Standards Based Heterogeneous Metacomputing:  
The Design of HARNESS II

Mauro Migliardi  
University of Genoa, DIST  
Via Opera Pia 13  
16145 Genoa, Italy  
om@dist.unige.it

Dawid Kurzyniec, Vaidy Sunderam  
Dept. of Math and Computer Science, Emory University  
1784 North Decatur Road, Suite 100  
Atlanta, GA 30322, USA  
{dawidk,vss}@mathcs.emory.edu

Abstract

Emerging trends in heterogeneous distributed metacomputing and in Web Services technologies exhibit several commonalities that each domain can exploit. In this paper, we present an architectural model and design issues in leveraging Web Services to construct metacomputing frameworks. Our design is based on a combination of concepts currently embodied in the Harness system and those implemented by the Web Services Description Language and associated technologies. We begin by drawing parallels between component based metacomputing and the use of web services in the electronic commerce domain. We then suggest that the direct use of web service technologies is inappropriate and inefficient for high performance distributed computing, but that natural extensions are possible to enhance the suitability of these technologies. Based on our proposed extensions, specifically with regard to component deployment, localization, and encoding, we describe an operational model, architectural layering, and high-level design for the Harness II metacomputing framework.

1. Introduction

Heterogeneous computing technologies and tools have matured considerably in the last decade, and current trends are closer to realizing the vision of seamless integration across multiple distributed platforms. Most recently, two types of technologies, namely computational grids and Web Services, have become predominant arenas in network computing, the former for large scale high performance scientific applications and the latter for electronic commerce and business uses. Distributed computing for high performance applications has built on the traditional foundations of SPMD and cluster computing as exemplified by PVM [10] and MPI [13]; frameworks such as Legion [13], Globus

[0] and Harness [17] are extending the scope and applicability of network based concurrent computing in far reaching ways.

The Harness project focuses primarily on improving robustness, enabling resource sharing, and adaptation, by proposing a component-based architecture that greatly facilitates configuration and reconfigurability, while other projects are oriented towards the “grid” abstraction. The Legion and Globus projects are widely known realizations of the concept of computing grids, intended to replicate the notion of electrical power grids, where computing power may be obtained “on demand” from a network of “suppliers”. This approach to metacomputing is intended to permit users to tap computational power from a virtually location-less reservoirs or to aggregate computational resources available within a corporate entity. Commercial attempts to realize this concept, albeit with practical constraints to the pure grid abstraction, also exist; examples include the systems available from Sun Microsystems [24] and Avaki [2]. In essence, all these projects have progressed from enabling communication across heterogeneous clusters as in first generation systems, to dynamically aggregating resources from multiple administrative domains into a coherent metacomputing system, with auxiliary systems providing scheduling and allocation facilities.

In sharing resources among multiple domains, a number of new issues need to be addressed. A comprehensive list is outside the scope of this paper, although a detailed discussion may be found in other publications [20]. The issues relevant to the topics under discussion here include:

- resource description and registration – resources that are being contributed by suppliers should be described with sufficient semantic information for users to determine their suitability, and should be published in accessible locations.
- lookup and discovery – mechanisms for determining available resources and the conditions under which
they may be accessed are required to find such resources.

- access methods – resources may be accessed in different ways, e.g. multiple transport mechanisms may be available to utilize a remote service; both suppliers and users should possess appropriate frameworks to enable resource access and use.

- mapping, scheduling, access control – resources should be mapped into usable aggregates (e.g. a distributed virtual machine), allocation of resources to multiple requesters should be performed, and most importantly, secure access control and unified authorization mechanisms must be provided.

These are among the myriad issues that middleware layers need to provide in order to realize the vision of computational grids. Currently, grid platforms implement their own systems for each of these needs, typically using proprietary methods that do not interact with other systems or other technologies. Examples include the Globus resource discovery and management system GRAM [8], and the services provided by the Master Host daemon in the Sun Grid Engine software architecture [24].

Concurrent with developments in the scientific computing community, but independent of these, the business and electronic commerce communities have been attacking problems relating to: (1) describing distributed software components in a language- and possibly paradigm-neutral manner; (2) enabling self discovery among such components; and (3) providing components with an interoperable, standardized mechanism to request and obtain each other’s services. In the pursue of these goals, CORBA [25] has made tremendous progress, and has been the first successful framework in terms of allowing language and vendor neutral object interoperability. At the same time however, CORBA’s strictly layered architecture is also a limitation. In fact, true peer-to-peer object interaction is not possible with CORBA because each object either needs to interact with any other object through the underlying middleware or to bypass CORBA entirely (e.g. by opening an independent network connection and adopting a proprietary protocol).

Furthermore, CORBA limits itself by defining objects only through their functional interface, i.e. by means of IDL; in fact, CORBA does not define or support meta-data descriptions. For this reason, while CORBA allows easy and straightforward syntactical checks of object compatibility, it is impossible to express semantic compatibility within its model. In order to allow semantic match, it is necessary to allow the definition of the meta-data onto which the objects are operating. Enabling this mode of operation is the main goal of recent XML technologies [28] such as the Web Services Description Language (WSDL) [5], and the associated set of technologies and methodologies that are collectively referred to as “Web Services”. Literally, the term Web Service refers to a software component that is capable of providing services to clients distributed over the public Internet. Such an approach is consistent with a model in which distributed components communicate with each other on a peer-to-peer basis.

WSDL is a new specification method to describe networked XML-based Web Services. It provides a simple way for service providers to describe the basic format of requests to their systems regardless of the underlying protocol (e.g. Simple Object Access Protocol (SOAP) [4] or XML), or encoding, e.g. Multipurpose Internet Messaging Extensions (MIME). WSDL is a key component of the effort concerning the Universal Description, Discovery and Integration (UDDI) [26] initiative to provide directories and descriptions of such on-line services for electronic business. The UDDI Project is an industry initiative that is working to enable businesses to quickly, easily, and dynamically find and transact with one another. UDDI enables a business to (1) describe its business and its services; (2) discover other businesses that offer desired services; and (3) integrate with these other businesses. The resulting benefits to the commercial community and its clients are obvious, and it is therefore no surprise that this movement has garnered an enormous amount of support and backing. However, by focusing almost exclusively on a truly widely distributed model with HTTP-based communication, the Web Services world seems to have neglected the significant potential for application of the model in more tightly coupled systems. In particular, the Web Services model appears to be relevant, at a very high level of abstraction to heterogeneous metacomputing environments and computational grids – but in practice, locality, number or data crunching orientation, and the close coupling found in high performance grid applications preclude direct use of Web Services frameworks.

Nevertheless, it is our opinion that adoption of the standards fostered by the Web Services community could improve the level of usability and interoperability of grid systems and lead to the development of metacomputing systems based on reusable, interoperable components. However, in order to make those standards well-suited to the requirements of high performance computing modification are needed, and it is necessary to exploit their extensibility. Early investigations have shown that the direct adoption of those standards result in overheads and bottlenecks that may be intolerable in the low communication/computation ratio realm of business applications, but would degrade the performance of high communication/computation ratio scientific programs to unacceptable levels.

The remainder of this paper is organized as follows. Section 2 outlines other projects exploring the intersection between Web Services and grid computing realms. In Section 3, we provide an overview of the Harness Metacomputing...
experiments with adaptation of the Common Component Ar-
component-based scientific frameworks. For example, ex-
the two concepts in the very near future. Between these communities will lead to further diffusion of
[27] and OMG itself. It is expected that the collaboration
members from both the scientific and commercial comput-
ware Services Grid Workshop [21] organized in July 2001
of large scale distributed computing. Until 2001, the two
concepts existed solely in their own domains. Since then,
however, they have gradually started to penetrate other’s do-
mains [23]. In this section, we briefly mention some repre-
sentative, recent, and still-evolving efforts to leverage and
combine the two types of technologies.
Sun Microsystems’ Grid Engine framework [24] focuses
on harnessing the distributed computing power within an
organization. It adapts the concept of grid computing to allow
businesses to increase utilization of the hardware resources,
and thus only addresses a limited form of resource sharing.
Another commercial entity, Killdara Corporation, portray
their Vitiris [15] system to be a foundation for a global Web
Services grid for electronic commerce, allowing small busi-
nesses to cooperate on a peer-to-peer basis. Vitiris provides
a lightweight Java-based application server capable of run-
ning on the Java Micro Edition platform, and a set of fun-
damental business-oriented web services. These efforts are
examples of business entities adopting grid technologies to
enhance commercial applications.

From the other side of the fence, major grid technol-
ogy providers like Globus and Avaki have acknowledged
the appropriateness of the Web Services model in the scien-
tific computing domain [6, 3]. Making grids Web Service-
enabled is currently an active area of research. The Soft-
ware Services Grid Workshop [21] organized in July 2001
by the Object Management Group (OMG) [22] gathered
members from both the scientific and commercial comput-
ing arenas, as well as standard bodies such as Global Grid
Forum (GGF) [11], World Wide Web Consortium (W3C)
[27] and OMG itself. It is expected that the collaboration
between these communities will lead to further diffusion of
the two concepts in the very near future.

Web Services are also being adopted by other
component-based scientific frameworks. For example, ex-
periments with adaptation of the Common Component Ar-
chitecture (CCA)-based [11] XCAT framework has been
documented [9]. This work introduces the concept of an
application factory service that is designed to simplify ap-
plication deployment. The function of an application fac-
tory service exhibits some similarities to the concept of a
component container as described in this paper. We intend
to explore the potential interoperability benefits that may
arise as a result from moving both projects adhere to Web
Services standards.

3. Harness Framework Architecture

The initiative described in this paper is being undertaken
in the context of Harness, a research metacomputing sys-
tem that is based on the notion of a software backplane
into which component modules are plugged in. These com-
ponents coordinate with each other to realize the various
functions required for loosely coupled distributed comput-
ing; the backplane and component model improves flexibil-
ity and functionality, and permits reconfiguration of system
capabilities as needed. In the current version of Harness,
some plug-ins are provided as part of the system distribu-
tion, while others might be developed by individual users
for special situations, while yet other plug-ins might be ob-
tained from third-party repositories.

Figure 1 depicts the overall architectural model of Har-
ness, as implemented in the current version. As the fig-
ure shows, the fundamental abstraction in Harness is a Dis-
tributed Virtual Machine (DVM) that provides a context for
the execution of concurrent programs. A DVM is associated
with a symbolic name that is unique in the Harness name
space. DVM’s are created by users and “constructed” by
first adding nodes (A, B, C, D in the figure) to the DVM, and
subsequently deploying plugins on each node (p2p, mmul,
ing, etc in the figure). Some plugins may be node specific
while others are replicated; typically, a set of replicated plu-
gins for primitive functions such as message passing and
process management are loaded on all nodes thus provid-
ing a consistent baseline for common parallel processing
applications. Users or applications may also load appli-
cation specific plugins, i.e. domain specific computations
that are themselves implemented as components consistent
with the Harness plugin interface. Alternatively, users may
first load plugins that emulate distributed computing envi-
ronments (currently PVM, MPI, and JavaSpaces plugins are
available), thereby creating a framework within which their
legacy codes may run.

In addition to the significant advantage of reconfigura-
bility, the plugin based scheme of Harness greatly facil-
itates service-based leveraging of functionality among plug-
ins. In other words, plugins that implement a certain func-
tion can exploit the services provided by other plugins that
are already loaded within the same Harness DVM, thereby

2. Related Work

The concept of Web Services, and technologies associ-
ated with this model, evolved within the electronic com-
merce world to support business-to-business interaction.
On the other hand, computational grids were originally de-
veloped to support massive, large scale scientific comput-
ations involving supercomputing centers and other forms
of large scale distributed computing. Until 2001, the two
concepts existed solely in their own domains. Since then,
however, they have gradually started to penetrate other’s do-
mains [23]. In this section, we briefly mention some repre-
sentative, recent, and still-evolving efforts to leverage and
combine the two types of technologies.

Sun Microsystems’ Grid Engine framework [24] focuses
on harnessing the distributed computing power within an
organization. It adapts the concept of grid computing to allow
businesses to increase utilization of the hardware resources,
and thus only addresses a limited form of resource sharing.
Another commercial entity, Killdara Corporation, portray
their Vitiris [15] system to be a foundation for a global Web
Services grid for electronic commerce, allowing small busi-
nesses to cooperate on a peer-to-peer basis. Vitiris provides
a lightweight Java-based application server capable of run-
ning on the Java Micro Edition platform, and a set of fun-
damental business-oriented web services. These efforts are
examples of business entities adopting grid technologies to
enhance commercial applications.

From the other side of the fence, major grid technol-
ology providers like Globus and Avaki have acknowledged
the appropriateness of the Web Services model in the scien-
tific computing domain [6, 3]. Making grids Web Service-
enabled is currently an active area of research. The Soft-
ware Services Grid Workshop [21] organized in July 2001
by the Object Management Group (OMG) [22] gathered
members from both the scientific and commercial comput-
ing arenas, as well as standard bodies such as Global Grid
Forum (GGF) [11], World Wide Web Consortium (W3C)
[27] and OMG itself. It is expected that the collaboration
between these communities will lead to further diffusion of
the two concepts in the very near future.

Web Services are also being adopted by other
component-based scientific frameworks. For example, ex-
periments with adaptation of the Common Component Ar-
preserving the benefits of component architectures, even in complex distributed situations. Consider, for example, the PVM plugin for Harness [16] whose structure is shown in Figure 2. The hpvmd plugin emulates the PVM daemon on each host, but leverages process spawning, message transport, general event management, and table lookup from other plugins -- both within the same address space (same Harness kernel) as well as in remote Harness kernels (not shown explicitly in the figure). While such synergism may be possible to realize in monolithic architectures with substantial effort, the degree of enhanced functionality attainable by using a plugin model is far superior.

However, in the current version of Harness, plugins provide services to other plugins and to applications by following framework specific paradigms. All Harness plugins (currently) are implemented in Java, and must implement Java interfaces that are specified by the Harness model. Harness plugin services may be accessed only through implicit knowledge of the interfaces they export. Furthermore, that functional interface is currently accessible only through Java RMI. Finally, independent discovery and lookup mechanisms are not supplied.

While this approach does work well in certain circumstances, the lack of openness and standardization with respect to descriptions of plugins, registration, discovery and lookup schemes, and service invocation mechanisms can be limiting in many cases. With the recent advent of Web Services, however, an opportunity has been created to overcome all of these drawbacks by leveraging standards, tools, and methodologies that are likely to become extremely widespread and popular in the near future. The design of Harness II presented in this paper proposes to combine plugin based distributed computing with Web Services standards to enhance the viability and effectiveness of heterogeneous metacomputing.

4. Web Services

The standards- and service-oriented Web Services model possesses certain attributes that are critical to extending them for heterogeneous metacomputing. These characteristics are primarily concerned the publishing of interfaces and access methods, and service deployment; these issues are discussed in detail below. The deployment of a Web Service takes place after the service has been built and before it can actually run. The deployment of a Web Service requires the following, non strictly sequential, activities (see Figure 3):

- The publication of the Web Service interface into a lookup system;
- The publication of the access mechanisms the Web Service supports (e.g. SOAP and HTTP) into a lookup system;
- The actual deployment of the run time code in the service container and, if it is eventually necessary to do so (e.g. if the Web Service wraps a legacy application), the connection of the run time code with a legacy system.

It is necessary to perform these steps for each Web Service that a provider wants to deploy. As an example, the deployment shown in the figure requires the execution of the steps listed above for each of the three services (A, B and C). The actual details of this process and the actions involved in each of these steps depends on the type of lookup
service used (e.g. UDDI, WSIL, etc.), and on the actual proxy-to-service communication technologies (e.g. SOAP, HTTP, etc.) supported by the service. As an example, a service accessible through SOAP requires the setup of a SOAP compatible HTTP daemon such as Apache/Tomcat. As a second example, to interrogate a UDDI registry for a specific service it is necessary to format the query so that it contains details of the business entity providing the business service and on the so-called tModel (the technology adopted to describe the service) [26]. A UDDI registry can be accessed through a Java library called UDDI4J, or through a set of tools that facilitate generating, deploying and searching the service entries.

After these activities have taken place, it is possible for a client to discover the service by querying the lookup system. The lookup system returns the information the clients need to contact the service – specifically, the Web Service interface and the definition of the access points available for the given service. This step completes the intervention of the lookup service in the Web Service server to Web Service client loop. Once the client has acquired the documents defining the service, the lookup service is out of the loop, and interaction takes place directly between the Web Service and the client. There is no need for further interrogation of the lookup service.

The two documents stored inside the lookup service, namely the interface description and the access points description, together constitute the WSDL description of a Web Service. The separation of the abstract, interface description part from the concrete, implementation dependent access point description part, allows the reuse of WSDL documents and also allows for searches that are both implementation independent and implementation dependent.

IBM provides a WSDL4J software package (actually a set of Java packages) to facilitate the generation, manipulation and interpretation of WSDL documents in Java programs. These classes, together with the classes in the org.w3c.dom package, allow the functional generation, manipulation and analysis of a WSDL document in a Java program. In particular, WSDL4J defines data types for the extensibility elements of a WSDL document. This class can be extended by users to define new access mechanisms and protocols for a Web Service. In particular, this capability is demonstrated by IBM in another software package, namely the Web Services Invocation Framework (WSIF) [19]. This package contains the extension classes that allow the definition of a direct Java object-to-object connection between a Web Service and its client. Moreover, this package provides a skeleton implementation for the dynamic, run-time generation of Web Service stubs. Thus, it is possible for a client both to select the type of protocol it wants to use to access a service (e.g. SOAP) or to let the framework dynamically generate the required stub.

As a canonical example, consider the development of a trivial Web Service, a Time service. Although the functionality of such a service is trivial, it is important to note that common services (e.g. stock quote services) possess very similar characteristics. We develop a Time service by first specifying the Java class implementing the service:

```java
public class WSTime
{
    public String getTime()
    {
        return (new java.util.Date()).toString();
    }
}
```

The next step is to generate the WSDL description for this Web Service; the corresponding document is shown in Figure 7. The code can be generated by hand or semi-automatically, e.g. with the wsdlgen tool provided by IBM in the Web Services Toolkit. Currently, automatic generation is limited to SOAP bindings; however, it is possible to extract the abstract interface description from the automatically generated file and to integrate it manually with the required bindings.

As can be seen from the example, a WSDL definition is composed of two main parts. The first part defines the abstract service interface in terms of messages, port types, operations and bindings. An operation is an exchange of messages between the client and the server. A port type is a group of operations. A binding is the association of a name, a port type and a binding type. A WSDL document can specify different bindings for different port types and several bindings for each port type. At present there are only
three kinds of bindings standardized by the W3C consortium, namely SOAP, HTTP and MIME. These bindings are the ones commonly adopted in the business world. However, WSDL is extensible and it is possible to define new bindings to suit the needs of non-business applications. In our example, the WSTime service defines a single, non-standardized binding for its WSTime_Service port – the Java binding. This example of an extended binding allows access to a Java object in a local, non-mediated way. In the second, concrete part of the WSDL definition the binding is described according to its own details. In our example we specify that the actual server can be found by querying the run time for the WSTime class.

5. Web Services and Component-Based Scientific Computing

Metacomputing environments such as Harness are component-based, i.e. the heterogeneous computing environment is aggregated and a concurrent programming platform emulated thereupon through a set of coordinated software components. Component-based metacomputing frameworks utilize the concept of a Distributed Virtual Machine (DVM) as an abstract boundary for a collection of active components. The DVM defines a name space and provides generic discovery mechanisms, i.e. a lookup service. Typical interactions between components within a DVM are shown in Figure 3. When a component (e.g. component A in the figure) is created inside a DVM, it is registered in the DVM lookup service. Other components may then query the lookup service to get a handle for A. The handle usually contains a form of proxy that hides the remote connection details. The client component can then communicate with A using the provided proxy.

The overall scheme is very similar to the Web Services usage scenario as described in the previous section. However, the significant difference in metacomputing environments is that the components are no longer tightly bound to specific computational resources. Both client and server components may reside on the same host. Mobile components may even move from one host to another during run time. In fact, the terms “client” and “server” became blurred as every component can play both roles at the same time. For the discovery mechanism, there is a whole range of implementation approaches. At one extreme, there are centralized lookup services. They are easy to implement and use, but they introduce a single point of failure and a potential scalability bottleneck. At the other extreme, a completely decentralized approach leads to a registration phase that is fully localized and does not involve any network traffic, whereas the discovery phase performs an active lookup that can be expensive and difficult to manage. Most frameworks provide solutions that are intermediate to these extremes.

As mentioned earlier, the two architectures (Web Services as shown in Figure 3 and Component-based Metacomputing as shown in Figure 4) are similar at a very high level of abstraction. Currently, however, metacomputing frameworks perform the major phases in specifying, registering, and deploying components, and lookup, discovery, and access, using framework specific methods. This constrains usability, re-use, and applicability, since implicit and proprietary knowledge between component developers and component invokers must exist. More importantly, components developed for different metacomputing frameworks cannot interact, thereby severely limiting interoperability. The most important issue that limits interoperability between multiple component-oriented frameworks is that they are not based on broadly accepted standards. To describe components, various frameworks use IDL, Java interfaces, COM interfaces, or proprietary formats. The communication protocols adopted include various forms of IIOP, Java RMI, DCOM, or custom protocols traditionally built on top of TCP/IP. Examples of discovery mechanisms are LDAP, Jini, JNDS, DNS, and Globus MDS.

The Harness II system is being designed to overcome these intra- and inter-operability limitations by leveraging Web Services standards. We propose to use WSDL for component description in place of Java interfaces that are used currently. We also plan to take advantage of SOAP for inter-component communication. This will allow Harness II plugins to be registered in any WSDL-aware lookup service, and used by any SOAP-aware client. For instance, lightweight clients (e.g. handheld devices) will be...
able to control distributed scientific applications running inside Harness II distributed virtual machines. Harness II applications themselves will be able to access and use services provided by other parties. We anticipate that as other vendors move into the Web Services world, the level of interoperability between distributed systems, including Harness II, will dramatically improve.

Web Service technologies were designed to support business-to-business (B2B) interaction. Services in the electronic commerce domain are usually static, long running processes, deployed on a companies’ application servers. The amount of data exchanged between parties in B2B transactions is usually reasonably small (i.e. expressed in kilobytes) and consists mainly of structured data. On the other hand, components of scientific applications are highly dynamic. They are created and assembled on demand. They often exchange mega- or even gigabytes of data, consisting usually of plain arrays of numbers. In mobile component frameworks the active component (or agent) can sometimes avoid exchanging large amounts of data by instead moving itself, and performing computations on the host when data is stored. When we try to adapt Web Service technologies – as they exist today – to the area of high performance computing, the following three issues emerge:

- **Deployment issue.** Due to the static nature of electronic commerce services, deployment technologies do not provide adequate support for automated service instantiation. Solutions vary among different application servers and they usually require human interaction. Furthermore, business oriented discovery services are biased towards storing persistent information about long-lived services rather than volatile information related to fluid components.

- **Localization issue.** Although WSDL is extensible and can facilitate many kinds of service access types (bindings), its current definition approved by W3C only standardizes the SOAP/HTTP and MIME bindings. HTTP is an excellent choice for point to point communication due to its ubiquitous availability and the fact that it is traditionally tolerable to firewalls. However, in case of components running in the same local system, exchange of data through an HTTP server and TCP/IP stack is an obvious overhead.

- **Data encoding issue.** SOAP, being an XML-based protocol, is suitable mostly for exchanging structured data in reasonably small quantities as it is the case in the e-commerce world. Unfortunately, the default BASE64 encoding adopted by SOAP for XSD data types introduces unacceptable overheads for scientific data both in terms of the network bandwidth and the encoding/decoding time [12].

The design of Harness II proposes to address these issues by devising solutions to each of the above problems, while retaining full compatibility with Web Service standards. In order to solve the deployment related issues our design incorporates plans to: (1) develop a registry/lookup framework based on the capability of querying XML documents (actually WSDL descriptions) for specific nodes and values. This will allow us to map the generic queries generated by our framework onto the specific queries of commercial implementations (e.g. UDDI); (2) develop specialized lightweight component container for volatile DVMs and short lived applications; and (3) develop automated tools based on the Java libraries WSDL4J and UDDI4J to facilitate the interaction with Web Services compatible commercial component containers.

The Harness II design also proposes to leverage the extensibility of WSDL in order to solve the localization and data encoding issues. In particular we plan to develop a set of bindings capable of: (1) bypassing the text encoding requirements introduced by XML messaging; and (2) minimizing the number of entities that need to be traversed in order to deliver a message. An example of the envisioned operational schema of the Harness II system is shown in the Figure 5.
As an example, a JavaObject scheme dedicated to find a specific local instance of a Java object is currently under design and development. Our scheme is similar to the Java binding scheme defined by IBM in the Web Services Invocation Framework. However, the Java binding does not allow selecting a specific, pre-existing instance of the Java class defined. On the contrary, it only allows selecting the type of Java object that is going to provide the service. In our scheme the binding not only defines the object type but also a specific instance. Obviously, this kind of binding is not supported by any commercially available application server as it needs some special features offered only by the Harness II. In fact, while the Java binding requires the run time to retrieve a port factory that needs only to be capable of instantiating a new object of the selected type, our JavaObject scheme requires the run time to query the local component container to obtain a reference to an already instantiated, stateful object.

As a second example we plan to develop an XDR binding capable of delivering numerical data on direct socket level connections. The XDR scheme is designed to be limited to the transfer of numerical data. As such, the only type of complex data available is the array. This binding, instead of constructing an XML document as is done by SOAP stubs, relies on the capability of Java I/O streams to encode numeric data in XDR format. On the server side, the scheme mimics the behavior of the RMI daemon to select the actual target component.

Let’s consider as an example, the development of a simple matrix multiplication Web Service. The first step is to provide the Java implementation:

```java
public class MatMul
{
    public double[] getResult(
        double[] mata,
        double[] matb)
    {
        // actual implementation omitted
        return new double[0];
    }
}
```

Executing the servicegen tool provided by the Web Services Toolkit [14] generates the WSDL description and the Java classes for the port type interface and the stub. In Figure 8 we show the WSDL definition for the MatMul Web Service. In our example we use both a standard SOAP and a local Java binding. The standard SOAP binding introduces an encoding overhead as well as several intermediate steps in the execution that are generally unacceptable for high performance distributed computations. However such a standard access mechanism might be useful in the case of clients with very limited resources such as handhelds or or wearable devices. High performance applications might take advantage of the local, unencoded access provided by the Java binding if they can deploy the client onto the same component container where the MatMul Web Service is hosted. In fact, the Java binding allows the automatic retrieval of the class code and its instantiation.

### 6. The Design of Harness II

In the design of the second generation of the Harness metacomputing framework, we combine the architectural and technological facets of component based distributed systems with those of the Web Services computing model. In the previous sections we have presented the parallelism between these, identified their differences, and proposed schemes to reconcile the underlying assumptions of Web Services with the performance and coupling requirements of distributed metacomputing. In this section, we present a consolidated view of our proposed Harness II architecture.

In Harness II, Distributed Virtual Machines are constructed by assembling independent, web-based components. Thus the whole framework is based on the concept of cooperating, interoperable Web Services. In Figure 6 we show the general architecture of the framework and how different services cooperate to form a Distributed Virtual Machine. In the Harness II framework there is no mandated level of component exposure. Although every entity is potentially a public service (i.e. has a well defined, XML-based description), it is the provider’s run time decision whether the component is to be registered in one or more publicly available lookup services, or if it is to be kept private. The decision can be reviewed at any time, thus allowing published services to be removed and private services to be published. Also, different instances of a same service can have different levels of exposure. This flexible model has several advantages; for example, it allows an organization to test a service implementation internally and to publish it only after a sufficient level of reliability and performance has been achieved.

Containers constitute a special category of services. They represent an aggregation point, provide local component management, define a local name space and supply appropriate lookup capabilities. However, they are full-fledged services themselves. The service provider can either expose them to the public or keep them for private use, e.g. inside a departmental metacomputer. We define 3 abstraction layers (see Figure 6). Each of these layers is represented by a Web Service. The service exposes an XML interface that is bound onto one or more endpoints to form a complete WSDL document. The lowest layer represents the abstraction of a computational resource in terms of the runner box service. The runner box defines only the lim-
Figure 6. Harness II architecture

The design and architectural composition of Harness II, a proposed follow on project to the Harness metacomputing system, is described in this paper. Harness is based on the concept of reconfigurable distributed virtual machines, and uses a component based model, supplemented by a software backplane scheme to realize dynamic and highly flexible heterogeneous distributed computing platforms. With recent advances in Web Services, widely available and interoperable standards are becoming available for the specification, publication, deployment, discovery, and invocation of component based services. By leveraging the facilities that

Further, he can load his application component to the same container that hosts the LAPACK service itself, and take advantage of local bindings in order to minimize latency.

The next level of abstraction is the distributed component container. It implements a set of services similar to that provided by a Harness I Distributed Virtual Machine, i.e. it supplies a unified name space, status query, lookup service and management point for a set of component containers. In effect, that level of abstraction introduces the notion of a distributed global state. However, the Harness II framework defines only the DVM API and does not mandate any particular solution to maintain global state coherency. Concrete implementations are provided by the DVM-enabling components that may vary in implementation from the full synchrony method to complete decentralization. In the full synchrony scheme, the entire state information is replicated across all participating nodes. All system events are synchronously distributed to maintain coherency. This scheme, currently the only one supported in Harness version 1.9, may be appropriate for relatively small DVMs running applications with many critical components. In contrast, in a fully decentralized scheme state change events are not propagated to other nodes. Instead, every request for state information triggers a distributed query spanning across the DVM. As this approach minimizes network traffic during state changes but introduces overheads for state inquiry, it may be appropriate for loosely coupled, massively distributed applications such as Seti@home. Mixed solutions are possible as well. For example, mesh-structured applications may benefit from a scheme that provides full synchrony across small neighborhoods but facilitates distributed queries for farther hosts. Obviously, DVM-enabling components implementing different state coherency protocols cannot coexist in a single DVM. However, they always expose the same functional interface as defined in Harness II framework, so that applications can be deployed and run on any Harness II DVM regardless of the underlying state management solution adopted.

7. Conclusions

The design and architectural composition of Harness II, a proposed follow on project to the Harness metacomputing system, is described in this paper. Harness is based on the concept of reconfigurable distributed virtual machines, and uses a component based model, supplemented by a software backplane scheme to realize dynamic and highly flexible heterogeneous distributed computing platforms. With recent advances in Web Services, widely available and interoperable standards are becoming available for the specification, publication, deployment, discovery, and invocation of component based services. By leveraging the facilities that
Web Services offer, the Harness framework can be significantly enhanced, enabling Harness components, their discovery, and access to be framework independent, and furthermore, permitting Harness to interoperate with other grid technologies. However, Web Services, due to their strong bias towards typical electronic commerce applications, are inappropriate for direct use in high performance distributed metacomputing situations. The design of Harness II proposes several unique schemes to remedy these deficiencies, primarily via extensions to Web Services, that will permit standard frameworks such as WSDL, UDDI, WSIF, SOAP, and associated technologies to be practicable in distributed HPC environments. These proposed extensions primarily address issues relating to component deployment, localization, and data encoding, which can be realized efficiently in metacomputing scenarios by utilizing the techniques employed in our design. The Harness II framework is comprised of a layered architecture that defines the notion of component containers and distributed component containers to aggregate multiple types of service components within a hardware resource and across a distributed virtual machine respectively. Coupled with standards-based specification, lookup, and access to components, this architecture exploits the benefits of Web Services to create dynamic, flexible, and interoperable heterogeneous distributed computing environments without sacrificing performance and efficiency.

Acknowledgements

The authors wish to thank the anonymous reviewers for their insightful comments and suggestions that have resulted in significant enhancements to the paper. This work was supported in part by NSF grant ACI-9872167 and DoE grant DE-FG02-99ER25379.

References


Biographies

Mauro Migliardi was born in Genoa (Italy) in 1966; he obtained a Laurea degree from the University of Genoa in 1991 and a PhD in 1995. From 1995 to 1997 he was a research associate at the University of Genoa where he studied hybrid SIMD-MIMD computers for image processing in research projects funded by the EU and the Italian government. Recently he was a research associate at Emory University and one of the principals in the HARNESS project. Currently he is an assistant professor at the University of Genoa. His main research interests are parallel, distributed, heterogeneous computing systems and architectures, metacomputing and high performance networking.

Dawid Kurzyniec has received a M. S. degree in Computer Science from University of Mining and Metallurgy, Kraków, Poland in 2000. Since 2000 he is a research associate at the Emory University in Atlanta. He is involved in the research on frameworks for reconfigurable metacomputing systems, and he is currently one of the main architects of the Harness project. His research interests include distributed computing, metacomputing, heterogenous systems, object oriented technologies, and compilers.

Vaidy Sunderam is a professor of Computer Science at Emory University. His research interests are in parallel and distributed processing systems, and infrastructures for collaborative computing. His prior and current research efforts have focused on system architectures and implementations for heterogeneous metacomputing, including the PVM system and several other frameworks such as IceT, CCF, and Harness. Professor Sunderam teaches computer science at the beginning, advanced, and graduate levels, and advises graduate theses in the area of computer systems.
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WSTime_Service"
    targetNamespace="http://www.wstimeservice.com/WSTime"
    xmlns="http://schemas.xmlsoap.org/wsdl/
        xmlns:interface="http://www.wstimeservice.com/WSTime-interface"
    xmlns:java="http://schemas.xmlsoap.org/wsdl/java/
    xmlns:types="http://www.wstimeservice.com/WSTime"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!--Definition of message types-->
    <message
        name="IngetTimeRequest"/>
    <message
        name="OutgetTimeResponse">
        <part
            name="meth1_outType"
            type="xsd:string"/>
    </message>
    <!--Definition of port types-->
    <portType name="WSTime_Service">
        <operation
            name="getTime">
            <input
                message="tns:IngetTimeRequest"/>
            <output
                message="tns:OutgetTimeResponse"/>
        </operation>
    </portType>
    <!--End of the abstract, reusable part.-->
    <!--Start of the implementation specific part-->
    <!--Association of port types with specific bindings-->
    <binding name="WSTime_ServiceBinding" type="tns:WSTime_Service">
        <java:binding/>
    </binding>
    <!--Detailed parameters for the binding: definition of the specific SAP-->
    <service
        name="WSTime_Service">
        <documentation>Trivial example of a Time Web Service</documentation>
        <port
            binding="WSTime_ServiceBinding"
            name="WSTime_ServicePort">
            <java:address class="WSTime"/>
        </port>
    </service>
</definitions>

Figure 7. WSDL definition for a WSTime service described in Section 4
<?xml version="1.0" encoding="UTF8"?>
<wsdl:definitions targetNamespace="http://DefaultNamespace"
xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:impl="http://DefaultNamespaceimpl"
xmlns:intf="http://DefaultNamespace"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
<schema targetNamespace="http://DefaultNamespace"
xmlns="http://www.w3.org/2001/XMLSchema">
<complexType name="ArrayOf_xsd_double">
<complexContent>
<restriction base="soap:Array">
<attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:double[]"/>
</restriction>
</complexContent>
</complexType>
<element name="el0" nillable="true" type="intf:ArrayOf_xsd_double"/>
<element name="el1" nillable="true" type="intf:ArrayOf_xsd_double"/>
<element name="el2" nillable="true" type="intf:ArrayOf_xsd_double"/>
</schema>
</types>
<wsdl:message name="getResultResponse">
<wsdl:part name="getResultResult" type="intf:ArrayOf_xsd_double"/>
</wsdl:message>
<wsdl:message name="getResultRequest">
<wsdl:part name="arg0" type="intf:ArrayOf_xsd_double"/>
<wsdl:part name="arg1" type="intf:ArrayOf_xsd_double"/>
</wsdl:message>
<wsdl:portType name="MatMulPortType">
<wsdl:operation name="getResult">
<wsdl:input message="intf:getResultRequest"/>
<wsdl:output message="intf:getResultResponse"/>
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="JavaBinding" type="intf:MatMulPortType">
<java:binding/>
</wsdl:binding>
<wsdl:binding name="MatMulSoapBinding" type="intf:MatMulPortType">
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="getResult">
<soap:operation soapAction="" style="rpc"/>
<wsdl:input>
<soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
namespace="http://DefaultNamespace" use="encoded"/>
</wsdl:input>
<wsdl:output>
<soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
namespace="http://DefaultNamespace" use="encoded"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="MatMul">
<wsdl:port binding="intf:MatMulSoapBinding" name="MatMulSOAPPort">
<soap:address/>
</wsdl:port>
<wsdl:port name="MatMulJavaPort" binding="intf:JavaBinding">
<java:address class="MatMul"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>

Figure 8. WSDL definition for a MatMul service described in Section 5