client environment tries to retrieve a proxy implementation from the remote repository using a dynamic proxy mechanism.
- Often times, scientists repeatedly run the same simulation by changing just some of the execution parameters. For such cases, a scripting interface to the CCA API is invaluable. XCAT-C++ provides a Python interface enabled by the Simple Wrapper Interface Generator (SWIG) [23] to translate calls between Python and the C++ library of XCAT-C++. Currently, the XCAT-C++ team is exploring the use of this scripting interface in incorporating XCAT-C++ components in work-flow specifications and portal environments.
- An interface one level higher than the data delivery layer, described in Section IV-A.5 above, implements the Legion method invocation protocol, which defines how parameters and return values are packed and delivered, how remote functions are identified, and how processes are named, among other things. This layer is abstracted within client code by a local data structure called a "program graph." A Legion program graph abstracts one or more run-time invocations, by holding the name of the function to be called, all the necessary parameters, and enough information to find and deliver all resulting return values. Program graphs are intended to be built up from within automatically generated code. Legion contains a stub generator that builds client-side stubs for calls on remote Legion objects (through program graphs), given remote object interfaces. LegionCCA adds support for expressing these remote interfaces in CCA-based SIDL. The Legion method invocation layer can also be generated by Legion-targeting compilers of higher-level parallel or Grid-enabled languages, including, for example, Mentat.

7) Quality of Service:

- VGE has been designed to support a multi-phase service access model that may comprise an administrative phase, a QoS negotiation phase, and the application execution phase. Using the VGE QoS module, parallel applications may be provided as dynamically configurable Grid services, which, depending on the requirements of a client or client application, may be executed on many processors in short time but for a higher price, or on a few processors with a lower price. The VGE-CCA client environment integrates QoS support by providing means for qualitatively describing a VGE component, which are dynamically selected by the component framework using a negotiation-broker service.
- In XCAT-C++, a common-denominator protocol can be used to negotiate the use of the most optimized protocol available with both the frameworks. The negotiation and switch to the appropriate protocol can be handled by the framework and remain transparent to the application. A similar design can be used by LegionCCA, which has a successfully incorporated the proteus library with the Legion communication system [21].

V. Interoperability Between Distributed CCA Frameworks

To leverage the strengths of each distributed CCA framework, it is important to develop an interoperability standard to facilitate the design and development of applications that can transparently span multiple frameworks. The format and implementation of inter-component calls are not prescribed by the CCA specification, since no single communication protocol suits all applications. By not mandating such implementation details, the CCA specification provides considerable flexibility to each framework to customize the implementation in accordance with the needs of target applications. However, as a result, an interoperability standard will require a strategy outside of the CCA. Elsewhere [22], we discuss five requirements for component framework interoperability and three approaches to meet them. For each pair of communicating frameworks, the appropriate approach must be applied for all the requirements. This effectively defines a design space for framework interoperability approaches.

The important interoperability requirements include (1) description and specification of the port interface, (2) specification of the communication protocol, (3) mechanism for naming interoperability for both components and interfaces/ports, (4) standard for discovery of distributed components, and (5) a common standard for creation of components within a framework.

These requirements can be met in several ways. We have classified possible approaches into three categories.

- an interoperability *standard* that explicitly specifies the interfaces and protocols that all implementations are required to follow,
- *adaptors* that can translate objects and other entities of one framework into those of another, and
- *proxies* that can help span frameworks by providing representatives of one framework in another.

All three frameworks, VGE, XCAT-C++, and LegionCCA have the capability to communicate with Web services. This common characteristic makes the use of WSDL documents an easy and effective choice for a common way to specify and describe interfaces. However, given that CCA ports are specified in SIDL, it is important to develop a mapping between SIDL and WSDL descriptions, for interfaces and data types.

The SOAP protocol has emerged as the standard Web services communication protocol. SOAP has many features that make it appropriate for interoperable data exchange in heterogeneous environments. SOAP can therefore serve as the default communication protocol between the distributed CCA frameworks. Two communicating frameworks can first exchange information on their supported protocols via messages in the mutually understood SOAP protocol. After the initial handshake, further communication can then take place in a high performance communication substrate that is agreed upon during the handshake.

In distributed frameworks, specialized handles serve as *remote pointers* (also called *global pointers*) to provide access to objects that live in remote address spaces. Apart from remote pointers, each framework also has the notion of a global namespace, making entity names understood in every component of the framework. This also allows data types that are received on the wire to be mapped to local objects within the receiving process. For applications to span different frameworks, well known registries must be used to convert remote references from the format of one framework to that of the other. Also, for each pair of frameworks, adapters/proxies must be built to convert data types from the sending to the receiving framework.

Our study on a common standard for discovery and creation is in the intial stages and is the subject of active future work. We plan to build on the ideas mentioned in this section to develop a prototype system that exhibits interoperability between different distributed CCA frameworks, and present it to the CCA community for feedback.

VI. Conclusion

Implementing distributed and Grid-based component frameworks imposes special constraints and requirements on the system design. A framework needs to support mechanisms for remote invocation and remote creation as applications typically span multiple address spaces. Inter-component communication must be accomplished remotely and be able to cross address spaces, machine boundaries, and administrative domains. Furthermore, components should be able to self-register with the component framework and allow dynamic discovery. XCAT-C++, LegionCCA and the VGE component framework are built upon Web service and Grid technologies. In this paper, we have shown how the high-level CCA specification can be applied to distributed computational environments. We presented key design choices for Web services and Grid-based frameworks. The design and implementation choices of CCA features decides the kind of applications that each framework is suited to support. In future work, we plan to study the performance tradeoff of these frameworks for representative scientific application workloads

References

[1] CCA Forum. Common Component Architecture Forum. [Online]. Available: {http://www.cca-forum.org}

[2] Office of Science, Department of Energy, "Top 10 DOE Science Achievements in 2002."

[3] R. Schmidt, S. Benkner, I. Brandic, and G. Engelbrecht, "Component based Applications Programming within a Service-Oriented Grid Environment," *Workshop on Component Models and Frameworks in High Performance Computing (CompFrame 2005)*, Atlanta GA, June 2005.

[4] S. Benkner, I. Brandic, G. Engelbrecht, and R. Schmidt, "VGE - A Service-Oriented Grid Environment for On-Demand Supercomputing," in *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (Grid 2004)*, November Pittsburgh, PA, USA, November 2004.

[5] The GEMSS Project: Grid-Enabled Medical Simulation Services. (2005) EU IST Project, IST-2001-37153. [Online]. Available: {http://www.gemss.de/}

[6] S. Benkner, G. Berti, G. Engelbrecht, J. Fingberg, G. Kohring, S. Middleton, and R. Schmidt, "GEMSS: Grid Infrastructure for Medical Service Provision," *Journal of Methods of Information in Medicine*, Vol. 44, 2005.

[7] 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*, Goa, India, December 18-21.

[8] 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*, vol. 63, pp. (525–538), 2003.

[9] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley, "Merging the cca component model with the ogsi framework," in *Proceedings of CCGrid2003, 3rd International Symposium on Cluster Computing and the Grid, Tokyo, Japan*, May 12–15 2003, pp. 182–189.

[10] M. Govindaraju, H. Bari, and M. J. Lewis, "Design of Distributed Component Frameworks for Computational Grids," in *Proceedings of the International Conference on Communications in Computing (CIC)*, June 2004.

[11] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou, "A component architecture for high-performance scientific computing," *Intl. J. High-Perf. Computing Appl.*, 2006, submitted to ACTS Collection special issue.

[12] 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.

[13] 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.

[14] Globus Alliance, IBM and HP . (2004) The WS-Resource Framework. [Online]. Available: {http://www.globus.org/wsrf/}

[15] Web Service Level Agreement (WSLA) Language Specification. (IBM 2001-2003). [Online]. Available: {http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf}

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

[17] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon, "Requirements for and Evaluation of RMI Protocols for Scientific Computing," in *Proceedings of SuperComputing 2000*, November 2000.

[18] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey, "Legion: An operating system for wide-area computing," *IEEE Computer*, vol. 32, no. (5), 1999.

[19] G. von Laszewski, J. Gawor, S. Krishnan, and K. Jackson, *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, 2003, ch. 25, Commodity Grid Kits - Middleware for Building Grid Computing Environments.

[20] K. Chiu, "XBS: A streaming binary serializer for high performance computing," in *Proceedings of the High Performance Computing Symposium 2004*, 2004.

[21] D. C. Erdil, K. Chiu, M. Govindaraju, and M. J. Lewis, "A Proteus-Mediated Communications Substrate for LegionCCA and XCAT-C++," *Workshop on Component Models and Frameworks in High Performance Computing (CompFrame 2005)*, Atlanta GA, June 2005.

[22] 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.

[23] SWIG. (1997) Simplified Wrapper and Interface Generator. [Online]. Available: {http://www.swig.org}