Comparison of Web Services, Java-RMI, and CORBA service implementations

N.A.B. Gray
School of Information Technology & Computer Science,
University of Wollongong
nabg@uow.edu.au

Abstract

This paper reports on comparisons of Web Service, Java RMI, and CORBA solutions for example applications. Performance problems, identified in earlier studies of Web Services, have been significantly reduced in the current implementations. The newer Web Service APIs realize a model that has significant overlaps with distributed object technologies, allowing in some cases for the use of a common code base in either a readily deployed Web Service or in a higher-performance distributed object style implementation.

1. Introduction

Web Services [1] present another alternative distributed computing infrastructure; an alternative that is being strongly promoted as preferable to the use of distributed object middleware such as Java RMI [2] or CORBA [3].

Web Services differ from the distributed object technologies in that they have reverted to an earlier "remote service" model similar to that in DCE. There is no concept of an object reference; instead a service is defined simply by an end-point that supports various operations. In terms of Java-RMI or CORBA, a Web Service is like a singleton server object. The singleton server character of a Web Service means that a stateless-server architecture is preferred; though there are mechanisms permitting the implementation of stateful-servers.

Web Service implementations support different client-side application programmer interfaces; client code may work by constructing "call" objects that are dispatched to the server, or may use a higher level interface that hides the communications level entirely through the use of client-side stub objects with an operational interface that mimics that of the server. The client-stub approach results in code that is very similar to Java-RMI or CORBA clients.

The analogous mechanisms for generating client and server components for Web-Services, Java-RMI, and CORBA are illustrated in Figure 1. When auto-generated client-side stubs are used for Web Services, the development processes and the code complexity for both client- and server-side are virtually identical for Web Service, Java-RMI, and CORBA solutions. Typically one starts with an interface definition for the service. A client-side stub is auto-generated from this interface. On the server-side, the interface is processed to yield a base class for the implementation class that must be written by the developer. With the mechanisms and costs of development being very similar, other factors will determine the choice of a technical solution. Systems developers will have to choose between interoperability where Web Services have advantages, and performance that will favor Java RMI or CORBA.

Java RMI and CORBA use optimized connection-oriented communications protocols that are either language specific, or have detailed rules defining how data-structures and interfaces should be realized. In contrast, Web Services (application-to-application) are based on the ubiquitous technologies that have grown up to support WWW-services (human-via browser-to-application). Communications use HTTP. HTTP is universally supported, and HTTP-traffic can normally pass through firewalls. Tiresome practical details like common data representations are avoided through the use of textual representations. All numeric and other data are converted to text. Meta-data, defining structure, are provided in situ as XML mark-up tags. XML parsers allow client and server implementations to construct their distinct but equivalent representations of any data structures

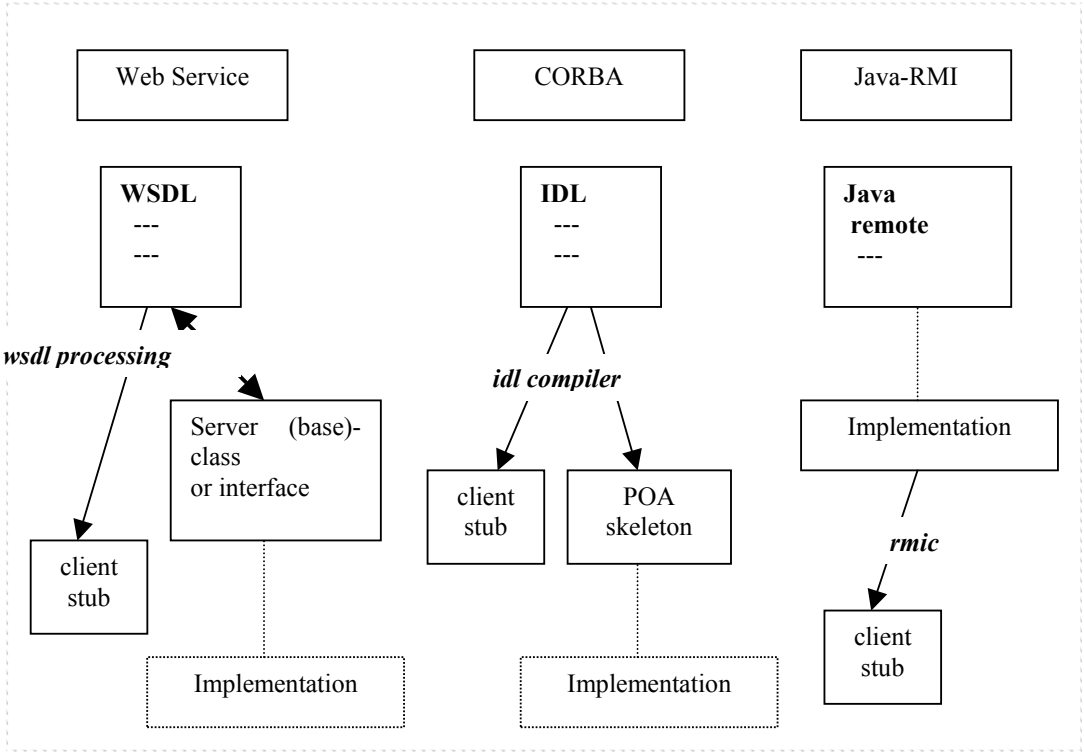

Figure 1 Generation of client and server components from interface for Web Services, CORBA, and Java-RMI.

The use of HTTP, and XML text documents, supports increased interoperability but also represents a significant increase in run-time cost for Web Service solutions as compared with Java-RMI or CORBA solutions. The stateless hypertext transfer protocol was devised originally for downloading individual files and is not ideally suited to applications where multiple requests and responses may be need to be exchanged. The XML formatted documents are inherently more voluminous than the binary data traffic of the other approaches. More data have to be exchanged across the network, and more control packets are required. Conversion to text format and parsing of XML documents is inherently more costly than the alternative mechanisms used to convert data to a common data representation for the network. The additional communications and processing costs are frequently perceived as a potential barrier to the use of Web Services technologies.

The work reported in this paper extends that in earlier studies on the costs associated with Web Service SOAP communications. Elfwing et al. compared Web Services to CORBA both as Java implementations, finding a degradation factor of up to 400 in performance; performance was improved by modifications of drivers and parsers [4]. Davis and Parashar compared several Web Service implementations (Java, Perl, .Net) with Java-RMI and CORBA [5]; they were qualified in their conclusions but tended to prefer Java-RMI until SOAP-based Web Service technology had further improved. Both groups identified specific issues with XML parsers and the HTTP transport protocol.

This work uses more recent implementations of the Java Web Service components. These offer a higher-level client-side API, and achieve better use of HTTP. Earlier studies have mainly looked at examples where the defined service returned either single data elements or data arrays of primitive types such as integers, reals, or strings. The examples here include those more representative of intranet applications that involve multiple queries and queries that yield large amounts of data. If Web Services are to replace CORBA and Java RMI, they must handle such queries effectively.

The Middleware Technology Evaluation project, hosted by the CSIRO division of Mathematical and Information Sciences, has demonstrated the

practicality and value of comparative performance studies on commercial implementations of middleware [6]. Limits on resources restricted this study to the use of the Sun reference implementations of Java-RMI, CORBA, and WebServices. The coding of these reference implementations is expected to be of similar quality, so that the results should fairly represent the intrinsic merits of the different technologies.

2. Scope of study

2.1. Technologies and measures

The technologies used in this study were:

- JAXRPC from Sun's Java Web Services Development kit (JWSDP) [7];
- Tomcat 5 (as incorporated in JWSDP) [8];
- Java RMI from Java 1.4 SDK [2];
- Java CORBA from Java 1.4 SDK [9];
- Ethereal network traffic analysis program [10].

Tests were performed on a network system with a 100Mb/sec Ethernet backbone. Performance tests utilized a Sun v480 (1030MHz) server and a group of SunBlade 150s (650MHz) running client applications. Network traffic analysis was conducted using the Ethereal tool to capture packets transferred over the 100Mb/sec Ethernet between a SunBlade 100 server and a client 2GHz Dell Optiplex GX260 machine running Win2000.

The machine groups were workstations in a teaching laboratory and a new server. The study was conducted out of academic session when these machines had no other users, so timing measurements were not distorted by process switching etc. The network carried some other light traffic, but was not approaching saturation so network latencies should not be distorted.

Measures include:

- memory usage in the client,
- CPU-times for both client and server,
- overall latency time for a single request and for sequences of requests,
- total byte transfers,
- packet counts.

Memory usage was estimated using differences in "total memory" and "free memory" as measured using calls to `java.lang.Runtime` in the client code.

Preliminary tests had involved running the server with increasing number of concurrent clients to determine the loads at which the server became fully utilized and service times increased. The results presented here are for situations where the server was lightly loaded by small numbers of concurrent clients and still had idle CPU time. Each client machine ran only a single copy of a client process so that contention was not an issue on the clients.

A small part of the study focused on the cost of a single request from client to server. These single request tests provide comparative costs for the establishment of a connection. Most of the reported results are from tests where several thousand requests are handled in the course of a client-server session. In these tests, set-up costs are amortized over many requests giving a better measure of the actual request handling costs. Typically tests involved 5000 requests; where the processing time was still short, runs of as many as 50,000 requests were used. The results presented are averages from ten or more test runs. With the tightly controlled conditions, variations in execution and run-times were small.

Some of the tests involve pseudo-randomly generated data sets that vary in size. For these tests, identical sequences of seeds were used for the random number generators when running the same test with different technologies. Naturally, time variations of successive runs were greater, up to 10%, reflecting the varying sizes of datasets generated and transmitted across the network.

The Ethereal tool provided the data on packet and byte counts. It also permits detailed analysis of the packet sequences generated by the different technologies. The technologies do vary in the efficiency of their use of the underlying TCP/IP communications.

2.2. Example services

Three example services were implemented. They differ mainly in the type and amount of response data; one of the efficiency issues being how well the technologies handle significant data volumes.

The first service, "calculator", actually involves a stateful server (issues of state maintenance in Web Service applications are discussed in section 3.5 below). The service defined a simple four-function calculator with operations that each required an integer input argument and generated an integer response. Clearly, such limited data should involve minimal overheads.

The second service, "data", involved a string input argument and a returned structure with integer, string array, and double array fields. Here, the XML encoding has to include some more significant structural meta-data.

The final service, "large data", models a real-world data retrieval application. Intranet use of Java-RMI and CORBA frequently has a client running in one departmental system that uses a middle-layer server to access a database in some other departmental system. It is this kind of application that Web Services will have to take on if they are to establish themselves as a preferred technology for intranet systems. The demonstration service was a reduced version of an application used to retrieve data on books from a database. The demonstration implementation has only a single function that effectively simulated the application query "*get books that include keywords …"*. Database performance issues were avoided by having the service pseudo-randomly generate and populate anything from a few dozen to several hundred records in response to each request.

The IDL interfaces for these services are:

"Calculator" (minimal complexity data transfers of integers):

```
   interface Demo {
      long long clear();
      long long add(in long long  val);
      ...
// quit – in JAXRPC version will release
// session storage; in JAVARMI version
// will do nothing (rely on garbage
// collector); in CORBA version will do
// a deactivate object
      void quit();
   };
// Factory component only relevant in
// RMI and CORBA implementations
   interface DemoFactory {
      Demo createDemo();
   };
```

"data" (transfer of simple record structure):

```
   typedef sequence<string> strings;
   typedef sequence<double> doubles;
   struct Data {
      long long  _d1;
      strings    _d2;
      doubles    _d3;
   };
   interface Demo {
      Data f1(in string key);
   };
```

"large data" (transfer of potentially large data structure):

```
   typedef sequence<string> strings;
   typedef double priceType;
   // Record defining a book
   struct Data2 {
      string     title;
      strings    authors;
      priceType     price;
      priceType     listPrice;
      string     publisher;
      long       publicationYear;
      long       publicationMonth;
      strings    keyWords;
      string     isbn;
      long       starRating;
      string     url;
   };
   typedef sequence<Data2> Data2Seq;
   interface Demo {
// Simulated method corresponding to
// find books with keyword like ...
      Data2Seq search(
         in string request);
   };
```

The `java.rmi.Remote` interfaces are closely similar, mainly involving substitution of arrays for IDL sequences.

The developer of a Web Service can start by composing the Web Service Description Language (WSDL) description of the service. However, both the JAXRPC and the .Net development environments permit a developer to start by defining a service interface in terms of a programming language class (C#, VB etc) or interface (Java) and using a helper tool to generate the WSDL document. This bottom-up approach is easier for most developers. This study used the same `java.rmi.Remote` interfaces as the starting points for both Java RMI and JAXRPC Web Service implementations. Client-side stubs are auto-generated from the WSDL service definition.

A WSDL document (an XML file), either composed manually or generated from a programming language class or interface, will define "messages" (input messages are essentially function signatures, output messages are responses), "port-types" (a service interface or class) with "operations" that are defined in terms of their input and output messages, "binding(s)" - specifications of a transfer protocol, typically HTTP, an encoding scheme (choice among SOAP encoding styles), and finally the service definition(s) with end-point URI(s). The inclusion of the service end point's URI is really the only major semantic difference from an IDL or `java.rmi.Remote` interface declaration.

2.3. Service implementations and deployment

A JAXRPC servant can be instantiated inside a servlet-container such as Tomcat, or can be implemented as a stateless EJB session bean. If the servlet model is used, the developer defines a class that implements the `java.rmi.Remote` interface

declaration; an EJB session bean defines an interface with equivalent operations. This study used the Tomcat-hosted servlet style. The developer must provide effective implementations for all defined operations, together with any life-cycle methods that are required by the servlet- or EJB- container.

In a servlet-based implementation, system supplied servlets take configuration data that define the Web Service classes that they manage. The servlet code deals with the HTTP data traffic, and invokes XML parsing to prepare arguments, and then invokes the service operations. The actual JAXRPC servant can be given access to its servlet-container by having its class implement the optional `javx.xml.rpc.server.ServiceLifecycle` interface. The context can include resources shared with other servlets such as in-memory data structures or database connections. The context can also manage session state for a stateful service.

XML deployment files for the servlets and their configuration data are all generated automatically. WWW-servers like Tomcat are designed for easy administration; services can be added or existing services replaced in a running server. The control data for the hosted services are persistent. A system restart will require minimal or no action by an administrator. The server for WWW and Web Service applications should simply re-initialize itself.

A CORBA developer has the choice of extending the auto-generated "POA" class, or of using an instance of a generated POATie class that works with a servant that is an instance of a class implementing the operational interface but which is otherwise independent of the CORBA class system.

The complexity of CORBA implementations is often overrated. In these demonstration applications, the server-side consisted of single server process. This initializes its ORB, instantiates the implementation class, activates this instantiated servant via the default POA, and publishes an IOR in a file. A more typical server system would involve a CORBA name service (or trader) and a CORBA daemon process. The server process would utilize a POA that supports persistent references; the servant identifier would be published in the name service. The CORBA daemon process would deal with issues like restarting a server process as needed. The name-service and CORBA daemon should both use persistent data stores and be capable of surviving system restarts. In the Sun Java 1.4 implementation, the "orbd" process takes on both the role of a persistent naming service and of the CORBA daemon that can launch server processes.

A Java-RMI servant class normally extends the class `java.rmi.server.UnicastRemoteObject` and implements the defined remote service definition; e.g. `public class DemoImpl extends UnicastRemoteObject implements Demo { ... }`.

The Java-RMI deployment is the most elaborate, and most fragile. A server system will comprise at least the actual server process, the rmiregistry (equivalent to a CORBA name-service), and a HTTP-server from which a would-be client can download the ".class" files of client-side stubs. An rmiregistry does not maintain persistent data so all records of registered servers are lost on system restart. Further, the registry must be restarted if there are changes in the implementation of any one of the services registered with it. The developer of a Java-RMI system has to take careful account of rules regarding access to .class files by server process, HTTP-server, and rmiregistry, and must also compose security policy constraint files. The rmid daemon process can be employed to support on-demand services; but rmid again does not use persistent data storage making the system fragile in regard to system restarts. The use of rmid also necessitates the use of extra setup programs. There are mechanisms for integrating Java-RMI with Jini technology that provide for more stable deployments; but these require skill sets and knowledge that are uncommon.

2.4 Client-side stub and client coding

The WSDL for a Web Service must be made available to the developer of the client-side code. If a UDDI registry were being used, the client developer might be able to download the WSDL from the registry. The Tomcat server configuration for a JAXRPC implementation can act as an alternative source of the WSDL. Both .Net and JAXRPC development systems include helper applications that can generate a client-side stub class, and other helper classes, from a downloaded WSDL definition.

A CORBA developer has to obtain a copy of the IDL interface definition (this too could come from a UDDI registry for these are not restricted to handling only WSDL defined services). The developer then generates a client stub in the required implementation language via an IDL compiler.

The developer of a Java-RMI client works solely with the remote interface. The client-side stub class has to be downloaded at run-time from a HTTP server co-located with the actual RMI server and the rmiregistry name-server process.

Each test client had code that established a connection and then in a loop repeatedly invoked service operations. Such tightly coded loops result in client behavior that is much more demanding than a typical real-world application.

Client application code is essentially identical for all technologies. All implementations will work with an object reference of the interface type; most of the code will simply invoke operations via this reference. The codes differ in the half dozen lines needed to create the stub object associated with this reference.

The JAXRPC code utilizes an instance of an auto-generated helper-class to create a stub object (an instance of an application-specific subclass of a generic `Stub` class). The end-point URL should be encoded in the generated stub, but can be overwritten. There is no further requirement to invoke a naming service or other helper.

A Java-RMI implementation will require a principal server-side object whose identity is published via the rmiregistry naming service. This could be the singleton object in a singleton stateless server, or a factory object for a stateful service. The client obtains a reference via a lookup operation on the naming service. The underlying Java-RMI runtime arranges to download the stub class code from an HTTP server as part of the lookup process.

A CORBA client might obtain a reference to the service for its stub from a `CosNaming` name service, or from a trader, or from a stringified IOR in a file in a shared file space.

3. Results

3.1. Costs associated directly with communications protocols

The typical illustrative Web Service application has a client connect to a service, submit a single request for data, and terminate. Such applications are unlikely to put any great demands on either a server host's CPU, or a communications network; consequently, performance issues are not that important. In any case, for such applications, Web Services technologies perform well in comparison with the distributed object systems.

The data shown in Table 1 show relative performances for four implementations of the "data" application with a request for a single data structure retrieved according to an argument key. (Here, times are measured from first to last packet of TCP/IP communications. The services were all fully initialized having handled previous clients.)

Technology	Total Latency	Total Packets	Total data transferred in bytes
WS	0.11s	16	3338
CORBA	0.48s	8	1111
CORBA & name server	0.86s	24	3340
Java RMI	0.32s	48	7670

Table 1. Costs of single shot request using various technologies.

Any remote-connection incurs the establishment costs of a TCP/IP connection. A Web Service implementation, using a static stub previously generated from WSDL, incurs the cost of only one connection. Subsequent exchanges using HTTP may be non-optimal. A request is sent in two parts (HTTP headers in the first message, SOAP "envelope" with request in second part), and these parts are acknowledged. A response is again multipart, a first part with the HTTP response header and subsequent continuation parts containing the XML response document. Despite the relative inefficiency of HTTP, the JAXRPC solution performs best.

A simple CORBA solution, using an endpoint address read as a stringified IOR taken from a file, is comparable in complexity with the JAXRPC solution. It represents the most efficient solution in terms of network traffic (in this case, half of the packet traffic being that needed to establish and terminate a TCP-IP connection) but is relatively slow. There are significant delays between the establishment of the connection and the issue of the first request by the client, and between the acknowledgment of the request and generation of the response on the servers.

A more realistic CORBA solution has the client contact a name service to obtain a reference to its server. This entails an extra TCP/IP connection and several exchanges resulting in a significant further decrease in performance.

The Java-RMI solution has the most complex mechanism; establishment of a connection to the rmiregistry and submission of lookup request, establishment of a connection to an HTTP server and posting of a request to download a class file, and finally exchanges with the actual server. The RMI protocol entails a certain amount of additional data traffic even in this short example with RMI "ping" operations etc that are used to keep open leases on server-side structures. Despite the overheads of the

extra connections and additional traffic, the Java-RMI solution appears better than CORBA.

Intranet style applications will more typically involve a client that submits numerous requests to a server. Here, the additional set-up overheads of the object technologies (contacts with name services, download of stubs) are amortized over many requests and so count for less.

The earlier studies of Elfwing et al. and Davis et al. looked at applications involving multiple requests, comparing average times for requests as obtained from a series of requests (and excluding establishment costs for the object technologies) [4, 5]. In both those studies, the Web Service (SOAP-communication) solutions performed poorly relative to alternatives. A significant factor was the use of HTTP 1.0 in the implementations studied. There were really two problems. Firstly, the HTTP 1.0 connections were not persistent; secondly, there were specific issues relating to delays in the pattern of requests and acknowledgements for the various parts of messages.

The original version of HTTP was intended for the download of individual documents; a client connects, the server returns a document and closes the connection. When documents ceased to be purely textual and started to require supplementary data such a style-sheets or image data, the protocol was seen to be inefficient as each supplementary file transfer required re-establishment of a TCP/IP connection. A "keep-alive" feature was introduced on the connections. The server would keep the connection open for a short (configurable) time after returning an initial document; a client could submit supplementary requests over this open connection. This feature was not formally part of the HTTP 1.0 protocol and was only implemented on an ad hoc and frequently inconsistent manner. The HTTP 1.0 used with earlier implementations of SOAP communications did not exploit the "keep-alive" feature. Consequently, each request in a sequence involves establishment and shut down of a TCP-IP link.

Elfwing et al. studied delays associated with the closure of the TCP/IP connection after each HTTP 1.0 request. The protocol used a "graceful" closedown with a TCP/FIN message from server to client, a client TCP/ACK and then TCP/FIN, and final TCP/ACK from the server. The problem was not in the exchange of TCP/IP handshake messages but in an initial delay of ~0.4 seconds prior to the server initiating the closedown. One of the changes made by Elfwing et al. was to modify the `java.net` code to allow the client to initiate the graceful closedown sequence. Both Davis et al. and Elfwing et al. identified other problems with delayed acknowledgements of the request header.

These problems are significantly reduced in the current HTTP 1.1 implementations. Firstly, HTTP 1.1 incorporates the "keep alive" feature as standard. The time-out on the server-side defaults to about 60 seconds in the JAXRPC configured Tomcat; so if a client process submits a subsequent request within this time there is no need to close the original and establish a new TCP/IP connection. The closedown sequence has in any case been changed; the client issues a TCP/RST and shuts down communications; there is no longer any need to modify low level `java.net` socket code.

Another significant change that comes with HTTP 1.1 is the use of a "chunked" response format. A HTTP 1.0 response should contain a `Content_Length` header; this necessitates buffering of an entire response in the server, which could be problematic for applications such as the "large data" application (the book records in that example exceed 1000 bytes and a response can have as many as 600 records). With "chunked" encoding, the server can send chunks with length component, data, and a separator. The chunked XML data can be processed on the receiving side because SAX style parsers are used rather than DOM parsers.

Earlier studies looked mainly at simple operations where the responses were single integers or strings. Such small responses can often be fitted into single packets. More realistic applications will have large responses that must be split into separate packets. The HTTP 1.1 implementation tries to use relatively large sizes (1460 bytes data, this is about the largest data packet that can be transmitted as a single unit over typical Ethernet connections). If a CORBA response is too large to be sent as a single unit, it is also fragmented (IIOP fragments); the Sun Java 1.4 implementation of CORBA uses a fragment size of 1024 bytes. A Java-RMI implementation returns large structures with RMI-continuation messages that are of apparently arbitrary size (with the "large data" demo, successive RMI-continuation messages in a response had sizes such as 338, 242, 497, 409, ...).

The results from this examination of protocol imply that the concerns raised by Elfwing et al. and Davis et al. have been largely addressed with the adoption of HTTP 1.1. The actual data traffic between client and server is reasonably efficient; it is simply that there is a lot more data to be transferred for Web Service implementations as compared with the object technologies.

3.2. Costs of document transfer

Table 2 shows results of traffic analysis for each of the technologies. These data show the total byte transfers and total number of packets and are averages of repeated tests. (The "iterator" variants are discussed in section 3.5.2.)

The image of large XML documents replete with textual data, mark-up tags, and data type specifiers had lead to an expectation of even poorer performance from the JAXRPC solutions. The relative performance is poor with the simpler data transfers, as in the calculator example, but the difference is less marked where the data volume is greater (the "large data" example includes numerous string elements, some quite lengthy; here mark-up and related overheads count for less). The JAXRPC solutions require from three to five times as many data packets as the alternatives, and from three to ten times as many bytes. (There are proposals relating to the use of text compression for SOAP data transfers, but nothing is yet standardized.)

The CORBA implementation performed best with the large data transfer example. Generally, Java-RMI has a superior performance [11, 12]. Here CORBA has an advantage over Java-RMI primarily because of its use of an 8-bit character encoding for data transfers as opposed to Java-RMI's 16-bit coding. A secondary factor is Java-RMI's inclusion of some class meta-data in responses that include class instances such as strings. The Java-RMI solution performed relatively poorly in terms of total number of packets in the large data case; this relates to its selection of a small size for continuation messages.

The five-fold or so increase in data transfer costs for Web Service style implementations may be tolerable for an intranet application. Local bandwidths should be high, and local switched Ethernet systems should reduce contention. If the application requires encryption of data traffic (not yet standardized for Web Services) then the larger amounts of data will incur additional processing costs for encryption; however, with intranet usage, encryption may not be vital.

3.3. Costs of XML generation and parsing

There are time and space costs associated with the use of XML encoding. Table 3 shows the measured CPU times for applications on both client (SunBlade, 650MHz) and server (Sun v480 1030MHz) as measured by Unix "time" command. The server required ~115s to generate 5000 large data records without any output of these records (same for all technologies). In this case, the data in the table include overall time and estimated time for technology-specific communications.

In terms of CPU usage, Java-RMI is generally the best. However, the test with large data example shows poor performance on the client-side (the Java-RMI solution requires something like 2.8 times as many packets as the CORBA solution and this seems to be impacting client-side performance more than server-side performance). The JAXRPC solutions require up to six times as much CPU power on the server; the client-side parsing of the large data documents has an even higher cost. The Java-RMI solution was consistently the best in terms of overall runtimes (actual elapsed time for the client to complete its sequence of requests); the CORBA solutions take about 10% longer. This is despite the fact that RMI exchanges many more messages than CORBA does in the "large data" case.

Example	Technology	Packets	Total data transferred
"Calculator"	WebServices	48,931	10,360,814
	CORBA	10,007	1,400,851
	Java-RMI	10,098	1,017,477
"data"	WebServices	55,617	16,053,312
	CORBA	10,007	2,236,661
	Java-RMI	10,050	2,451,790
"large data"	WebServices	1,143,608	1,134,974,047
	CORBA	475,235	344,363,683
	Java-RMI	1,354,377	449,330,931
"iterator"	CORBA	501,266	348,321,700
	WebServices	1,257,678	1,157,156,203

Table 2: Traffic analysis for illustrative applications with different technologies

Application	*"calculator"*		*"data"*		*"large data"*	
Technology	**Client CPU**	**Server CPU**	**Client CPU**	**Server CPU**	**Client CPU**	**Server CPU**
WS	15.0s	6s	22.8s	8.4s	1087s	551s (436s)
Java-RMI	2.3s	0.8s	4s	1.1s	148s	212s (97s)
CORBA	3.2s	1.9s	3.6s	2.1s	54.2s	250s (136s)

Table 3: Measured CPU times for client and server with varying technologies and applications.

The other resource used is memory space in client and server. The current use of chunked responses with HTTP 1.1 eliminates one major space requirement on the server-side - the buffering of a complete response prior to its transmission. Significant memory may still be used for XML parsing on both client and server. (All parsers are SAX-style, so there is never any need to build a complete parse tree as for a DOM parser; such a tree-structure would incur much larger memory costs.) In this study, approximate memory measures were recorded and only on the client-side.

Memory usage was averaged over a series of tests runs on the "large data demo. Average memory usage for the systems were: JAXRPC 4.75Mbyte, Java-RMI 2.6Mbyte, CORBA 2.0Mbyte. "Iterator" based models (where large collections a returned in blocks whose size is client-selected) reduce memory usage. The mechanisms are discussed below in section 3.5.2. The difference in memory usage is small in the CORBA implementation; the JAXRPC memory usage was reduced to ~2.1Mbyte.

3.4. Common implementation for Web Service and Java RMI

If a service has heavy internal use on an intranet, and also some external Internet users, then one possible approach is to use a common implementation for both JAXRPC and Java RMI services.

The typical Java-RMI server extends `java.rmi.server.UnicastRemoteOject`; a JAXRPC implementation class does not. However, a Java-RMI server can use the implementation created for the JAXRPC system; it simply has to connect the server object to the Java-RMI runtime system and register it with the rmigrestry naming service.

A dual technology solution might be expedient in some situations where the lower performance of a JAXRPC server was perceived as problematic. However, it is unlikely to prove a long-term solution. There would be pressures for divergent implementations to evolve. State has to be hacked in Web Service models, while being handled naturally in Java-RMI. Developers of a JAXRPC style servlet based service will naturally tend to want to take advantage of services offered by the servlet container.

3.5. State

The traditional design for a stateful service, as in CORBA and RMI, involves two classes; there is a factory class, and a stateful server class. The factory class is a singleton class, an instance of the factory is created when the server process starts and it is the identity of this factory object that is published in naming services. Clients establish initial contact with the factory object and use a "create" method to instantiate a session-server object; the create method returns an object-reference that the client builds into a stub. It is this object that is subsequently used via the client's stub and, hopefully, is neatly disposed of when the client's session is completed. The typical tutorial example has a "calculator-factory" and "calculators" for the individual clients. The state data for a calculator will be the contents of its register(s).

It is not possible to build a stateful server of this type within the Web Services model because this model does not support the concept of a returnable object reference. Archetypical Web Services are stateless; and developers are encouraged to think only in terms of stateless services [13]. However, if really required, stateful services can be implemented using mechanisms similar to those employed for WWW-services that maintain things such as "shopping carts".

When WWW services grew beyond the simple download of static pages, state became an issue and "hacks" had to be developed to permit state maintenance. Solutions involving the server sending existing state data back to the client as part of an intermediate response have limited application. The typical solution has state data maintained in the server (in memory, file, or database) with simply a client identifier key sent back to the client. This key can be held in a "cookie" that will be returned to the server or may be embedded in the path of all URLs.

Either way, the key gets returned as part of subsequent invocations. These WWW-hacks can be adapted to Web Services that use HTTP.

Details vary but the major implementations of SOAP/HTTP servers and clients all support state via cookies (or if necessary via URL-rewriting, though this appears poorly supported in the .Net implementation [14]). With JAXRPC, the server-side defaults to having session support; with .NET and Apache-SOAP, server-side support can be enabled via configuration variables. The client-side must also enable session handling, and may need to allocate and register structures to hold cookie data.

3.5.1. Implementing a stateful service. Service containers, such as Tomcat or the .Net system, typically support in-memory session state via some form of hash-map. The container hides all the detail of dealing with the cookie or re-written URL, supplying instead an operation that the service object can use to retrieve the hash map with the session data appropriate to the current client.

Direct use of data in the hash map results in somewhat unnatural code. For example, a stateful JAXRPC calculator service would have to store its register value in a `java.lang.Integer` variable in this session hash map. The code would be something like the following:

```
public long add(long val)  {
   int result = 0;
// First get session for client
   HttpSession session = null;
   try{
      session = endPointContext.
         getHttpSession();
   }
   if(session!=null) {
// If existing Session, get state data
// variable with current value
         if(!session.isNew()) {
           Integer cVal =
              (Integer) session.
        getAttribute("myRegister");
           result = cVal.intValue();
         }
   }
   result += val;
   if(session!=null) session.
      setAttribute("myRegister",
                new Integer(result));
   return result;
}
```

CORBA's POATie mechanism offers a model for a more natural implementation of a stateful service. The POATie mechanism has two classes that implement the same operations interface as defined for the service; the POATie class deals with issues of integration with the ORB and delegates all actual work to an independent implementation class. If this model is used for something like a JAXRPC calculator service, one would have a `Calculator` class and a `java.rmi.Remote` class ("`Demo`") that ties in with the servlet container. The `Calculator` class would use data members to store state in a natural way. The tie-class, `Demo`, would create an instance of this `Calculator` class as its state data saved in the system provided session hash map. Code for the tie-class would be something like the following:

```
public class DemoImplementation
   implements Demo,
   javax.xml.rpc.server.ServiceLifecycle
{
   private final String
      myKey = "DemosCalculator";
   private ServletEndpointContext
      endPointContext;

   public void init(Object cntxt) {
     endPointContext =
      (ServletEndpointContext) cntxt;
   }

   public void destroy() {    }

   public long add(long val) {
      return getCalculator().add(val);
   }

   ...

   private Calculator getCalculator() {
// Retrieve real object from session
// (or create it if necessary)
// code, as above, to get session
      ...
      if(session!=null) {
         if(session.isNew()) {
           Calculator c =
              new Calculator();
           session.setAttribute(
              myKey, c);
           return c;
         }
         else return (Calculator)
            session.getAttribute(myKey);
      }
      // problem - no session
      ...
    }
}
```

3.5.2. A stateful iterator. Although often discouraged [13], a stateful Web Service may be of advantage in cases where large quantities of data must be returned. In CORBA, it has been traditional to define search operations that may result in large amounts of response data as yielding references to stateful iterators rather than sequences of records.

On the server-side, the stateful iterator is created with ownership of a collection of those records that are to be returned, and a counter that identifies the subset of records that will be returned in the next request. The client gets a stub that allows it to use such an iterator to retrieve data segments of specified size. The server-side iterator and its data collection are discarded when all data have been returned.

This model can be adapted to Web Services. Of course, a search operation cannot return a reference to an iterator object. It can instead create the iterator, and make this an element in the client's session-data record. An additional method can then be provided that retrieves records from this iterator. The "large data" example was adapted to use a very simple stateful iterator. The revised service definition is:

```
public interface Demo extends Remote {
   public Data2[]
      getMoreRecords(int blksize)
         throws RemoteException;
   public int search(String request)
         throws RemoteException;
}
```

The implementation of the search function finds the required records and stores them in an in-memory collection. When the search is complete, the search function creates an instance of an `Iterator`, giving it the collection and storing it as a state variable in the client's session. The `getMoreRecords` method retrieves this state variable and invokes a method that returns the next block of records.

A naive implementation of an `Iterator` class requires less than twenty lines of Java.

The client application was modified to use this iterator-based service, requesting result records to be returned in blocks of at most fifty. Test results of this version of the application showed a 10% increase in the total number of packets transmitted, and only about 2% increase in byte transfers. It does however lead to considerably less memory usage in the client; this version of the client having an apparent maximum memory footprint of 2.1Mbyte instead of the 4.6Mbyte for the standard implementation.

4. Conclusions

Improvements in implementations of SOAP communications have significantly reduced the performance failings noted in earlier studies. While a Web Service solution will still be slower, consume more memory, more network bandwidth, and more CPU cycles than an alternative solution, the differences are less marked in realistic applications such as the "large data" application than in toys like the "calculator" application.

The substantial commonality of code base for the various technologies makes dual deployments possible - a Web Service implementation for external users and an alternative higher performance variant of internal use. Some problems associated with very large responses from a Web Service can be alleviated through the use of a stateful iterator approach.

The example applications did not involve any sophisticated features such as encryption and transactions. Currently, there are competing proposals for Web Service transaction management, and there is an elaborate network of related investigations into security frameworks. However, these features are still at prototype stage. If such features are required, developers will have to look to commercial CORBA implementations. (There are transaction managers and security systems for freeware CORBA but a lot of work is needed to develop an application that uses these in a reliable manner.)

References

[1] W3C Organizations Web Services definitions, http://www.w3.org/2002/ws/

[2] Java: Remote Method Invocation; http://java.sun.com/j2se/1.4.2/docs/guide/rmi/index.html

[3] CORBA: Common Object Request Broker Architecture, http://www.omg.org.

[4] Elfwing, R., Paulsson, U., and Lundberg, L, *Performance of SOAP in Web Service Environment Compared to CORBA,* In Proceedings of the Ninth Asia-Pacific Software Engineering Conference, IEEE, 2002

[5] Davis, D., and Parashar, M., *Latency Performance of SOAP Implementations,* In Proceedings of 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE, 2002.

[6] Gorton, I., Liu, A., and Brebner, P., "Rigorous evaluation of COTS middleware technology"*, Computer*, IEEE, March 2003, pp. 50-55.

[7] JWSDP Java Web Services Developer Pack http://java.sun.com/webservices/jwsdp/index.jsp

[8] *Tomcat server:* http://jakarta.apache.org/tomcat/index.html

[9] Java 1.4 CORBA implementation; http://java.sun.com/j2se/1.4.2/docs/guide/corba/index.html

[10] "*Ethereal: a free network protocol analyzer"*, http:// www.ethereal.com

[11] Juric, M.B., Rozman, I., and Hericko, M., *Performance Comparison of CORBA and RMI,* Information and Software Technology 42, pp 915-933, 2000.

[12] Juric, M.B, Rozman, I., Stevens, A.P., Hericko, M. and Nash, S., *Java 2 Distributed Object Models Performance, Analysis, Comparison, and Optimization,* In Proceedings of 7th International Conference on Parallel and Distributed System, IEEE, 2000.

[13] Zimmermann, O., Tomlinson, M., and Peuser, S., *Perspectives on Web Services*, Springer-Verlag, Berlin, 2003.

[14] Powell, M., *Using ASP.NET Session State in a Web Service,* as http://msdn.microsoft.com/library/ default.asp?url=/library/enus/dnservice/html/ service08062002.asp