GRID AND COMPONENT TECHNOLOGIES IN PHYSICS APPLICATIONS

S. Shasharina, N.Wang, S. Muszala, R. Pundaleeka, Tech-X Corporation, Boulder, CO 80303, USA

Abstract

Recent advances in computer science have made development and management of large-scale distributed and high-performance applications possible. Examples of such advances are Grid and Component technologies. In this paper we share our experience in bringing Grid and Component technologies into fusion applications.

INTRODUCTION

The paper consists of two parts. The first part is devoted to Grids and describes our experience working with Globus in developing a service for accessing fusion simulations data. The second one introduces Components and describes some efforts using them for the fusion integrated modeling applications.

GRID APPLICATIONS

Grids

There are many detailed papers rigorously defining the concept of a Grid [1], and here we give only a simplified definition of it - based on our experience. Simply put, a Grid is a set of computers connected via a network and shared by a group of people performing various computing and data management tasks. Grids use middleware that makes them more valuable than just a sum of its constituents and protects resources via some kind of user authentication and authorization.

Without a value added, a Grid does not make too much sense. A question like "Why do I need a Grid, since I can just login to all our computers, *scp* needed data and run my codes without a pain of installing new software and extra pain of getting a certificate?" is a valid question. The value is added by the services offered by the middleware. Each service is typically a reflection of a particular virtualization model. Virtualization is an abstraction that presents the resources based on the conceptual view of a particular application.

For example, instead of asking the system to bring file A from computer B and file C from computer D, one can request all the data which is generated by discharges with the current higher than x. In this example, data becomes virtualized, in that it is characterized by a particular attribute, rather than by its form and location. Another example would be if a person wants to submit a computational job needing M processors with software Z, but does not care where exactly to run it or store the results. Virtualized resources (processors and data storage) can figure out how to allocate themselves.

Grids can be used to save storage, because one does not need to replicate data anywhere; save travel funds, because one can work with the whole system and also collaborate remotely; and perform data operations more efficiently, because data analysis and visualization use attribute queries rather then directly referencing data by location and name. Grids can enable things that were not possible before, such as global data and network analysis and global detection, and automate error-prone manual workflows.

The most successful and widely used Grid middleware is Globus [2]. Conceptually Globus exists in the pre-Web Service and Web Service form. The pre-Web service Globus introduced an efficient data transfer (GridFTP), means to run remote jobs and do this with adequate security. The Web Service Globus added a capability to create stateful services explicitly allowing virtualization based on a desired interface. In the next session, we describe one of several Globus-based Web Services that we designed in our projects.

Fusion Grid Service

Fusion and plasma simulations typically run on remote supercomputers and can generate terabytes of 3D data. The data is commonly stored in the HDF5 [3] format, which has a convenient API to query and access data of interest through methods that extract data into the local memory directly. The goal of the Fusion Grid Service was to mimic this local HDF5 API so that scientist could access remote data as if it was local. Such access to remote data makes the transfers of the whole (possibly very large) file unnecessary and storing accessed data optional as data can be analyzed in the client memory.

The development of such a service starts with defining its interface using the Web Service Definition Language (WSDL). WSDL requires extension to accommodate complex types such as multidimensional arrays. To extract and transfer HDF5 datasets (which are multidimensional arrays) we had to define two methods. The first method performs a query for the array metadata (numerical type, rank and dimensions). Using this method, the client allocates the needed memory, and then uses GridFTP (not a SOAP call) in the second method to copy the extracted dataset into the client memory (Figure 1). Using GridFTP rather than a SOAP call required the extra step of saving the extracted data in a temporary file on the server, as we could not find a way to do GridFTP transfers from memory to memory. The decision to use GridFTP was made despite this extra step because GridFTP allows using multiple, parallel data streams and is much more suited for large data transfers.

In doing this work we used Globus 4.0.2 and faced a couple of inconveniencies. First, Globus generates Makefiles for client and server, but does not provide a means to include external files and libraries prior to the generation of Makefiles. We had to replace the Globus configuration files, and create custom shell scripts to generate the final Makefiles in order to use HDF5 in the implementation. We also found that we had to put all

implementation in the generated service directory. Later we figured out that this can be avoided by putting implementation into a shared library and use dlopen() to access implementation in the generated directory.

Second, we discovered that only the person who starts the service (C Globus container) could use the service. This problem appears to be only in the C Core, and is not present in Java bindings. The Globus team provided us with a patch that fixed the problem but we do not know if Globus is patched in the later release. Generally, it is our impression that the Java Core of Globus is more developed than the C Core.

The service is now fully implemented and has methods to query datasets metadata, get a dataset, get a hyberslab, select a stride and perform the full file transfer using multiple GridFTP streams. It is installed at NERSC, PPPL and Tech-X.

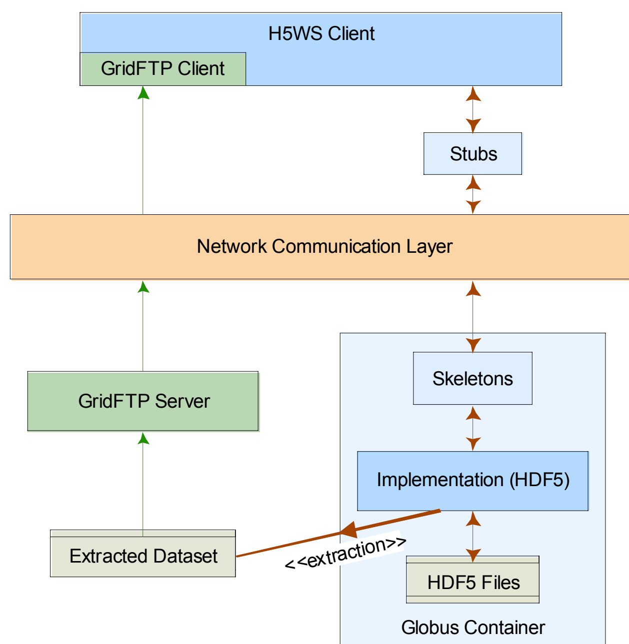

Figure 1: The architecture of the Fusion Grid Service.

Performance Tests

Prior to the full development of the service we performed timing tests comparing the service's performance with a CORBA C++ system using TAO-1.5.2 [4] and gSoap C++ service using gSoap-2.7 [5] with the Direct Internet Massage Encapsulation (DIME) attachment using the following three connection setups (see Table 1):

1. LAN: the server and the client are connected via a 100Base-T switch.
2. WAN: the server and the client are connected via the "regular" Internet backbone.
3. ESnet: the server and the client are connected via a high speed wide-area network, such as the ESnet [6] (OC-3: 155 Mbps), sponsored by the DOE.

Table 1: The connectivity of the test cases. RTT is the round trip delay time. BDP is the bandwidth-delay-product.

Setup	Bottleneck bandwidth (MByte/sec)	RTT (msec)	BDP
LAN	12.5	0.27	3.4 KB
ESnet	125.0	72.0	9.0 MB
WAN	0.19	162.0	31.4 KB

Each test had the following steps: establish the connection, query dataset metadata, allocate adequate memory on the client side, and send data to the client. In the case of our service, there was an extra step to save dataset in a temporary file on the server. Figure 2 shows the throughput of each of the HDF5 retrieval solutions using different networks (see also [7]).

As seen from Figure 2, in the LAN connection, our service has too much overhead (related to security and storing data in a temporary file) and is less efficient than the other solutions. CORBA system saturates the connection in this case, while gSoap saturates about 50% of it.

In the ESnet scenario, our service seemingly loses if a single stream is used for GridFTP. But with an addition of just one more stream, it fares better than CORBA and gSoap (see Fig.3). None of the systems reached the bottleneck throughput. This is probably due to the fact that the receiving buffer at Tech-X is 85K, which is much smaller than the ESnet BDP.

In the WAN scenario, our service gives the best results and uses the bandwidth well with just one stream, while other systems underperform. Though the service fared well versus CORBA and gSoap in this setting, not much difference was observed in bandwidth utilization when multiple streams were added to GridFTP client. This is due to the fact that just one stream (0.16 MB/sec, WAN results on Figure 2) is close to the bottleneck (0.19MB/sec in Table 1) so adding more streams does not improve the throughput.

Our performance tests proved that using multiple streams of GridFTP works best in networks that have large bandwidth and large latency connections and justified our choices of the service technology (Grid Service using GridFTP).

This said, we would like to note that in choosing distributed solutions, developers might not be directed only by the performance tests. Other criteria include provided security (our service using Globus is the winner here), size of the footprint (gSoap is the lightest, then CORBA followed by Globus), and the ease of running the service (CORBA and gSoap need only the server and the client processes, while our solution needs a GridFTP server running on the server side). This means that other teams might find approaches different from ours more plausible.

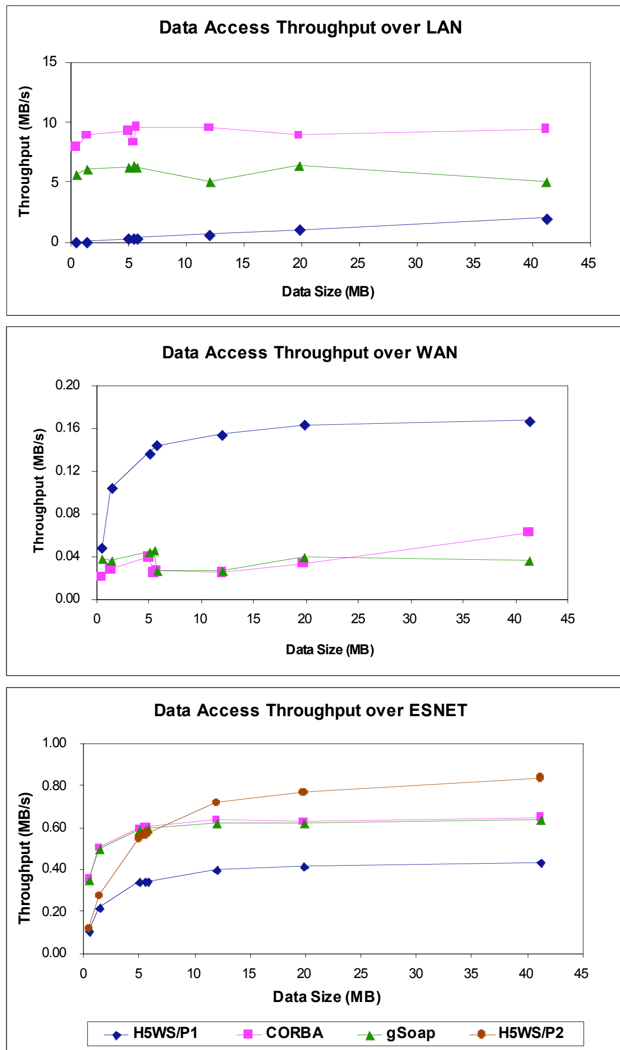

Figure 2: Data access throughput versus the size of transferred data for LAN, ESnet and WAN setups. H5WS is a short name for the Fusion Grid Service.

COMPONENTS APPLICATIONS

Component Technologies

The increased complexity of software requires a scalable mechanism for building systems consisting of multiple interacting, interchangeable and reusable parts. Such parts should be designed in such a way that evolution of particular subsystems does not affect the design and functioning of the whole. These demands are addressed by *software components*.

There are many frameworks, scientific and commercial, that claim to support component model. We prefer to use a somewhat maximal definition, which is driven by physics projects in which we are participating. Software components should:

- interact through well-defined interfaces (encapsulation);
- separate implementation from interfaces;
- be able to be deployed independently;
- explicitly support composability by distinguishing between the interfaces that they provide and use.

Demanding that components should have two kinds of interfaces clearly separates them from objects, since objects also interact through interfaces, can separate interfaces from implementation by using interfaces or abstract classes and inheritance, and support composability by, for example, using pointers to each other. Often a part of an interface that expresses the connectivity of the component is called a port. *Provides-port* is a set of methods that are provided by the component, i.e. can be called by other components who have a matching *Uses-port*, expressing the needs of these components in this functionality.

As an example illustrating ports roles, lets consider an application that advances a variable obeying a non-linear diffusion equation:

$$\frac{\partial\phi(x,t)}{\partial t} = \frac{\partial}{\partial x}\left(D(\varphi,x)\frac{\partial\phi}{\partial x}\right)$$

To solve this equation we introduce three components. The Driver/State component keeps the current value of t and the current value of ϕ as a function of x. The Solver component knows how to advance ϕ from moment t to t+dt given the value of D. The Transport component knows how to calculate D given x and ϕ. Hence the advancement of ϕ can be then presented as composition of these components shown on Figure 3. Driver asks Solver to Advance by dt and passes dt and the state: the value of ϕ. Within the Advance call, Solver invokes Eval Transport port of Transport to evaluate diffusion coefficient D using current ϕ, and then calculates new ϕ. This new value is then returned by the Advance method to Driver.

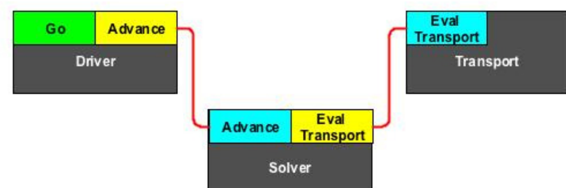

Figure 3: Three components connected to form an executable for solving a diffusion equation. A blue section on the left side of each component represents a Provides-port, and the yellow part on the right represents a Uses-port. The Green Go port is specific for the driver component that starts the execution.

Why one would want use such components rather then implementing a sequence of steps? The advantage transpires only if one would like to use multiple implementations of such components communicating through the same Advance and Eval Transport ports. Designed correctly, these ports could be used as standard connectors between multiple solvers and various transport

models with varying implementations. One then can have multiple combinations of them – each new combination affecting only a subset of components. For example, one can have multiple models for anomalous transport in tokamaks and use solvers with different discretizarion schemes and implicitness.

In addition to the component requirements listed above, it is desirable that components:

- support multiple languages;
- support High Performance Computing (HPC);
- allow remote invocation.

Multilanguage support additionally separates components from object-oriented models and is needed for development of integrated simulations combining modern and legacy components written in different languages. The requirement for remote invocation comes from the need to support distributed computing as many applications need to run on distributed data and separate the compute-intensive "solving" part running remotely from a less demanding data analysis running locally.

Industry has recognized the benefits of the component approach a long time ago. Examples are Java Beans and COM components. But since the needs of commercial applications differ from the needs of scientific computing, Java Beans and COM components do not satisfy all our requirements. For example, neither model supports two kinds of interfaces. Also, Java Beans are language specific, while COM is platform specific and primarily for C/C++. They do support remote invocation, but are seldom used for HPC applications.

Of all the commercial products, CORBA Component Model (CCM) [8] is the closest to support our component definition. In fact, the CCM specification does satisfy all the required and two of the desirable requirements listed above - multilanguage support and remote invocations. But while many CORBA implementations provide support for Java, C++ and Python and CORBA objects (different from components) written in various languages can interoperate, there are too few implementations of CCM, and in reality they support either C++ or Java, thus not allowing components in many languages. Moreover, none of CORBA implementations supports Fortran, which is heavily used in scientific computing. Finally, CORBA is seldom used in HPC applications.

Due to the limitations of commercial components, several scientific components frameworks have been developed that addressed some of the limitations. The domain-specific ESMF [9] and SWMF [10], for example, are high performance frameworks, but do not support multiple languages (only C++ and F90, respectively). Moreover, their interfaces do not discriminate between Provides and Uses types. Finally, they do not support remote invocations.

The most promising and well-developed scientific component framework is the Common Component Architecture (CCA) [11]. CCA component model satisfies all the requirements of our component definition and has strong support from a wide scientific community.

Language Interoperability Tool Babel

An integral part of CCA is Babel [12], a language interoperability tool that is used to define components, generate language mappings for them and provide for remote invocations. Babel uses SIDL (Scientific Interface Definition Language) to define interfaces and relationships between components. The Babel interoperability tool reads the SIDL description and generates the necessary "glue" between different languages. The "glue" mostly consists of client stubs and server skeletons.

For example, here is an example of a SIDL definition of a function taking double as an argument:

```
package newPackage version 1.0{
        class newClass{
                void doWork(inout double varX );
        }
};
```

Imagine that we want to call F90 from C++. Then we use Babel to generate a C++ client stub and a F90 skeleton. The relevant part of the C++ client stub will look like:

```
namespace newPackage{
        class newClass{
                public:
                 void doWork (double& varX);
        };
};
```

The Fortran 90 implementation is done by inserting the logic into the server skeleton:

```
recursive subroutine
 newPackage_newClass_doWork_mi
        (self, varX, exception)
!Implementation goes here
end subroutine newPackage_newClass_doWork_mi
```

Then in the main program one can call the C++ client as follows:

```
main(){
  newPackage::newClass B =
        newPackage::newClass::_create( );
        B.doWork(5.);
}
```

The same SIDL interface defined above can be used for making remote calls between various languages. In this case one needs to start a server process that runs Babel's ORB on a particular port and creates the worker object. The client than needs to specify the host and the port to access the object. In Java, this will look as follows.

```
  public static void main (String args []) {
    Sidl.rmi.ProtocolFactory.addProtocol
      ("simplehandle",
       "simple.rmi.SimHandle");
    newPackage.newClass obj =
      newPackage.newClass.connect
        ("simhandle://hostname:9000/1000");
    obj.doWork (100.0);
  }
```

One can specify different choices of the client and server languages. Currently, the Babel language bindings include Fortran 77, Fortran 90, C, C++, Python, and Java.

Babel is implemented very efficiently. Before we decided to use it in our projects we performed performance comparison between Babel and a standard F2003 mechanism for interoperating with C (ISO_C_BINDING module). From these tests we concluded that using Babel should have a very small overhead for our mixed language applications. The results of one such test is shown on Figure 4.

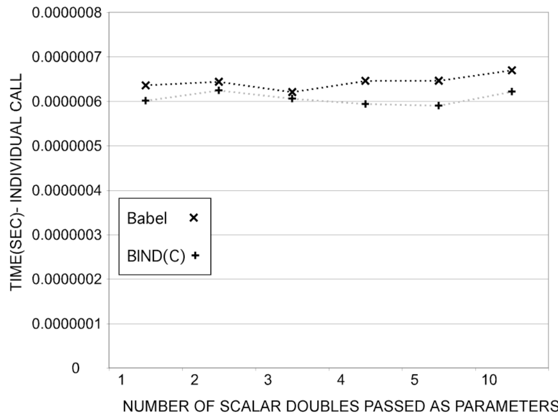

Figure 4: Comparison of performance of Babel and F2003 ISO_BINDING_C module. The graph shows the time for an individual call from C to Fortran versus the number of scalar arguments.

The role of Babel will be growing as many scientific disciplines strive to provide comprehensive modeling of various phenomena and need to mix many codes written in various languages in one application. To do this, scientists first needs to standardize the interfaces of conceptually similar modules, express them using some common language and provide some "glue" allowing their integration. All this can be done by Babel.

Use of Components in Fusion Applications

An example of a discipline that started looking at components in order to unify interfaces and develop complex applications is fusion theory. Its current thrust is to provide integrated modeling of tokamaks. In 2006, two SciDAC projects, SWIM (Center for Simulation of RF Wave Interactions with Magnetohydrodynamics) [13] and CPES [14] (the Center for Plasma Edge Simulation0, were funded. In 2007, another integrating SciDAC project, lead by Tech-X Corporation, FACETS [15] (the Framework Application for Core-Edge Transport Simulations) started.

Since migration to components is not simple and requires a lot of learning and a paradigm shift, all projects at the moment use a much weaker definition of component than the one defined above. For example, SWIM provides Python wrappers with standard interfaces for each code involved in their integration and uses files exchange for modules integration. FACETS project uses standard C++ interfaces to define its modules. It strives for the high performance in the beginning and supports data exchange in memory rather than using files. That is why FACETS uses Babel for calls from its C++ framework to the legacy modules transport, wall and edge modules implemented as Fortran, C, and C++ libraries with the interfaces standardized between types (Figure 5).

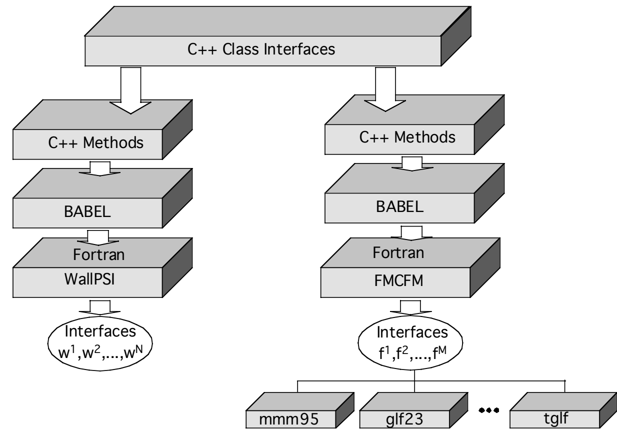

Figure 5: FACETS uses Babel to call legacy modules from the C++ framework.

All three projects might unite in the upcoming Fusion Simulation project some time next year and hopefully will merge with the components technologies as the work progresses and the technologies mature.

REFERENCES

[1] I. Foster, "What is the Grid? A Three Point Checklist," GRIDToday, July 20, 2002.
[2] http://www.globus.org/toolkit/.
[3] http://hdf.ncsa.uiuc.edu/HDF5/.
[4] www.cs.wustl.edu/~schmidt/TAO.html.
[5] www.cs.fsu.edu/~engelen/soap.html.
[6] http://www.es.net.
[7] Svetlana G. Shasharina, Chuang Li, Nanbor Wang, Rooparani Pundaleeka, David Wade-Stein, "Distributed Technologies for Remote Access of HDF Data", to appear in proceedings of the 16th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2007).
[8] http://ditec.um.es/~dsevilla/ccm/.
[9] http://www.esmf.ucar.edu/.
[10] http://csem.engin.umich.edu/swmf/.
[11] http://www.cca-forum.org/.
[12] http://www.llnl.gov/CASC/components/babel.html.
[13] http://cswim.org/.
[14] http://www.cims.nyu.edu/cpes/.
[15] http://www.facetsproject.org/facets.