An Evaluation of XML-RPC*

Mark Allman
NASA Glenn Research Center/BBN Technologies
mallman@bbn.com

Abstract

This paper explores the complexity and performance of the XML-RPC system for remote method invocation. We developed a program that can use either XML-RPC-based network communication or a hand-rolled version of networking code based on the java.net package. We first compare our two implementations using traditional object-oriented metrics. In addition, we conduct tests over a local network and the Internet to assess the performance of the two versions of the networking code using traditional internetworking metrics. We find that XML-RPC reduces the programming complexity of the software by roughly 50% (across various metrics). On the other hand, the hand-rolled java.net-based implementation offers up to an order of magnitude better network performance in some of our tests.

1 Introduction

The notion of *remote procedure calls* (RPC) was first outlined in [4]. The idea behind RPC is that a software developer is already familiar with the idea of making a procedure call (or a method call in object-oriented programming vernacular). Therefore, to make the development of distributed systems more accessible to all programmers RPCs provide the developer an interface to communications code that is as close as possible to simply making a procedure call. Using this notion the developer does not have to write networking code, thus allowing programmers who do not happen to be experts in developing network code to write large complex systems distributed across any number of hosts on a network.

Many systems have tried to implement the notion of RPC in various ways. Sun RPC was an early, widely deployed and used variant of RPC, with CORBA [15], DCOM, JavaRMI, SOAP [6] and many others following. While these systems are all meant for slightly different environments (e.g., CORBA is an object-oriented version of RPC) they all share the same major goal of making distributed applications easier to write and maintain. In the study presented in this paper we focus on a remote procedure call system called XML-RPC[1].

At its core, XML-RPC defines a framework for transmitting method calls and the resulting responses between processes across hosts. The transactions are encoded in a standard way using XML [7]. The XML-RPC approach differs from more traditional RPC systems in several ways:

- In some RPC systems, such as Sun RPC, the RPC system (with input from the programmer) generates stubs for the programmer to call. For instance, if the programmer wanted to call a remote method `foo()` they would call a local method `foo()` which would be a stub generated by the RPC system. XML-RPC differs from such systems, as it does not generate stubs for the programmer. Rather, XML-RPC provides several primitives for programmers to use to construct method requests and obtain the corresponding response. The XML-RPC system reduces the work required by programmers because the programmer does not have to tightly specify their remote procedures for a stub generator. On the other hand, XML-RPC requires developers to know more about the underlying system than an RPC system that provides a stub generator.
- Since XML-RPC uses a standard XML encoding strategy the system is highly interoperable. A system like Sun RPC can be used across architectures, operating systems and languages. However, the programmer must have the same RPC system on all platforms. On the other hand, XML-RPC represents a loose coupling between hosts. As long as the hosts both work to the specification they can communicate trivially.
- Argument marshaling is a non-issue in XML-RPC since all data is encoded as text before transmission.
- XML-RPC is easily layered on top of existing application protocols (e.g., the HyperText Transfer Protocol (HTTP) [3, 8]). Therefore, integrating XML-RPC with other applications is straightforward.

While the motivation behind RPC (and in particular XML-RPC) is compelling we wondered about the benefits versus the costs. The XML-RPC system is quite flexible and generic and therefore likely not optimized for any particular situation. While this is a feature it may also be a disadvantage if an implementer is trying to obtain good network performance. The goal of this paper is to assess XML-RPC by comparing a set of simple remote procedure calls implemented in XML-RPC, as well as using hand-rolled networking code based on the java.net package. While we use the Java version of the XML-RPC system, we note that XML-RPC is implemented for many other languages.

Our comparison of manually written networking code with

*This paper appears in ACM Performance Evaluation Review, March 2003.

[1] http://www.xmlrpc.org

XML-RPC-based code has two major thrusts. First, we compare the two systems in terms of object-oriented code metrics. That is, we assess the difficulty of writing and maintaining the two variants, as well as the complexity of the resulting code. The second area for comparison is in terms of the network performance attained by the two variants.

The remainder of this paper is organized as follows. § 2 outlines our testing program and the environment in which our tests were conducted. § 3 outlines the results of applying object-oriented metrics to both versions of the communications subsystem that we implemented. § 4 outlines the results of applying traditional network performance metrics to our two versions of networking code. § 5 discusses using compression techniques to reduce the size of XML-RPC transactions – which is found to be a major cause of performance problems for XML-RPC. Finally, § 6 gives our conclusions and outlines future work in this area.

2 Test Environment

To examine the differences between our manually written code based on the java.net package and XML-RPC code we wrote two modest testing programs (a client and a server) that perform a number of operations. The client can use the java.net or XML-RPC code to call on the server to perform the requested operations based on command-line options given by the user. The server program receives requests from the client, performs the requested operation and returns the results. Like the client, the server can use either the java.net-based or XML-RPC code for communication.

The four operations the program performs were chosen to use different kinds of requests and responses. While we only scratch the surface of all possible method calls and responses we believe the following methods offer a useful exploration of the space. The operations we use are:

- `boolean IsPrime (int n)`
  This function determines whether the given integer is prime and returns this determination. This method represents a small request/small response transaction.
- `double Average (Vector numbers)`
  This function returns the average of the given vector of double-precision numbers. The server can handle a vector of arbitrary size. In the tests reported in this paper we use a list of 50,000 randomly chosen numbers. This method represents a big request/small response transaction.
- `Vector GetRandNums (int n)`
  This function returns a vector of n double-precision random numbers. In the tests reported in this paper the client requests 50,000 random numbers. This method represents a small request/big response transaction.
- `Vector LogTransform (Vector numbers)`
  This function takes a vector of double-precision numbers, performs a log transformation on each value and returns a vector containing the transformed values. The server can handle vectors of arbitrary size as input. In our experiments, the client requests the log transformation of 50,000 randomly chosen values. This method represents a big request/big response transaction.

Most of the code in the testing programs is the same regardless of which mechanism is being used to communicate over the network. For instance, the code that actually performs the above actions is implemented in a `PerfActions` class and is shared.

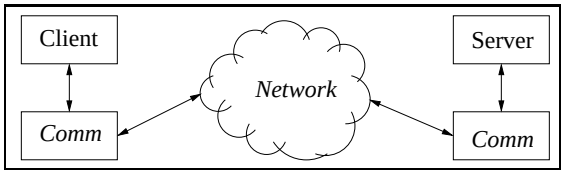

Figure 1: Layout of testing programs.

The general program flow is shown in figure 1. As shown, both the client and server applications communicate through some communications system which, in turn, exchanges messages over some network. The communication system used in our test program depends on command-line input from the user. The three currently implemented subsystems are a hand-rolled application layer protocol based on java.net, an XML-RPC-based system for remote method invocation and a mechanism that simply invokes the methods locally rather than running the method on another host (for debugging purposes). The first two subsystems are explained in the following subsections.

2.1 java.net Code

The first communication subsystem we discuss is based on the java.net package. We use the `Socket` and `ServerSocket` classes to setup a separate connection for each transaction. In principle, we can leave the connection open to serve more than one transaction. However, as an initial study, we did not want to call multiple methods through a single connection (likewise, we did not use XML-RPC's *boxcar* or multi-call feature). We used the Transmission Control Protocol (TCP) [14] to ensure reliable data delivery across the network. In addition, in some of our tests (as outlined in § 4.3) we increase the size of the send and receive socket buffers to 60 KB.

We implement the client networking code in one class and the server code in another. In addition, we use a third class for generic networking routines to implement three methods needed by both the client and server. A longer discussion of the complexity of the implementation is given in § 3.

We implement our own simple application layer protocol to conduct the transactions. The first item sent from the client is a 2 byte short integer identifying the remote method we wish to invoke. Everything sent after this identifier in both the request and the response is specific to the particular method being invoked. For example, the `IsPrime()` request sends a 4 byte integer after the 2 byte method identifier and receives

a 1 byte response from the server (whether the given integer is prime).

2.2 XML-RPC Code

Next we implement a communication subsystem for our testing program based on XML-RPC. TCP is used by XML-RPC for reliable data delivery. In addition, the HyperText Transfer Protocol (HTTP) [3, 8] is used as the application protocol due to its vast deployed base. The data portion of the payload is an XML request to run a particular method with the given parameters or an XML response that encodes the results of the remote method call. As with the implementation based on the java.net package, the XML-RPC code uses one TCP connection per transaction. Future work in this area should include reusing TCP connections and assessing the impact of conducting multiple transactions in short amounts of time.

Unlike our hand-rolled java.net-based implementation we do not have to specify the application layer protocol used by XML-RPC. As discussed in § 4 we captured the packets involved in the XML-RPC transactions to measure performance. A side-effect of this is that we are able to show a sample of an XML-RPC transaction from our tests, which is given below[2]:

```
POST /RPC2/ HTTP/1.1
Content-Length: 164
Content-Type: text/xml
User-Agent: Java1.3.0
Host: mercedes:8081
Accept: text/html, image/gif, [...]
Connection: keep-alive

<?xml version="1.0" encoding=
                     "ISO-8859-1"?>
<methodCall>
 <methodName>act.IsPrime</methodName>
 <params>
  <param>
   <value><int>82</int></value>
  </param>
 </params>
</methodCall>
```

3 Object Oriented Metrics

In this section we compare the complexity of the java.net-based and XML-RPC-based communications code. Note that we only consider the networking code developed by the application designer in this section since the non-communications code is the same regardless of communication subsystem. Also, we do not consider the underlying complexity of the XML-RPC implementation itself, since that is not of concern to the application programmer[3]. This section is divided of the into two parts. The first subsection presents a brief side-by-side comparison of one of the stubs in the communications subsystem to give the reader a feel for the code. The second subsection uses object-oriented metrics to quantify the complexity of the code for each communications subsystem.

3.1 Qualitative Comparison

In this subsection we examine the client-side stubs for both the java.net and XML-RPC based implementations. Figure 2 shows the java.net-based stub for the `LogTransform()` method and figure 3 shows the XML-RPC version of the same stub. The first thing we note is that the java.net code represents more work for the application designer than the XML-RPC code because the programmer has to implement all the details of the transactions. In the java.net code we explicitly open and close the TCP connection (via alternate methods that we also must implement). In addition, as discussed in § 2 we implement our own application protocol, sending the identification number for the remote method we want executed followed by the parameters for the method. While we slightly generalized the parameter passing for lists of double-precision numbers by using the two generic methods `WriteDoubleVector()` and `ReadDoubleVector()` the code cannot be made arbitrarily reusable since each remote method will have its own set of arguments and will return its own set of results. In addition, note that if we wanted to pass a vector of integers we would have to add new methods to the java.net code to accomplish this task (leading to *more complexity*).

On the other hand, the XML-RPC code is short and easy to understand, even without studying XML-RPC. All parameters are inserted into a vector that is then passed to the XML-RPC system with the name of the remote method to be invoked ("act.LogTransform" in the case shown). The results are passed back in a table that provides easy access for the developer. The client and server must agree on a naming scheme for the remote method and the results. However, this is true no matter what type of system is used for communication.

Also note that in both versions of the networking code presented we have shown a *stub* for `LogTransform()` method. However, unlike systems like Sun RPC we are not required to have a stub for the remote methods in XML-RPC. In Sun RPC the program calls a stub rather than the desired method. However, in XML-RPC (and our hand-rolled version) the programmer can use the primitives to directly call a remote method without writing a stub (or having a tool write a stub).

3.2 Quantitative Comparison

We now focus on using coding metrics to assess the two versions of the communications code. The first metric we use to compare our java.net and XML-RPC code is the number of lines of code required to implement the required functionality. The number of lines of code required is a first-order examination of the complexity of the code and of how difficult to maintain that code may be. In this paper, we report the number of lines of code containing actual *program statements*

[2] Note that the XML has been reformatted for presentation in this paper. The line breaks are not present in the code that is transmitted across the network.

[3] However, the complexity of the underlying code will impact the performance of the XML-RPC-based system and therefore of interest to the application programmer. This is discussed in more detail in § 4.

```
public Vector LogTransform (Vector nums)
{
    Vector newnums = null;

    try
    {
        open_conn ();
        out.writeShort (PerfActions.LogTrans);
        out.writeInt (nums.size ());
        NMisc.WriteDoubleVector (out,nums);
        out.flush ();
        newnums = NMisc.ReadDoubleVector (in,nums.size ());
        close_conn ();
    }
    catch (Exception e)
    {
        System.err.println ("LogTransform: " + e.toString());
    }
    return (newnums);
}
```

Figure 2: The java.net-based version of the client stub for the `LogTransform()` method.

```
public Vector LogTransform (Vector nums)
{
    Vector params = new Vector ();
    Vector ln_nums = null;
    Hashtable result;

    params.addElement (nums);
    try
    {
        result = (Hashtable)server.execute ("act.LogTransform", params);
        ln_nums = (Vector)result.get ("logtrans");
    }
    catch (Exception e)
    {
        System.err.println ("LogTransform: " + e.toString());
    }
    return (ln_nums);
}
```

Figure 3: The XML-RPC-based version of the client stub for the `LogTransform()` method.

Program	java.net	XML-RPC
Client	110	93
Server	136	76
Generic	39	0
Total	285	169

Table 1: Lines of code in each communication subsystem (excluding blank lines and lines containing only comments).

Program	java.net	XML-RPC
Client	1, 8, 5	1, 6, 2
Server	1, 7, 6	1, 6, 2
Generic	1, 3, 1	0, 0, 0
Total	3, 17, 12	2, 11, 4

Table 2: Number of classes, methods and data members needed to each communication subsystem.

(i.e., not counting blank lines and lines that contain nothing but comments).

Table 1 shows the lines of code required to implement the two forms of communication. As show in the table, the java.net code is a factor of nearly 1.7 larger in total lines of code when compared to XML-RPC. Most of the savings realized by XML-RPC comes from the server code and the lack of requiring the generic routines used in the java.net version of the code.

Next, table 2 takes a deeper look into the complexity and manageability of the code. This table shows the number of classes, methods and data members required to implement each version of the communication subsystem. As with the above discussion of the number of lines of code, in all metrics presented in table 2 XML-RPC appears to be less complex than the java.net code. At a minimum the client and server class must each have five methods (according to the interface they implement). Of these, four methods act as stubs for the four operations outlined in § 2. Further, the client is required to implement a method that returns a string identifying the type of networking being used (for logging purposes) and the server is required to implement a `Start()` method that is used to begin a transaction (but, is different from the constructor which is used to setup the parameters of the transaction, but not any particular transaction). Given these requirements XML-RPC uses one method more than the minimum in both the client and the server (the constructor in both cases). Meanwhile, the java.net implementation requires several additional methods (e.g., open a TCP connection). Also, as noted above, the java.net code requires 3 generic methods used by both the client and server code and statically defined

Program	java.net	XML-RPC
Client	76	42
Server	77	33
Generic	15	0
Total	168	75
Mean/Class	56.0	37.5
Mean/Method	9.5	7.0

Table 3: Cyclomatic Complexity of both versions of the networking code.

Program	java.net	XML-RPC
Client	13	8
Server	13	8
Generic	6	0
Total	32	16

Table 4: Amount of coupling between the networking code and outside classes.

in a non-instantiated generic class. The XML-RPC code has no such dependency. Finally, note that the number of data members used by the java.net code is three times higher than those kept by the XML-RPC code. This is caused by the need for the java.net code to handle all the networking objects, while XML-RPC abstracts these details from the programmer.

In summary, table 2 shows that the java.net code is more complex code that will be more difficult to implement, test and maintain when compared to the XML-RPC implementation – which nicely abstracts the networking code away from the application developer.

We next we use the Cyclomatic Complexity (CC) [9] to gauge the complexity and amount of testing required for the methods in each implementation of the communication subsystem. Given a method flow graph, the Cyclomatic Complexity is defined as:

$$V(G) = e - n + p + 1 \quad (1)$$

where e is the number of edges in the graph, n is the number of nodes in the graph and p is the number of connected components found in the graph. We calculate CC for each method in our code.

Table 3 shows the CC for the java.net and XML-RPC versions of the code. As shown, the java.net-based code has higher total complexity (by more than a factor of 2). But, this is largely because the java.net implementation has more methods than the XML-RPC version of the code as indicated by the mean CC per method – 9.5 for the java.net code versus 7.0 for the XML-RPC code. So, on a per-method basis the java.net code involves a more complicated flow graph than the XML-RPC code (more conditionals, more loops and more reliance on other methods). Using XML-RPC abstracts the details of the conditionals, loops, etc. from the developer.

Finally, we examine the coupling required between our communication subsystem and outside classes. Table 4 shows the number of outside classes that each of our networking implementations access. As shown, the total number of classes that the java.net-based implementation communicates with is higher than that of the XML-RPC version of the code by a factor of 2. This indicates that the java.net code is more dependent on other portions of the system and therefore may be more difficult to maintain due to changes in the outside classes.

4 Network Performance

4.1 Methodology

We performed two sets of experiments to assess the network performance of the java.net and XML-RPC based communication systems. The first set of experiments involves a local 10 Mbps Ethernet network with only a simple hub between the client and server. The second set of measurements are from transactions over the Internet between NASA's Glenn Research Center (GRC) and Ohio University (OU). The path between the client and server encompassed roughly 15 router hops at the time of our experiments (although, as shown in [13] routes can change arbitrarily and so our measure of the hop-count may not be accurate for the entire test period). The hosts used in the local network tests were Pentium III 400 Mhz FreeBSD 4.4 machines. In the Internet tests, a Pentium III 400 Mhz FreeBSD 4.4 machine at NASA GRC was used as the client, while the server at OU was a dual-processor Sun Enterprise 250 running Solaris 8.

We invoke all four methods using both the java.net and XML-RPC based communication framework at roughly 30 second intervals (the exact interval is determined using a Poisson process with a mean of 30 seconds). The testing program wrapped around the communication systems takes timestamps before and after the transaction to measure the length of the remote method call from the application's vantage point. In addition, we captured all packets transmitted by the clients using *tcpdump*[4]. The packet traces show the length of time each *network transaction* takes (e.g., without accounting for the time needed for pre- or post-processing as the application-level measurement includes). We also use the packet traces to determine the number of data bytes sent by each version of the networking code. *Tcpdump* indicated that no packet filter drops occurred during our experiments.

4.2 Local Network Tests

The distributions of the transaction times for both the java.net-based and XML-RPC-based implementations for the tests over the local network are shown in figure 4. From the figure we observe that the performance does not vary much between runs. This is expected due to relatively static condition of the local network. In addition, we see that the java.net implementation of the networking code always outperforms the XML-RPC code. With the exception of calling the `IsPrime()` method, the java.net transactions takes roughly an order of magnitude less time than the XML-RPC transactions.

For the smallest transaction, `IsPrime()`, the XML-RPC transactions take roughly 70 ms. While this is slightly more

[4] http://www-nrg.ee.lbl.gov or http://www.tcpdump.org

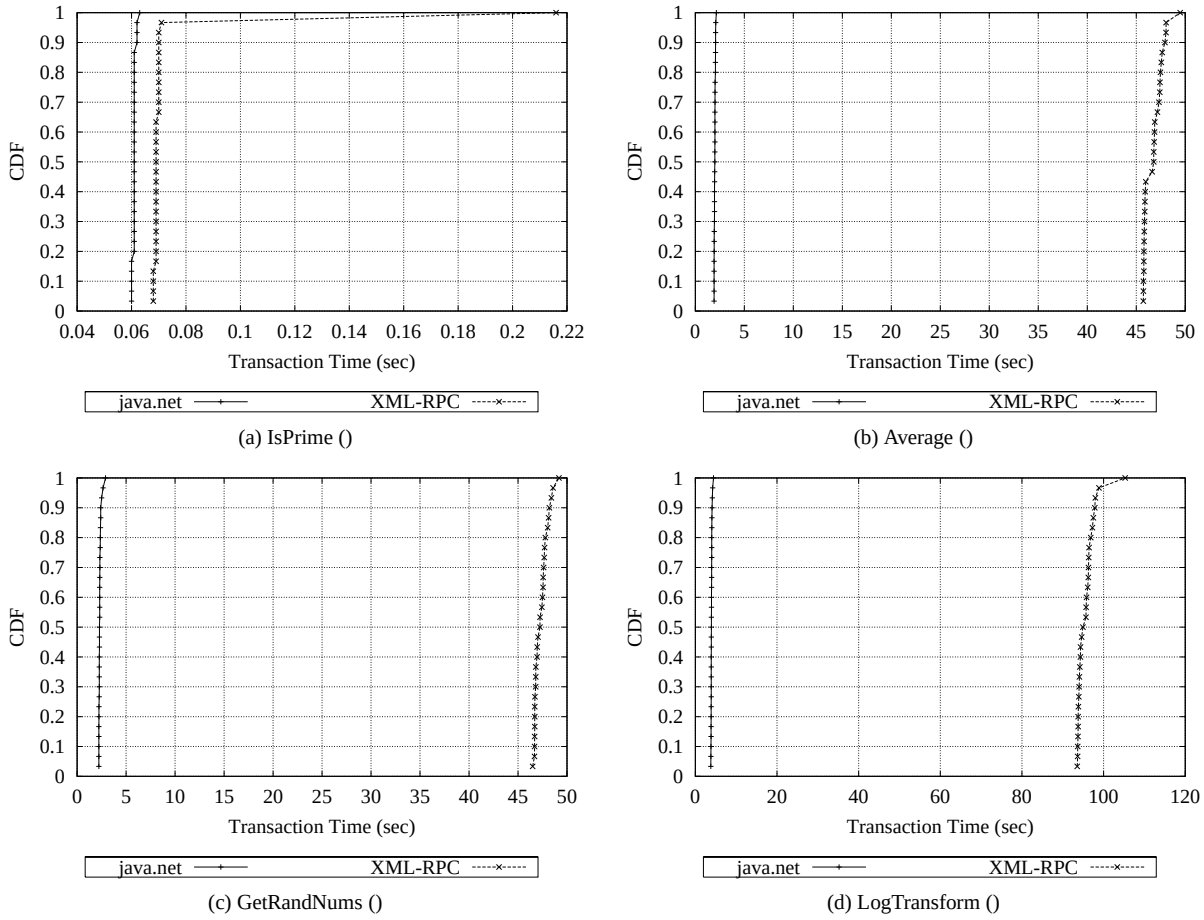

Figure 4: Distribution of the transaction times for the local network experiments.

than than the time required by the java.net version of the code (which takes roughly 60 ms) the difference is likely not significant in systems involving human interaction. For instance, when conducting a single small transaction a user will not perceive the difference between the two versions of the code. However, if many small transactions are performed back-to-back (e.g., by an automated system) the increased amount of time required by the XML-RPC version of the code may add up to an interval that is perceptible and meaningful to the user. Furthermore, the extra time required for the XML-RPC transactions may hinder the performance of automated systems that utilize numerous RPC calls in completing their task.

We now turn our attention to explaining the discrepancy in performance between the java.net and XML-RPC communication systems. Table 5 shows the number of unique bytes transmitted for each transaction[5]. With the exception of the `IsPrime()` method call the results show that the XML-RPC version of the program sends roughly six times more data than the java.net version, which explains some of the overall disparity between the two communication systems.

A large disparity (a factor of roughly 50 to over 300) in transaction size between the two `IsPrime()` versions is shown in table 5. However, the difference between the median transaction times is only about 10 ms because both transactions are small enough to fit in a single packet. So, even though the XML-RPC transaction is larger none of TCP's congestion control algorithms [2] come into play in either case.

Next, we examine the overhead incurred by the two versions of the code that happens before or after the transaction is sent across the network. Table 6 shows the median transaction time for the XML-RPC code measured by the application, as well as the median difference between the transaction time measured by the application process and the length of the TCP connection measured from the packet traces. From this table we make several observations:

- There is little pre- or post-processing overhead involved in the `IsPrime()` method call.
- The `Average()` and `GetRandNums()` methods take roughly the same amount of time to execute when measured by the application. This is expected as roughly the same amount of data is exchanged in both cases – just in

[5]The java.net version transfers numbers as binary data and therefore the transaction is always the same size. However, XML-RPC transmits a textual representation of the data and therefore the size of the transaction can vary from run to run (e.g., one run might send "3.1" while the next sends "541.78234", yielding a difference of six bytes between the two numbers. Therefore, the XML-RPC results presented represent the median size of the all the runs and therefore are denoted as approximate in the table.

Transaction	Request Size (bytes)	Response Size (bytes)
IsPrime – java.net	6	1
IsPrime – XML-RPC	≈ 339	≈ 332
Average – java.net	400,006	8
Average – XML-RPC	≈ 2,513,756	≈ 334
GenRandNums – java.net	6	400,000
GenRandNums – XML-RPC	≈ 346	≈ 2,513,812
LogTransform – java.net	400,006	400,000
LogTransform – XML-RPC	≈ 2,459,521	≈ 2,466,477

Table 5: Transaction sizes in bytes. The numbers designated as approximate represent the median number of bytes transferred. Exact numbers of bytes are given where the number of bytes does not vary across transfers.

Method	Transaction Time (sec)	Time Delta (sec)
`IsPrime ()`	0.069	0.003
`Average ()`	46.787	16.559
`GetRandNums ()`	47.242	0.065
`LogTransform ()`	95.303	16.706

Table 6: Comparison of median time required for XML-RPC `LogTransform()` transaction and the difference between the total elapsed time of the transaction and the time required by the underlying TCP connection.

different directions.

- We observe a difference of over 16.5 seconds between the total transaction time and the network transfer time in the `Average()` and `LogTransform()` calls. In these two calls the client encodes a vector of 50,000 numbers before starting the TCP connection and pushing the data over the network.
- We note little difference (roughly 65 ms) between the total transaction length and the duration of the TCP connection for the `GetRandNums()` method call. As noted above, the total transaction time for the `GetRandNums()` call is similar to that of the `Average()` call. However, the `GetRandNums()` function cannot encode data before the start of the TCP connection because the server creates the vector of data based on input from the client.

From the above table we can sketch a back-of-the-envelope analysis to measure the difference between the java.net and XML-RPC versions in terms of network usage for the `LogTransform()` method. We know that encoding a vector of 50,000 values takes roughly 16.5 seconds from the above table. So, approximately 33 seconds of the `LogTransform()` transaction are spent encoding data to be transmitted. If we assume that parsing the incoming XML and adding the values to the vector takes the same amount of time as encoding the data for transit we need another 33 seconds for decoding for the `LogTransform()` routine. So, of the roughly 95 seconds that the transaction takes (on median) 66 seconds are spent encoding/decoding the data. We also know (from table 5) that the XML-RPC code sends roughly 6.15 times as much data as the java.net code. So, if we simply multiply the transaction time experienced by the java.net by 6.15 we expect an XML-RPC transaction time of just over 24 seconds. Combining the expected transfer time (≈ 24 seconds) with the encoding/decoding time (≈ 66 seconds) we get an expected transaction time of 90 seconds. Even if this analysis is a bit off we believe that the XML-RPC processing and the added bytes that XML-RPC sends across the network explain the majority of the performance difference between the two communication systems.

4.3 Internet Tests

We now describe measurements taken on the Internet path between NASA GRC and Ohio University. We tested both the java.net and XML-RPC communication subsystems over the path as described above (see § 4.1). Preliminary measurements illustrated a performance problem across the Internet caused by FreeBSD's default TCP advertised window size, which in some cases was too small to fully utilize the available bandwidth[6]. Therefore, for our Internet tests we added a variant of the hand-rolled java.net code, denoted "java.net+" that increases the socket buffer sizes (and, therefore, TCP's advertised window).

TCP throughput, T, is ultimately limited based on the size of the advertised window, W, and the round-trip time, RTT, of the network path, as follows [14]:

$$T = \frac{W}{RTT} \tag{2}$$

Throughput can be further limited by TCP's *congestion window* [10, 2] which is based on the measured load on the network. When a TCP connection fills the entire advertised window with data the sender *may* be able to achieve higher performance if the advertised window were increased. In the java.net+ variant in our measurements the advertised window is increased from the default (32 KB) to 60 KB.

Figure 5 shows the distributions of transaction times for the four method calls across the Internet path. When compared to the local network tests, the Internet measurements show more variability, as expected. Again the plot shows that (with the exception of the `IsPrime()` method) the XML-RPC transactions take roughly an order of magnitude longer than the java.net (and java.net+) transactions. Given the analysis of the

[6] Note that [1] shows that this situation is not rare on the Internet.

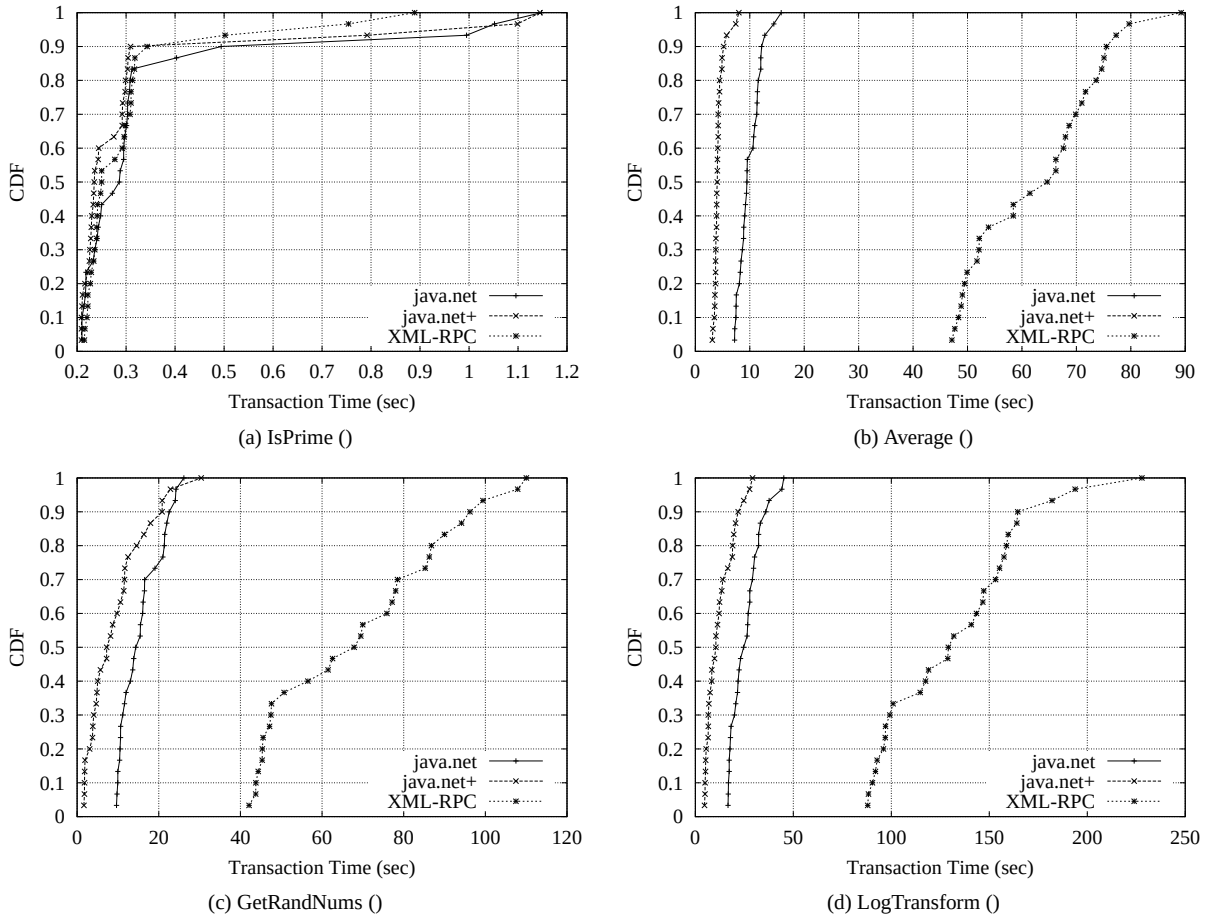

Figure 5: Distribution of the transaction times for the Internet experiments.

transaction sizes and the processing overhead presented in the last subsection these results are not surprising. In addition, we see that using larger TCP window sizes increases the performance over the stock TCP configuration. This shows that a programmer who knows how to tune the sockets[7] can induce better performance, which is a downside of using something like XML-RPC that hides these details from the programmer.

Finally, we note that the performance of the small `IsPrime()` transaction is generally invariant of the underlying communications scheme used. Since this transaction involves only one-segment in each direction for both the java.net and XML-RPC code the performance is dominated by the latency between NASA GRC and Ohio University.

5 Compression

As discussed in the last section, one of the largest performance problems with XML-RPC lies with the number of bytes transmitted into the network. Therefore, we now briefly analyze the usefulness of *abbreviations* and *compression* in mitigating this performance barrier. For our analysis we took one of our XML-RPC `LogTransform()` transactions and extracted the data bytes (excluding protocol headers) from the packet trace. While we are only analyzing a single transaction we note that the request/response size in this transaction is less than 0.04% different from the median transaction sizes reported in table 5 in § 4.2.

Table 7 shows the sizes of the XML-RPC request and response for the chosen transaction, as well as the sizes of the java.net requests and responses for comparison. The first compression technique we use is to simply *gzip* the request/response. This reduces the size of the request/response to less than a quarter of the size of the original XML-RPC transaction. However, the transaction size of the *gzipped* version is still on the order of 25% larger than the java.net transaction. On the Pentium III FreeBSD machines we ran our tests on the *gzip* operation took a little over 1 second, with the de-compression taking just under 0.25 seconds. Therefore, we believe that using *gzip* to compress the transactions would be a net win for large transactions. For comparison we *gzipped* the java.net transaction byte streams. The gain for the java.net-based transaction is not as substantial as for the text-based XML-RPC-based transactions, only yielding a savings of roughly 6% over the uncompressed version.

Our results show benefits to using *gzip* for XML-RPC transactions. However, there are also disadvantages, such as obscuring the payload such that debugging XML-RPC is more

[7] There are additional socket changes that could be made besides adjusting the advertised window size based on the application at hand (e.g., disabling the Nagle algorithm [12]).

Technique	Request Size (bytes)	Response Size (bytes)
XML-RPC	2,459,472	2,466,521
java.net	400,006	400,000
GZipped XML-RPC	527,105	495,890
GZipped java.net	376,967	375,040
Abbreviate	1,559,465	1,566,506
Abbreviate/Combine	1,309,472	1,316,521
New XML Tag	1,159,472	1,166,521
New XML Tag/Gzip	478,021	451,236

Table 7: Impact of compression on the `LogTransform()` transaction size.

difficult and added reliance by XML-RPC on another library. So, we next look at three techniques that can be completely implemented in XML-RPC.

First, as shown in § 2.2 XML-RPC is verbose with respect to the parameters being passed to a method. In the case of an vector of double precision numbers every number is transmitted as a string of the form:

```
<value><double>NN.NN</double></value>
```

One possible extension to XML-RPC is to use *abbreviations*. As shown in table 7, abbreviating "value" with "v" and "double" with "d" results in transmitting over 900,000 fewer bytes (or roughly 36% of the transfer size) in each direction when compared to using the current scheme.

The next row of the table shows the size of the transactions if XML-RPC were to *abbreviate and combine* the "value" and "double" tags to be transmitted like:

```
<vd>NN.NN</vd>
```

When compared to abbreviations alone this reduces the amount of transmitted data by roughly 200,000 bytes. When compared to standard XML-RPC using this technique reduces the transfer size by more than 1.1 MB.

Next we introduce a new (abbreviated) tag with an argument instead of using begin and end tags to get encoding like:

```
<v d=NN.NN>
```

This new tag saves 3 bytes per double-precision number transmitted when compared to the abbreviate and combine technique. Further, the new-style tag represents less than 20% of the overhead imposed by the current XML-RPC technique (6 bytes as opposed to 32 bytes).

As a last step we examine the efficacy of *gzipping* transactions that use the new-style tag introduced above. As the table shows, using the new tag with *gzip* reduces the transaction sizes by roughly 9% over using *gzip* on the standard encoding. However, the resulting transaction is still larger than the java.net encoding.

Since all the abbreviations and tag combinations introduced above are new (i.e., not understood by XML-RPC) they can be added without breaking any of the current functionality. That is, the current tags used by XML-RPC do not need to be replaced, just augmented to support any of the techniques outlined in this section. A method for the client to ask the server if the new technique is supported would have to be added to XML-RPC. One migration path may be to query the server only for large transactions that would benefit from sending significantly smaller transactions. For instance, paying the additional RTT cost to query the server for its supported techniques would not seem to be worth it when executing the `IsPrime()` method, but may well allow a performance improvement for a method like `LogTransform()` when working on a large list of numbers.

Also, note that the size of transactions impacts the end host's RAM usage. As shown in the last section, requests are encoded before being transferred over the network. Therefore, these transactions require a user-space memory "staging area". In addition, since XML-RPC connections last longer than hand-rolled java.net-based transactions kernel memory for TCP's retransmission buffer is also required for a longer amount of time (potentially impacting other network applications).

Finally, we note that TCP/IP header compression [11, 5] can also be employed at various links across a network path to reduce the number of bytes that must be transmitted. However, since TCP/IP headers generally represent a small fraction of the bytes transmitted in the course of a long transfer (3-8%) compressing only the headers leads to only modest decreases in transfer time. Also note that the W3C's Web Services Initiative[8] is also investigating ways to compress "on the wire" XML.

6 Conclusions and Future Work

The following are the major conclusions from our study of remote method invocation using the java.net package and the XML-RPC system:

- The java.net implementation is roughly 50% more complex to code and maintain when compared the the XML-RPC code across a variety of coding metrics.
- The time required for small transactions is roughly the same regardless of underlying communication system across both the local network and the Internet.

[8] http://www.w3c.org/2002/ws/

- The java.net code performs up to an order of magnitude better than the XML-RPC code when transfering a significant amount of data. The performance problems of XML-RPC are largely caused by (i) the increased size of XML-RPC transactions (an increase of over a factor of 6 when compared to the java.net code), and (ii) the overhead of encoding and decoding XML.
- We show several compression and abbreviation schemes to reduce the size of XML-RPC transactions. Even with the compression/abbreviation strategies the hand-rolled java.net transactions are always smaller than XML-RPC transactions. However, using some form of compression within XML-RPC will likely have a positive impact on performance.
- Tweaking socket options can yield positive performance gains. We show performance improvement by increasing the socket buffer sizes in our Internet tests (but, note that tweaking the advertised window size had no effect on the tests across the local network).

In addition, we have identified a number areas for future work in this area:

- The encoding/decoding of transactions is a barrier to good performance in the Java version of XML-RPC. Future work should quantify this further, keeping the following questions in mind: Is this purely a Java Virtual Machine issue? Does the encoding/decoding perform differently in other languages? Are there more efficient techniques to encode/decode that will aid XML-RPC transaction times?
- The XML-RPC system could be extended to give the programmer optional access to the underlying sockets. In doing this, XML-RPC would allow savvy programmers to tweak socket options to obtain better performance for their particular application.

Finally, this paper only scratches the surface of evaluating RPC mechanisms in general (and XML-RPC in particular). Analyzing additional RPC systems from a number of different vantage points (network performance, coding complexity, encoding/decoding algorithm complexity) is useful and should be undertaken more often as we attempt to write and refine complex systems.

Acknowledgments

Thanks to Ethan Blanton, Shawn Ostermann and Vern Paxson for arranging the remote hosts used to test against. Thanks to Doug Dechow, John Lambert, Rich Slywczak, Lee White and the anonymous reviewers for comments on an earlier version of this paper. I engaged in spirited discussions with Joseph Ishac, Eric Megla and Vern Paxson that improved this work. Finally, Doug Dechow helped a great deal with the analysis presented in § 3. My thanks to all!

References

[1] M. Allman. A Web Server's View of the Transport Layer. *Computer Communications Review*, 30(5):10–20, Oct. 2000.

[2] M. Allman, V. Paxson, and W. R. Stevens. TCP Congestion Control, Apr. 1999. RFC 2581.

[3] T. Berners-Lee, R. Fielding, and H. Nielsen. Hypertext Transfer Protocol – HTTP/1.0, May 1996. RFC 1945.

[4] A. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM TOCS*, 2(1):39–59, Feb. 1984.

[5] C. Bormann, C. Burmeister, M. Degermark, H. Fukushima, H. Hannu, L.-E. Jonsson, R. Hakenberg, T. Koren, K. Le, Z. Liu, A. Martensson, A. Miyazaki, K. Svanbro, T. Wiebke, T. Yoshimura, and H. Zheng. RObust Header Compression (ROHC): Framework and Four Profiles: RTP, UDP, ESP, and Uncompressed, July 2001. RFC 3095.

[6] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielson, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. Technical report, World Wide Web Consortium, May 2000.

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Technical report, World Wide Web Consortium, Oct. 200.

[8] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, Jan. 1997. RFC 2068.

[9] B. Henderson-Sellars. Modularization and McCabe's Cyclomatic Complexity. *Communications of the ACM*, 37(12):17–19, Dec. 1992.

[10] V. Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM*, 1988.

[11] V. Jacobson. Compressing TCP/IP Headers For Low-Speed Serial Links, Feb. 1990. RFC 1144.

[12] J. Nagle. Congestion Control in IP/TCP Internetworks, Jan. 1984. RFC 896.

[13] V. Paxson. End-to-End Routing Behavior in the Internet. In *ACM SIGCOMM*, Aug. 1996.

[14] J. Postel. Transmission Control Protocol, Sept. 1981. RFC 793.

[15] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications*, 14(2), Feb. 1997.