From Legion to Legion-G to OGSI.NET: Object-based Computing for Grids[1]

Marty Humphrey
Department of Computer Science
University of Virginia
Charlottesville, VA 22911

Abstract: *The object abstraction has long proven to be an effective foundation upon which to structure application codes; however, its application to Grid Computing contains many challenges related to the heterogeneous, dynamic, and cross-administrative-domain nature of Grids. This paper contains an overview of the succession of three projects at the University of Virginia that provide object-based support for Grid computing: Legion, Legion-G, and OGSI.NET. Throughout the three projects, the overall goal has remained to reduce the barrier for entry to Grid applications developers, thereby enabling next-generation Grid applications beyond those that have been provided by today's "heroic programmers". The successes of each project with respect to this overall goal are discussed.*

1 Introduction

Arguably, the most significant challenge facing applications developers for Grid Computing lies in the *programming model*. The Grid by definition attempts to collect disparate resources into a single logical platform by which to facilitate resource sharing and collaborations within and across virtual organizations. Applications executing on the Grid must be able to accommodate heterogeneity, satisfy cross-domain security requirements, and be fault-tolerant, opportunistic, and adaptive; Fundamentally, are new abstractions and infrastructure needed for the applications developers, or are the traditional ones that currently exist for desktop (as well as for high-performance computing) sufficient?

Since 1996 and the days of "metacomputing", we have advocated the object-based abstraction as the core upon which to build a Grid Computing infrastructure. Object-based design has long proven effective as a means by which to structure software according to sound fundamental designs, irrespective of Grids. The challenge, of course, is how to leverage object-based design principles and build the necessary additions to accommodate the challenges of Grids.

In this paper, we describe the progression of object-based support for Grid Computing that has resulted from our research at the University of Virginia. In Section 2, we describe the first project, *Legion* [5,6], which started from scratch to build an object-based Grid infrastructure. As the Legion software stabilized and hardened, and the user base grew, there was a desire to combine the features of Legion with the toolkit approach of Globus [3]; this merging is represented by *Legion-G,* which is roughly an applications-level port of Legion onto the Globus infrastructure. Section 3 details the goals, design, and successes of Legion-G. The goal of the third (and current) project is to apply the lessons learned with Legion and Legion-G to contribute to the emerging community-based *Open Grid Services Architecture (OGSA)* [4]. Section 4 describes the goals, specific problems being address, and the early successes of *OGSI.NET*, which is an attempt to synergize the .NET architecture from Microsoft with the Web-standards-based approach of OGSA. Throughout the three projects, the overall goal has remained to reduce the barrier for entry to Grid applications developers, thereby enabling next-generation Grid applications beyond those that have been provided by today's "heroic programmers". The successes of each project with respect to this overall goal are discussed.

2 Legion

2.1 Motivation

In the mid 1990s, it was becoming increasingly evident that emerging ubiquity in high-speed networks unto itself was not going to provide the increase in productivity for the sciences that relied heavily on computers. Simply, people were still being forced to use *telnet* and *ftp* (and their more secure analogs over *SSL*) to utilize computing resources. This required end-users to manage information that was often idiosyncratic and unpredictably changing: on a per-machine basis, each user had to remember his/her account ID, the version of the operating system on the machine, the versions

[1] This work is supported in part by the National Science Foundation grants EIA-9974968, ANI-0222571, and ACI-0203960; the NPACI Partnership; the NASA Information Power Grid program; and Microsoft Research.

of the compiler and run-time support structure on the machine, the amount of his/her allocations, the policy on the use of scratch space on the machine, etc.

The core design principle of Legion is that, in the face of the onrush of hardware, Grid computing (then known as *metasystems*) should not focus on stretching an existing paradigm, interacting autonomous hosts, into a regime for which it was not designed. Extending the model of autonomous hosts will result is a collection of partial solutions — some quite good in isolation, but lacking coherence and scalability — that make the development of even a single wide-area application demanding at best. Instead, a software infrastructure should hide the underlying physical infrastructure from users and from the vast majority of programmers, and enable inter-operability of components. The software should support construction of larger integrated components using existing components, provide a secure environment for both resource owners and users, and scale to millions of autonomous hosts. A *virtual machine* abstraction could likely provide such a set of abstractions.

2.2 Description

From the high-level user perspective, Legion provides both *capability* and *capacity* computing for high-performance application developers. For example, Legion contains support for *parameter-space studies*, in which multiple independent jobs can be easily scheduled and executed by Legion in parallel. A typical use would be when an engineer is designing a particular product and then wants to perform multiple simulations but may not care where they execute. In this case, Legion could select machines (based on user-defined criteria), perform binary management (copying the executable binary to the selected machines), interact correctly with the selected machines (perhaps via the queuing system such as PBS or LSF installed on the machine), and transparently copy the output files back to the user's desktop machine.

Legion creates the illusion of a single virtual machine that provides secure shared object and shared name spaces. Legion provides Operating System-like abstractions of the underlying hardware and provides the glue to couple diverse applications together. Unlike a conventional operating system that provides an abstraction of a single physical computer, Legion aggregates a large number of diverse computers running different operating systems into a single abstraction, thus vastly simplifying the task of writing applications in heterogeneous distributed systems.

All system and application components in Legion are objects; the default implementation provided by the core developers can be overloaded and replaced by users, which facilitates rapid customization. Legion provides process management, inter-process communication, persistent storage, a file system, and security services. Legion supports PVM, MPI, C, Fortran (including an object-based parallel dialect), a parallel C++, Java, and the CORBA IDL. Legion is layered on top of host operating systems. From its inception Legion was designed to deal with tens of thousands of hosts and millions of objects—a capability lacking in other object-based distributed systems.

At the core, Legion objects are logically address space-disjoint, active entities. Objects represent coarse-grained resources and entities such as users, hosts, storage elements, schedulers, metadata repositories, files, and directories. Each Legion object belongs to a class, and each class is itself a Legion object. The complete set of method signatures exported by an object defines its interface. Much of the Legion object model's power comes from the role of Legion classes; much of what is usually considered system-level responsibility is delegated to user-level class objects. For instance, Legion classes are responsible for creating and locating their instances, and for selecting appropriate security and object placement policies. Legion objects may be active or inactive, and store their internal state on disk (either periodically or during deactivation). Objects may be migrated simply by transferring this internal state to another host. The object's class then spawns a process that is instantiated with the migrated internal state.

Legion objects are identified using a three-level naming hierarchy. At the highest level, objects are identified by user-defined text strings called *context names*. These user-level context names are mapped by a directory service called *Context Space* to system-level, unique, location-independent binary names called *Legion object identifiers* (*LOID*s). For direct object-to-object communication, LOIDs must be bound to low-level addresses that are meaningful within the context of the transport protocol that will be used for message passing. These low-level addresses are called *Object Addresses* and the process by which LOIDs are mapped to Object Addresses is called the Legion *binding process*. An Object Address (OA) represents an arbitrary communication endpoint, such as a TCP socket. Context Space appears to the user as a Global tree-like directory and enables the ability of Legion programmers to refer to objects in a location-independent manner. An important capability for fault tolerance is provided by LOIDs and Class Objects: when a "server" object is

```
Main() {
  int a=10, b=15, x,y,z;
  MyObject A,B;

  x=A.op1(a);
  y=B.op2(b);
  z=A.op3(x,y);
  printf("%d\n", z);
}
```

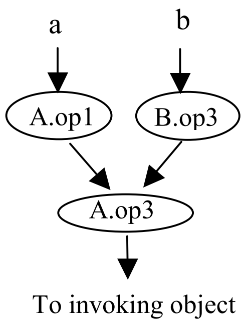

Figure 1: Example Legion Program Graph

migrated, current clients are transparently redirected to the server's new location via a new LOID-to-OA binding provided by the Class Object.

The Legion run-time library (LRTL) is the cornerstone of the Legion Grid infrastructure. Legion object implementations are linked with LRTL, which provides the basic mechanisms to allow Legion objects to communicate with one another using Legion-compliant mechanisms. LRTL is intended to be used both by Legion-targeting compilers and by user-level code; thus, when we refer to LRTL's "users", we mean both compiler writers and applications programmers. In building LRTL, we were driven by several sub-goals and constraints. First, we wanted to abstract much of the complexity that is inherent to heterogeneous distributed computing. For example, we wanted to alleviate the need for LRTL's users to deal directly with the varying data formats on different machine architectures. More importantly, in accordance with the overall Legion philosophy that one size does not fit all, we wanted LRTL to become a useful software tool with which users could build different policies and algorithms along many different dimensions, without having to build an entirely different library. Thus, we built LRTL itself to be extensible and configurable.

Methods on Legion objects are executed using a macro dataflow model. This model requires that any method invocation sent to an object include (in addition to its parameters) a description of where the results produced by the method should be forwarded. For example, instead of being returned to the caller as in an RPC model, the result of a method invocation might be forwarded directly to some other object as a parameter to one of its methods. This dataflow model is called the *Legion program graph*. In a program graph, nodes represent method invocations on Legion objects and arcs represent data dependencies between the method invocations. Figure 1 shows a simple user program and the resultant data dependencies expressed as a program graph. Note that object *A* and object *B* can be executing on different machines of different architectures, in different parts of the world. The implementation of program graphs in Legion enables two important properties of the Legion system, (1) support for concurrency and parallel processing, and (2) support for graphs as first class objects. The latter aspect is important because some applications may require the specification of a computation in one object and the initiation of that computation in a different object.

A more recent development of the Legion project is LegionFS [8], which is a peer-to-peer, Grid-enabled distributed file system based on Legion services. LegionFS is a specialization of the Context Space. Unlike directories, ContextObjects may contain references to arbitrary objects such as hosts. LegionFS provides a flexible framework that can span the range of geographic environments, usage scenarios, and security requirements. The design of LegionFS is based on the premise that the underlying file systems upon which Legion executes are largely competent in the actual storage of data; the challenge is to provide a cross-architecture, cross-organization, and scalable file system that both exploits and expands the mechanisms provided by the heterogeneous, isolated data stores. The key features of LegionFS are:

- Naming: The three-level naming system is used, enabling location-independent references to data objects and large repositories.
- Security: Each component of the file system may exist independently, represented as an object. Each object is its own security domain, controlled by fine-grained Access Control Lists (ACLs). The security mechanisms can be easily configured on a per-client basis to meet the dynamic requirements of the request.
- Scalability: Individual files within a directory sub-tree can be distributed throughout the storage resources in an organization and I/O operations on the files can be conducted in a peer-to-peer manner, which is a natural consequence of the LegionFS object-based system. This holds also for the directory service and eliminates centralized components that can be performance bottlenecks.
- Extensibility: Every object publishes an interface, which may be inherited, extended, and specialized to provide an object supporting additional semantics, alternate policies, or a novel implementation.

- Adaptability: LegionFS maintains a rich set of system-wide metadata that may be used by objects to tailor their behavior to environmental changes.

The core of LegionFS functionality is provided at the user-level by Legion's distributed object-based system. As such, the file and directory abstractions of LegionFS may be accessed independently of any kernel file system implementation through libraries that encapsulate Legion communication primitives. This approach provides flexibility as interfaces are not required to conform to standard UNIX system calls. To support existing applications, a modified user-level NFS daemon, *lnfsd*, has been implemented to interpose an NFS kernel client and the objects constituting LegionFS. This implementation provides legacy applications with seamless access to LegionFS.

2.3 Assessment

There are number of ways in which to gauge Legion's impact in the community, broadly covering four areas. First, Legion has been deployed in a number of early Grid prototypes, most notably across the NASA information Power Grid, across the DoD Major Shared Resource Centers (MSRCs), and across NPACI resources. This deployment has been crucial in establishing a concrete dialogue of expectations and requirements between the computer science community and the end-users. Second, the origins of the Global Grid Forum (GGF) lie in the desire to standardize Grid computing "best practices", protocols, and interfaces. Without the experiences of Legion users and Globus users largely motivating the creation of this organization, arguably, Grids would not be experiencing the successes they are today. Third, a large number of research papers utilizing the Legion infrastructure have been written and appear in the top HPC workshops, conferences, and journals (see http://legion.virginia.edu for a subset of these papers). Fourth, the Legion technology has been successfully transferred to private industry; the Boston-based Avaki bases its commercial grid offering on the Legion technology (although it has changed significantly from Legion's roots).

Perhaps the biggest challenge associated with Legion is its complexity. That is, the Legion goals were ambitious, requiring a sophisticated software solution. As a result, it took a relatively long amount of time to get the first version of Legion to the general public. At times, it also required a substantial knowledge to administer a Legion network. However, we believe the broader impacts of Legion outweigh these difficulties.

3 Legion-G

3.1 Motivation

During the time that Legion was being hardened and being deployed at a number of sites, the Globus approach to Grid computing was also experiencing tremendous visibility and success. But deployers, Grid programmers, and end-users were finding that Legion and Globus were each uniquely geared for different problems. The combination of Legion and Globus would represent a truly state-of-the-art Grid computing paradigm. The Legion user would benefit from a higher-performance cross-machine MPI implementation available from the Globus toolkit (MPICH/G2), be able to utilize the years of development of the GRAM protocols, and in general gain all the benefits of an arguably more standards-based approach (such as the Globus security model, Grid Security Infrastructure, or GSI). The Globus user would gain access to LegionFS, be able to utilize Legion's graphical tools for parameter-space studies, use the Legion general-purpose scheduler, and in general have access to a Grid-enabled object model. Both Legion users and Globus users would benefit from the immediate redundancy of certain Grid services (such as the information services). The Legion and Globus development teams would benefit from a closer working relationship, for example to more quickly foster the identification of Grid computing requirements (and subsequent, shared solutions). Clearly, the motivation existed to attempt to combine the Legion approach and the Globus approach into a unified Grid infrastructure solution.

3.2 Description

Legion-G is, in essence, Legion running on top of Globus. The design of Legion-G is shown in Figure 2, which shows that the Legion-G object model ultimately relies on the four key protocols or components of the Globus Toolkit: GSI, GRAM, GRIP, and GridFTP. Figure 2 shows the integration with other key Grid technologies such as the Network Weather Service (NWS) and the Storage Resource Broker (SRB). Also shown is the support for a particular user application, CHARMM [2].

Our initial work focused on the integration of LegionFS with the Globus toolkit. A key value added in our desire to utilize LegionFS was to capitalize on the flexibilities provided by location-independent naming. Globus is a powerful toolkit but lacks location transparency in its naming system, due to a reliance on URLs. In practical terms, this means that a Grid user (or software running on behalf of the user) must know precisely *where* Grid entities are.

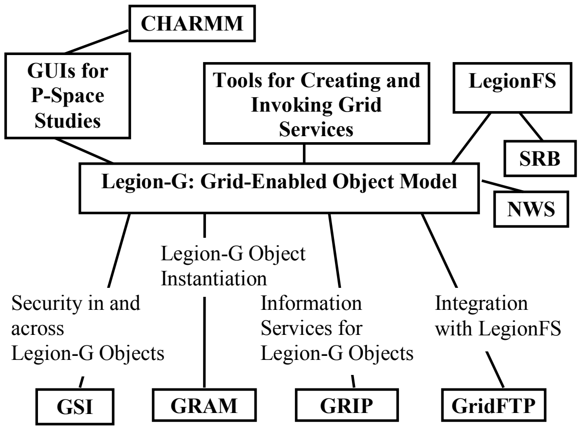
CHARMM
GUIs for P-Space Studies
Tools for Creating and Invoking Grid Services
LegionFS
SRB
Legion-G: Grid-Enabled Object Model
NWS
Legion-G Object Instantiation
Security in and across Legion-G Objects
Information Services for Legion-G Objects
Integration with LegionFS
GSI
GRAM
GRIP
GridFTP

Figure 2: Legion-G Architecture

For example, if a particular user's computation reads input from some file, that user is generally required to know not only the name of the file but the location as well. This requirement extends to all "Grid services" that may be engaged. The problem with URLs is that hardware reconfiguration, file system reorganization, and changes in organizational structure can often result in dangling links. All users of the World Wide Web have inevitably encountered this problem, resulting in a cryptic "document not found" error message that often leads to increased frustration. To a certain extent, organizationally, transparency can be achieved via virtual frontends, of which requests are then dynamically farmed out to back end servers. This is often the case for large corporations. However, this does not work if the entity moves across organizational boundaries, as can occur in Grid computing.

In December, 2001, we demonstrated the ability of an MPI application running across the NASA Information Power Grid to access information contained in LegionFS. We modified a simple MPI code to use a Legion-G BasicFileObject (a component of LegionFS) as input. The simplified MPI code is shown in Figure 3. The code on the left represents a style of computation in which the input file is staged to the computing platform before the process is spawned. This is appropriate for small input files, and is supported by Legion and Globus. This style can be inefficient when files are large and/or only a portion of the file is to be read. In the right of Figure 3, the code has been modified to read its input from LegionFS. The boldface indicates changes to lines that exist in the original version, while the boldface-italicized represent lines that had to be added to the source code. Note that it is not strictly necessary to change the source code in order to access LegionFS—the use of *lnsfd* obviates the need to recompile. Modified source code is shown in this example for clarity of presentation.

```
main () {                            main () {
  MPI_Init();                          MPI_Init();
  MPI_Comm_rank(…);                    MPI_Comm_rank(…);
  MPI_Comm_init (…);                   MPI_Comm_init (…);
                                       BasicFiles_init();

  fp = fopen("input.dat",              fp = BasicFiles_open(
             "r");                              "input.dat", "r");
  read(fp, buf, 100);                  BasicFiles_read(fp, buf,10);
  /* compute */                        /* compute */

  MPI_Send(…);                         MPI_Send(…);
  MPI_Recv(…);                         MPI_Recv(…);

 /*                                   /*
  * read more data                     * read more data
  * from file, compute                 * from file, compute
  */                                   */
 fclose(fp);                          BasicFiles_close(fp);
                                      BasicFiles_done();
 MPI_Finalize(…);                     MPI_Finalize(…);
}                                    }
```

original *Legion-G version*

Figure 3: MPI Code Modified to Read Input from LegionFS

The following describes the nature of the experiment in more detail. A small Legion-G network was booted across two linux machines at the University of Virginia, *deneb-uva* and *algol-uva*. A BasicFileObject was instantiated on *algol-uva,* and given the context name *input.dat*. (The code shown in the right of Figure 3 works because in Legion there is a "current working context" similar to the UNIX "current working directory", facilitating the use of relative context names.) The code in the right of Figure 3 was compiled and executed using an unmodified Globus installation on two NASA IPG SGI machines at NASA Langley, *whitcomb-nasa* and *rogallo-nasa*. To allow the MPI jobs to access LegionFS, it was necessary to link against certain Legion libraries. A two-node MPI job was executed, with one of the MPI instances on *rogallo-nasa* and one on *whitcomb-nasa.* Each of the computations were designed to first read some data from the *input.dat* Legion-G object, perform some computation, exchange MPI messages, read more data from *input.dat,* perform more simple calculations, and then terminate. In the baseline case, the invocation of *BasicFiles_open* causes the operations of Figure 3 to take place. That is, the Legion Run-Time Library (LRTL) engages the appropriate Legion-G objects to determine the object address of the Legion-G BasicFileObject that encapsulates *input.dat*. The invocation of *BasicFiles_read* uses this binding and retrieves the data. Each MPI instance reads data from this Legion-G object, computes, and then retrieves more information from this Legion-G BasicFileObject. The MPI program then terminates successfully.

The baseline experiment illustrates the flexibility that LegionFS offers computations through its naming system and naming resolution infrastructure. In this case, the MPI program running across *rogallo-nasa* and *whitcomb-nasa* never realized that it was pulling its input from a non-local machine. Two other experiments were executed to test the functionality of the binding and re-binding mechanisms. In the first experiment, the BasicFileObject was migrated from *algol-uva* to *deneb-uva* after the first file access. It is possible to migrate a BasicFileObject at any point, perhaps because the resource on which it currently resides has suddenly become overloaded. Other reasons include moving closer to clients or moving to a more physically secure server. This experiment also completed successfully, with a minor delay due to the rebinding process. That is, after the first file access, both MPI computations (in their LRTLs) believe that, upon invoking the second file access, that the BasicFileObject is on *algol-uva*. A message is sent to the old address; after a timeout occurs (either there is nothing at that port on *algol-uva* or there is a new process that does not understand the message from the MPI client), the LRTL asks the BasicFileClass for the new location of the *input.dat* BasicFileObject. The BasicFileClass has this new address, because the BasicFileObject reported in upon (re-starting) on *deneb-uva*. Upon receiving this new binding, the MPI computations re-try their requests to read the data, this time successfully ending to a particular port on *deneb-uva*. In the second experiment, *input.dat* is first configured as a redundant server (the *"Simple K-Copy Class"* functionality of Legion), with the primary copy on *deneb-uva* and the secondary copy on *algol-uva*. After the first access, we simulated the loss of a machine or of network connectivity but manually killing the process on *algol-uva* that was the encapsulation of *input.dat*. Again, upon the second access by each of the MPI jobs, the message fails to the server process on *algol-uva,* which causes the rebinding process. The MPI computations again experience a short delay but are ultimately able to complete their functionality.

3.3 Assessment

The integration of Legion and Globus (through focusing on LegionFS) in December 2001 represented a major success of Legion-G and, we believe, was strongly indicative of the successes to come. However, as we were just beginning to show the value of the object model itself to the Globus community, the Open Grid Services Architecture (OGSA) was created [2]**.** OGSA was a major break-through and quickly became the focus of the Grid community. After a careful and detailed analysis, it was determined that the planned "added value" of Legion-G could be attained through an implementation based on OGSA, so the resources of the Legion-G development were re-directed at an OGSA-compliant effort. In other words, the goals of Legion-G remain, but, instead of addressing these goals through a "Legion-port-onto-Globus", we were instead compelled to create solutions for these goals/problems in an OGSA-compliant software setting, which Legion-G unto itself would not have achieved. Making Legion-G itself OGSA-compliant was deemed too difficult.

4 OGSI.NET

4.1 Motivation

By many accounts, the Open Grid Services Architecture (OGSA), and more properly the Open Grid Services Infrastructure (OGSI), endorses/conveys an object-based Grid computing infrastructure, albeit in the terminology of Web Services (i.e., portTypes and "Grid Services"). However, there were two problems. First, the development model of OGSI itself was largely influenced by Java/AXIS/Tomcat, and it was not clear if the underlying heterogeneity of the Grid would be accommodated/embraced through the development work centered at ANL. Second, and more importantly, we believed that OGSA/OGSI represents a great step forward for the community, but the very difficult problems of scheduling, debugging, scalability, manageability and especially *programming* grid applications remain unsolved. We had made progress with Legion and then Legion-G, but there were many challenging issues remaining.

4.2 Description

OGSI.NET is an attempt to utilize the .NET Framework (and its support for Web Services) to provide Grid-specific tools and programming models. The overall goal is to integrate *e-science* with *e-business* by merging OGSA with the Global XML Web Services Architecture (GXA) [1]. A key difference between OGSA and GXA is that OGSA is *service-centric,* and GXA is, instead, focused on the protocols *on the wire* that are necessary to facilitate a global computing infrastructure. A careful analysis shows that the two approaches are not necessarily consistent.

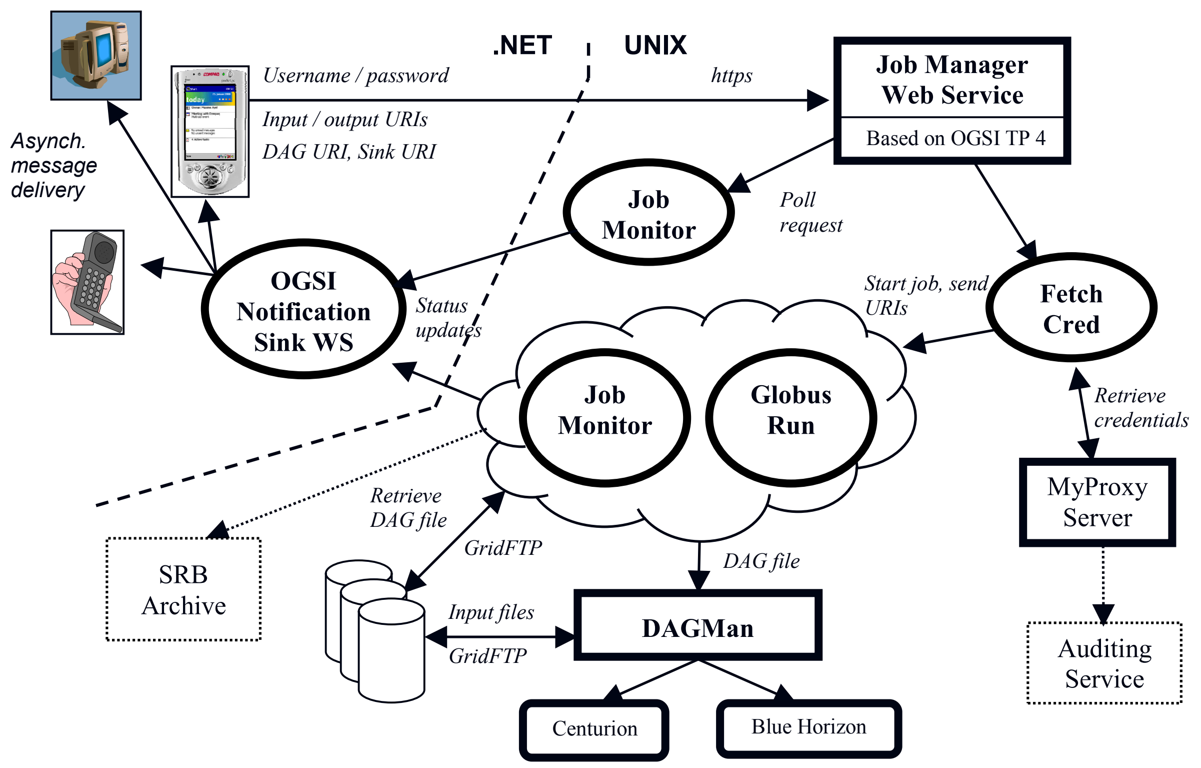

Figure 4: Executing CHARMM via OGSI.NET

The baseline capabilities of .NET, independent of OGSA, are impressive. .NET has multi-language support (e.g., C#, C++, COBOL, Fortran, Python, Scheme, Visual Basic, etc.), via mappings to the Common Language Runtime (CLR). The CLR also contains sophisticated support for security enforcement, memory, process, and thread management, life-cycle management, strong type naming, cross-language exception handling, and dynamic binding. The Microsoft ADO.NET data classes support persistent data management and include SQL classes for manipulating persistent data stores through a standard SQL interface. The Microsoft ASP.NET classes support the development of Web-based applications and XML Web services. The Windows Forms classes support the development of Windows-based smart client applications. The plan is to provide client-side support for GSI in .NET, integrate OGSI security with .NET security, capitalize on the Visual Studio IDE in creating Grid applications, provide server-side support for GSI in .NET, and integrate with the .NET Compact Framework.

We have recently prototyped this work and used it as the basis of a demonstration in the NPACI booth at *Supercomputing 2002* in Baltimore. As shown in Figure 4, this represents an integration of many of the emerging approaches for Grid computing. Because domain scientists want easy access to high-end computing resources, we have constructed the necessary infrastructure by which a biologist can submit a CHARMM execution on NPACI resources via a PocketPC (shown at the left of the diagram). The user specifies the input URI, output URI, username, password, and "Sink URI". (The "Sink URI" is a Web Service that receives notification events regarding the status of the CHARMM job.) Upon hitting "SUBMIT", the Web Service on the upper right of the screen fetches a credential from the MyProxy server; this credential will ultimately be used to authenticate to the back-end computing resource (Centurion or Blue Horizon, although only Centurion is enabled at this time). After retrieving the credential (based on the username and password), the Web Service spawns a GlobusRun, which is actually a precedence-related pair of CHARMM tasks. These tasks are executed via DAGMan, which is a Condor-G-enabled mechanism. In the demo in the NPACI booth, we had one person submit the CHARMM job from his/her PocketPC and had multiple people subscribe to the Notification Sink. That is, one person submitted, and multiple people received asynchronous notifications on their PocketPCs regarding the status of the job ("job submitted", "job in queue", "job transitioned to running", "phase 1 complete", "phase 2 complete", etc.) The nature of the PocketPC is that the person could be mobile or in some meeting away from their desk and still receive important status updates. This is a new way to collaborate, as the scientists need not be co-located and instead could be receiving real-time notifications around the country or around the world. If the user does not have a PocketPC handy (say that she is traveling home from work), then the job can even be configured to send a text message to their phone when the job is done!

Note that we recognize that some scientists prefer the "traditional" look-and-feel of the Web Browser instead of the PocketPC platform. While the demos were meant to illustrate the secure "Grid plumbing" and experimental use of the PocketPC, we are currently working with researchers at The Scripps Research Institute (TSRI) to develop the appropriate look-and-feel (rendered through Internet Explorer) that CHARMM users want to see from their desktop. This front end will utilize the entire Grid plumbing (the right side of the slide) we have constructed for the NPACI booth demos. Of course, the user can submit via the Browser, and then receive notifications on their PocketPC, thus combining the best of stationary and mobile computing.

The next steps, specifically with regard to the software of Figure 4, are to create the Web-based front-end submission, integrate with more NPACI back-end resources, and address some of the security issues still unresolved (most notably, we are crossing between traditional Grid techniques and more cutting-edge Web Services techniques; there is not currently a mechanism by which authentication and authorization credentials easily translate between the two worlds.)

4.3 Assessment

The potential impact of OGSI.NET is substantial. We have shown a prototype that works, and we have even more recently demonstrated the .NET hosting environment at GlobusWorld 2003 (January, San Diego). Once the hosting environment stabilizes, we intend to focus on two other projects: [2] *Continuous Scheduling in OGSI.NET*, by which scheduling decisions for components on Grid computations are repeatedly and opportunistically rescheduled; and [3] the *Grid Debugging Visualizer (GDV),* by which system administrators and users can more easily determine the errors (and performance characteristics) of Grid computations. OGSI.NET is the groundbreaking new approach for Grid computing and provides the foundation on which to support Continuous Scheduling and GDV. The research leverages and greatly extends our experiences and philosophy of Legion and Legion-G.

5 Conclusions

Throughout our Grid research, we have attempted to provide the Grid system administrator, the Grid programmer, and the Grid end-user with highly-usable, efficient abstractions given the complexities of the operating environment. First with Legion and then with Legion-G, we believe that the object foundation has proven extremely effective for Grids. With OGSI.NET, we believe that we will continue to provide the infrastructure necessary for next-generation, complex software systems.

Acknowledgments

This work would not be possible without the support and collaboration of Andrew Grimshaw, who created Legion and has provided many insights that we have utilized in Legion-G and OGSI.NET. Glenn Wasson, Norm Beekwilder, and Shaun Arnold have been instrumental in the development of many of these ideas described in this paper, particularly on OGSI.NET.

References

[1] Box, Don. Understanding GXA. Microsoft Corporation, July 2002. Available for download at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dngxa/html/gloxmlws500.asp

[2] Brooks, B. R., Bruccoleri, R. E., Olafson, B. D., States, D. J., Swaminathan, S., Karplus, M., *CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations*, Journal of Computational Chemistry, Vol. 4, 1983.

[3] Foster, I. and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit", *International Journal of Supercomputing Applications*, 1997.

[4] Foster, I., Kesselman, C., Nick, J., Tuecke, S. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration.* Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[5] Grimshaw, A. S, A. Ferrari, F. Knabe, and M. Humphrey. Wide-Area Computing: Resource Sharing on a Large Scale. *Computer*, 32(5):29-37, May 1999.

[6] Grimshaw, A.S., W. Wulf, and the Legion team. "The Legion Vision of a Worldwide Virtual Computer", *Communications of the ACM*, 40:1, pp. 39-45, January 1997.

[7] National Partnership for Advanced Computational Infrastructure, http://www.npaci.edu

[8] White, B., M. Walker, M. Humphrey, and A. Grimshaw. LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications. In *Proceedings of Supercomputing 2001*, Denver, CO, November 2001. *Best Student Paper Finalist.*