Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications

Dennis Gannon, Randall Bramley, Geoffrey Fox, Shava Smallen, Al Rossi, Rachana Ananthakrishnan, Felipe Bertrand, Ken Chiu, Matt Farrellee, Madhu Govindaraju, Sriram Krishnan, Lavanya Ramakrishnan, Yogesh Simmhan, Alek Slominski, Yu Ma, Caroline Olariu, Nicolas Rey-Cenvaz

Department of Computer Science, Indiana University

Abstract

Computational Grids [Grid, Grid1] have become an important asset in large-scale scientific and engineering research. By providing a set of services that allow a widely distributed collection of resources to be tied together into a relatively seamless computing framework, teams of researchers can collaborate to solve problems that they could not have attempted before. Unfortunately the task of building Grid applications remains extremely difficult because there are few tools available to support developers. To build reliable and re-usable Grid applications, programmers must be equipped with a programming framework that hides the details of most Grid services and allows the developer a consistent, non-complex model in which applications can be composed from well tested, reliable sub-units. This paper describes experiences with using a software component framework for building Grid applications. The framework, which is based on the DOE Common Component Architecture (CCA) [CCA, CCAT, CCA1,CCA2], allows individual components to export function/service interfaces that can be remotely invoked by other components. The framework also provides a simple messaging/event system for asynchronous notification between application components. The paper also describes how the emerging Web-Services [WSDL] model fits with a component-oriented application design philosophy. To illustrate the connection between web services and Grid application programming we describe a simple design pattern for application factory services which can be used to simplify the task of building reliable Grid programs. Finally we address several issues of Grid programming that better understood from the perspective of Peer-to-Peer (P2P) systems. In particular we describe how models for collaboration and resource sharing fit well with many grid application scenarios.

1. Introduction - Computational Grids

A computational Grid consists of a set of resources, such as computers, networks, on-line instruments, data servers or sensors that are tied together by a set of common services

which allow the users of the resources to view the collection as a seamless computing/information environment. The standard Grid services include

- security services which support user authentication, authorization and privacy
- information services, which allow users to see what resources (machines, software, other services) are available for use,
- job submission services, which allow a user to submit a job to any compute resource that the user is authorized to use,
- co-scheduling services, which allow multiple resources to be scheduled concurrently,
- user support services, which provide users access to "trouble ticket" systems that span the resources of an entire grid.

Many more Grid services are described in [ipg]. Some Grid middleware systems [legion, Grid2] also support global namespaces for files and other objects. Others [globus] provide access to metadata catalogs and storage resource brokers [srb]. Large Grid efforts like Griphyn [griphyn] and the European Data Grid and the Particle Physics Data Grid [edg,ppdg] seek to build wide area data management systems that allow file caching across the network to provide higher throughput access to many users.

Unfortunately, building applications that run reliably and efficiently on these "Grid Platforms" is often a difficult task. The reasons for this are numerous. Often these applications consist of a heterogeneous collection of sub-applications that are stitched together to form one large distributed application. The difficulty lies in making all the pieces work together in a consistent and predictable manner. In many cases, the reliability of these assemblages is the source of the problem. For example if one of the sub-computations is itself a parallel program running on parallel computing platform and scheduled by a conventional batch system, how should it synchronize and communicate with the other parts of the application? Often the problems encountered involve complex interactions between various grid services, schedulers, security systems and network requirements. In many cases the source of difficulty lies in application designs that are overly complex and not well supported by any Grid programming tools. Our experience in building complex Grid applications has led to three conclusions. First, it is important to distinguish between Grid developers and users and to provide the appropriate tools to each group. Second, in the case of Grid developers, a software component model that incorporates a wide-area, publish-subscribe messaging system, can provide a powerful mechanism for building Grid applications. With these tools, complex distributed applications can have their workflow scripted and legacy application component can be wrapped and controlled. Third, our experiences also have revealed weaknesses in current Grid programming methodology, which we believe can be addressed by adapting Web services and P2P concepts to the Grid.

1.1 Science Grid Portals

We argue that there are three types of Grid application developers and users. The most numerous group are end users who program pre-packaged grid applications by using a

simple graphical or Web interface to supply application specific parameters and simple execution configuration data. In an ideal world this group of people need know little about actual Grid protocols or services. The second group of Grid programmers are those that know how to build a Grid application by composing them from existing application "components" and Grid services. The third group consists of the researchers that build the individual components of a distributed application, such as simulation programs or data analysis modules that make up the basic sub-computations of a wide-area Grid application. Often, this group has little or no experience with building distributed applications. While some users have skills that span all three categories, Grid application development systems should allow programmers to work in one without becoming expert in the others. It is our experience that the first group of users is best served by "Grid Portals", which are web servers that allow the user to configure or run a class of applications. The server is then given the task of authenticating the user with the Grid and invoking the needed grid services required to launch the user's application. Grid portals in current use include the XCAT Science Portal [XCAT], Gateway [webflow,gateway, foxport], Mississippi Computational Web Portal [miss], Discover [Rutgers], NPACI Grid Port [hotpage], Nimrod-G [nimrod], NASA IPG Launchpad [ipglp], Cactus [cactus] and many others. The basic architecture of a Grid Application Portal is illustrated in Figure 1 below. While no two portal designs are the same, they all share characteristics of this model.

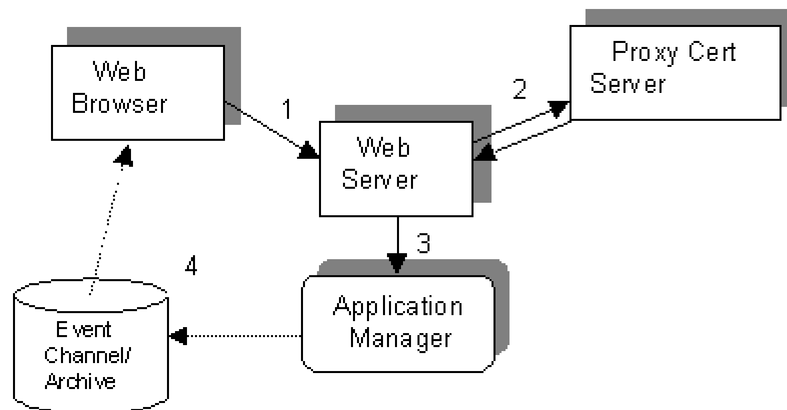

Figure 1. Grid Portal Architecture. 1. The User makes a secure connection from the web browser to the portal server. 2. The portal server then obtains a certificate from a proxy certificate server and uses that to authenticate the user with the Grid. 3. When the user completes defining the parameters of the computation the portal web server launches an ***application manager***, which is a process that controls and monitors the actual execution of the grid computation. The web server delegates the users proxy credential to the application manager, so that the application manger may act on the users behalf. 4. In some systems, the application manager publishes an event/message stream to a persistent event channel-archive. This event stream describes the state of the Grid application execution and can be monitored by the user through the browser.

In some cases the application manager is just the remote grid application itself. However, in many other cases it is a "wrapper" around a legacy application. In the case of applications that involve concurrent computations on multiple resources, the application manager is an agent that is responsible for launching and coordinating these remote computations. Often the application manager is simply a script, which drives the workflow of a complex set of tasks that the Grid application must accomplish.

The design of the application manager script/workflow for a particular Grid application and its web interface is the task of our second level of Grid programmers. It is this level of Grid programming that this paper primarily addresses.

In the paragraphs that follow we shall describe how a software component architecture can be used in the task of building distributed Grid applications. We will then argue that this is just a special case of a more general web services framework for Grid systems. In the last section of this paper we will discuss the ways in which Peer to Peer concepts can provide additional, important support to building Grid applications for groups of collaborating users.

2. Software Component Models

Software component technology is not new. It is now a standard part of many software design practices. Microsoft COM and much of .NET [net] is based on component concepts. Enterprise Java Beans [ejb] is another very important technology for building the business end of large-scale, e-commerce applications.

The basic concepts behind a software component architecture are not difficult to understand. A software component model is a system for assembling applications from smaller units called components. The system defines a set of rules that specify the precise execution environment provided to each component and the rules of behavior and special design features that components must have in order to be considered true "components". A component is then nothing more than an object (or collection of objects) that obey the rules of the component architecture. A component framework is the software environment that provides the mechanisms to instantiate components, compose them, and use them to build applications. The execution environment the component architecture provides a component instance is often called the component's "container."

2.1 CCA Concepts

The DOE Common Component Architecture (CCA) [CCA, CCA1, CCA2, CCAT] is one such component architecture designed for use in large scale scientific and engineering applications. It is the work of two US universities (Indiana and Utah) and five US national laboratories (Sandia, Livermore, Argonne, Oak Ridge, Los Alamos). The basic ideas are drawn from early versions of the OMG Common Component Model (CCM) [ccm]. CCA components are characterized by their external interfaces, called ***ports***, which each take one of two different forms.

- ***Provides Ports*** are component access points that provide an interface of functions that the component will evaluate on behalf of its client. A provides port can be thought of as a "service" provided by the component. A component may have zero or more provides ports.
- ***Uses Ports*** are component features that represent a reference to an external object from within the component. It can be thought of as a call-site within a component where it may *use* a service *provided* by some other component. A component may have zero or more uses ports.

The key architectural idea behind CCA is that if component A has a "uses" port of interface type T that means it may require a service of type T provided by another component B. More specifically, at some point in the execution of component A, it will invoke a method in the interface T that is supplied by some provider. CCA provides a mechanism to connect uses ports of one component to provides ports of another as shown in Figure 2.

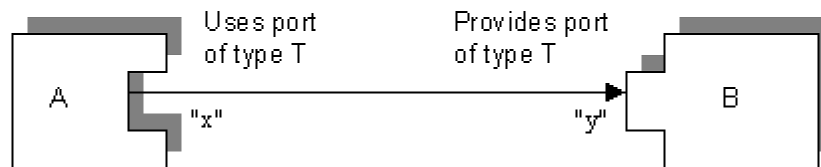

Figure 2. Connecting a uses port named "x" of type T of component A to a provides port named "y" of type T on component B means that the uses port object identified by name "x" has become a remote reference in A for invoking methods on interface "y" of B.

The connection between components can be made at runtime and the components need not reside on the same host. Ports themselves are dynamic entities: A component can create a new provides or uses port at any time or it can remove an existing port.

In the XCAT implementation [xcat] of CCA, components are described by an XML document which contains information about where a component is installed, how an instance can be created, the port names, and links to another XML file that defines their type. In XCAT components can be instantiated and their ports connected together from a Java, C++ or Jython (Python in Java) [python] program. For example, in Jython, given the XML document describing a component and the name of a host on which it has been deployed, it can be instantiated as follows

```
instanceA = cca.createComponent("XML_description_of_component")
cca.setMachineName(instanceA, "modi4.ncsa.uiuc.edu")
cca.setCreationMechanism(instanceA, "gram")
cca.ccaCreateInstance(instanceA)
```

In this example, it is assumed that the XML description file provides the instructions for how an instance of the component can be created on the host "modi4" using the Globus "gram" protocol. When the createInstance method completes the instanceA object is a proxy for the remotely executing instance of the component. Given two remote instances of components we can connect a uses port on one to an identically typed provides port on the other with the call

```
cca.connectPort(instanceA, "usesportname", instanceB, "provportname")
```

In XCAT the communication protocol used to implement the remote procedure call between a uses port method invocation and the connected provides port remote objects currently is based on XSOAP [xsoap], a Java and C++ implementation of the simple XML over HTTP protocol SOAP [soap] (see also [globwrap].

To illustrate how XCAT is used in Grid applications, we describe the "Application Manager" (AM) component which launches and manages a single application code, or scripts the workflow of a set of applications. The AM is just a componentized version of the Jython script interpreter running inside a simple CCA component. The way this component is used is very simple. A Jython script is written to manage the workflow of the Grid application. The portal web server launches or authenticates itself with a running instance of an AM and then downloads the application script into it. If this application manager script needs to be connected to another component the appropriate port class is loaded into the AM. Once this is done, the AM can be started and stopped by invoking methods on its control port. The loaded script is then capable of staging files, running applications or communicating with other components as needed. AM's allow legacy or non-source code applications to be used within a distributed CCA framework, by separating the necessary Grid communications and framework-specified behavior from the application [xcat]

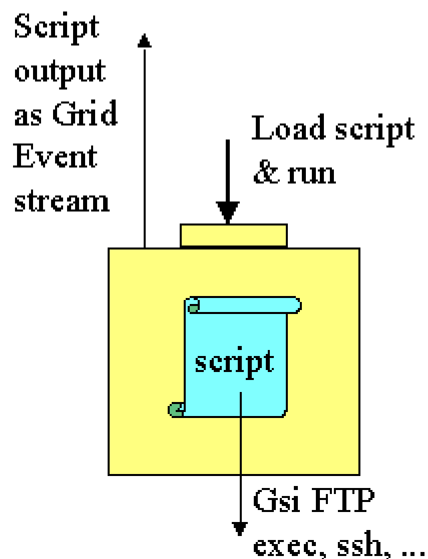

Figure 3. Scriptable Application Manager. (Control Port not shown).

The component has two provides ports.

- **ScriptProvidesPort** of type ScriptPort which has five methods

  ```
  void setParam(Object[] parms);
  void int runScript(String script);
  void killScript(int id);
  void addProvidesPort(String name, String interfaceName, String
  className);
  void addUsesPort(String name, String interfaceName);
  ```

  This port is used to load a Jython script into the component, dynamically add uses and provide ports, and then run the script.
- **ControlProvidesPort** of type ControlPort which has two methods

  ```
  int start();
  int kill();
  ```

  which is used to start and stop the Jython interpreter. Kill causes the component process to terminate.

The application manager is, in effect, a remote Python shell encapsulated as a CCA component. As shown above, the executing Python script is capable of instantiating other components via the CCA creation/connection API and then connecting itself to these new instances. The ability to dynamically add provides ports allows the AM component to expose any arbitrary interface to the user. For example, if the Grid application has a real-time application steering interface this may be exposed as a provides port, which can be invoked through a desktop component connected to the remote AM.

2.2 Events and Messaging

It is often the case that component port connections are not the most convenient or robust model for communicating information between the parts of a Grid application. In many cases Grid applications require an asynchronous messaging system for reporting on application events such as message to say that a file has been written or an error condition has been noted, or a subtask of a large application has completed. CCA ports are synchronous and designed for direct transmission of data or control signals between components. To solve this problem we need some form of messaging/event system and there are many to choose from. For our purposes we have built a simple XML event system using XSOAP.

An event is an XML object that has the following required fields: a namespace, a type name, a timestamp and a source name. Additional fields may be added as extensions to this base type. For example, the following event has one additional field, a string that contains a message.

```
<event xsi:type='ns1:common.test' xmlns:ns1=
      'http://www.extreme.indiana.edu/soap/events/v11/'>
  <eventNamespace xsi:type='xsd:string'>
        http://www.extreme.indiana.edu/soap/events/v11/
  </eventNamespace>
  <eventType xsi:type='xsd:string'> common.test </eventType>
  <source xsi:type='xsd:string'> test-event-source </source>
  <timestamp xsi:type='xsd:timeInstant'> 2001-11-26T23:04:51.411Z
</timestamp>
  <message xsi:type='xsd:string'> Application buffer overflow
</message>
</event>
```

An event namespace plus the event type uniquely identifies the "class" of the event. These class names follow a hierarchical scheme with the event namespace giving the base domain defining that eventand a 'dot' naming convention for the event type to represent subclasses. E.g: ResourceData may be the base event with sub-classes ResourceData.machine, ResourceData.machine.utilization. This enables effective querying/filtering based on the event type. Filtering on ResourceData returns all ResourceData and sub-class events while filtering on ResourceData.machine.utilization returns only the machine utilization events. The timestamp is the time at which the event was created by the publisher of the event, represented using the ISO-8601 standard.

The event system follows a simple publish-subscribe model, which is very similar to CORBA events and notification and to systems like Java Message Service. It is based on several very simple APIs. An event subscriber can either be a push or pull subscriber. A push subscriber will implement and interface with a single method

```
void handleEvent(Event ev);
```

which will be invoked by publishers when they wish to deliver a message. Though the message is an XML string, XSOAP will automatically convert it to a Java or C++ class object.

A pull subscriber invokes a pull method on a pull publishers. This method takes the form

```
Event[] pullEvent(Filter f)
```

This method will return any buffered and previously undelivered events to the calling subscriber.

An event publisher can either be a push publisher, which means it delivers methods to subscribers by invoking the handleEvent(…) method or a pull publisher, which means it will keep an archive of events to deliver to pull subscribers.

A Generic Event Channel

In a simple implementation, we can have a simple listener listening to a particular publisher. But to be more effective, we need a mechanism to store the events being

published and allow filtering and query of the events. This is done by using an event channel. Simply put, it acts as a listener accepting events from publishers and is also a publisher providing interfaces to pull events based on some filter or query events from the past. Realtime events can be filtered and forwarded to the listeners while querying for past events requires a persistent store. Components using events to communicate need not know each other's location, and instead just need the location of an event channel. This also allows components to lose contact or be migrated during a long-running application, and still continue to function and reconnect later. This additional robustness is crucial for Grid-distributed applications which use resources and components that are potentially unreliable.

We also have the concept of a generic channel i.e. the channel need not be aware of the specific subtype of the event that it receives. It just extracts the common information present in all events and uses it for filtering or indexing while storing the event in a persistent store. The original event in raw XML form is sent when a request is received from a listener. Our event channel is implemented on top of a relational database that stores event streams for pull subscribers.

A particular event can be identified using the event's namespace and type and the timestamp at which the event was received by the event channel. By storing the timestamp that the event was received with the event helps avoid problems due to clock skew between the listeners and publishers. Depending on the event type, this may be the machine name, application name or any such string that provides information about the source.

3. Application Examples

Three examples illustrate how these ideas are used in practice. The first is a common case: a small number of stand-alone applications that run on remote sites need to be linked together to form a larger, "multi-disciplinary" application. The second case illustrates another class of applications where a single problem can be divided into a large number of smaller pieces and the solution to one of the sub-computations is selected as the overall or best solution. This is similar to some "parameter space" studies. The third example is the most complex, where the end user dynamically creates a large network of different components to analyze and solve a problem in different ways.

3.1 Wrapping and Coupling Applications: Chemical Engineering

The work done with the Chemical Engineering team from NCSA is an example of the kind of science problems the portal is intended to solve. The simulation models copper electrodeposition in a submicron-sized trench which forms the interconnection on microprocessor chips. The simulation consists of two linked codes. One consists of a continuum model of the convection-diffusion processes in the deposition bath adjacent to the trench. The second consists of a Monte Carlo model of events that occur in the near-surface region where solution additives influence the evolution of deposit shape and roughness during filling of the trench. The codes communicate by sharing data files about

common boundary conditions. Figure 5 shows the coupled codes and the associated application managers.

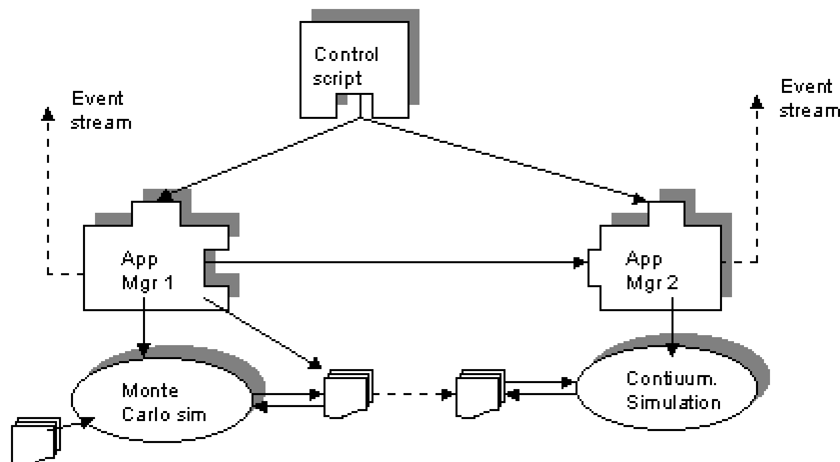

Figure 5. Two coupled Chemical Engineering simulation programs. Application Manger 1 signals Application Manger 2 when the Monte Carlo simulation completes a time step and the associated output state files have been migrated. Upon receipt of the message Application Manager 2 runs the continuum simulation. When this terminates control is returned to the first AM.

The codes are run separately on the Grid. The transfer of files is done using Grid based file-management and transfer utilities. The interface to the Grid is provided by Application Managers. These wrappers provide access to Grid services such as GSI [gsi], grid-events, etc. to the codes making them "grid-aware". Each execution is set up and controlled from the controlling Jython script, which runs inside the portal. The primary mechanism for getting feedback is the event system. Grid file-management tools can be used to transfer output files that are generated. Events from the applications are handed off to event handlers that have been registered or are logged. Special events can also trigger event handlers that can change or control the course of the execution.

This application illustrates several interesting scenarios in collaboration. The experiment is set up by the chemical engineers using the tools provided in the portal. Simple web forms are created for parameter input which will control the experiment. Subsequent users do not need to know about these parameters or the mechanics of the grid computation. They will interact with only the portal web interface and event notification mechanisms

3.2 Embarrassingly Parallel Search: Computing Orbit Intersections

The CRASS application uses Grid technology to build a distributed parallel solution to the problem of deciding if a proposed artificial satellite orbit collides with space debris or known operational satellites. Over 7,000 observable orbiting objects larger than 10 cm in

size have been catalogued. Of those, only 6% are operational spacecraft. The rest include decommissioned satellites, spent upper stages, and mission related objects. Typical earth orbits also include debris from 129 on-orbit fragmentations (almost all of them explosions) which have generated 70,000 to 120,000 objects larger than 1 cm. Most of the debris is located in Low (less than 2000 km) and Geostationary Earth Orbits, sharing space with a large number of operational spacecraft. Because the difference in speed among orbiting objects can be immense, even a small debris object can totally destroy a target. Collisions may also produce other small fragments that would increase the population of debris. About one out of ten shuttle missions have had to perform collision avoidance maneuvers. CRASS uses the USS SpaceCom data base of TLEs (Two Line Elements) for space debris, which gives orbital parameters for the objects. CRASS uses those parameters and a model of the forces acting on the object to find its state at a later time. Doing this for both debris and satellites allows prediction of when two objects will pass close enough to warrant a collision avoidance maneuver.

CRASS first decides if two objects can collide after it has applied some simple filters, and then propagates both the operational satellite and the debris orbits for some period of time in the future (no more than one or two weeks because the predictions become very poor after that). This collision detection is easily divided into smaller independent problems by dividing the debris objects into several groups and assigning them to different computers. The operational satellites are replicated in all the nodes, and the propagation of their orbits is performed by all nodes. The overhead produced by such replication is small because the number of operational satellites is very small compared to the number of debris objects. Also note that there is no additional overhead involving communication between nodes, because the only collisions that we want to detect are between operational satellites and debris objects. The design of CRASS implements a client/server architecture involving a single master component and numerous distributed worker components (see Figure 6).

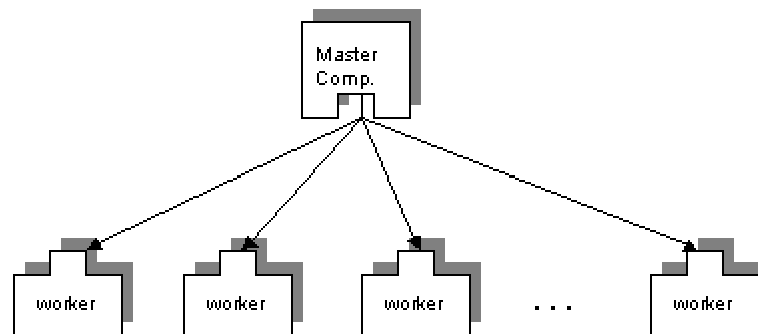

Figure 6. Two types of components, master and worker are used to partition the problem and distribute the work

When the computation is started, the master component sends the relevant TLE entries to the workers and issues a separate requests to each of them asking to check for collisions in a small simulation time period. After that, the master component issues new requests for succeeding time periods as soon as a previous request is completed. This loop repeats until the whole period of time under consideration is covered. The size of the time period is a configurable parameter and should be small if the master component is to have frequent updates of the state of the workers.

The master component displays the collisions as soon as they are reported and can visualize the orbits in an animated 3D model of the earth. Additionally, the master component allows the user to cancel, pause and restart the current computation. It is also responsible for the management of the worker components, like remotely shutting down the remote components using a special procedure call when the user exits the application. The user is responsible for setting the parameters of the computation, like the location of the TLE database, the period of simulation time, the visualization options, and other configuration options related to the computation. Figure7 shows the CRASS portal interface.

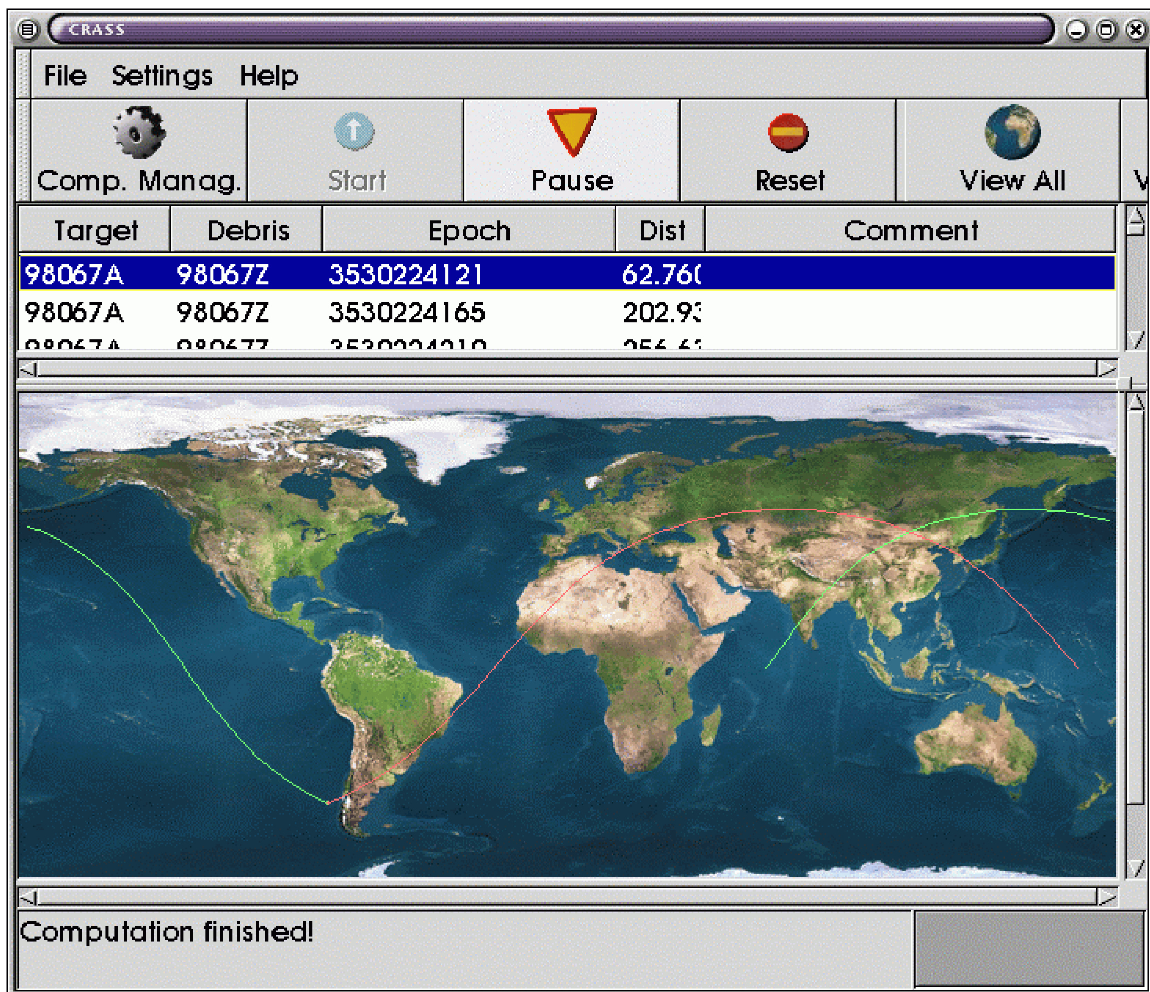

Figure 7: CRASS Science Portal Interface

The implementation in terms of CCA components is straightforward. Each worker component, written in C++, has a CCA provides port (Crass_Port) with the method:

```
std::string CrassImpl::findCloseEncounters(std::string* targets_tle,
                                           std::string* debris_tle,
                                           double* distance,
                                           double* from,
                                           double* to,
                                           double* step)
```

This method finds close encounters between the targets and the debris, with TLE's entered as strings. The method also takes as arguments the distance below which to report the close encounters, the time bounds, and the step size. It returns information about close encounters as strings.

The master component, also in C++, has for each worker a CCA uses port of the same type as above. The master and the workers are launched using Jython scripts. The number of workers are specified as a parameter to the Jython script, which uniformly distributes the workers on the set of target machines. The script also connects each uses port of the master with the provides port of a single remote worker as follows :

```
for i in range(0, numWorkers):
     cca.connectPorts(master, "masterMainPort" +
(i+1).toString(),workers[i], "inputCrassPort")
```

When the computation is started, the master component sends the relevant TLE entries to the workers, using the RMI call "findCloseEncounters" on its uses port, along with a simulation time period on which to operate. After that, the master component issues new requests for succeeding time periods as soon as a previous request is completed. However, it doesn't need to send the TLE entries again as the workers now have a local copy of the TLEs they need, thus saving bandwidth. This continues until the whole period of time under consideration is covered.

Some machines on which the workers are scheduled may be more powerful than others, and the workers running on those machines may finish faster than other ones. So a limited support for heterogeneous workload distribution is provided. At the beginning of the computation the worker nodes are asked to complete a short performance test by calling the method `int CrassImpl::performance()` on their provides port. Results from this calibration are used to divide unequally the set of debris objects, so that a larger set is assigned to the more powerful nodes. The other method that the provides port exposes is `void kill()` which is responsible for cleanup and exit of the worker when all the work is done. This is invoked by the master component on each of the workers, before it shuts down.

CRASS uses the publish/subscribe event model to notify the interface of possible collision events, allowing visual tracking. It allows an end user to access and effectively

use Grid resources, by providing the necessary Grid interactions via the Jython script underlying the Portal.

3.3 Complex Interactions: Linear System Solvers

A more complex example that show how multiple components can be linked to build a distributed algorithm test environment is given by the Linear System Analyzer (LSA) [lsa1, lsa2]. The LSA is a rapid prototyping tool for analyzing a sparse linear system of equations and testing various solution strategies on them. It is designed for the large, unstructured, sparse linear systems of equations which often occur in computational science and engineering. LSA components operate on SLS (sparse linear system) objects, and are in four categories: *I/O* ones which extract a SLS from a running application or a URL, *filters* which modify a SLS by scaling or reordering, *solvers* which solve the SLS and generate a solution vector, and *informational* modules which provide analysis of an SLS (e.g., spectral information, structure and storage information, or visualization.) Choosing a solution strategy for large sparse linear systems in realistic applications relies heavily on experimentation and exploration, and much time and effort is spent in recompiling code, trying to understand adjustable parameters in solvers, and trying to form a coherent picture of results from a variety of output . The LSA instead lets a user dynamically create a tree of components designed to provide information about the solution process and effect of each step on the sparse linear system. Using CCA components allows this to be done without recompiling code, and by launching the components on as many Grid resources as needed . Figure 8 illustrates a simple test configuration of components created by the LSA.

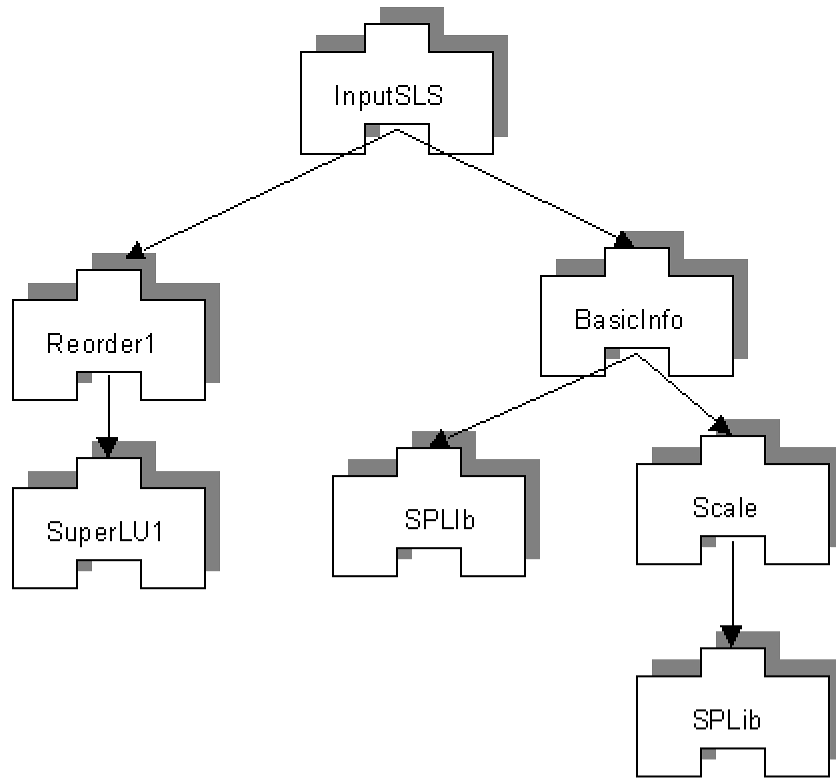

Figure 8. A small LSA session; arrows show the flow of a sparse linear system from input to various solvers.

```
def LSAbuildgraph():
   G = graph_lib.Graph()
   L.create_and_add_component(G,0,'InputSLS')
   L.create_and_add_component(G,1,'BasicInfo')
   L.create_and_add_component(G,2,'Reorder')
   L.create_and_add_component(G,3,'Scale')
   L.create_and_add_component(G,4,'Splib')
   L.create_and_add_component(G,5,'SuperLU')
   L.create_and_add_component(G,6,'Splib')
   L.add_edge(0,1,G)
   L.add_edge(1,2,G)
   L.add_edge(2,3,G)
   L.add_edge(3,4,G)
   L.add_edge(2,5,G)
   L.add_edge(2,6,G)
   return G

def DescendTree(L, G, root):
   for edge in G.out_arcs(root):
      L.LSAconnect(G.node_data(root),
G.node_data(G.tail(edge)))
      DescendTree(L, G, G.tail(edge))
   G = LSAbuildgraph()
```

```
DescendTree(L, G, 0)
```

The `create_and_add_component`() method calls and hides the XCAT invocations shown in Section 2.1. The `LSAconnect()` method similarly creates the CCA port connections. The output resulting from each component is accessible as a Web page on the remote machine, with their URL's published and added to the end user's browser page by the publish/subscribe event system described earlier. This is an example of the utility of using different communications mechanisms for different needs: the SLS sent between LSA components must be delivered synchronously or the computation will fail. However, getting the URL of a result file back to the LSA portal can be asynchronous, and if the message fails to arrive in a timely fashion it should not cause the overall application to fail.

A numerical linear algebraist can use these results to craft a solution strategy for sparse linear systems. For example, the difference between SPLib on the left and SPLib on the right is the second one had the linear system scaled first, by being fed through the Scale_1 component. However, comparing the Web page results of the two SPLib instances shows that it actually took more iterations (48 versus 46) and more time (5.19 versus 4.98 seconds) to solve the scaled system – telling the end user that scaling is ineffective for this problem.Equally important, the user can then add more components to the tree, or disconnect and terminate branches of the tree, from the notebook interface. Each time the tree is changed, a simple image of the new tree like that in Figure 7 is created by the Jython script and added to the LSA portal as a web page, giving a visual representation of the current state of configuration and a record of the user's sequence of steps.

4. Web Services and Grids

In 2001 several large software companies began to consider what could be done to make the concept of business-to-business distributed applications work. The early attempts in 1999-2000 by several B2B initiatives were based on linking together html based web sites that provide services to businesses. This approach failed because HTML descriptions of web sites did not carry enough semantic information for one site to invoke the services of another. By developing a precise XML description of the services provided by a site, along with a specification of the protocols needed to invoke the service, one would have a standard framework for B2B operations. This framework would allow businesses to provide services that other client businesses could invoke remotely and reliably directly from their own software. Furthermore, it would be possible to build automated service brokers that would give users a choice of implementations of similar services to solve a particular problem. In 2000 and 2001 this consortium of companies, working with W3C released a set of standards which defined web services. These standards include

- The Web Services Description Language WSDL [wsdl] that defines the XML Schema used to describe a web service. Each Web Service is an entity, which is defined by **ports** that are service "endpoints" capable of receiving (and replying

to) a set of messages defined by that port's type. Each port is, in fact a binding of a port type and an access protocol that tells how the messages should be encoded and sent to the port. A service may have several different access points and protocols for each port type.

- The Universal Description, Discovery and Integration (UDDI [uddi]) and the Web Services Inspection Language (WSIL) [wsil] provide the mechanism needed to discover WSDL documents. UDDI is a specification for a registry that can be used by a service provider as a place to publish WSDL documents. Clients can then search the registry looking for services and then fetching the WSDL documents needed to access them. However, not all services will be listed on UDDI registries. WSIL provides a simple way to find WSDL documents on a web site. These discovery mechanisms correspond to the Grid Information Service [mds2] in Globus terms.

In addition several other standards have been proposed that provide additional features. For example, IBM has proposed the WSFL [wsfl] which is a mechanism for scripting the workflow for integrating multiple services together to accomplish a complex task. A workflow engine acts as the agent that follows the WSFL specification document and contacts each of the services required by the specification following the order (an directed graph) specified. This workflow engine and WSFL document plays the same role for web services as the Jython script engine and script used to connect and manage XCAT components.

In many ways, the Web Services framework that is emerging is no more powerful than any other distributed object system. For example, CORBA [omg] has many of the features WS support and many more. So why are Web Services interesting? The answer is that the WS standards are simple, they are based on standard web technologies, and they are focused on making interoperability possible and easy. WSDL documents allow multiple protocols to be associated with a given service and WSFL provides a very high level mechanism for describing the way services can be combined together to accomplish a task.

There are several obvious ways that the web services model can be used in Grid systems. The first is to redefine many of the standard grid services as web services. For example, some interesting and useful Grid-Web services would be

- Grid Authorization Service: This service would provide a place where policy questions such as "Is John authorized to access resource X?" can be evaluated.
- Grid Application Resource Broker. This service would select the best compute resource from all those available on the grid for a particular application to run on.
- Grid co-Scheduling Service. Many resource schedulers are now incorporating advance reservation features. In Grid applications where more than one resource is required this service would negotiate a time when both are available and notify the client.
- Grid File Object Metadata Directory. Being able to fetch metadata associated with a Grid file object handle is very important for many applications. For example

File Object meta data can direct me to any special file reader I may need to access the object. Searching Grid metadata for files with specified metadata attributes is also very important.

Many more of the standard Grid Services can, and will, be given WS interfaces. This will greatly simplify the task of making Grid Services available to applications that need them and to the portal tools users will use to access them.

As can be seen from the definition of a web service, it is not substantially different from a XCAT Grid component. In fact, the standard "default" access protocol for web services is the same (SOAP) as we use for our default XCAT communication protocol. Furthermore, the port interface types supported by XCAT are easily described by a subset of the XML Schemas used by WSDL. Hence an XCAT component instance is web service.

But not every aspect of XCAT components can be described by the current web services standards. For example, CCA components also have "uses" ports that represent the call points in a component to an external "provided" port on some other component. It is by connecting uses ports to provides ports that give CCA its programming-by-composition nature. It is our feeling that this would be a useful addition to the web services standard.

4.1 Grid Application Factory Service

One of the common difficulties with the current Grid programming model involves the deployment of applications that are to be shared by a group of users. While this is often dismissed as a simple management problem, it is often a great source of frustration when groups of Grid users attempt to collaborate. Grid frameworks like Globus provide a uniform mechanism for submitting jobs to batch queues on remote systems, but Globus does not currently provide a mechanism to deploy an application on that resource. Hence the deployment task (tracking down all needed libraries and installing and testing the application in that environment before it is made available to others) is left to a user (probably the application author) or a system administrator who is probably unfamiliar with the application. Unfortunately, a user-installed application is frequently difficult to invoke by another user. This is because environment variable settings in user environments differ greatly from one user to another. Also user applications tend to read and write local files and, unless the application designer has taken this into account, there will often be errors caused when user A invokes an application that attempts to write a temporary file in user B's directory. Furthermore, if multiple users concurrently want to invoke the application provided by user B, then user B better have managed the name space for temporary files correctly or there will be collisions when different instances of the application attempt to write to the same temporary files.

To illustrate how we can use the Web Services + CCA model together, we can define a Grid Factory Service (GFS) as a Grid Service that exists solely to instantiate instances of a specific application for an authorized set of users. A GFS provides an interface that allows a client to specify application parameters and resource requirements for the application to run and the GFS creates a running instance of that application and returns

some for of handle to it to the client. For example, the client may submit a request of the form "take input parameters from the file identified by this URN and put the output on file system x and identified by this URN." and "run this with 4 gigabytes of memory and 132 processors." It is the responsibility of the GFS to negotiate with resource brokers and lower level services like Globus Gram to make this happen.

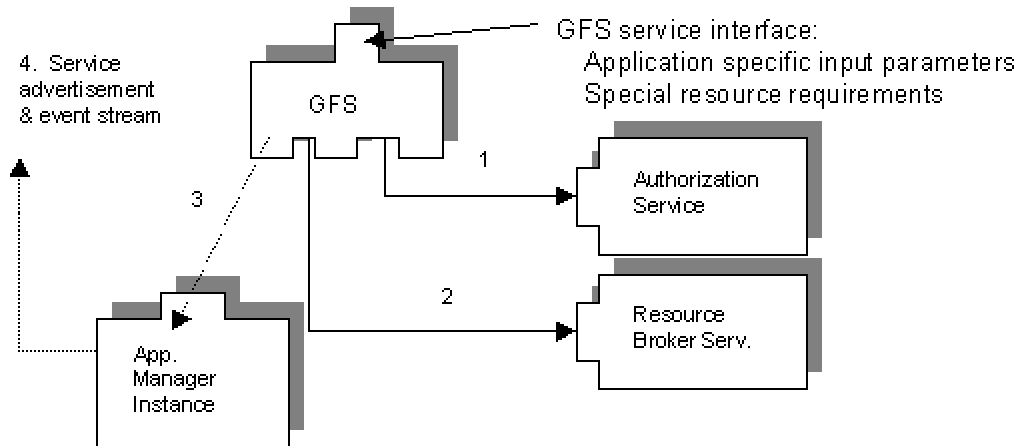

Figure 9. Grid Factory Service is contacted by the client who supplied application specific parameters and special resource requirements. The Factory service first (1) contacts an authorization service to verify that the user is authorized to execute this application. Next (2) it contacts a resource broker service to find a suitable execution host. It then (3) launches an application manager instance and returns a handle to that instance. This handle may be the name of the instance, which the client can use to discover it when it actually starts running. When the client does run it may publish an event stream and a WSDL service advertisement the client can use to contact it.

The GFS provides a level of abstraction to the client that is much higher than grid services like Gram because it takes low level job submission details like environment variable and temporary file management out of the hands of the client.

4.2 Providing Grid Security to Grid Web Services

In the example of the Grid Application Factory Service described in the previous section, we have some distinct requirements for security. The requirements can be classified as follows. At the lowest level we need transport layer security included both client and server authentication. Next we need authorization and, when these issues are resolved for a specific connection, we must consider delegation of authority.

Transport Layer Security (TLS via SSL) takes care of basic authentication of server and client. It is essential for the client to know that it is talking to the right server. A similar identification process may be needed for the server to trust the client. SSL/TLS is the

most pervasive underlying protocol. We use the Grid Security Infrastructure (GSI) [gsi] to provide Public Key Infrastructure. The use of Java CoG kit [cog] that uses IAIK's SSL as the underlying protocol provides the authentication. The underlying implementation can be easily replaced by a similar system like Sun's Java security infrastructure.

XSOAP/Java uses Java CoG kit to process the Globus user proxy certificates and then to manage secure connections with the server. The default mode of XSOAP using Globus grid proxies is as a *personal web service*, i.e. when a web service is started using a CoG Java provider it accepts only connections that uses the same user proxy for the client authentication.

We are also working on providing multi-user authentication. In this mode the server will trust a number of clients based on stored public key information.

The higher level of security would be the authorization of the client. Authorization means only certain types of clients may be given different rights to the services. This will be determined by the policy information on the server side and from the Grid Authorization Service and the credentials that the client presents on the connection.

The policy information will need to have flexibility to store information about individuals, groups of people and the type of accesses allowed. Since X.509 certificates are passed as credentials during the authentication process the Distinguished Names in the certificates can be used to identify the client.

There is also a need for a service to access another service to satisfy the user's requests. The user may decide to delegate his credentials to the server that can be in turn used by the service to access other services on behalf of the user. To provide this functionality we are using GSI delegation capability.

In current XCAT Provides Port provides no security features for the component. Now that XSOAP, the underlying framework for XCAT has authentication and authorization capabilities component writers will have the ability to add security features to their components. The first step will be to have a Grid Personal Application Event Service that will accept events from only the user who started the service. In current XSOAP implementation different security features are easily pluggable that will make it scalable for inevitable future changes and will provide good flexibility for XCAT component framework.

5. Peer to Peer Grid Concepts

Peer-to-Peer (P2P) systems can be divided into two categories:

- File sharing utilities such as Napster, Gnutella and Freenet [p2p], which can be characterized as providing a global namespace and file caching and a directory service for sharing files in a wide-area distributed environment. In the most interesting cases, the resources and all services are completely distributed. Each

user client program, which can access local files, is also a data server for files local to that host.

- CPU cycle sharing of unused user resources usually managed by a central system, which distributes work in small pieces to contributing clients. Examples of this include Seti-at-home, Entropia [entropia] and Parabon [parabon].

Both of these cases are interesting distributed system architectures. From the perspective of Grid computing there are several compelling features to these systems. First, the deployment model of P2P systems is purely user-space based. They require no system administrator. Security, when it exists in these systems is based on users deciding now much of their resource they wish to make public to the collective or to the central resource manager. Also P2P systems are designed to be very dynamic, with peers coming and going from the collective constantly as users machines go on and off the network. P2P systems are also do a good job of bypassing firewalls. This stands in contrast to large-scale scientific Grid systems, which manage large expensive resources and must be constantly maintained and managed by the system administration staff.

SUN Microsystems has released a package JXTA [jxta] into the public domain that provides a simple toolkit for building P2P systems. There are several important contributions that JXTA makes to P2P. The most important is concept of **PeerGroups**. In JXTA-like systems a PeerGroup is a distributed collection of people that want to collaborate in some way. Consequently there must be a P2P service that allows groups of individuals to define an entity that represents the group. JXTA includes a series of protocols for PeerGroups including Discovery, Membership, Sharing, etc. These provide the ability for members of a PeerGroup to identify each other, to agree on membership and to exchange information. For example, a simple way to do discovery and membership resolution is to use a simple directory that associates group names with lists of users and their public keys. Presenting a signed certificate and a group name to this service will validate the user as an authentic member of the group. Once membership in a peer group can be established in a secure way, the individual members can deploy services that can be shared by others. For example, a user with a special application he or she wishes to share with others in a group can deploy a Grid Application Factory service that authorizes only those users that are members of the group.

6. Conclusions

In this paper we have addressed the problem of Grid programming from the perspective of building distributed applications by composing them from application components and services. We look at three approaches to the problem of designing distributed applications: software component systems, web services and peer-to-peer frameworks. We argue that software component systems and web services share many important characteristics and can work together well as a foundation for building Grid applications. Peer-to-peer systems show the important concepts of collaboration and light-weight, easily deployed services to Grid computing.

The component model we discuss is the DOE CCA. In this model each application component can present two types of interfaces. The first type, called a provides port, is an interface of functions that can be invoked by remote clients. The second type, called a uses port, is a call-site in a component where it uses the services of a providing component. If two components have matching interface types on uses and provides port pair, they may be connected. In the Grid implementation of CCA, called XCAT the communication between a uses port and a provides port is a remote procedure call. The protocol for this RPC is based on SOAP as the default communication layer. While the point-to-point style of port communication works for many applications, there are many other Grid applications that need an asynchronous messaging system. XCAT uses a simple SOAP based event messaging system. Events, which are small XML documents, can be sent from publishers to persistent event channels and then broadcast to subscribers.

Unfortunately, many distributed Grid computations must be composed of components that are legacy codes that were never designed to run in any other mode than "stand alone". To address this problem, we have designed a special "Application Manager" (AM) component that can be used to be a proxy for a legacy application. The AM can stage files, launch the legacy application and signal, via events or port connections, when the managed application changes state. Application managers are programmed by simple Jython (Python in Java) scripts, which are encapsulated as standard CCA/XCAT components.

The Web Service model is also a component-like framework. Individual components are services that present a provides-style port and a protocol binding described by WSDL. Standards like UDDI and WSIL provide a discovery mechanism for services that is very similar to the Grid information system model, but it is applied at the level of application services instead of Grid hardware resources. Workflow in WS applications is defined by WSFL which is executed by a flow engine in a manner that is similar to the way CCA scripts are evaluated by the application manager component. We note that this service model can be used to good advantage in building application factory services that launch instances of applications on behalf of distributed Grid applications or remote users. We have also shown that WS applications can also be enhanced by adding Grid security protocols by layering SOAP on top of SSL using standard Globus and other X.509 certificates.

We concluded with a brief discussion of the role peer-to-peer systems can play in defining Grid collaboration tools. It is our experience that often Grid users would like to be able to set up small, private, temporary distributed collaborations. P2P technology and concepts can be used to extend the reach of conventional Grid systems to allow this to happen.

7. References

[CCA1] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt and J. A. Kohl, "The CCA Core Specification In a Distributed Memory SPMD Framework," submitted to *Concurrency : Practice and Experience.*

[CCA2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. Mcinnes, S. Parker, and B. Smolinski, "Toward a Common Component Architecture for High Performance Scientific Computing," *High Performance Distributed Computing Conference*, 1999. See http://z.ca.sandia.gov/~cca-forum.

[CCAT] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, M. Yechuri, "A Component Based Services Architecture for Building Distributed Applications," *Proceedings of HPDC*, 2000.

[lsa1] R. Bramley, D. Gannon, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju, "Component architectures for distributed scientific problem solving," IEEE Computational Science and Engineering, 5, no. 2 (1998) pp.50-63.

[lsa2] R. Bramley, D. Gannon, J. Villacis, A Whitaker, "Using the Grid to Support Software Component Systems," SIAM Conference on Parallel Processing 1999.

[cactus] The Cactus Code. See http://www.cactuscode.org

[cog] Commodity Grid Kits. See http://www.globus.org/cog

[CCA] The Common Component Architecture Technical Specification, Version 0.5. See http://www.cca-forum.org.

[cbiowb] Computation Biology Workbench. See: http://workbench.sdsc.edu for the current version. The Biology workbench now resides at SDSC.

[omg] *The Common Object Request Broker: Architecture and Specification*, Object Management Group, February 1998. See http://www.omg.org/corba.

[ccm] *CORBA Components*, Object Management Group, OMG TC Document orbos/99-02-95, March 1999. See http://www.omg.org.

[mds2] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman, "Grid information services for distributed resource sharing", Proc. 10th IEEE HPDC, Aug 2001.

[Rutgers] DISCOVERY: An Interactive Computation Collaboratory for Grid Applications. See http://www.computingportals.org/CPdoc/discover.pdf

[colab] The DOE2000 collaboratory project is documented at their web site. See: http://www-unix.mcs.anl.gov/DOE2000/collabs.html

[enote] The DOE2000 electronic notebook project resources can be found at the following location: http://www.csm.ornl.gov/enote

[entropia] Entropia Distributed Computing, see http://www.entropia.com

[edg] European Data Grid, http://www.eu-datagrid.org/

[Grid1] G. C. Fox, D. Gannon, "Computational Grids", IEEE Comput Sci Eng. Vol 3, No. 4, , pp. 74-77, 2001

[foxport] G. C. Fox, "Portals and frameworks for web based education and computational science", 2nd Int. Conf. On Practical Applications of Java, The Practical Application Co., http://www.practical-applications.co.uk/Proceedings/index.html\#PAJAVA, 2000.

[webflow] G. C. Fox, W. Furmanski, "High-performance Commodity Computing", Chapter 10, The GRID: Blueprint for a New Computing Infrastructure, Ian Foster, Carl Kesselman, eds. 1998, Morgan Kaufmann

[Grid2] D. Gannon, and A. Grimshaw, "Object-Based Approaches", (*The Grid: Blueprint for a New Computing Infrastructure*}, Ian Foster and Carl Kesselman (Eds.), pp. 205-236, Morgan-Kaufman, 1998.

[Gateway] Gateway Computational Portal. See http://www.computingportals.org/CPdoc/Gateway_CP.doc

[globus] Globus, Argonne National Lab, Math and Computer Science Division, http://www.mcs.anl.gov/globus

[GRADS] GrADS Testbed: Grid application development software project. http://hipersoft.cs.rice.edu.

[GCE] GridForum Grid Computing Environment working group (www.computingportals.org) survey of existing grid portal projects. www.computingportals.org/cbp.html.

[Grid] *The Grid: Blueprint for a New Computing Infrastructure*, Ian Foster and Carl Kesselman (Eds.), Morgan-Kaufman, 1998.

[Griphyn] The Grid Physics Network, http://www.griphyn.org/

[gsi] The Grid Security Infrastructure, see http://www.globus.org/security.

[legion] Andrew Grimshaw. Legion: A Worldwide Virtual Computer. See http://www.cs.virginia.edu/~legion.

[globwrap] Kieth Jackson. Globus Wrap. http://www.itg.lbl.gov/~kjackson/globusWrap

[ipg] W. Johnston, D. Gannon, B. Nitzberg, A. Woo, B. Thigpen, L. Tanner, "Computing and Data Grids for Science and Engineering," Proceedings of SC2000.

[KG97] K. Keahey and D. Gannon, "PARDIS: A Parallel Approach to CORBA," *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computation*, August 1997.

[KG98] K. Keahey and D. Gannon, Developing and Evaluating Abstractions for Distributed Supercomputing, Journal of Cluster Computing, special issue on High Performance Distributed Computing, Vol. 1, No. 1, May 1998.

[net] Microsoft Corporation. ".NET", See http://www.microsoft.com

[Miss] Mississippi Computing Web Portal. See http://www.computingportals.org/CPdoc/mcwp.doc.

[jms] R. Monoson-Haefel, D. Chappell, "Java Message Service", O'Reilly, 2000

[ipglp] NASA IPG Launch Pad Portal. See http:/www.computingportals.org/Cpdoc/LaunchPad.doc

[nimrod] Nimrod: A tool for Distributed Parametric Modeling", see http://www.csse.monash.edu.au/~davida/nimrod

[p2p] A. Oram, "Peer-to-Peer: Harnessing the Power of Distributed Technologies", O'Reilly, 2001.

[parabon] Parabon Computation, see http://www.parabon.com

[Pallickara] S. Pallickara, "A Grid Event Service", Ph.D. Thesis, Syracuse University, 2001

[jxta] Project JXTA, http://www.jxta.org

[ppdg] "Particle Physics Data Grid", see http://www.ppdg.net/.

[python] The Python Programming Language. See http://www.python.org for complete details.

[soap] Simple Object Access Protocol. See http://www.w3.org/TR/SOAP

[xsoap] A. Slominski, M. Govindaraju, D. Gannon, R. Bramley, "Design of an XML Based Interoperable RMI System: SoapRMI, PDPTA, June 25, 2001. see also http://www.extreme.indiana.edu/soap.

[srb] "Storage Resource Broker", San Diego Supercomputer Center, http://www.npaci.edu/DICE/SRB/.

[ejb] A. Thomas, "Enterprise JavaBeans Technology: Server Component Model for the Java Platform", http://java.sun.com/products/ejb/white_paper.html, 1998.

[hotpage] Mary Thomas. Hot Page. USCD User Portal http://www.computingportals.org/ CPdoc/HotPage.doc.

[uddi] UDDI: Universal Description, Discover and Integration of Business for the Web. See http://www.uddi.org.

[CAT] J. Villacis, M.Govindaraju, D. Stern, A. Whitaker, F. Breg, P. Deuskar, B. Temko, D. Gannon, R. Bramley, "CAT: A High Performance, Distributed Component Architecture Toolkit for the Grid," Proceedings High Performance Distributed Computing Conference 1999.

[wsdl] "Web Services Description Language (WSDL) 1.1", W3C, http://www.w3.org/TR/wsdl

[wsfl] "Web Services Flow Language (WSFL)", see http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[wsil] "Web Services Inspection Language (WSIL), see http://xml.coverpages.org/IBM-WS-Inspection-Overview.pdf

[nws] Rich Wolski. The Network Weather Service. See http://www.npaci.edu/NWS for detials.

[xcat] "The XCAT Science Portal", S. Krishnan, R. Bramley, D. Gannon , M. Govindaraju, R. Indurkar, A. Slominski, Proceedings, SC 2001, Denver, Nov. 2001

[xport] "The Xport Project", http://www.cs.indiana.edu/ngi/, 3/9/2001.