Introducing WEDS: a WSRF-based Environment for Distributed Simulation

Technical Report Number UKeS-2004-07

Peter Coveney*, Jamie Vicary†, Jonathan Chin‡, Matt Harvey§

Centre for Computational Science
Department of Chemistry
University College London
Christopher Ingold Laboratories
20 Gordon Street
London WC1H 0AJ
United Kingdom
http://www.chem.ucl.ac.uk/ccs/

October 11, 2004

Abstract

Web services have the potential to radically enhance the ability of researchers to make use of distributed computing resources, but jargon and a plethora of standards make their use almost impossible for the scientist who has no prior experience of the necessary technologies. A powerful and simple WSRF-based middleware scheme is presented, designed to let scientists remotely deploy single or multiple instances of a pre-existing code across multiple resources, and giving steering, visualisation and workflow functionality with only simple modifications to program code.

It is hoped that development and implementation of such a toolkit will be relevant not only to the problem of deploying workstation-class codes in real time, but also move towards more tractable alternatives to the Globus toolkit for deployment of processes in a high-performance computing environment.

1 Background and motivation

Grid computing is distributed computing performed transparently across multiple administrative boundaries. Most so-called "grid projects" in existence today do not fit this definition: either there is no transparency so far as the user is concerned, or no administrative boundaries are involved, or both.

In our work to date [1, 2, 3] we have sought to perform scientific research on existing "grids", specifically the US TeraGrid and various versions of the developing UK Grid (originally the "Level 2 Grid" and now the National Grid Service). While these fulfil the criterion of involving multiple administrative boundaries, none is what a user would call "transparent". In particular, existing middleware obstructs rather than facilitates access to grid resources [4], and none currently exist [5, 6] that handle all aspects of job submission, steering, visualisation and workflow manipulation.

We have therefore, of necessity, been obliged to look for and develop much more usable middleware solutions. Middleware which is "lightweight", both in terms of ease of use ("transparency") and resource footprint [4, 7], is much more attractive than existing heavyweight solutions (all existing Grids we have to work with use Globus Toolkit Version 2, GT2). Early successes [2, 8] in using this – as OGSI::Lite [7] – have led to the support of new research projects aimed (i) at widening the scope of scientists able to use this middleware in order to deploy a wide range of applica-

*p.v.coveney@ucl.ac.uk
†jamie.vicary@mansfield.oxford.ac.uk
‡jonathan.chin@ucl.ac.uk
§m.j.harvey@ucl.ac.uk

tions [9] and (ii) to harden it into a product that is robust, reliable and resilient, under the auspices of the Open Middleware Infrastructure Institute's Managed Programmes [10].

However, in addition to ongoing work in these projects which intends to expose applications themselves as Grid services, scientists require easy and effective methods for launching and interacting with their computations (here assumed to be computer simulations). No effective means for doing this in a flexible way currently exists [4]. It is the purpose of this article to describe a new and "transparent" means for doing this, which can be coupled directly to the WSRF::Lite middleware. The architecture and communications protocols used by WEDS are broadly compatible with the technologies and interfaces embraced by OGSA [11], although it moves beyond OGSA in several ways to accomplish goals which are *technically* but not *realistically* attainable through its direct implementation.

The great power of web services is their flexibility; the term "loosely coupled" is often used and well describes the way a library of web services, in an e-Business or an e-Science context, could be easily connected to run simulations or analysers which automatically exchange their results between each other. Perhaps a molecular dynamics simulation service could feed its output to a Fourier transform service or an autocorrelation function service, which feed their results to a visualisation service, each component in the "pipeline" or "workflow" being essentially unaware of its context. It is the ability to construct such schemes, across platform, architecture and geographical boundaries, where the real power of web services lies, but such a vision will never become a practical reality until enough well-written services exist for such workflows to be constructed.

As so often with emergent technologies, it is a chicken-and-egg situation; services will not appear until there are users to write them, but users will not be prepared to face the steep learning curve necessary to enter the web-services world while so few services exist for them to use as a part of their research. The jargon is specialised and the existing middleware opaque and complex [4]; it is clear that to help resolve this deadlock the learning curve must be flattened, and it is to this end that WEDS is being drawn up.[1]

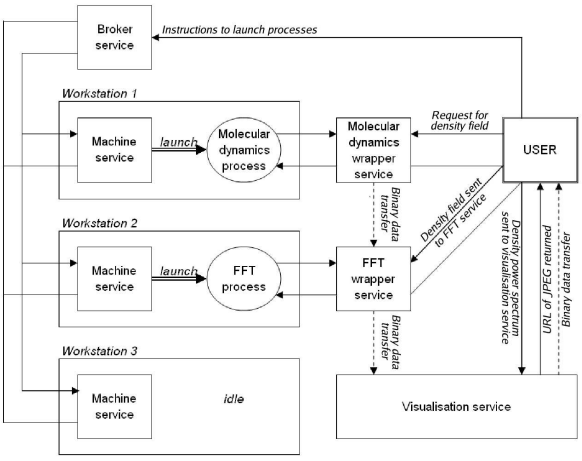

Figure 1: An example WEDS implementation. A user script coordinates a workflow between molecular dynamics, Fourier transform and visualisation services. The binary data moves directly between elements, not via the user's machine, halving bandwidth demands.

This article does not fully address grid computing in that, so far, we have not tried to extend the WEDS environment across multiple administrative boundaries, although it is a next logical step. To do this in an acceptable manner requires WS-Security to be built in, which is now under development in our OMII funded project [10].

2 The WEDS framework

WEDS is built around four key WSRF services, each of which perform different roles. They are described here along with the key methods that they expose to the user, and are depicted in a workflow implementation in Figure 1.

- **Machine services** represent individual machine resources, and must run on the machine that they represent. They make available information such as the specification of the machine, and whether it is currently busy running a simulation. If they receive a command to

[1] This document is intended as a discussion of the reasons for constructing WEDS and an overview of its structure, not a practical guide to implementing it. Such a document is expected to follow in the near future.

start a new simulation, they will fork off a simulation process and initiate a wrapper service to represent it within the WEDS framework.

- A **wrapper service** represents a *single* simulation, either completed or in progress. It runs on any machine which has access to the working directory for the simulation which it represents; this could of course be the simulation machine itself, but such a choice is not ideal as it places extra demand on the simulating CPU. If the steering API (see Section 5) is implemented in the currently active simulation, then commands from the user of the form setValue(*ParameterName*, *NewValue*) and getValue(*ParameterName*) can be used to steer and visualise the simulation respectively. The method getOutputFileList(*Glob*) can be used to return a list of output files created by the program whose file names match a given expression, which may use the wildcards * and ?. These output files can be recovered from the simulation machine to the user's machine via the DataProxy binary file transfer mechanism, or transferred directly to another simulation as a part of a workflow. The piped output from the simulation is made available through the getSTDOUTCookie() and getSTDERRCookie() methods. (For a discussion of cookies and the DataProxy mechanism, see Section 4.)

- The **file service** runs on the user's machine and serves requests for input files (for example, representing initial conditions across an entire area or volume) needed to initiate a simulation, transferring these through the DataProxy file transfer mechanism.

- The **broker service** acts as a registry, linking together all of the above services and storing their contact addresses, called WS-Addresses, which each comprise a URL and a unique identification number. This service is the initial point of contact between the user and the WEDS framework. When a user calls the startSimulation(*SimulationName*, *Arguments*, [*FileServiceEndpoint*, *FileServiceID*]) method, the broker communicates with each of the machine services registered with it and chooses the fastest one available to conduct the simulation, ignoring machines which do not meet the requirements if any have been specified. It passes to the chosen machine service the argument list, and optionally a file service WS-Address if any input files will have to be provided, and returns to the user a small positive integer uniquely identifying the simulation, or 0 if the simulation could not be started. This *SimulationNumber* can be passed to the broker in a future call to the getWrapperWSAddress(*SimulationNumber*) method to discover the location of the wrapper service which represents the simulation.

A fuller description of the methods these services make available is not appropriate in this summary document, but the short descriptions above suffice to make clear the division of labour between the various parts which comprise the WEDS framework. It is worth making clear that even if steering and visualisation support has not been compiled into the code, the calls getSTDOUTCookie() and getOutputFileList(*Glob*='*'), along with the functionality presented by the file service, allow the executable to be deployed to a remote machine, its input files retrieved and its output retrieved.

It is worth emphasising that simulation resources only have a transient existence, as this is not the usual way in which web services are used. A business, for example, may operate a stock level web service which takes as an argument the name of a product and returns the number of items present in the warehouse. Such a service would be expected to exist for a time large compared to the timescale on which goods move in and out of the warehouse, serving many independent requests.

Scientifically, such long-lived services also have a useful role to play. For example, the visualisation service in Figure 1 is presented as a static entity, not being created only when visualisation is required, but preexisting on a machine which need not be running a WEDS machine service. Simulation wrapper resources do not operate in this way; they are created, along with the simulation processes themselves, in response to requests by a user that certain simulations be performed and destroyed automatically after their simulation has been completed. These different modes of operation are appropriate as a visualisation service can often be expected to be able to deal with many requests quickly; unlike individual simulations, it

may not be useful to consider each separate visualisation request as a separate entity. A useful distinction can be made between *resources*, which are transient, and *services* which are not, although these terms are often interchanged.

3 User interaction

Under this framework, an executable can be deployed remotely using a "standard" script as if it were being run locally. The script takes the name of the simulation to execute and the arguments to pass to it, and passes these details to the broker, which selects a machine on which to deploy the process. A file service is also launched on the user's machine, its WS-Address passed to the broker along the other information. The wrapper starts on the remote resource, and attempts to resolve each argument passed as if it were an input file by negotiating with the file service; successfully resolved files are copied to the remote simulation working directory. The simulation is then launched. The script waits for the simulation to complete, transferring standard output from the process as it is produced, and then transfers all of the produced output files back to the user's local working directory.

Such a script is as important a part of the WEDS framework as the methods and services described above – it is very important indeed that this basic remote deployment functionality works "out of the box". More complex uses for WEDS will need to be coordinated with more complex scripts or programs, and these can of course be written in any language for which there exists a SOAP API. An object-oriented representation of a simulation wrapper has been written in Perl, which allows scripts (or users) to start, visualise and control simulations with commands as simple as

```
$g = $sim->getValue("gravity");
```

or

```
$sim->setValue("pressure", 200);
```

which would, respectively, set the variable `$g` to equal the current acceleration due to gravity, or set the pressure in the simulation to 200.

It is the view of the authors that the need for flexible, *easy* interfaces such as these has been overlooked for too long, and that considerations of this kind are actually more important than the design of the resource framework itself as they are the key to the generation of a substantial user base, something which distributed computing is at this stage greatly in need of.

Key to the interaction between WEDS and other WSRF implementations is the WSDL description of the framework. The Web Services Description Language is used as a standard way to define the procedures made available by a web service, along with the data types that they expect and return. No WSDL has yet been written for WEDS, as development has been in Perl, a loosely-typed language that does not need WSDL to operate. One of the possible next stages in the development of WEDS is to produce WSDL documents for the WEDS services, and to demonstrate interoperatation with a range of SOAP implementations, but such work is not required for WEDS to be fully exploited in its current implementation.

4 Transferring binary data

The least tractable problem with using SOAP-mediated web services in e-Science is the unsuitability of XML – a human-readable, text-based protocol – to represent binary data, which very often needs to be transported by the gigabyte for both simulation input and output. Firstly, key to the parsing of XML is the presence of tags such as <data> and <\data> to delimit sections of the document, these being identified by their < and > symbols. If a chunk of binary data contained either of these characters the resulting XML document would very likely not be "well formed", and would consequently not constitute a viable message. One solution to this is to restrict the number of possible bytes from 256 to some lower number, excluding those which might interfere with the syntactic structure of the document, and such base-64 binary encodings[2] can indeed be included in a valid SOAP message. This is, however, less than ideal – a binary message in base-64 is a third longer than the same message in base-256, requiring an equivalently longer transfer time and, more importantly, expensive encoding and decoding at each end. It is not obvious how this pressing problem should be solved [12]; unfortunately, while web services can in

[2] 64, surprisingly, being the largest power of 2 of contiguous ASCII characters which will neither render the XML document invalid, nor be mangled by servers re-encoding the text as the message is enroute.

principle support communication via other protocols, in reality SOAP is the only widely supported format for message exchange.

The solution to this within WEDS is to transmit binary data directly between sockets, outside of the WSRF specification, with which WEDS is otherwise fully compliant. This functionality exists in the form of a library called DataProxy, which handles this for the application. The initial request for the file is made conventionally, by calling a remote method on a file or wrapper service, as described in Section 2. This method locates the necessary file and invokes the DataProxy API to copy it to a temporary directory, giving it a unique name to avoid clashes with other files. This new filename is appended to the URL of the machine on which the file is stored to form a long string, referred to as a cookie. This cookie is returned to the user or service which originally made the call requesting the file. The requestor can now use a DataProxy library function cookieToFile(*Cookie*) on his own machine, to open a connection to a daemon listening on a prearranged port at the URL contained in the cookie. This daemon reads the file and streams it across the socket; the cookieToFile method saves this received file and returns its filename on the local filesystem to the process which called it.

This process, while apparently somewhat involved, is powerful because it enables files to be passed between services via other intermediate services, which do not themselves need to actually download the file to their own machines. Consider an output file produced by a simulation, which a user would like to transfer to a visualisation service. She can obtain the cookie which represents the output file from the wrapper service which represents the simulation, and give this cookie to the visualisation service. The cookie is resolved, and the data contained in the output file streams directly from the wrapper service's machine to the visualisation service's machine. While the *location* of the file must be transmitted via the user who is coordinating the transfer, the actual *data content* of the file need not be, which can speed transfer significantly, especially if the wrapper and visualisation services are running at the same location distinct from that of the user.

5 Steering and visualisation

A steering and visualisation API has been written in C, allowing simulation code to easily import and export data in the well-established XDR [13] format. A comprehensive RealityGrid steering and visualisation API already exists [14, 15], but it was felt that its complexity and its implementation, based around OGSI rather than WSRF, made it unsuitable for incorporation into WEDS.

The underlying mechanism for the WEDS steering API is file-based. Such an implementation has advantages, both in speed and in its capacity for asynchronous transfer of large datasets, but also has limitations; most obviously, the wrapper service as described in Section 2 must have access to the simulation working directory. The client side interface has been designed so that a possible future socket-based development of the protocol could expose the same methods as the current API, ensuring compatibility between current and future versions.

The data transferred consists of an XDR binary stream with an ASCII header, which describes the content of the file and the dimensionality and type of the data it contains. It is data of this type which is transferred from program to program to operate a workflow. Tools have been written to convert files between this WEDS format and others which are more widely used, such as HDF5, VTK and plain XDR.

6 Overview

Grids can exist in different scales, shapes and sizes. The computational intensity involved in maintaining information about resource state and deciding where jobs should be deployed is the key measure by which the *complexity* of the grid can by understood. WEDS is suitable for low-complexity grids, for which a single broker process can manage all resource registration and job submission tasks; a volume of around 30 simulations deployed per minute would be a feasible upper bound on broker performance, which is not insubstantial. A much more complex grid, on which thousands of jobs might be scheduled, would not be suitable for construction with the framework presented here.

A framework of WSRF-compliant services has

been written which allows an existing, local workstation-class code to be deployed in a large-scale, taskfarmed environment with no modification at all, and steered and visualised with minimal modification involving calls to the WEDS steering and visualisation API, currently implemented in Perl. The services-oriented architecture allows scientists to easily link together multiple programs in a "pipeline" or "workflow" fashion, with a minimal learning curve.

Such a services-oriented architecture as is presented here could be developed into a framework to deploy not only workstation-class codes, but high-performance supercomputing-class codes, and to organise the flow of data between elements which is so crucial in large computing projects.

7 Acknowledgements

We thank EPSRC for the RealityGrid grant GR/R67699 which funded this work, and Mark McKeown, Stephen Pickles and Dave Berry for stimulating discussions.

References

[1] *The RealityGrid Project*
`http://www.realitygrid.org/`

[2] Jonathan Chin, Jens Harting and Peter Coveney *The TeraGyroid project – collaborative steering and visualisation in an HPC Grid for modelling complex fluids*
`http://www.allhands.org.uk/proceedings/papers/181.pdf`

[3] *The TeraGyroid Project*
`http://www.realitygrid.org/teragyroid-index.html`

[4] Jonathan Chin and Peter Coveney *Towards tractable toolkits for the Grid: a plea for lightweight, usable middleware* UK e-Science Technical Report, number UKeS-2004-01
`http://www.nesc.ac.uk/technical_papers/UKeS-2004-01.pdf`

[5] *Condor Project Homepage*
`http://www.cs.wisc.edu/condor/`

[6] *The Internet Backplane Protocol*
navigate from `http://loci.cs.utk.edu/`

[7] Mark McKeown *WSRF::Lite and OGSI::Lite*
`http://www.sve.man.ac.uk/Research/AtoZ/ILCT`

[8] Jonathan Chin *Adventures with LB2D and OGSI::Lite*
`http://the.earth.li/~jon/work/ogsi/writeup/`

[9] *Rapid prototyping of usable grid middleware* (EPSRC Grant GR/T27488)

[10] Peter Coveney (PI) and Stephen Pickles *Robust Hosting of WSRF::Lite* (OMII funded research project)
`http://www.omii.ac.uk/`

[11] The Globus Alliance *Towards Open Grid Services Architecture*
`http://www.globus.org/ogsa/`

[12] Adam Bosworth et al. *XML, SOAP, and Binary Data*
search from `http://search.microsoft.com/`

[13] R. Srinivasan *RFC 1832 - XDR: External Data Representation Standard*
obtain from `http://www.faqs.org/rfcs/`

[14] Stephen Pickles, Robin Pinning, Andrew Porter et al *The RealityGrid Computational Steering API*
`http://www.sve.man.ac.uk/Research/AtoZ/RealityGrid/Steering/ReG_steering_api.pdf`

[15] Stephen Pickles, Robert Haines, Robin Pinning and Andrew Porter *Practical Tools for Computational Steering*
`http://www.allhands.org.uk/proceedings/papers/201.pdf`