A Distributed System Based on Web Services for Computational Science Simulations

Keshav Pingali and Paul Stodghill
Department of Computer Science
Cornell University
Ithaca, NY 14853 USA
{pingali,stodghil}@cornell.edu

ABSTRACT

In this paper, we describe the ASP system, a testbed based on Web Services for coupled multi-physics simulations. 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 a typical application and for 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 potential performance bottlenecks identified in the literature may not be major issues in practice, and that a standards-compliant implementation like ours can delivery excellent scalable performance even on coupled problems, provided Web Services are used judiciously.

1. INTRODUCTION

The term *grid computing* is used to describe a wide range of applications including applications that require a large number of small, independent tasks as well as applications that access remote instruments within a federated environment [20].

Most computational science applications do not fall into these categories since they are not embarrassingly parallel, and also do not require on-line interaction with instruments or other data sources. Nevertheless, we believe that grid computing is useful for implementing a subset of these applications that perform large-scale, *loosely-coupled* computations. 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 both legacy code modules and new modules to the project. Modules from different team members may be written in different programming languages and developed for different computing platforms. For inter-operability, these modules are usually ported to a single high-performance computing platform.

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) [2]. Each site maintains its own code on whatever platform the module was developed, but it provides a server that can be invoked from remote sites to access the functionality of that module. Instead of exporting code, each site therefore exports only the interface to 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 [41] and the Java RMI [44].

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 RPC mechanism was intended for a stateless, service-oriented architecture in which the service is relatively light-weight, and clients and servers exchange only small amounts of data. As a consequence, most RPC standards and implementations have features that make 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 that may need to exchange data sets of many megabytes or gigabytes.
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 acceptable 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

[0]Acknowledgements: This research was supported by NSF Grants ACI-0085969, ACI-0406345, ACI-0509307, ACI-0541193, and ACI-0615240.

ICS06 June 28-30, Cairns, Queensland, Australia

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. The overhead of doing this might out-weigh the benefits of this style of computation.

We believe this conventional wisdom must be re-examined in light of recent developments in Web Services in the World Wide Web. 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" [26]. 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 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 [26], 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'SOAP that is layered on top of SOAP. It is described in Section 4. O'SOAP permits asynchronous client-server interactions in which arbitrarily large amounts of data can be exchanged. A particularly useful feature of O'SOAP is that it permits legacy command-line-oriented applications to be deployed as Web Services without any modification. We believe that O'SOAP addresses the second and third problems with conventional RPC's described above.

The remaining problem that must be addressed is the overhead of data exchange between distributed components using SOAP and XML. Two previous studies of this issue [11, 40] 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. We argue in this paper that these studies are misleading. We have used our system to implement a large multi-physics, coupled fluid/thermal/mechanical/fracture simulation of a problem from the aerospace domain, and we have found that the overheads are less than 5%. To the best of our knowledge, this is the first performance evaluation of an entire state-of-the-art scientific application built using the Web Services framework.

The rest of this paper is organized as follows. In Section 2, we describe the aerospace problem that is being solved and the workflow used to implement it. In Section 3, we describe the design requirements and goals that we set for our distributed simulation infrastructure. In Section 4, we describe O'SOAP, the Web Services framework that serves as our infrastructure. In Section 5, we present experimental results obtained by using O'SOAP to deploy the workflow described in Section 2. We also resolve the paradox of why our results appear to contradict the results in the literature. In Section 6, we compare our approach with related approaches, and conclude in Section 7.

2. THE APPLICATION

In this section, we describe the aerospace problem and how simulation is used to solve it. This will motivate the design and implementation that we discuss in subsequent sections.

The Pipe Problem involves an idealized segment of a rocket engine modeled after NASA's next-generation space shuttle engine. This object, shown in Figure 1, is a curved pipe segment that transmits chemically-reacting, high-pressure, high-velocity gas through the inner, large diameter passage, and 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 create non-uniform thermo-mechanical stress and deformation fields within the pipe segment. Suppose a hairline crack, shown in Figure 2, forms in the interior of the pipe wall. The problem is to simulate the growth of this crack under the influence of the mechanical stresses.

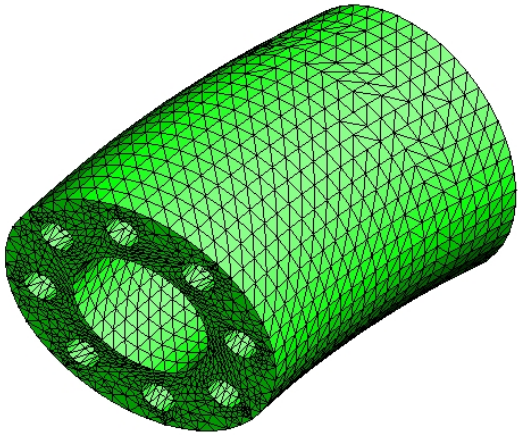

Figure 1: The Pipe

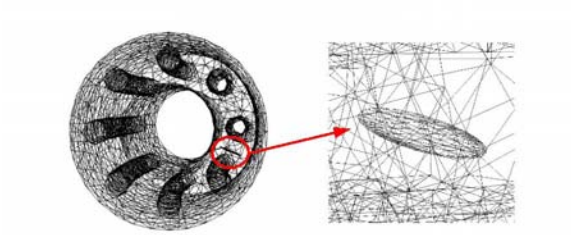

Figure 2: The Pipe with Crack

2.1 The Workflow and components

To appreciate both the problem of setting up the simulation and our solution to this problem, it is important to understand at a high level what computations are done and what data transfers are involved. The workflow for the Pipe Problem simulation is shown in

Figure 3. The components of our system appear like this, and the intermediate data sets appear like ≪this≫.

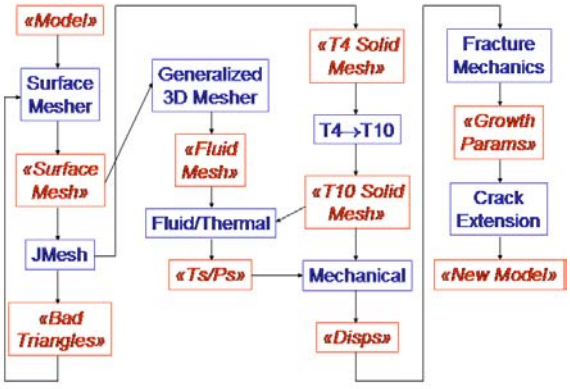

Figure 3: Workflow for the Pipe Problem

The input to the simulation is a model of the geometry of the pipe, together with material properties and initial conditions. The model is built using GGTK, a geometry manipulation tool from Bharat Soni's group at the University of Alamaba [24].

The first step of the simulation is to produce a triangulated *surface mesh* for the geometric surfaces of the component. The geometry of the problem changes from one time step to the next because the crack propagates through the material. A new surface mesh is required at each time step, so it is not feasible to manually inspect and correct the surface meshes. Therefore, we use a guaranteed-quality surface mesh generator developed by Paul Chew [7]. The surface mesher is new code that was developed for this project.

The next step is to generate a mesh for the solid part of the pipe, starting from the surface mesh. This is a discretization of the solid part of the pipe, and it is required to perform finite-element analysis of stresses and strains. Our project uses an advancing-front mesh generator called the *JMesh* [4] that generates unstructured, four-noded tetrahedral meshes for arbitrarily shaped three-dimensional regions. JMesh was implemented in Tony Ingraffea's Computational Structures Group (CSG) at Cornell to handle the unique geometric problems that occur in Fracture Mechanics. It is one of the pre-existing components that we marshalled for use in this project.

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 *T4 to T10* component converts the four-noded tetrahedra in the volume meshes produced by JMesh into equivalent meshes of ten-noded tetrahedrons.

To simulate the chemically-reacting flow in the interior of the pipe, it is necessary to discretize the interior of the pipe as well. The meshes generated by the solid mesher are not appropriate for fluid simulation since highly anisotropic elements are desirable for simulating viscous fluid flows in regions near no-slip boundaries, i.e., boundary layers. Therefore, we use the *Generalized 3D Mesher* [6, 5], which generates high-quality meshes consisting of extruded triangular prisms, tetrahedral elements, and generalized prisms, starting from the surface mesh. This code was implemented by David Thompson's group at Mississippi State University (MSU), and it is one of the pre-existing components used in our project.

The fluid mesh is used by the *Fluid/Thermal Solver* to simulate the 3-D chemically-reacting flow in the interior of the pipe. This solver models the fluid as a thermally-perfect, calorically-imperfect gas, and it is based on the CHEM code written by Ed Luke at MSU [29, 30]. The component we use is a modified version of the existing code at MSU.

The *Mechanical Solver* takes the solid mesh and the boundary conditions for pressure and temperature provided by the Fluid/Thermal Solver as input, and solves the equations of linear elasticity to determine the stresses and mechanical displacements in the pipe. It was written for this project by Gerd Heber at the Cornell Theory Center.

The *Fracture Mechanics* component implements a state-of-the-art crack propagation model that uses the displacements to predict the new crack front at the next time step. It is an existing component from Ingraffea's CSG group at Cornell. The *Crack Extension* component updates the crack geometry within the model based upon the crack front parameters computed by the Fracture Mechanics component. This component uses the GGTK tool described earlier.

Once the crack has been propagated, the new geometry is handed back to the surface mesher, and a new time step begins. The simulation is terminated when the crack reaches the surface of the engine wall.

2.2 Barriers to inter-operability

Making these components inter-operate is difficult because of the following facts.

- The components are written in a variety of languages including C, C++, C#, Python, Perl. If third-party libraries are counted, the source code for all of these components is between 1 and 2 million lines of code. There is no single platform on which all of the components run.
- Virtually all of these components are used for other projects. Their developers are extremely reluctant to write specialized versions of their components for this project.
- There are IP issues that prevent the authors of some of the components from releasing their source code to other project members. However, they are willing to run the component on behalf of other project members.
- One of the components, the Mechanical Solver, uses a Relational DBMS for storing and processing data. This component is coded for Microsoft's SQL Server and uses the .NET framework extensively. Currently, it runs only on the Windows-based clusters at the Cornell Theory Center, and porting it to our collaborators' UNIX and LINUX-based clusters would be a herculean task.

3. SYSTEM DESIGN

In this section, we discuss why we selected an architecture based upon distributed software components and an infrastructure based upon Web Services protocols.

We had the following design goals.

- Build a general infrastructure; do not tailor the system for a single problem.
- Use existing standards as much as possible.
- Keep the overheads of using the infrastructure low.
- Enable the creation of *adaptive* simulations in which software modules can be replaced with other modules with the same functionality.

3.1 The need for distributed components

We believe these goals require a design based upon *distributed* software *components*.

Why components? Componentization is a necessary (but not sufficient) condition for interchanging modules within an adaptive system. Consider an application that switches from one algorithmic technique to another: if the two techniques do not have clearly defined interfaces or use similar parameter (or data) types, then dynamically switching between them would be impossible.

Why distributed? Our project members use many different architectures and operating systems, and it would be a tremendous burden if every developer had to port their code to every other platform. Ideally, a component could be deployed on just one platform and invoked remotely by project partners. In other words, our components should be "write once, run *from* anywhere". In addition, some adaptive systems (e.g, DDDAS ([14, 15])) include data collection devices (e.g., VLA radio telescopes, sensor arrays) that are necessarily physically separated.

Our project therefore required that we build a distributed component-based system whose configuration evolves dynamically and whose implementation and execution spans institutions. In the parlance of Grid computing, we need to build a *virtual organization* ([22]). To build such a virtual organization, we need a system infrastructure with certain functionality, some of which is specified in the Open Grid Services Architecture ([21]). In this paper, we will focus on one aspect of this infrastructure, namely the computing model and its implementation.

Our current computing model is based upon the concept of the *remote procedure call (RPC)*. In our system, a client implements the simulation workflow by using the RPC mechanism to execute components on remote servers. This is a baseline; in a more advanced system, parts of the workflow might be executed by agents or other specialized servers.

A number of frameworks and standards have been proposed for developing RPC-style component-based systems. Perhaps the best known are CORBA [35] and COM [32]. We investigated these frameworks, but found that using them would require us to make extensive modifications to our existing applications. We also found that existing frameworks were primarily designed for deploying applications within a single machine. DCOM [31] is one exception to this. It is interesting to note that existing component frameworks are evolving towards interoperability with other standards (witness .NET subsuming COM and DCOM, and the OMG's adoption of a specification on CORBA-WSDL/SOAP Interworking [34]).

3.2 Web Services

One set of Internet protocols came close to meeting our needs, namely SOAP-based Web Services [47]. Many businesses need to form collaborations that span company boundaries, and they need a software infrastructure that will support their efforts and not force them to use a small set of proprietary client and programming tools. To meet this need, the W3C, Oasis and other organizations have started to define the necessary Web Services protocols. Many systems have been developed for deploying Web Services. These range from large systems, such as Apache Axis [23], Microsoft .NET [13] and Globus [19] to more modest frameworks such as, SOAP::Lite [28], SOAPpy [37] and GSOAP [16].

Unfortunately, these systems present a relatively high entry point for application developers. Let us consider what is required to deploy an existing application using these systems. First, the developer must write code to interface the application with a Web Services framework. While this code is often short, it presents a learning curve that can discourage computational scientists from experimenting with Web Services. The second difficulty is that there are many issues that arise in deploying an existing application in a Web Services environment, which do not arise in the traditional interactive environment, including the following.

- Data management: In computational science applications, data set sizes can vary greatly. Small data sets can be included within the SOAP envelope that is passed between the client and the server, which eliminates the need for a second round of communication to retrieve the data. However, embedding large data sets in SOAP envelopes is problematic for several reasons. One reason, which has been observed by others [11, 40], 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.
- Asynchronous client-server 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 response[1]. Unfortunately, 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.
- Generating WSDL: WSDL is the means for documenting a Web Service's interface. Some Web Service frameworks provide tools for generating WSDL documents automatically, but many require that the developer write these documents by hand.
- Authentication, Authorization and Accounting (AAA): The developer will certainly wish to restrict which remote users are able to use the Web Service.
- Job scheduling: Very often, the machine that is hosting the Web Service is not the same as that on which the component will run. If so, the Web Service has to interact with a job scheduling system to run the component.
- Performance: High performance is an important consideration for most computational science applications.

The existing tools offer a blank slate for the programmer. This enables the experienced and knowledgeable Web Services developer to write efficient and robust solutions for each application. For the novice Web Service developer, this presents a tremendous hurdle that will only be tackled if absolutely necessary.

To summarize, the very general nature of existing Web Service tools makes deploying Web Services a very costly undertaking for a novice Web Services application developer. This cost makes it unlikely that computational scientists will try to build a system based upon distributed components unless there is an absolute need to do so. What is needed is a new Web Services framework that is designed to address the needs of the scientific application developer. Ideally, this framework would address the considerations listed above, and enable a computational scientist to deploy new and existing applications as Web Services with little or no interfacing code.

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

4. IMPLEMENTATION

We describe O'SOAP [42], an O'Caml [38] based Web Services framework that we have developed for distributed computational science applications. The primary benefits of O'SOAP over other frameworks are its support for legacy applications and the way in which it builds upon the basic SOAP protocol to enable efficient interactions between distributed scientific components.

4.1 XML-based data formats

For interoperability, we have established a set of common, XML-based [48] formats for some of our data sets. These formats are described elsewhere [8, 10].

4.2 Server-side tools

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 4.

```
#! /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 4: 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 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 as listed below.

- 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. O'SOAP also automatically generates a WSDL [12] document that describes these methods, their arguments, and additional binding information.

4.3 Client-side tools

On the client, the `oids_tool` program provides a command-line interface to remote Web Services. Figure 5 shows how it can be used to invoke a service deployed using the script in Figure 4.

4.4 Asynchronous interactions

O'SOAP's server-side programs provide basic job management by exposing a number of methods to the client.

Spawn. This method invokes the component on the server and returns a job id to the client.

Running? The client can pass a job id to this method to discover whether or not the application has finished execution.

Results. Once completed, the client uses this method to retrieve the results of the component's execution.

Call. This wraps calls to the previous three methods into a single method that behaves synchronously. There is a slight performance advantage to using this method when the full power of the asynchronous methods is not needed.

There are several additional methods, such as "kill", for remotely managing the application process. Since the server is able to generate a response for these methods (except "call") 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).

O'SOAP also provides basic session management. If enabled by the component, the client is allowed to create a session on the server in which one of a number of application programs can be executed. Disk space is allocated to the session so that data can be shared between the application programs without having to be sent back to the client.

WS-Context [3] 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.

```
$ oids_tool \
   http://somewhere.com/sample.cgi?wsdl \
   --usage
call
    input:
        x: int
        y: int
    output:
        result: int
[ ... etc ... ] $ cat two 2 $ oids_tool \
   http://somewhere.com/sample.cgi?wsdl \
   call x=1 y:two out:result
$ cat result 3
```

Figure 5: Invoking sample server using `oids_tool`

4.5 XDescr

As was discussed in Section 3, one of the requirements of any infrastructure for distributed scientific applications is that it must be able to support the transfer of very large files between clients and servers. O'SOAP supports the XDescr protocol [27], which enables data sets to be separated from the SOAP request and response envelopes. If a data set is included, it is encoded using XML or Base64 ("pass by value"). If it is not included, then a URL to the data set is included ("pass by reference"[2]). Furthermore, XDescr enables clients and servers to dynamically specify whether a data set will be passed by value or reference.

Figure 6 gives an example of a SOAP request message using XDescr that a client might send to the Meshing Loop component. In this case, the client has chosen to include the XML description of the geometric model within the message. This is denoted by the

[2]The terms "by value" and "by reference" are only descriptive. Clearly, pass by reference in the usual language design sense would require a complex implementation in a grid setting.

presence of the `xdescr:XMLData` element, inside of which is the `GeoModel` element, which is the model.

Figure 7 shows the SOAP response message that might be sent from the Meshing Loop component to the client. In this case, the component has decided to return the three resulting meshes "by reference". Notice the presence of the `wsa:EndpointReference` elements in place of the mesh values. These elements contain URL's which the client can use to retrieve the files.

XDescr supports XML, ASCII, and binary data. Currently, HTTP, FTP, and MAILTO (email) protocols are supported for transmitting data "by reference". The scheme is very easily extended to handle other protocols, such as IBP [36].

On the server side, O'SOAP manages a pool of disk space that is used for storing data sets downloaded from the client and data sets generated by the application that will be accessed remotely.

Another feature provided by O'SOAP is a mechanism to pass data efficiently between two components hosted on the same server. If component A generates a large data set that is input to a component B on the same server, O'SOAP will recognize that the URL to the data set points to a local file, and will cause B to use that file directly.

```
<?xml version="1.0"?>
<soap_env:Envelope
    xmlns:xdescr="http://..."
    xmlns:s="http://..."
    xmlns:soap_env="http://...">
  <soap_env:Body>
    <s:spawn>
      <s:model>
        <xdescr:XMLData>
          <GeoModel GeoModelID="PipeExample">
            <!-- Model goes here -->
          </GeoModel>
        </xdescr:XMLData>
      </s:model>
      <s:size>1</s:size>
      <s:solid_volume>
        OuterVolume
      </s:solid_volume>
    </s:spawn>
  </soap_env:Body>
</soap_env:Envelope>
```

Figure 6: Message with in-lined data

Figure 8 shows the time needed to transfer files of various sizes. The line labeled "Local file copy" shows the time taken to copy each file on one local disk. Notice that for small and medium size files, the copy is virtually instantaneous. For larger files, the time is roughly linear. The line labeled "By value" shows the time taken to copy a file from an O'SOAP client to an O'SOAP server and back. In this case, the file is encoded in Base64 and inserted into the SOAP request and response messages. Notice that it takes significantly longer than the file copy for smaller files. Notice also that the times are 100M and 1G files are missing. This is because the SOAP requests were too big for the server to process.

The line labeled "By reference" shows the time taken to copy a file between an O'SOAP client and server using the XDescr protocol. In this case, the SOAP request and response messages contain a URL that can be used to pull the file across the network using the HTTP protocol. For small files, the time for "By reference" is the same as "By value". This suggests that most of the time in both cases is spent in generating, transmitting, and processing the SOAP request and response messages, and not in handling the data. For larger files, the cost appears to converge to the "Local file copy"

```
<?xml version="1.0"?>
<soap_env:Envelope
    xmlns:soap_env="http://..."
    xmlns:wsa="http://..."
    xmlns:s="http://...">
  <soap_env:Header/>
  <soap_env:Body>
    <s:results_response>
      <s:surface_mesh>
        <wsa:EndpointReference>
          <wsa:Address>
            http://foo.com/aaaa
          </wsa:Address>
        </wsa:EndpointReference>
      </s:surface_mesh>
      <s:solid_surface_mesh>
        <wsa:EndpointReference>
          <wsa:Address>
            http://foo.com/bbbb
          </wsa:Address>
        </wsa:EndpointReference>
      </s:solid_surface_mesh>
      <s:solid_mesh_t4>
        <wsa:EndpointReference>
          <wsa:Address>
            http://foo.com/cccc
          </wsa:Address>
        </wsa:EndpointReference>
      </s:solid_mesh_t4>
    </s:results_response>
  </soap_env:Body>
</soap_env:Envelope>
```

Figure 7: Message with data passed by URL

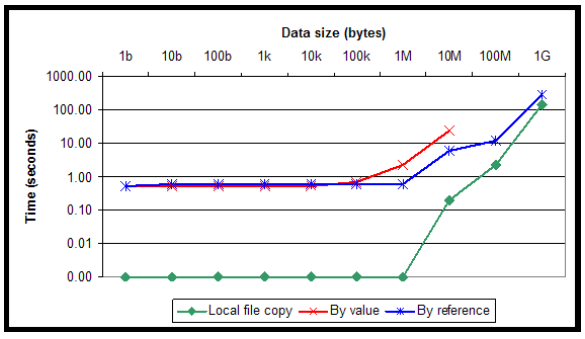

Figure 8: Performance of XDescr protocol

times. This suggests that, in these cases, most of the time is spent copying the file on the server's local disk.

These results show that the XDescr protocol can be used to efficiently transmit large files between clients and servers.

5. EVALUATION

In this section, we describe an end-to-end performance evaluation of our Web Services-based system solving the Pipe Problem described in Section 2. To the best of our knowledge, this is the first performance evaluation of an entire state-of-the-art scientific application built upon Web Services.

Web Services have been designed primarily with business applications in mind, so it is natural to wonder whether they are suitable for scientific applications like ours. Previous papers ([11, 40]) have looked at the use of XML and SOAP for scientific applications. Their conclusion was that there is a large performance penalty and they recommended several sophisticated implementation strategies and changes to these protocols to get around the performance problem. Contrary to previous studies, our results suggest that Web Services can be used for computational science applications, with little performance penalty.

5.1 Experimental setup

The following machines were used for the experiments below,

- The *ASP* cluster is housed in the Cornell Computer Science department in Ithaca, New York 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.
- The *CUCS* machine is another machine in Cornell Computer Science that was used for this experiments. It is a Dell PowerEdge 1750 server with dual 3 GHz Intel Xeon processors. It has 4 GB of memory and is running Red Hat Enterprise Linux 4.
- Two machines host components at the Engineering Research Center at Mississippi State University in Starkville, Mississippi. *SAVAGE* is an IBM x330 server, with dual 1.266GHz Intel Pentium III CPUs and 1.25GB RAM running Red Hat Linux version 7.3. *LOCUS* is a Dell Precision 420 workstation, with dual 933 mhz Pentium III processors with 1GB RAM and runs Red Hat Linux Version 9.
- 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.

The following components were executed on these machines. These components are executed immediately and directly on the servers on which they are deployed.

- Meshing Loop on ASP.
- Generalized Mesher on SAVAGE.
- T4 to T10 on ASP.
- Fluid/Thermal solver on LOCUS.

The Mechanical Solver is executed on a large Windows-based clusters via a batch queue. The presence of the batch queue makes it very difficult to isolate the true running time of the component, which makes it impossible to establish a baseline that can be used for measuring overheads for this component.

We used the adaptive Meshing Loop discussed in Section 2 to generate two different problem sizes[3] 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.

In these experiments, all of the components as well as the client were deployed using the O'SOAP framework.

5.2 Performance Results

Table 2 shows the running time for all of the components up to and including the Fluid/Thermal solver.

The column labeled "Baseline 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 column labeled "CUCS runtime" shows the running time when the client runs on a machine that is within the same building as the ASP server, while the column labeled "UAB runtime" shows the running time when the client runs on a machine at UAB. Each row shows the running times for the individual components, and the row marked "Total" shows the aggregate results for the entire run. All runtimes reported are the average over several runs. We were not able to arrange for identical machines to run the client at the two sites, so it is important not to read too much into the relative runtimes for the two client deployments.

5.3 Performance Analysis

There are a number of interesting points in the performance results of Table 2.

There are several components whose relative overhead is very large (e.g., the overhead of the Generalize Mesher for size 1 is 168.17%). This overhead is associate with marshaling the data through the Web Services stack. Notice that the absolute overhead is roughly the same for all components (on the order of 10 seconds). When the baseline running time of the component is small relative to the absolute overheads, the relative overhead appears large. In contrast, consider the Meshing Loop and Fluid/Thermal components. Notice that the overheads for both components decrease as problem size is increased. Each component runs for a relatively long time, so the cost of invoking each component using Web Services is small relative to its total running time.

Overall, the total overheads for both clients fall as the problem size increases. The overheads for the largest problem size are 1.85% and 1.74% for the "CUCS" and "UAB" clients, respectively. This is because there are several components whose running times dominate the total running times, so the overhead introduced by the use of Web Services is relatively small.

These results are in marked contrast to previous studies in the literature [11, 40] that concluded that the use of SOAP and XML adds enormous overhead to computational science applications. This paradox is resolved as follows. The previous studies measured the overhead of using Web Services to execute matrix-multiplication and other small kernels on machines scattered over the Internet. In addition, the problem sizes used were very small. Therefore 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., so the relative overhead of communicating data across the Internet and invoking Web Services is small.

Another way to understand this point is that it is an instance of

[3]Earlier drafts of this paper included partial information about a larger third problem size. Unfortunately, this project has ended and several of the machines hosting component were decommissioned before complete experimental results for the third problem size could be collected. Interested readers can find earlier results from all three problem sizes in [43].

Problem size	Solid Mesh vertices	triangles	tet's	Interior Mesh 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

Table 1: Pipe Problem Sizes

Problem size	Component	Baseline runtime	CUCS runtime	CUCS overhead	UAB runtime	UAB overhead
1	Meshing loop	348.34	357.83	2.73%	362.17	3.97%
	T4 to T10	17.80	24.69	38.76%	33.37	87.50%
	Generalized Mesher	12.28	33.47	172.58%	32.93	168.17%
	Fluid/Thermal	3,020.13	3,076.35	1.86%	3,087.87	2.24%
	total	3,398.55	3,492.35	2.76%	3,516.33	3.47%
2	Meshing loop	1,003.15	1,012.64	0.95%	1,019.43	1.62%
	T4 to T10	44.49	56.53	27.07%	68.71	54.46%
	Generalized Mesher	37.48	65.73	75.35%	64.06	70.90%
	Fluid/Thermal	7,814.79	7,929.96	1.47%	7,902.78	1.13%
	total	8,899.91	9,064.85	1.85%	9,054.99	1.74%

Table 2: Pipe Problem Runtimes

the well-known fact that the efficiency of parallel computation depends critically on the computation to communication ratio. 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 [46], a message-passing library designed for this purpose. If this component were to be implemented in a distributed way across the Internet, the overheads would likely be prohibitive.

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 interoperate. 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.

6. RELATED WORK

Perhaps the most widely know paradigm for distributed scientific computing is Grid computing [18], and the most widely known grid system is the Globus Toolkit [19]. The Open Grid Services Infrastructure (OGSI) specification [45] and WS-Resource Framework (WSRF) proposed specifications [25] build upon the SOAP protocol to define additional protocols that are useful for distributed computing, such as resource management, event notification, etc.

OGSI and WSRF. Grid Services, as they have been standardized by the Global Grid Forum (GGF), *are* Web Services. In fact, some of the standards being considered by the GGF have been submitted to Web Services standards bodies for consideration (e.g., WS-Agreement, WS-Notification). However, the ASP project predates the core protocol adopted by the GGF, namely the Open Grid Services Infrastructure (OGSI), by almost 2 years. So, OGSI was not an option for us when we started.

At this point, OGSI does exist, so a natural question is whether or not we plan to migrate towards it. We do not. One reason for not migrating to OGSI is that several of the key OGSI developers have proposed a new set of standards, the Web Services Resource Framework (WSRF), to supercede OGSI. One of the key advantages of WSRF over OGSI is that there are plans to pursue its standardization, not only within the GGF, but within OASIS, as well. Fundamentally, the functionality of 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.

Globus. The Globus Toolkit is arguable the most widely known framework for grid computing. Presently, it does not run on Microsoft Windows. Since we need to deploy components on Windows based clusters, in addition to UNIX and LINUX clusters, Globus was not an option.

We have another reason for not using Globus, and that is that we are very interested in developing alternative Web Services frameworks that address different design goals than Globus. For instance, we have designed O'SOAP to support legacy applications. Systems like Globus, Microsoft .NET, or Apache Axis require a certain amount of coding in Java, C# or Perl to interface an application with the framework, but O'SOAP enables existing applications to be deployed as Web Services without any modifications.

Other Grid Frameworks. There are at least two systems that implement RPC for virtual organizations: Ninf ([33]) and NetSolve ([1]). The Global Grid Forum (GGF) GridRPC Working Group is attempting to define standard API's for these systems ([39]), but the protocols that these systems use for communicating over the Internet are not standardized.

Ninf [33] 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. There are also efforts to build similar systems in the context of Matlab [17].

The lack of standard protocols presents two difficulties. The first

is philosophical: we have argued that an adaptive system like ours should be built from distributed components with standardized interfaces. The second difficulty is practical: if we use standard API's but proprietary protocols for invoking components, we lose many of the benefits of standardization. These include the ability for our users to choose from a suite of standard-based client-side and server-side toolkits.

7. 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, with the basic system and initial performance being described in [9]. 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 provided us with a number of development and software maintenance benefits.

We have also described a set of performance experiments on our system. 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 [22], 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, to obtain 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 multi-investigator projects and that Web Services may be appropriate for many of these as well.

It would be interesting to redeploy our components using frameworks built around the emerging grid standards such as WSRF.NET and Globus, and rerun the experiments. These frameworks provide several features not found in our system such as sophisticated security mechanisms and integration with grid portal systems. Whether this increased sophistication comes without an increased cost in performance and ease of use remains to be seen.

8. REFERENCES

[1] Dorian C. Arnold and Jack 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] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*, page 3, Bretton Woods, NH, 1983. Association for Computing Machinery.

[3] Doug Bunting, Martin Chapman, Oisin Hurley, Mark Little, Jeff Mischkinsky, Eric Newcomer, Jim Webber, and Keith Swenson. Web services context (ws-context) ver1.0. Available at `http://developers.sun.com/techtopics/webservices/wscaf/wsctx.pdf`, July 28 2003.

[4] J.B. Cavalcante-Neto, P.A. Wawrzynek, M.T.M. Carvalho, L.F. Martha, and A.R. Ingraffea. An algorithm for three-dimensional mesh generation for arbitrary regions with cracks. *Engineering with Computers*, 17:75–91, 2001.

[5] S. Chalasani and D. Thompson. Quality improvements in extruded meshes using topologically adaptive generalized elements. *International Journal for Numerical Methods in Engineering*, (submitted).

[6] 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.

[7] 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.

[8] L. Paul Chew, Stephen Vavasis, S. Gopalsamy, TzuYi Yu, and Bharat 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.

[9] Paul Chew, Nikos Chrisochoides, S. Gopalsamy, Gerd Heber, Tony Ingraffea, Edward Luke, Joaquim Neto, Keshav Pingali, Alan Shih, Bharat Soni, Paul Stodghill, David Thompson, Steve Vavasis, and Paul Wawrzynek. Computational science simulations based on web services. In *International Conference on Computational Science 2003*, June 2003.

[10] Paul Chew and Steve Vavasis. Proposal for mesh representation. Internal draft, January 21 2003. Accessed February 13, 2003.

[11] Kenneth Chiu, Madhusudhan Govindaraju, and Randall 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.

[12] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. Available at `http://www.w3.org/TR/wsdl`, March 15 2001.

[13] Microsoft Corporation. Microsoft .NET. Accessed February 11, 2003.

[14] Frederica Darema. Dynamic data driven applications systems: A new paradigm for application simulations and measurements. In *International Conference on Computational Science*, pages 662–669, Krakow, Poland, June 6–9 2004.

[15] Frederica Darema. Dynamic data driven applications systems: New capabilities for application simulations and measurements. In *International Conference on*

Computational Science, pages 610–615, Atlanta, Georgia, May 22–25 2005.

[16] Robert A. Van Engelen and Kyle A. Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, page 128, Berlin, Germany, May 21 – 24 2002.

[17] M.H. Eres, G.E. Pound, Z. Jiao, J.L. Wason, F. Xu, A.J. Keane, and S.J. Cox. Implementation of a grid-enabled problem solving environment in Matlab. In *Proceedings of the International Conference on Computer Science (ICCS 2003)*, pages 420–429. Springer-Verlag, 2003.

[18] Global Grid Forum. Global Grid Forum home page. Accessed February 13, 2003.

[19] I. Foster and C. Kesselman. The Globus project: A status report. In *IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[20] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, second edition edition, 2004.

[21] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, June 22 2002.

[22] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001.

[23] The Apache Foundation. Webservices - axis. `http://ws.apache.org/axis/`.

[24] GGTK home page. Accessed February 13, 2003.

[25] Globus Alliance. The ws-resource framework. Available at `http://www.globus.org/wsrf/`, January 24 2004.

[26] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP version 1.2 part 1: Messaging framework. Available at `http://www.w3.org/TR/SOAP/`, June 24 2003.

[27] Gerd Heber and Paul Stodghill. X-descriptor: A protocol for specifying datasets in web services, June 16 2005. Available at `http://www.asp.cornell.edu/specs/2005/01/xdescr/xdescr.pdf`.

[28] Paul Kulchenko. Web services for Perl (SOAP::Lite, XMLRPC::Lite, and UDDI::Lite). Accessed on June 3, 2003.

[29] E. A. Luke. *A Rule-Based Specification System for Computational Fluid Dynamics*. PhD thesis, Mississippi State University, 1999.

[30] E. A. Luke, X.L. 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.

[31] Microsoft, Inc. Distributed component object model (DCOM). Accessed February 13, 2003.

[32] Microsoft, Inc. Microsoft COM technologies. Accessed February 13, 2003.

[33] Hidemoto Nakada, Mitsuhisa Sato, and Satoshi Sekiguchi. Design and implementations of Ninf: towards a global computing infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 15(5-6):649–658, 1999.

[34] Object Management Group. CORBA to WSDL/SOAP interworking, v1.1. Available at `http://www.omg.org/technology/documents/formal/CORBA_WSDL.htm`, February 2005.

[35] Object Management Group, Inc. Welcome to the OMG's CORBA website. Accessed February 13, 2003.

[36] James S. Plank, Micah Beck, Wael R. Elwasif, Terry Moore, Martin Swany, and Rich Wolski. The internet backplane protocol: Storage in the network. In *NetStore99: The Network Storage Symposium*, Seattle, WA, USA, 1999.

[37] Python web services. `http://pywebsvcs.sourceforge.net/`.

[38] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *In Theory And Practice of Objects Systems*, 4(1):27–50, 1998.

[39] K. Seymour, H. Nakada, S. Matsuoka, D. Dongarra, C. Lee, and H. Casanova. GridRPC: A remote procedure call API for grid computing. ICL Technical Report ICL-UT-02-06, Innovative Computing Laboratory, Department of Computer Science, University of Tennessee, June 2002.

[40] Satoshi Shirasuna, Hidemoto Nakada, Satoshi Matsuoka, and Satoshi Sekiguchi. Evaluating web services based implementations of GridRPC. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, 2002.

[41] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. IETF RFC 1831, August 1995.

[42] Paul Stodghill. O'SOAP - a web services framework in O'Caml. `http://www.asp.cornell.edu/osoap/`.

[43] Paul Stodghill, Rob Cronin, Keshav Pingali, and Gerd Heber. Performance analysis of the pipe problem, a multi-physics simulation based on web services. Computing and Information Science Technical Report TR2004-1929, Cornell University, Ithaca, New York 14853, February 16 2004.

[44] Sun Microsystems. Java RMI specification. Available at `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`.

[45] Steve 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.

[46] D. W. Walker and J. J. Dongarra. MPI: a standard Message Passing Interface. *Supercomputer*, 12(1):56–68, 1996.

[47] World Wide Web Consortium. Web services activity. Accessed June 11, 2003.

[48] World Wide Web Consortium. Extensible markup language (XML) 1.0 (second edition). W3C Recommendation, October 6 2000.