

Performance Analysis of a Multi-Physics Simulation System Based on Web Services*

Paul Stodghill, Rob Cronin, Keshav Pingali
Dept. of Computer Science
Cornell University

Gerd Heber
Cornell Theory Center
Cornell University

Abstract

The ongoing convergence of grid computing and web services has inspired a number of studies on the use of SOAP-based web services for scientific computing. These studies have exposed several performance problems in using SOAP-based communication; to eliminate these bottlenecks, extensions to the SOAP standard and sophisticated implementation strategies have been proposed. In this paper, we will describe the ASP system, a simulation testbed based on web services for simulating multi-physics, coupled fluid/thermal/mechanical/fracture problems. The system is organized as a collection of geographically-distributed software components in which each component provides a web service, and uses standard SOAP-based web service protocols to interact with other components. There are a number of advantages to organizing a system in this way, which we discuss. We have analyzed the performance of our system for several applications and a number of problem sizes and have found that the overhead for using SOAP-based web services is small and tends to decrease as the problem size increases. Our results suggest that the previously identified potential bottlenecks may not be major issues in practice, and that a standards-compliant implementation like ours can deliver excellent scalable performance even on tightly-coupled problems, provided web services are used judiciously.

1 Introduction

Grid computing is being used for a restricted class of applications, such as problems that require a large number of small, independent tasks, or problems that access remote instruments [14]. The majority of computational science applications however do not fall into these categories. Most of these applications are not embarrassingly parallel, so they

cannot be decomposed into tasks that execute independently on a computational grid. In addition, most of them do not require on-line interaction with instruments or other data sources.

Nevertheless, we believe that the metaphor of grid computing is useful for implementing large-scale, *loosely-coupled* computational science applications. To appreciate this point, it is useful to consider how these applications are usually created. Almost invariably, large applications are created by a multi-institutional team, whose members contribute legacy and new modules to the project. Modules from different members may be written in different programming languages and developed for different computing platforms. Since re-implementing all the software in a single programming language is not practical, all the code must be ported to a single high-performance computing platform so that these modules can inter-operate with each other.

Building a monolithic application in this way has several disadvantages. Porting code from one platform to another takes time and effort. Moreover, thorny intellectual property (IP) issues may arise if the common platform is at a remote institution. Even if these problems are overcome, the contributed code modules are usually under continuous development, so the process of porting code to the common platform may need to be repeated every time there is a new release of these modules.

In principle, these problems can be avoided by designing the system as a collection of distributed components that interact by using a mechanism like remote procedure call (RPC). Each site maintains its own code on whatever platform the code was developed on, but it provides a server that can be invoked from remote sites to access the functionality of that code. Instead of exporting code, each site therefore exports only the functionality of the code, thereby implementing a *write once, run from anywhere* philosophy. The distributed systems community in particular has explored RPC mechanisms extensively, and there are many standards and implementations such as Sun RPC [25] and the Java RMI [28].

*This research is partially supported by NSF grants EIA-9726388, EIA-9972853, and ACIR-0085969.

Although RPC has been around for two decades, this architecture is used by few if any computational science applications. The conventional wisdom about why this is so is summarized by the following points.

1. There is no RPC standard supported by all vendors, so interoperability is a problem.
2. The basic RPC mechanism was intended for a stateless, service-oriented architecture in which the service is relatively light-weight, and the client and server exchange only a small amount of data. As a consequence, most RPC standards and implementations have features that made them unsuitable for use in computational science programs. For example, many RPC implementations use UDP for data transport, which restricts RPC calls to 8KB of data. This is not acceptable for computational science programs which may need to exchange data sets that may be many megabytes or gigabytes in size.
3. Similarly, in most RPC implementations, a client is required to block after making a remote request, until it receives a response from the remote server. This is fine if the service is light-weight, but if the component that is invoked takes many minutes or hours to produce a result, most RPC implementations will time-out and assume that the remote server has crashed. An asynchronous interaction mechanism in which notification of completion of remote requests is decoupled from the request itself would address the problem, but this requires a stateful message-exchange paradigm.
4. Perhaps the most important issue is the overhead of data transfer between distributed components. Two procedures in the same program can exchange data by passing pointers to data structures, which is a very low-cost operation. If the two procedures are in components at different sites on the Internet, exchanging data is a far more elaborate and expensive operation - the calling component must linearize the data structure, convert it to some common data exchange format like XDR and transmit it to the remote site which reverses this process to rebuild the data structure.

Because of recent developments in the area of web services for business applications, this conventional wisdom must be re-examined. To support seamless application-to-application communication in a decentralized, distributed environment, the web services community has defined the Standard Object Access Protocol (SOAP) which can be viewed as a “protocol specification that defines a uniform way of passing XML-encoded data” [17]. While SOAP can be used to implement many kinds of interactions between applications, the SOAP standard also specifies a protocol for performing RPC’s, using HTTP as the underlying communication protocol. Most computer vendors are com-

mitted to supporting this standard, which addresses the first problem discussed above.

Nevertheless, like existing RPC implementations, SOAP is intended for light-weight services that exchange small amounts of data. Although the amount of data that can be passed in a SOAP message is implementation-dependent, our experiments show that it is at most a few megabytes on all implementations we have looked at. Moreover, SOAP is “fundamentally a stateless message-exchange paradigm” [17], so it does not directly support the stateful interaction paradigm that is better suited for computational science applications as described above.

To address these concerns, we have implemented a system, called O’SOP, that is layered on top of SOAP and is described in Section 2. It permits asynchronous client-server interactions in which arbitrarily large amounts of data can be exchanged. A particularly useful feature of O’SOP is that it permits legacy command-line-oriented applications to be deployed as web-services without any modification. We believe that O’SOP addresses the second and third problems with conventional RPC’s described above.

The final problem that must be addressed is the overhead of data exchange between distributed components using SOAP and XML. Two previous studies of this issue that appeared in HPDC’02 reported that the use of SOAP and XML imposed a large performance penalty in scientific applications, and concluded that SOAP was not practical for computational science applications unless a number of sophisticated optimizations and changes to the protocols were made [10, 24].

We argue in this paper that these studies are misleading. In Section 3, we describe three large computational science applications that we have implemented using our infrastructure. Two of these applications are coupled fluid-thermal-mechanical computational fracture simulations, while one is a simulation of blunt trauma on human brains. In Section 4, we describe performance results that show that the overhead of using O’SOP based distributed components to implement these applications is small. To the best of our knowledge, this is the first performance evaluation of entire state-of-the-art scientific applications built using the web-service framework.

In Section 5 we discuss other related work. Finally, in Section 6, we highlight lessons that we have learned from this implementation.

2 O’SOP

O’SOP [26] is an O’Caml [23] base, web services framework that we have developed for distributed computational science applications. The primary benefits of O’SOP over other frameworks are its support for legacy scientific applications and the manner in which it builds

upon the basic SOAP protocol to enable efficient interactions between distributed scientific components.

2.1 Deploying Applications as Distributed Components

On the server side, O'SOAP enables existing, command-line oriented applications to be made into web services without any modification. The user only needs to write a small CGI script that calls O'SOAP server-side applications. Placed in the appropriate directory on a web server, this script will execute when the client accesses its URL. An example of such a script is shown in Figure 1.

```
#!/bin/bash

oids_server \
  -n arithmetic-test -U urn:test \
  -N 'Arithmetic Server' \
  -- ./add.sh '[in val x:int]' \
    '[in val y:int]' \
    '>' '[out file result:int]'
```

Figure 1. Sample O'SOAP Server

The `oids_server` program, which is provided by the O'SOAP framework, processes the client's SOAP request. The `-n`, `-N`, and `-U` parameters specify the short name, full name, and namespace, respectively, of the web service. What appears after `--` is a template of the command line that is to be used to run the legacy program, `add.sh`. The text that appears within `[. . .]` describes the arguments to the legacy program. Each argument specification includes at least four properties,

- The directionality of the parameter, i.e., “in”, “out”, or “in_out”.
- Whether the parameter value should appear directly on the command line (“val”) or whether the parameter value should be placed in a file whose name appears on the command line (“file”).
- The name of the parameter, e.g., “x”, “y” and “result”.
- The type of the parameter value, i.e., “int”, “float”, “string”, “raw” (arbitrary binary file), “xml” (a structured XML file).

A component implemented using O'SOAP will expose a number of methods, discussed below, that can be invoked using the SOAP protocol. The component also provides a means for generating a WSDL [11] document that describes these methods, their arguments, and additional binding information.

On the client-side, O'SOAP provides two tools for accessing remote web services. The `osoap_tool` program provides a command-line interface to remote web services.

In addition, the `wSDL2m1` program generates stub code for invoking web services from O'Cam1 programs.

To summarize, O'SOAP is a framework that hides most of the details of the SOAP protocol from the client and server programs. With this in place, we can now discuss how the interactions between the clients and servers can be organized to support distributed computational science applications.

2.2 Asynchronous interactions

The SOAP protocol was designed for synchronous client-server interactions. That is, the client sends a SOAP request to the server and then waits to receive a SOAP response¹. However, many computational science applications can take a very long time to execute. Using the synchronous interaction model directly in this case is often not possible. For instance, many SOAP clients will signal an error if a response is not received within a fixed timeout interval. While it might be possible to increase this timeout interval, a better approach is to use an asynchronous interaction model.

O'SOAP's server-side programs provide basic job management by exposing a number of methods to the client. The “spawn” method invokes the application on the server and returns a job id to the client. The client can then pass this job id as the argument to the “running” method to discover whether or not the application has finished execution. Once completed, the client uses the “results” method to retrieve the results. There are additional methods, such as “kill”, for remotely managing the application process.

Since the server is able to generate a response for these methods almost immediately, the synchronous SOAP protocol can be used for such method invocations. Also, since a new network connection is established for each method invocation, detached execution and fault recovery are possible without additional system support (e.g., to re-establish network connections).

2.3 Support for small and large data sizes

Data set sized in computational science applications can vary greatly. For example, for the Fluid/Thermal solver in the Pipe problem described in Section 3, the input boundary conditions are a few kilobytes, but the results of the solver can be tens of megabytes. For small data sets, it makes sense to include the data within the SOAP envelope that is passed between the client and the server. This eliminates the need for a second round of communication to retrieve the data.

However, there are several reasons why embedding large data sets in SOAP envelopes is problematic. One reason

¹Other modes of interaction were defined by the SOAP 1.1 Specification, but were dropped in SOAP 1.2.

that has been observed by others [10, 24] is that translating binary data into ASCII for inclusion in the SOAP envelope can add a large overhead to a system. The second reason is that many SOAP implementations have preset limits on the size of SOAP envelopes. Many of our data sets exceed these limits.

For these reasons, O'SOAP enables data sets to be optionally separated from the SOAP request and response envelopes. If a data set is included, it is encoded using XML or Base64. We call this case "pass by value". If it is not included, then a URL to the data set is included instead. We call this "pass by reference". Furthermore, O'SOAP enables clients and servers to dynamically specify whether a data set will be passed by value or reference.

O'SOAP currently supports the HTTP, FTP, and SMTP protocols, and we have plans to provide support for IBP [22].

3 Application Description

In this section, we give a high-level description of the three large-scale, distributed, computational science, simulations we have implemented in the Adaptive Software Project (ASP) [18].

3.1 Pipe Problem

This application simulates an idealized segment of a rocket engine modeled after actual NASA experimental spacecraft hardware. The object is a curved, cooled pipe segment that transmits a chemically-reacting, high-pressure, high-velocity gas through the inner, large diameter passage, and a cooling fluid through the outer array of smaller diameter passages. The curve in the pipe segment causes a non-uniform flow field that creates steady-state but non-uniform temperature and pressure distributions on the inner passage surface. These temperature and pressure distributions couple with non-uniform thermomechanical stress and deformation fields within the pipe segment. In turn, the thermomechanical fields act on an initial crack-like defect in the pipe wall, causing this defect to propagate.

The workflow for the Pipe simulation is shown in Figure 2. The components of our system appear like this, and the intermediate data sets appear like «this».

In order to enhance interoperability, we have established a set of common, XML-based [31], file formats for some of our data sets. These formats are described elsewhere [7, 9].

Some of the components used in the Pipe Problem are the following.

- The *Surface Mesher* produces triangular surface meshes for each of the model's geometric surfaces.

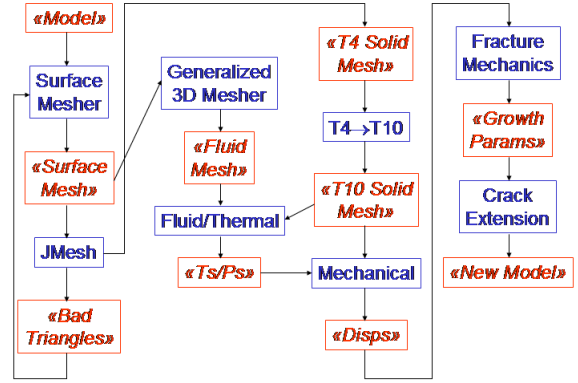


Figure 2. Workflow for the Pipe Problem

This component produces surface meshes with certain quality guarantees [6].

- *JMesh* [3] generates unstructured tetrahedral meshes for arbitrarily shaped three-dimensional regions, and was designed to handle the unique geometric problems that occur in Fracture Mechanics.
- If the surface mesh is too coarse to allow a quality volume mesh to be produced, *JMesh* will produce a list of surface mesh triangles that require refinement. This list is passed back to the *Surface Mesher*, which then passes a new surface mesh to *JMesh*, etc. The loop between the *Surface Mesher* and *JMesh* components for automatically and adaptively producing surface and volume meshes will be referred to as the *Meshing Loop*.
- The *Generalized 3D Mesher* [5, 4] generates high quality meshes consisting of extruded triangular prisms, tetrahedral elements, and generalized prisms. These highly anisotropic elements are required for simulating viscous fluid flows required in regions near no-slip boundaries, i.e., boundary layers.
- The *Fluid/Thermal Solver* is based upon the CHEM code [19, 20], which simulates 3D chemically reacting flows of thermally-perfect, calorically-imperfect gases.

3.2 Single Element Injector

The Single Element Injector (SEI) Problem models a component of a next-generation rocket engine in which oxygen and hydrogen are combined and ignited. One very important aspect of the geometry of this problem is the tip of the post at which oxygen and hydrogen first combine. The design of this post is critical to the performance and robustness of the SEI component. One challenge in simulating the SEI is to accurately model the way in which the flame

adheres to the surface of the post.

The workflow of the SEI simulation is shown in Figure 3. One way in which the SEI simulation differs from the Pipe simulation is that two different surface meshes are generated for the SEI simulation. This creates the need for an interpolation component for conveying boundary conditions between the Fluid/Thermal and Mechanical solvers. Another difference is that, because the SEI model is axially symmetric, the Fluid/Thermal solve is done on a small wedge of the SEI model.

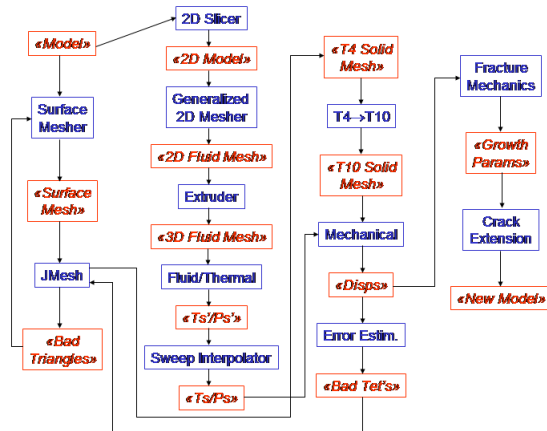


Figure 3. Workflow for the SEI Problem

Here is a brief description of some of the components used in the SEI Problem that differ from those used in the Pipe Problem

- The *2D Slicer* component produces a 2D model representing a slice along the SEI's axis of symmetry.
- The *Extruder* produces a 3D mesh from a 2D mesh by rotating the mesh a few degrees along the axis of symmetry. The resulting 3D mesh corresponds to a pie-shaped wedge of the original SEI model.
- Since the Fluid/Thermal solver computes temperatures and pressures for only a small wedge of the original model, these results have to be interpolated back to the full SEI model. The *Sweep Interpolator* component performs this calculation.
- The *Error Estimator* component is used to evaluate the quality of the displacements produced by the Mechanical Solver. If the quality is poor, the Error Estimator will produce a list of tetrahedra that require refinement. This list is passed back to the Meshing Loop, which is rerun to generate new and finer surface and volume meshes. This process is repeated until the quality of the solution is acceptable.

3.3 Brain Trauma

The third application that we have developed is a simulation of blunt force trauma on a skull and brain. Like the previous two problems, this involves a coupled fluid/mechanical/fracture simulation. Unlike the previous two simulations, contact between the brain and the skull can occur, so explicit dynamics are modeled.

The workflow of the Brain simulation is shown in Figure 4. Because of the explicit nature of the solution, the solver component is different than those used for the Pipe and SEI Problems; however the JMesh component is the same.

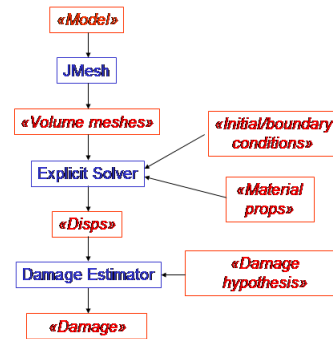


Figure 4. Workflow for the Brain Problem

Here is a brief description of some of the components used in the Brain Problem that differ from those used in the Pipe and SEI Problems

- The *Explicit Solver* component currently only implements a structural mechanics model. In other words, the cerebral fluid is modeled as a solid and not a fluid. This component shares some code with the Mechanical Solver used for the Pipe and SEI problems, but because it is an explicit solver, it is implemented as a separate component.

4 Performance Experiments

This section describes performance results and analysis for the Pipe Problem. Performance results for the SEI Problem and the Brain Problem will be included in the final version of this paper.

4.1 Experimental setup

The following machines were used for the experiments below,

- The *ASP* cluster is housed in the Cornell Computer Science department and consists of 5 Dell PowerEdge 1650's, each with Pentium III's at 1.26GHz (1 dual and 4 single). Each node has 512MB-1GB RAM and runs Red Hat Linux 8.0.
- Web services at Mississippi State University, or *MSU*, were executed on an IBM x330 server, with dual 1.266GHz Intel Pentium III CPUs and 1.25GB RAM running Red Hat Linux version 7.3.
- The machine used at the University of Alabama at Birmingham, or *UAB*, is an IBM x335, with dual 2.4GHz Xeon and 2GB RAM, and runs Red Hat Linux release 7.3.

Except where noted, the components used in these experiments were deployed on the *ASP* cluster.

We used the adaptive Meshing Loop discussed in Section 3 to generate three different problem sizes for the Pipe Problem to understand how increasing the problem size changes the performance of our system. The sizes of the meshes for the solid and interior volumes of the Pipe, generated by JMesh and the Generalized Mesher respectively, are shown in Table 1.

The clients used in these experiments were all developed using O'SOAP. Except for the Generalized Mesher, all of the components used in these experiments were deployed using the O'SOAP framework. The Generalized Mesher was deployed using SOAP::Clean [27, 8], a Perl-based ancestor of O'SOAP.

4.2 Performance Results

Table 2 shows the running time for all of the components up to and including the Fluid/Thermal solver. The Mechanical Solver, which is the next component, is the only component in our system that must be executed via a batch queue. Currently, it is impossible for us to measure the running time of the Mechanical Solver without including the time spent in the batch queue, so we have not included its runtimes.

The column labeled "Local runtime" shows the running time in seconds of each component when it is executed directly on the server, without using the web services infrastructure. The overheads are measured relative to these times. The columns labeled "Intra-campus runtime" and "Inter-state runtime" show the running times when the client is run on different machines than the components. The "Intra-campus" client runs on a machine on the same LAN as the *ASP* server, and the "Inter-state" client runs on a machine at *UAB*, roughly 1000 miles away.

Each row shows the running times for the individual components, The row marked "Download" shows the time taken to download the results from the server after all of the computations have completed. This operation is not

performed for the "Local" client. The row marked "Total" shows the aggregate results for the entire run.

For the final version of the paper, we will include results for the SEI and Brain Problem. Since these problems use many of the same components as the Pipe Problem, we expect the results to be similar. The one major difference that we expect involves the Explicit Solver component. Since this component runs significantly longer than either of the other two solvers, we expect that the overheads for the Brain Problem will be significantly less than either Pipe or SEI.

4.3 Performance Analysis

There are a number of interesting points in the performance results of Table 2 which we now discuss.

Consider the Meshing Loop and Fluid/Thermal components. Notice that for both clients the overhead for the Fluid/Thermal component consistently decreases over the range of problem sizes, while the overhead for the Meshing Loop components does not exhibit a consistent trend.

This difference can be explained by the components' architectures. The Fluid/Thermal Solver is a single component that runs for a relatively long time. There is a cost for invoking the solver using web services, but this cost is small relative to its total running time.

On the other hand, recall that the Meshing Loop is, in fact, two components, the Surface Mesher and JMesh, that are successively invoked until suitable meshes are produced. To produce the largest problem size, 18 separate invocations of the Surface Mesher and JMesh are required. Since the running time of each invocation is relatively short, the relative cost of the component invocations is larger.

Another difference worth noting involves the Generalized Mesher. Since this is the only component not hosted on the *ASP* cluster at Cornell, the "Local" runtimes are actually the time to perform the web service invocation between the *ASP* and *MSU* clusters. The "Local" and "Intra-campus" runtimes are within a few seconds of one another, but the "Inter-state" runtimes are measurable less. One explanation is that, since they are geographically closer together, there is less latency between the "Inter-state" client, which is running at *UAB*, and the Generalized Mesher at *MSU*.

Overall, the total overhead for both clients falls as the problem size increases. The overhead for the largest problem size is 3.2% and 0.1% for the "Intra-campus" and "Inter-state" clients, respectively. These results are in marked contrast to the studies in the literature [10, 24] that concluded that the use of SOAP and XML adds enormous overhead to computational science applications.

The explanation is the following. The previous studies measured the overhead of using web services to execute matrix-multiplication and other small kernels. In addition, the problem sizes used were very small. Therefore

Problem Size	Solid Mesh			Interior Mesh		
	vertices	triangles	tet's	vertices	tri's/quad's	tet's/prisms
1	4,835	4,979	22,045	19,242	3,065	38,220
2	16,832	10,322	83,609	41,216	5,232	85,183
3	54,849	21,127	289,500	79,407	9,074	170,179

Table 1. Pipe Problem Sizes

Size	Component	Local	Intra-campus		Inter-state	
		runtime (secs.)	runtime (secs.)	overhead	runtime (secs.)	overhead
1	Meshing Loop	228.62	250.80	9.70%	247.80	8.39%
	Generalized Mesher	40.96	44.63	8.96%	35.75	-12.72%
	T4 to T10	18.56	21.49	15.79%	20.67	11.37%
	Fluid/Thermal	1342.75	1401.73	4.39%	1390.02	3.52%
	Download	n.a.	0.79		1.00	
	Total	1630.89	1719.44	5.43%	1695.24	3.95%
2	Meshing Loop	813.88	884.95	8.73%	884.93	8.73%
	Generalized Mesher	79.99	86.01	7.53%	69.38	-13.26%
	T4 to T10	62.88	70.52	12.15%	73.07	16.21%
	Fluid/Thermal	4636.91	4734.15	2.10%	4715.69	1.70%
	Download	n.a.	0.92		2.71	
	Total	5593.66	5776.55	3.27%	5745.78	2.72%
3	Meshing Loop	2622.24	3234.93	23.37%	2699.85	2.96%
	Generalized Mesher	208.45	207.55	-0.43%	188.59	-9.53%
	T4 to T10	689.04	648.53	-5.88%	634.94	-7.85%
	Fluid/Thermal	18683.00	18808.16	0.67%	18690.11	0.04%
	Download	n.a.	2.22		9.00	
	Total	22202.73	22901.39	3.15%	22222.49	0.09%

Table 2. Pipe Problem Runtimes

the amount of computation was small relative to the amount of communication, and overheads were magnified dramatically. Our measured overheads are small because our components perform non-trivial computations like mesh generation, solving linear equations, etc. As Table 2 shows, most of the running time of our system is taken by the execution of the Fluid/Thermal Solver. Although the execution of this component may involve a large number of messages being exchanged between processors, all of these processors are part of a single cluster, and it is done using MPI [30], a message-passing library designed for this purpose.

Our conclusion is that the organization of a distributed simulation system makes more of a difference to its performance than the underlying web services infrastructure. We believe few applications will need to perform matrix multiplication or solve linear equations on several machines across the Internet. On the other hand, there is a growing need for infrastructures to build virtual organizations in which the code of different project partners can inter-operate. We believe most of these situations will be similar to ours - the modules contributed by different project partners will have some components that do non-trivial amounts of computation and internal communication - so a SOAP/XML-based infrastructure like O'SOAP is eminently practical.

5 Related Work

Perhaps the most widely know paradigm for distributed scientific computing is Grid Computing [12], and the most widely known grid system is the Globus Toolkit [13]. The Open Grid Services Infrastructure (OGSI) specification [29] and WS-Resource Framework (WSRF) proposed specifications [16] build upon the SOAP protocol to define additional protocols that are useful for distributed computing, such as resource management, event notification, etc. The functionality defined by OGSI/WSRF and O'SOAP is largely orthogonal, and we would expect our results to be similar if our components were deployed within either of these frameworks.

WS-Context [2] provides a mechanism for correlating SOAP messages over time. This can be used to implement stateful interactions, like transactions. Context information roughly corresponds to the job id's that are used by O'SOAP servers. OGSI and WSRF provide alternative mechanisms for identifying state.

Ninf [21] and NetSolve [1] are intended to allow existing numerical *libraries* to be executed remotely, while O'SOAP and the other elements of our infrastructure are intended to allow existing *applications* to be executed remotely. As a result, the type systems are different. For example, both Ninf and NetSolve provide array and subarray types, while O'SOAP provides simple scalars and arbitrary binary and

XML files.

6 Conclusions

We have described a multi-physics simulation testbed that consists of a loosely coupled set of distributed components implemented using a web services framework called O'SOAP which is based on SOAP/XML. To the best of our knowledge, this is the first system of its kind. This testbed has enabled us to develop state-of-the-art simulations without having to port codes between each other's machines. This approach has given us a number of development and software maintenance benefits.

We have also described a set of performance experiments of our system. To the best of our knowledge, this is the first such performance analysis of a web services or grid based simulation system that employs many components. Our results suggest that even a simple and standard-compliant web services infrastructure, such as O'SOAP, can be used directly in high performance distributed scientific computing without introducing performance bottlenecks. In fact, we observe that for larger problem sizes, the overhead of using distributed components is essentially negligible.

We believe that our work provides a number of important lessons for other researchers. First, with this sort of infrastructure, it is possible for multi-institutional, multi-disciplinary computational science projects to establish virtual organizations, as envisioned in [15], and build efficient, distributed, component-based applications. This is possible even with basic web services protocols, let alone the more recent OGSI or WSRF protocols .

Second, in order to achieve reasonable performance from a distributed simulation system, it is important to carefully chose the functionality that goes into each of its components. This is illustrated by the overheads that we observed for the Meshing Loop and Fluid/Thermal components. Loosely coupled codes that communicate infrequently can be placed in separate components, while tightly coupled codes should almost certainly be placed within the same component. For many applications, individual sites will provide enough resources to do matrix multiplication or solve large systems of linear equations, so the role of web services in such projects is to make it possible for large codes to inter-operate with minimal coordination and re-implementation.

We believe that this sort of decomposition is a natural result of, not only our physical problem, but of the fact that we are a multi-disciplinary project. In such a project, each member has a clearly defined research area, and the components seem to natural divide themselves along these lines. Put differently, our components are loosely coupled because our project members are! We expect that this will be true of most other multi-disciplinary projects, and we believe that

web services may be appropriate for many of these as well.

References

- [1] D. C. Arnold and J. Dongarra. The netsolve environment: Progressing towards the seamless grid. In *2000 International Conference on Parallel Processing (ICPP-2000)*, Toronto, Canada, August 21-24 2000.
- [2] D. Bunting, M. Chapman, O. Hurley, M. Little, J. Mischinsky, E. Newcomer, J. Webber, and K. Swenson. Web services context (ws-context) ver1.0. Available at <http://developers.sun.com/techtoc/web/services/wscaf/wsctx.pdf>, July 28 2003.
- [3] J. Cavalcante-Neto, P. Wawrzynek, M. Carvalho, L. Martha, and A. Ingrassia. An algorithm for three-dimensional mesh generation for arbitrary regions with cracks. *Engineering with Computers*, 17:75–91, 2001.
- [4] S. Chalasani and D. Thompson. Quality improvements in extruded meshes using topologically adaptive generalized elements. *International Journal for Numerical Methods in Engineering*, (submitted).
- [5] S. Chalasani, D. Thompson, and B. Soni. Topological adaptivity for mesh quality improvement. In *Proceedings of the 8th International Conference on Numerical Grid Generation in Computational Field Simulations*, Honolulu, HI, June 2002.
- [6] L. P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the Ninth Symposium on Computational Geometry*, pages 274–280. ACM Press, 1993.
- [7] L. P. Chew, S. Vavasis, S. Gopalsamy, T. Yu, and B. Soni. A concise representation of geometry suitable for mesh generation. In *Proceedings, 11th International Meshing Roundtable*, pages pp.275–284, Ithaca, New York, USA, September 15-18 2002.
- [8] P. Chew, N. Chrisohoides, S. Gopalsamy, G. Heber, T. Ingrassia, E. Luke, J. Neto, K. Pingali, A. Shih, B. Soni, P. Stodghill, D. Thompson, S. Vavasis, and P. Wawrzynek. Computational science simulations based on web services. In *International Conference on Computational Science 2003*, June 2003.
- [9] P. Chew and S. Vavasis. Proposal for mesh representation. Internal draft, January 21 2003. Accessed February 13, 2003.
- [10] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of soap performance for scientific computing. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, July 2002.
- [11] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. Available at <http://www.w3.org/TR/wsdl>, March 15 2001.
- [12] G. G. Forum. Global Grid Forum home page. Accessed February 13, 2003.
- [13] I. Foster and C. Kesselman. The globus project: A status report. In *IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.
- [14] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, second edition, 2004.
- [15] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001.
- [16] Globus Alliance. The ws-resource framework. Available at <http://www.globus.org/wsrf/>, January 24 2004.
- [17] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. Soap version 1.2 part 1: Messaging framework. Available at <http://www.w3.org/TR/SOAP/>, June 24 2003.
- [18] The itr/acs adaptive software project for field-driven simulation. Available at <http://www.asp.cornell.edu/>.
- [19] E. A. Luke. *A Rule-Based Specification System for Computational Fluid Dynamics*. PhD thesis, Mississippi State University, 1999.
- [20] E. A. Luke, X. Tong, J. Wu, L. Tang, and P. Cinnella. A step towards “shape-shifting” algorithms: Reacting flow simulations using generalized grids. In *Proceedings of the 39th AIAA Aerospace Sciences Meeting and Exhibit*. AIAA, January 2001. AIAA-2001-0897.
- [21] H. Nakada, M. Sato, and S. Sekiguchi. Design and implementations of ninf: towards a global computing infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 15(5-6):649–658, 1999.
- [22] J. S. Plank, M. Beck, W. R. Elwasif, T. Moore, M. Swamy, and R. Wolski. The internet backplane protocol: Storage in the network. In *NetStore99: The Network Storage Symposium*, Seattle, WA, USA, 1999.
- [23] D. Rémy and J. Vouillon. Objective ml: An effective object-oriented extension to ml. In *Theory And Practice of Objects Systems*, 4(1):27–50, 1998.
- [24] S. Shirasuna, H. Nakada, S. Matsuoka, and S. Sekiguchi. Evaluating web services based implementations of gridrpc. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, 2002.
- [25] R. Srinivasan. Rpc: Remote procedure call protocol specification version 2. IETF RFC 1831, Aug. 1995.
- [26] P. Stodghill. O'soap - a web services framework in o'caml. <http://www.asp.cornell.edu/osoap/>.
- [27] P. Stodghill. SOAP::Clean, a Perl module for exposing legacy applications as web services. Accessed February 11, 2003.
- [28] Sun Microsystems. Java rmi specification. Available at <http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html>.
- [29] S. Tuecke et al. Open grid services infrastructure (ogsi) version 1.0. Available at https://forge.gridforum.org/projects/ogsi-wg/document/Final_OGSI_Specification_v1.0/en/1, June 27 2003.
- [30] D. W. Walker and J. J. Dongarra. MPI: a standard Message Passing Interface. *Supercomputer*, 12(1):56–68, 1996.
- [31] World Wide Web Consortium. Extensible markup language (xml) 1.0 (second edition). W3C Recommendation, October 6 2000.