2005

Performance of Java Middleware - Java RMI, JAXRPC, and CORBA

N. A. B. Gray
University of Wollongong, nabg@uow.edu.au

Publication Details

This conference paper was originally published as Gray, NAB, Performance of Java Middleware - Java RMI, JAXRPC, and CORBA, The Sixth Australasian Workshop on Software and System Architectures (AWSA 2005), Brisbane, 29 March 2005. Original conference information available here

Performance of Java Middleware - Java RMI, JAXRPC, and CORBA

Abstract
Developers of distributed Java systems can now choose among Java-RMI, CORBA, and Web-Service (JAXRPC) middleware technologies. Performance is one factor that has to be considered in choosing the appropriate technology for a particular application. The results presented in this paper show that the nature of response data has a greater impact on relative performance than has been allowed for in most previous studies. Relative performances of the technologies as measured on simple requests and responses are not representative of the behaviour that can be expected in practical applications.

Keywords
Middleware, Java-RMI, JAXRPC, CORBA, Java systems

Disciplines
Computer Engineering

Publication Details
This conference paper was originally published as Gray, NAB, Performance of Java Middleware - Java RMI, JAXRPC, and CORBA, The Sixth Australasian Workshop on Software and System Architectures (AWSA 2005), Brisbane, 29 March 2005. Original conference information available here

Performance of Java Middleware - Java RMI, JAXRPC, and CORBA

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

Abstract

Developers of distributed Java systems can now choose among Java-RMI, CORBA, and Web-Service (JAXRPC) middleware technologies. Performance is one factor that has to be considered in choosing the appropriate technology for a particular application. The results presented in this paper show that the nature of response data has a greater impact on relative performance than has been allowed for in most previous studies. Relative performances of the technologies as measured on simple requests and responses are not representative of the behaviour that can be expected in practical applications.

1. Introduction

Distributed object systems, implemented in Java, can now be created using Java-RMI [1], CORBA [2, 3], and WebService (JAXRPC) [4, 5] technologies. Java-RMI and CORBA allow the implementation of sophisticated server-side architectures, but in many cases only a simple stateless server is required. WebServices can be adapted to support stateful services, but are primarily intended for stateless servers. The discussions in this paper focus on such stateless servers. A typical application would use such a server to interrogate or update a database. The appropriate technology for a specific application will be determined by a variety of factors including ease of programming, stability and ease of deployment, and performance. Published studies of empirical tests of performance can serve as a guide; this paper identifies a weakness, relating to network usage, that appears in many of these earlier studies.

The alternative technologies are similar with regard to ease of programming. In all cases, developers start with an interface that declares the operations that define the service. This interface will be a Java `Remote` interface for Java-RMI (or optionally for WebServices), or an IDL interface for CORBA, or a WSDL interface [6] for WebServices. Automatic code generators create client-side-stub classes, and corresponding server side “skeletons”. The client-side code that must be written by the developer is essentially identical for all technologies, differing only in the few lines needed to obtain a proxy (stub) object for the server. For the server side, the developer must define a class that either implements an interface or extends an auto-generated base-class. This server implementation class contains the business logic; its coding is almost identical for all technologies. For Java-RMI and CORBA, the developer must also create a simple driver program that instantiates an instance of the server class, binds it to the low-level object request broker runtime system, and publishes its identity. In a JAXRPC WebService, a standard servlet-based framework performs the equivalent services.

The auto-generated stubs and skeletons hide the networking and data-marshalling aspects. CORBA and Java RMI use communications protocols (IIOP and JRMP) that directly overlay the TCP/IP layer. WebServices, and optionally Java RMI, work with the HTTP protocol for requests and responses. Inevitably, HTTP-based systems are less efficient than the binary protocols but they do have advantages when the clients and servers in a distributed system are separated by firewalls. Firewalls are typically configured to admit HTTP traffic, but security administrators are loath to open up additional ports in firewalls or to install bridging applications in the firewall as needed to allow communication to arbitrary ports as used by CORBA and Java-RMI. The choice of technology for a particular distributed application will to some degree depend on an assessment of the additional costs of HTTP-based mechanisms (both processing cost and increased network traffic) as compared to their advantages with regard to firewall configuration. If an HTTP-based approach is adopted, the relative performances of different HTTP-based implementations must be assessed.

Quite apart from considerations of firewalls, performance may need to be traded against other deployment and stability issues. JAXRPC WebServices are deployed in WWW-servers such as Tomcat [7]. Such servers have persistent configuration data and will automatically restart all services after a system failure; systems administrators will be familiar with such servers and will not require training. CORBA systems typically have a CORBA-daemon process that restarts actual servers when needed, and a `CosNaming` nameserver process (in Sun’s Java 1.4, these tasks are combined in the `orbd` process). While a CORBA system is typically stable and fairly easy to administer, the technology is much less likely to be familiar to systems administrators.

Java-RMI has its `rmiregistry` program that fills a nameserver role; but this does not use persistent data defining its servers and cannot restart servers at a system-wide restart. Each server, or group of activatable servers, must be reregistered by launching the programs manually or via a script. In many ways, Java-RMI is the least resilient of the technologies; its apparent advantage has always been seen as its higher performance.

With the choice of technology often being determined by a trade-off between performance and configuration/ deployment issues, developers need to have a clear understanding of the factors affecting the performance of the technologies. The main contribution of this paper is to present data on comparative performance of these Java middleware technologies. The work extends a number of similar previous studies, taking into account some details of the actual data transfers on the network that have been inadequately considered. Section 2 of the paper summarizes previous studies; section 3 reports on the results from a series of tests, and section 4 presents concluding remarks.

2. Earlier studies

Juric et al. [8, 9, 10] have published results for a series of studies comparing Java-RMI and CORBA. In [8], they compared Java implementations of RMI and CORBA (the Visibroker implementation). The performance measures were based on roundtrip times recorded for invocations of service operations that returned simple data types or strings; typically, the time for 200 invocations was measured. For the simple data types, RMI performed significantly better than CORBA. However, CORBA showed better performance when returning large strings; this was attributed in part to CORBA using 8-bit characters for strings where Java-RMI was expected to use 16-bit characters. The later two papers in this series included measures of the original and optimized versions of RMI-IIOP.

Buble et al. [11] have identified problems with a number of such comparative studies. In particular they note the need for a “warm-up” period prior to measurements of latency times for communications. Each middleware system will involve activities such as priming of caches, and stabilization of adaptive resource allocation algorithms; further, Java “just in time” compilation cuts in at some point. Illustrative results from Buble et al. show quite dramatic decreases in roundtrip invocation times after about 1000 invocations, and further decreases after longer warm-up times. In their example study, some 15,000 requests were needed before round-trip times stabilized and showed no further decreases. The effect of warm-up is obviously important for servers that run for extended periods handling thousands of requests from various clients. Its importance for clients is less clear. Most applications have clients that connect for relatively short periods of time and submit only small numbers of requests. Comparisons based solely on round-trip times are probably best done using “warmed-up” clients, though actual timings in practical cases will be longer.

In their more recent study, Demarey et al. [12] have presented extensive results on benchmarking round-trip latency for various Java middleware platforms. They have tested numerous Java CORBA implementations, Java RMI, Java XML systems, and less common systems such as OpenCCM and Fractal. Their benchmark involves timing a `void ping()` operation on the server. These measurements were made on “warmed up” systems and incorporated other improvements as suggested by Buble et al. [11]. In their study, the ORBacus 4.1 implementation of CORBA actually outperformed Java-RMI (by about 7%); though most of the other CORBA versions had considerably inferior performances (by up to 500%). Their view was that a well implemented version of GIOP/IIOP could provide better performance than ad hoc protocols such as Java RMI's JRMP, and that WebServices platforms incurred high overheads that would likely preclude their use in cases where distributed systems would interact strongly.

The emergence of WebServices as a contender resulted in initial studies by Elfwing et al. [13] and by Davis and Parashar [14]. Elfwing et al. compared Web Services to CORBA, both in Java implementations, finding a degradation factor of up to 400 in performance. This performance impact could be reduced by recoding the `java.net` libraries to work around problems with HTTP as used by the WebService. Davis and Parashar compared several WebService systems (Java, Perl, .Net) with Java-RMI and CORBA. They noted similar problems with the behavior of HTTP. Their final conclusion was that Java-RMI was preferable to the Web Service systems available at the time of the study. The poor performance of WebServices in these two studies was due in large part to their use of the HTTP-1.0 protocol [15]. Every invocation of a remote operation required the costly construction and teardown of a new TCP/IP link; delays in the mechanism for closing each link caused the largest performance impacts. The current JAXRPC implementations of WebServices use HTTP-1.1 where the “keep-alive” protocol feature permits many requests to be made once a connection has been established.

In more recent work, Juric et al. [16] have compared Java-RMI, RMI-HTTP tunneling, and JAXRPC WebService implementations. This study allows some measure of the impact of a firewall on a system's likely performance. Their results showed that Java-RMI was more than 8 times faster than JAXRPC, which was again more than 3 times as fast as RMI-HTTP tunneling through a servlet based web server.

This study extends previous work in just two aspects. Firstly, an examination of recorded network traffic had shown anomalies in the relative network performance of Java-RMI and CORBA implementations when the size of a response message was changed. Investigation of these anomalies has exposed a deficiency in Java-RMI's handling of larger responses; a deficiency that reduces the generally observed performance advantage of Java-RMI. Secondly, examination of data traffic had also pointed to variations in the relative performances of JAXRPC and RMI-HTTP tunnelling. Again, the relative performances of the technologies change with the nature of the data being returned.

These empirical observations motivated the performance study reported here. The study examines firstly the simple case where the invoked server operation returns a character string of fixed size. For these examples, the results are consistent with most previous studies in that Java RMI performs best, RMI-HTTP worst. However, such simple requests are atypical of actual remote services. The rest of the study looks at invocations that return structured data (e.g. a small array of records such as might be obtained by a SQL select query on a datatable). With these data, the relative performance of the different technologies changes and CORBA proves most effective and WebService (JAXRPC style) most costly. The change in performance relates to the way that data are organized into packets for transmission on the network.

3. Performance investigation

This study used the Sun Java 1.4.2 reference implementations throughout. The tests were done on a system with a 100Mb switched Ethernet connecting a Dell Optiplex GX260 with a 2GHz CPU and 512Mbyte memory running Windows XP and a SunBlade 100 workstation running Solaris (tests were run with each machine in both client and server roles, differences in the implementation of TCP/IP result in some minor performance differences when a client creates a connection to submit only a single request). In their study, Demarey et al [12] ran both client and server processes on the same machine to avoid network perturbations; this configuration proved impractical for this study as a server system, such as the RMI-HTTP system with Tomcat/rmiregistry/server, ran at about 80% CPU utilization in some tests and simultaneous execution of the client on the same system would have just resulted in contention amongst processes. Network traffic analysis was conducted using the Ethereal tool to capture packets. Tomcat 5 was used as the servlet application server for the JAXRPC system, and as both the class file server for Java-RMI and host for the servlet used to enable RMI-HTTP tunneling. The tests did not involve any actual firewall; RMI-HTTP tunneling was forced using available configuration options in Sun's RMI implementation.

As Buble et al. [11] have noted, round-trip call times only become reliable after several thousand operations have been invoked so as to "warm-up" the systems. A server will often run for days without being restarted, so its "warmed-up" performance is appropriate. Clients do not typically make tens of thousands of requests. After all, the prototypical WebService example is a "stock quote" service where the client makes a single request. In this case, a WebService solution will perform best because it involves only a single connection to the server whereas both RMI and CORBA solutions will involve initial contact with a naming service to find the server (and RMI has the additional overhead of needing to contact a web-server to download client stub class files). Client performance is measured here with programs where the client connects and then runs a number of cycles, in each cycle measuring the time for 1000 operation requests. Buble et al. [11] note that the use of such aggregate times for many invocations loses a lot of information that could be used to characterize details of middleware performance, but these aggregates will suffice when looking for major differences in performance of different technologies. In this study, it was observed that the time for the first 1000 invocations could be twice that of the eventual stabilized time, but in most cases the times had stabilized after about 3000 invocations.

Demarey et al. [12] use a simple `void ping()` operation in their performance tests. Juric et al. use a server with an interface such as the following:

```
public interface IPerformanceTester
     extends java.rmi.Remote {
     int getInt() throws RemoteException;
     ...
     double getDouble() throws ...;
     String getString()
          throws RemoteException;
}
```

For this study, three different interfaces were used. The first server defined a `getString()` operation; the effect of string length on performance was an issue. (All `java.lang.String` data elements used in this study were in the default ISO-Latin font.) Performance was tested with strings of varying sizes in the range 64-8192 characters. HTTP-systems, IIOP systems, and JRMP (Java-RMI native encoding) have different approaches to choosing packet size and how to handle continuation packets when a response is large. The Java-RMI mechanisms (JRMP) appear to be the least effective; though they are not too troubled by regular sized structures such as strings.

The second server was defined via the following class and interface:

```
public class Data2 implements
     java.io.Serializable {
     private long          _d1;
     private String[]      _d2;
     private double[]      _d3;
     // accessor and mutator functions etc
     ...
}

public interface Demo extends Remote {
     public Data2[] f1(String str)
          throws RemoteException;
}
```

In this case, the server implementations each generated an array of `Data2` objects as the response to every `f1()` invocation. Each `Data2` had eight doubles and four strings whose lengths were varied “randomly” (using identical random sequences in every test run). A typical `Data2[]` was about 900 byes but they could exceed 1Kbyte.

The final server simulates a system that returns customer records retrieved from a database. It is defined in terms of the following classes and interface:

```
public class Address implements
     java.io.Serializable {
     private long   _postcode;
     private String _unit;
     private String _street;
     private String _city;
     private String _state;
     // accessor and mutator functions etc
     ...
}

public class SimpleDate implements
     java.io.Serializable {
     private long   _day;
     private String _month;
     private long   _year;
     // accessor and mutator functions etc
     ...
}

public class Order implements
     java.io.Serializable {
     private SimpleDate    _date;
     private String        _productcode;
     private int           _number;
     // accessor and mutator functions etc
     ...
}

public class Data3 implements
     java.io.Serializable {
     private String       _customer;
     private String       _salesrep;
     private Address      _address;
     private Order[]      _orders;
     // accessor and mutator functions etc
     ...
}
public interface Demo3 extends Remote {
     public Data3[]
          getCustomersForSalesRep(
               String salesrep)
               throws RemoteException;
     public Data3 getCustomerRecord(
          String customer)
               throws RemoteException;
}
```

The intent of this example was to explore the impact of any inefficiency in the SOAP XML encoding of structs containing structs, or the encoding of arrays of such structures.

3.1 Regular string data

Table 1 shows performance data for the `getString()` example with varying string sizes. The data include stabilized time for 1000 invocations, total data traffic for 10,000 invocations, and total number of packets. (The times shown are averages of the five lowest records. There can be slight variations in numbers of packets and total bytes transferred due to things like occasional “ping” requests with JRMP or acknowledgements of groups of packets rather than single packets; these variations are < 0.1%.)

These results are consistent with previous studies. Java-RMI (JRMP) is the most efficient in terms of both elapsed times and total data transfers. CORBA performs second best; its relative performance compared with RMI improves somewhat with larger strings. JAXRPC performs less well, but is better than RMI-HTTP.

Each protocol has its own “wire” representation for the data. All incur overheads that are significant for small sized responses such as the 64-character strings but become less marked for long strings as shown by the ratios of data on the wire to content data.

RMI’s JRMP protocol simply serializes the objects that are transferred. In this simple case, the response data consist of a single string object that is sent with its class specification and the content character data. If the data represent an array of objects (or a graph of objects) with duplicates (e.g. an array of “date” objects with duplicate string data for day and month names), the serialization mechanism encodes subsequent occurrences of an object by back-references rather than by duplicating the data; this typically results in some data compression. Contrary to reports in previous studies, RMI’s JRMP does not necessarily utilize a 16-bit character representation; the ISO-Latin strings used by the test application are transferred as 8-bit characters on the wire. The packets transferred for JRMP comprise those for exchanges with `rmiregistry` when finding the service, those needed to download the client stub class file, a few “ping” requests and responses, and the RMI “call” and “response” messages. JRMP has a relatively small preferred packet size for response data; when the string length exceeds about 300 bytes, JRMP splits the response into a first packet and one (or more) continuation packets. As will be illustrated by the data in the following sections, it is this approach to handling continuations that can lead to RMI-JRMP performing less effectively than CORBA-IIOP.

Technology	Length of string returned	Packets (10,000 invocations)	Total bytes transferred (10,000 invocations)	"Stabilized" time for 1000 invocations	Ratio data sent / actual content data
CORBA	64	20,034	3,374,400	1.29	5.3
	512	20,034	7,854,420	1.50	1.5
	4096	80,032	48,434,269	2.86	1.2
RMI-JRMP	64	20,100	2,412,400	0.77	3.8
	512	40,130	8,404,000	1.19	1.6
	4096	100,129	48,345,000	2.04	1.2
RMI-HTTP	64	40,965	7,948,300	11.42	12.4
	512	40,968	12,438,400	11.85	2.4
	4096	61,198	49,390500	13.57	1.2
JAXRPC	64	100,816	19,868,900	4.84	31.0
	512	100,816	24,348,800	5.04	4.8
	4096	130,869	61,830,000	7.03	1.5

Table 1 Performance of technologies for varying string length.

CORBA-IIOP generally performs best with regard to the number of packets transmitted. The IIOP data traffic comprises a few initial exchanges with a `CosNaming` nameservice to locate the desired service, and then the CORBA request and response packages. IIOP has a 1024 byte packet limit; larger responses require additional continuation packets. The IIOP overhead on a packet is higher than that for RMI-JRMP, but this extra overhead is offset by the reduction in the number of packets for large responses.

The RMI-HTTP has essentially the same data transfers as RMI-JRMP, but with the data being carried in HTTP text packets. The HTTP exchanges involve separate header and content packets for both requests and responses. Response data are buffered in the server and sent in large response and continuation packets (~1500 bytes). Apart from very short strings, the total data sent are only a little greater than for RMI-JRMP and actual packet use can be less because of the larger data and continuation response packets. Processing costs on the server are high, and the response times here are the poorest. (The system configuration used had a Tomcat WWW server hosting a servlet that acted as an intermediary to separate `rmiregistry` and server processes "behind the firewall"; every request and response involves additional inter-process communications between Tomcat and the application process.)

The JAXRPC solution is the most costly in terms of total data transfer and packets. Each request requires a header and a content packet; each response has a HTTP header packet, a SOAP envelope packet, one or more content packets with the rpc-encoded SOAP response XML data, and a final terminator packet. For this simple example, the extra data sent are those needed to encode the SOAP envelope; the actual excess data for the XML markup consist of a simple tag specifying the field name and type, `xsd:string`, of the response string. These overheads are naturally most notable for short strings. Although sending more packets and more total data than the RMI-HTTP solution, JAXRPC exhibits better response times. In the JAXRPC solution, the actual server object is instantiated in the same JVM in the same Tomcat process as the standard servlet that handles HTTP-communications and XML-encoding and decoding; there are no extra inter-process communication costs on the server side.

3.2 An array of simple "structs"

Table 2 shows performance data for the example where the server returns a small array of simple "structs" each containing integer, double, and string fields.

The relative performances of the different technologies now change quite radically. CORBA-IIOP shows the best performance in terms of number of packets, total data transfer, and round-trip times. RMI-HTTP and JAXRPC are now more equal in performance; JAXRPC performs slightly better with the small arrays but considerably less well for the larger array.

The data records, for the array size 4, in this example vary a little in size, most being about 900 bytes. Most such records can fit within CORBA-IIOP's 1024 response packets; a few require IIOP continuation packets. Java's JRMP protocol has much greater problems with these data. Rather than a single packet, JRMP uses several packets for the array (most likely encoding each struct in a separate continuation packet). Many more small packets are necessary. The total data transferred are only about 10% greater for JRMP than for IIOP. JRMP takes its performance hit because of the much greater number of packets that must be processed – with all the overheads of switching between JVM and the underlying native interface code that implements the actual TCP/IP handlers.

Technology	Array size	Packets (10,000 invocations)	Total bytes transferred (10,000 invocations)	“Stabilized” time for 1000 invocations
CORBA	4	28,264	13,161,991	1.9
	32	130,397	88,930,600	4.1
RMI-JRMP	4	51,679	15,431,012	2.3
	32	293,246	99,885,950	5.8
RMI-HTTP	4	41,071	18,857,000	13.4
	32	91,668	89,463,665	18.3
JAXRPC	4	130,835	72,015,000	14.0
	32	427,584	437,890,943	78.5

Table 2 Performance of technologies for a server returning a small array of “structs”.

Here, RMI-HTTP is actually more efficient than JRMP when measured in terms of total data transfer and packets, simply because it continues to buffer response data on the server and send ~1500byte HTTP response packets. Of course, RMI-HTTP still has a much higher processing cost and so has slower response times.

The costs of XML encoding of response data now become more apparent and the performance of JAXRPC lags. There are two problems. The main one is that with rpc-encoded SOAP responses, every data field is sent with identifying XML markup tags, and every tag contains attributes with a data type. Consequently, the total amount of data transferred increases, and continuation packets are increasingly required. A secondary problem, more apparent in the third example data set, relates to the form of the SOAP message structure. With arrays, and nested structures, the SOAP XML representation can be quite lengthy. For example, a response array is encoded as essentially an array declaration, then as an array with data elements that are essentially forward references to tagged elements that will appear later in the XML stream, and finally as the individual data elements. Every XML element is tagged with its `xsi:type=xsd:value` attribute, adding considerably to the text that must be transferred.

A typical remote method invocation is going to return an array of structs or a nested struct rather than a simple data type. Since the SOAP XML encoding of such structures is very lengthy, a WebService implementation like JAXRPC necessarily returns much more data, and uses many more packages. This extra data traffic can offset the greater processing efficiency that is apparent when the service returns only simple data types.

3.3 Nested structures

The `Data3` structures, with their nested `SimpleData`, `Address`, and `Order` structures, are typically about 1Kbytes. As expected, the CORBA implementation works best for a server that returns single `Data3` elements, or arrays of `Data3` elements. The IIOP wire representation of the data uses a relatively small number of continuation packets whereas the JRMP encoding uses a large number of small packets. In a test with a client making some 10,000 requests for single `Data3` elements and an additional 10,000 requests for small arrays of `Data3` elements, the CORBA implementation completed 1000 simple requests in 1.3 seconds, and 1000 array requests in 2.6 seconds where JRMP required 2.2 seconds and 4.2 seconds. For the total test run, the CORBA implementation used some 76,000 packets and 38Mbyte of data transfers where JRMP used 149,000 packets for 48Mbyte of total data.

HTTP-based protocols are naturally less efficient. Here, RMI-HTTP performs better than JAXRPC for the arrays but not for the single `Data3` responses. The comparable times for simple requests are 8.6 seconds, and 13.1 seconds for JAXRPC and RMI-HTTP. For the arrays, the times are 39.6 seconds (JAXRPC) and 15.8 seconds (RMI-HTTP). The total data transfer for the WebService solution was some 296Mbyte in 394,000 packets, while RMI-HTTP used 52Mbyte and 99,000 packets.

The complexity of the SOAP-encoding is shown in Figure 1 which illustrates the encoding of a response containing just two `Data3` elements in an array; each containing customer details, an `Address` object, and an `Order` array with a single `Order` element. As can be seen from this example, the encoding and decoding mechanisms will need a large amount of data buffering as each record in the `Data3` array is delivered in separate typed parts, with all `Address` and `Order` elements being defined prior to the first `SimpleDate` element.

The extra costs of generating and transmitting the lengthy XML data soon outweigh the more efficient structure of the JAXRPC solution, making RMI-HTTP the preferred technology for remote invocations that return significant data. A simplified version of the client server system was run for the JAXRPC and RMI-HTTP solutions. This version involved the client making 2000 requests for `Data3` arrays of differing sizes. The results are as shown in Table 3.

Technology		**Array size 1**	**Array size 2**	**Array size 4**	**Array size 8**	**Array size 16**
RMI-HTTP	time	1.3	1.36	1.41	1.5	1.66
	Total bytes data	2,663,145	2,977,242	3,607,242	4,973,958	7,597,547
JAXRPC	time	0.8	1.0	1.31	2.2	3.8
	Total bytes data	7,106,609	9,860,018	15,401,886	27,087,637	48,912,368

Table3 WebService (JAXRPC) and RMI-HTTP performance for complex data structures.

When the array of `Data3` structures is of realistic size, ~8, the extra costs of XML encoding and the increased volume of data, and consequently data packets, completely offsets the more efficient processing of the JAXRPC system.

4. Conclusion

The majority of previously reported studies have used benchmark examples that involve minimal data transfers. Their performance comparisons use "ping" operations that involve neither argument nor result data, or test the response time for operations that return a built-in data type such as a double or a short string. Such studies do provide a measure of one extreme of performance, but the results are not necessarily helpful to those needing to select a technology for practical applications.

Sun's Java-RMI clearly has problems where response data exceed about 300 bytes. Its use of numerous short continuation packets suggest that it may be using a strategy where it flushes the output stream and sends a packet after serializing each individual object in any large composite response. In an intranet, the resulting large number of small packets may cause congestion; on the Internet, there is greater chance of packet loss and hence costly delays as retransmission is required. Although Java-RMI is more efficient with regard to the handling of individual packets than most CORBA implementations, the fact that it may use four or five packets where CORBA uses a single 1024-byte packet can lead to Java-RMI showing a poorer overall performance than CORBA. It should be noted that the Sun Java 1.4.2 CORBA implementation used in this study was the poorest performing of the CORBA implementations reviewed by Demarey et al. [12] in their extremal "ping" test. JacORB 2.1 was more than twice as fast as Java 1.4.2, and ORBacus 4.1.0 was five times as fast performing as well as Java RMI in the extremal test. Contrary to most previous reports, it is possible that a CORBA solution will perform better in practice than a Java-RMI solution.

The results in the comparison of JAXRPC WebServices and RMI-HTTP tunnelling also show that relative performance depends on the types of data transferred. WebServices do not perform consistently better than RMI-HTTP tunnelling. If response data take the form of an array or sequence of structures, as would be typical for a server that is retrieving data from a database table, then the SOAP encoding scheme proves inefficient. The volume of data, and consequently number of packets transferred increases considerably. In these circumstances, RMI-HTTP tunnelling can be faster than a WebService solution.

Developers of distributed object systems would in general do better to prototype their proposed systems. Prototypes that model the kinds of response data that will be generated in the actual application can be used to test the performance of the different technologies. Such tests will provide a much better guide as to the appropriate technology than will the results of extremal tests as reported in the literature.

References:

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

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

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

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

[5] Java API for XML-Based RPC (JAX-RPC); http://java.sun.com/xml/jaxrpc/

[6] *Web Services Description Language* http://www.w3.org/ TR/2003/WD-wsdl20-20031110/

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

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

[9] M.B. Juric, I. Rozman, and S. Nash, *Java2 Distributed Object Middleware Performance Analysis and Optimization*, ACM Sigplan Notices, 35, 2000, 31-40.

[10] M.B. Juric, I. Rozman, I., A.P. Stevens, M. Hericko, and S. Nash, *Java 2 Distributed Object Models Performance, Analysis, Comparison, and Optimization*, In Proceedings of 7th

International Conference on Parallel and Distributed System, IEEE, 2000, 239-246.

[11] A. Buble, L. Bulej, and P. Tuma, *CORBA Benchmarking: A Course with Hidden Obstacles*, in Proceeding of the International Parallel and Distributed Processing Symposium, IEEE, 2003

[12] C. Demarey, G. Harbonnier, R. Rouvoy, and P. Merle, *Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms*,
http://www.lifl.fr/~merle/benchmarking.pdf

[13] R. Elfwing, U. Paulsson, and L. Lundberg, *Performance of SOAP in Web Service Environment Compared to CORBA*, In Proceedings of the Ninth Asia-Pacific Software Engineering Conference, IEEE, 2002, 84-93

[14] 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, 377-382.

[15] N.A.B. Gray, *Comparison of Web Services, Java-RMI, and CORBA service implementations*, in Proceedings of Fifth Australasian Workshop on Software and System Architectures, 2004, 52-63.

[16] M.B. Juric, B. Kezmah, M. Hericko, I. Rozman, and I. Vezocnik, *Java RMI, RMI Tunneling, and WebServices: Comparison and Performance Analysis*, ACM Sigplan Notices, 39 (5), 2004, 58-65.

```
<env:Body>
    response – forward reference to an array
    <ns0:getCustomersForSalesRepResponse
        env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <result href="#ID1"/>
    </ns0:getCustomersForSalesRepResponse>
    a definition of a two element array, forward references to elements
    <ns0:ArrayOfData3 id="ID1"
        env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xsi:type="enc:Array" enc:arrayType="ns0:Data3[2]">
        <item href="#ID2"/>
        <item href="#ID3"/>
    </ns0:ArrayOfData3>
    a definition of a Data3 element – simple string fields, and forward
    references to contained substructures
    <ns0:Data3 id="ID2"
        env:encodingStyle="http://..."       xsi:type="ns0:Data3">
        <_address href="#ID4"/>
        <_customer xsi:type="xsd:string">Customer C1</_customer>
        ...
    </ns0:Data3>
    definition of second Data3 element
    <ns0:Data3 id="ID3" env:encodingStyle="..."      xsi:type="ns0:Data3">
        similar data defining two strings and two forward references
        to later tagged elements
    </ns0:Data3>
    definition of first Address structure
    <ns0:Address id="ID4"
        env:encodingStyle="http:..."  xsi:type="ns0:Address">
        <_city xsi:type="xsd:string">Sydney</_city>
        ...
    </ns0:Address>
    definition of Order[] for first Data3 element – specifying its type
    as array with forward reference to single element
    <ns0:ArrayOfOrder id="ID5"
        env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xsi:type="enc:Array" enc:arrayType="ns0:Order[1]">
        <item href="#ID8"/>
    </ns0:ArrayOfOrder>
    definition of Address structure for second Data3 element in result array
    <ns0:Address id="ID6"
        env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xsi:type="ns0:Address">
        similar data defining second address record
    </ns0:Address>
    definition of Order[] for second Data3 element
    ...
    definition of first actual Order element – simple data fields and forward
    reference to SimpleDate structure
    <ns0:Order id="ID8"
        env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xsi:type="ns0:Order">
        <_date href="#ID10"/>
        <_number xsi:type="xsd:int">1</_number>
        <_productcode xsi:type="xsd:string">Product#1234</_productcode>
    </ns0:Order>
    definition of next Order element
    ...
    definition of SimpleDate structure that completes the first Data3 element
    <ns0:SimpleDate id="ID10"
        env:encodingStyle="..."       xsi:type="ns0:SimpleDate">
        <_day xsi:type="xsd:long">10</_day>
        ...
    </ns0:SimpleDate>
    definition of second SimpleDate structure
    <ns0:SimpleDate id="ID11" ...">
        similar data
    </ns0:SimpleDate>
</env:Body>
```

Figure 1 SOAP-XML encoding of simple array of structs.