# A Comparison of Two Alternative Implementations of Message-Passing in the Systems Biology Workbench

Michael Hucka, Andrew Finney, Herbert Sauro, Hamid Bolouri
{mhucka,afinney,hsauro,hbolouri}@cds.caltech.edu
Systems Biology Workbench Development Group
ERATO Kitano Systems Biology Project
Control and Dynamical Systems, MC 107-81
California Institute of Technology, Pasadena, CA 91125, USA
http://www.cds.caltech.edu/erato

Principal Investigators: John Doyle and Hiroaki Kitano

12 April 2001
Revised: 4 June 2001

## 1 Background and Introduction

The Systems Biology Workbench (SBW) is a broker-based architecture that provides infrastructure enabling software components to communicate with each other. A component in SBW (called an SBW *module*) can take many forms: it may be primarily computational and lack a GUI, or it may be a computational module having its own GUI, or it may consist solely of a GUI designed to control other tools. A critical requirement for SBW is supporting modules written in any major programming language.

For a variety of reasons that are outside the scope of the present report, the SBW project had at the time of this writing decided to avoid more complex technologies such as CORBA (OMG, 2001; Seetharaman, 1998; Vinoski, 1997) and ILU (Janssen et al., 1999), and settled instead on using a message-passing approach for communications between modules. We still needed to resolve two basic issues: the *message format* (how a structured piece of information is encoded prior to transmission; i.e., the content encoding) and the *transport protocol* (how a message is exchanged between agents, including issues of handshaking and network port numbers).

Our main criteria for choosing a suitable message-passing scheme were: performance, support for data types needed for SBW, simplicity, and portability. We examined a number of existing frameworks including XML-RPC (McLaughlin, 2000; UserLand Inc., 2001; Winer, 2001), SOAP (Box et al., 2000), MPI (Gropp et al., 1999), Java RMI (Steflik and Sridharan, 2000; Sun Microsystems, 2001), and a homegrown RPC scheme developed by Andrew Finney and Herbert Sauro in the first phase of the SBW project. We narrowed the possibilities to two candidates: our original "homegrown-RPC" implementation, and XML-RPC. In this report, we summarize the results of our comparison between these two alternatives.

# 2 Descriptions of the Two Alternative Schemes

It is first useful to clarify how a "message-passing system" differs from simply using sockets. Frankly, the distinctions are muddy. Nearly every network standard currently in widespread use already *is* a form of message passing: even a low-level protocol such as TCP/IP involves bundling data into messages that are sent from one agent to another using network sockets. The key difference is that this level of the protocol is hidden from a developer by an API that involves sockets and (perhaps) stream interfaces. So what we are doing is layering a higher-level message-passing scheme on top of the more basic socket functionality, with the details of the messages tailored specifically to the needs of the application we are developing.

## 2.1 Homegrown-RPC

The message-passing system implemented in the first phase of SBW development is a relatively simple scheme that encodes data values as bytes and sends them in a stream across a socket connection. The message encoding has a custom format, in which each data element in a message is prefixed by a byte that describes its data type. This "homegrown-RPC" implementation supports the following common and useful data types:

- Byte: An 8-bit quantity, equivalent to a Java `byte`.
- Boolean:: A true or false value.
- Integer: A 32-bit signed number, equivalent to a Java `int`.
- Double: A double precision floating-point number in IEEE 754 standard format.
- String: A sequence of characters, equivalent to `char *` in C.
- Array: A multidimensional homogeneous collection of data of arbitrary size.
- List: A sequence of heterogeneous data elements; the elements can be of any other types, including for example other lists.

The operations in the basic API center around blocking and non-blocking remote procedure calls. The blocking version (`call`) invokes a specific method in a specific module, handing it arguments serialized into a message data stream. The call waits until the method on the remote module returns a value. The non-blocking version (`send`) is similar, except that it does not wait for a return value. The homegrown-RPC scheme also provides a means for returning exception codes to a caller.

We were reluctant to invent a custom message encoding format, given that a number of alternatives already existed. We examined a number of well-known candidates, including Sun RPC XDR (Sun Microsystems, 1987), DCE NDR (Open Software Foundation, 1993), and the Java Object Serialization Specification (Sun Microsystems, 1998). However, from the standpoint of our requirements, each of these alternatives suffered from deficiencies. For example, both XDR and NDR are untagged formats, which means that two parties communicating data must know ahead of time the exact structure being sent, and Java serialization has substantial (and for our purposes, unnecessary) baggage for dealing with object classes.

## 2.2 XML-RPC

XML-RPC (`http://www.xmlrpc.com`) is a lightweight remote procedure calling protocol that uses HTTP as the transport and XML as the message encoding. The following is an example of an XML-RPC message body (minus the HTTP headers); this example calls a function named `sumAndDifference` on the object `sample`, giving it two integer arguments as parameters:

```
<methodCall>
  <methodName>sample.sumAndDifference</methodName>
  <params>
    <param><value><int>5</int></value></param>
```

```
    <param><value><int>3</int></value></param>
  </params>
</methodCall>
```

XML-RPC supports a variety of data types, including heterogeneous lists and homogeneous arrays; other types can be transmitted by encoding them in base64 format. The types supported are:

- Boolean:: A true or false value.
- Integer: A four-byte signed integer, equivalent to a Java `int`.
- String: A sequence of characters.
- Double: A double-precision signed floating point number.
- Date/Time: A date/time specification in ISO 8601 format.
- Base64: Base64-encoded binary data.
- Struct: A heterogeneous sequence of members, each of which contains named item-value pairs.
- Array: A sequence of (possibly heterogeneous) data items.

An XML-RPC transaction is synchronous and involves returning a result code; this code may be a fault/exception.

# 3 Experimental Methods

We tested Java implementations of both the XML-RPC and homegrown-RPC systems. The performance tests consisted of timing how long it took each messaging implementation to exchange a certain number of messages. We tested five different scenarios:

- Empty message.
- A short array of 10 double's.
- A long array of 1000 double's.
- A short string of 100 characters.
- A long string of 10 000 characters.

The tests of the XML-RPC and homegrown-RPC versions both used the same testbed driver; it is listed in Appendix B for reference. The communications tests involved only the local machine; the message exchanges were not inter-machine, which means that network latencies can be assumed to be zero, and the results can be assumed to measure the performance of message handling only.

We attempted to eliminate some confounding variables:

- We performed timing tests under both Linux (Red Hat Linux 7.0, kernel 2.4.0) and Windows (Windows 2000) systems, using identical hardware in both cases (733 Mhz Pentium-III based computer with 384 MB RAM). The test results were comparable and did not reveal a platform-specific advantage of using one OS over the other. Here we present only the results from the Linux-based tests.
- We tested different versions of the JDK, specifically Sun's version 1.3 JDK (Java HotSpot Client VM, build 1.3.0rc1-b17, "mixed mode") and IBM's version 1.3 JDK (build 1.3.0, J2RE 1.3.0 IBM build cx130-20010207, JIT enabled), both under the Linux system. We did not find an advantage to using one or the other JVM for either of the message-passing schemes. (The absolute performances differed when using different JVMs, but the relative performance difference between XML-RPC and homegrown-RPC stayed the same.) The results presented here were generated using the Sun implementation.

For the the XML-RPC portion of these tests, we started with an off-the-shelf solution available on the Internet (version 1.0beta4 of the Helma XML-RPC implementation, `http://helma.at/hannes/xmlrpc`). We substituted a different XML parser (MinML; Wilson, 2001), which we found to have higher performance on the test suite used here. (Appendix A lists the XML parsers that we examined for this report.) Initial tests showed that long arrays of `double`'s took inordinately long to transfer, so we performed one optimization: we changed the XML-RPC implementation to encode arrays of `double`'s as byte arrays. (Strictly speaking, this is not in the spirit of XML-RPC, but it is also not directly forbidden.) The test results here are based on this modified implementation.

# 4 Results

The timing results are presented in Table 1. The table shows that in terms of absolute performance, XML-RPC message exchanges for short messages took 1.3–1.6 ms mean time, whereas the homegrown-RPC solution took 0.19–0.28 ms mean time. This represents a factor of 5 to 8 in difference. This difference is similar for the case of empty messages; there again, the XML-RPC approach took approximately ten times as long to exchange messages as the homegrown-RPC method. For long messages, exchange times were similar for both methods.

| | **Mean Run Times (ms)** | |
|---|---|---|
| **Test** | **XML-RPC** | **Homegrown-RPC** |
| short array of `double`'s | 1.6 | 0.19 |
| short character string | 1.3 | 0.28 |
| long array of `double`'s | 11.3 | 8.6 |
| long character string | 8.9 | 8.2 |
| empty message | 1.2 | 0.10 |

**Table 1:** *Mean run times for each case of the timing experiments on a machine running Red Hat Linux 7.0, Linux kernel 2.4.0, 733 Mhz Pentium-III, 384 MB RAM. Tests were performed using Sun's JDK version 1.3 for Linux (Java HotSpot Client VM, build 1.3.0rc1-b17, "mixed mode"). Tests involved only the local machine (i.e., the loopback network interface) and therefore represent a best-case scenario free of network latencies.*

# 5 Analysis and Discussion

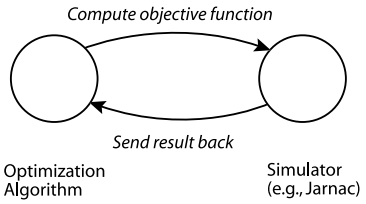
The results in the previous section imply that message exchanges using the XML-RPC approach take 5 to 10 times as long as the homegrown-RPC scheme. On the face of it, these results favor the homegrown-RPC scheme. However, to be fair, a number of additional factors should be considered:

1. How do the tested run times compare to the times expected for completing computations in SBW modules?
2. Did both systems (XML-RPC, homegrown-RPC) perform exactly the same operations?
3. Do the test results represent best-case performances, or can the implementations of each scheme be improved?
4. Aside from performance itself, what additional issues should be considered in choosing between the two alternatives?

## 5.1 How Do the Tested Run Times Compare to Expected Module Run Times?

Absolute message-exchange times are relatively meaningless unless placed in the context of an application's run times. In SBW, one of the situations that will demand high performance is optimization, in which

an objective function involves repeatedly performing a simulation (e.g., using Jarnac; Sauro and Fell 1991; Sauro 2000) under the control of an optimizer (e.g., Gepasi; Mendes 1997, 2001). The scenario is something like the following:




We attempted to estimate how long a typical simulation might take by examining a few example Jarnac simulations and then making some rough estimates for other cases. The results are shown in Table 2.

| Case | Execution times |
|---|---|
| *Run to steady-state* | |
| Small model | 1.5–5 ms |
| Large model (estimated) | 15–50 ms |
| *Time-course simulation* | |
| Small model | 10–30 ms |
| Large model (estimated) | 40–70 ms |

**Table 2:** *Approximate run times for simulations using Jarnac, based on a small sample problem involving a two parameter model with three species and non-trivial kinetics. The estimates for a "big model" are essentially educated guesses made by Jarnac's author.*

The estimates in Table 2 show that the run time for reaching steady-state in a model (the most likely situation used in an optimization problem) is 1.5 ms *at minimum*. The actual run times for real cases are almost guaranteed to be longer. Let us assume that actual ranges will be closer to the mean value of the extremes: so, instead of 1.5–5 ms for a small model and 15–50 ms for a large model, we take $(1.5 + 50)/2 \approx 30$ ms (rounded down) as the average time to compute a result for a realistic model in a system such as Jarnac.

This has implications for the message communications method. Let us suppose that message round-trip times are on the order of 1 ms, as in the XML-RPC tests. This implies that the total time spent on each computational cycle will be $30 + 1 = 31$ ms. A message time of 1 ms is equal to 1/31 of this total, or 3.2%. On the other hand, the homegrown-RPC case takes approximately 0.1 ms, which implies that the total time spent on each computational cycle will be $30 + 0.1 = 30.1$ ms. A message time of 0.1 ms, then, is equal to 0.1/30.1 of this total, or 0.33%.

Note, however, that although the difference in percentages of time spent in communications appears large (3.2% versus 0.33%), the difference in absolute times (31 ms versus 30.1 ms) is small: it is 2.9%.

## 5.2 Did Both Messaging Frameworks Perform the Same Operations?

Post-experimental analysis revealed that the homegrown-RPC test server code did not perform as many operations as the XML-RPC code. Here is an explanation of why.

The XML-RPC code used in these experiments was a full implementation that included code for invoking handlers for different message types. On the server side, each RPC message resulted in the server code parsing out the name of the method being invoked, looking up the handler for that method in a hash table, then invoking a specific method on the handler object. The handler object takes care of reading the arguments (either a string array or an array of `double`'s) and returning a result.

Conversely, the code in the homegrown-RPC test server specifically looked for one of two cases (either a string or an array), read that particular data type directly, and returned the result immediately. The following is the relevant code:

```
while (true)
{
    DataBlockItr itr = new DataBlockItr(in);
    itr.getInt();
    itr.getInt();
    if (itr.getNextType() == DataBlockBuilder.STRING_TYPE)
        itr.getString();
    else if (itr.getNextType() == DataBlockBuilder.ARRAY_TYPE)
        itr.get1DDoubleArray();
    DataBlockBuilder builder = new DataBlockBuilder(100);
    builder.add((int)100);
    out.write(builder.getData());
    out.flush();
}
```

This implementation does not reflect the complete set of operations that a production server would have to perform: a real-world server would need to provide hooks for invoking different handler objects based on different RPC method invocations. The XML-RPC test server therefore represents more of a real-world implementation than the homegrown-RPC test server.

Compared to the homegrown-RPC version, the extra operations performed by the XML-RPC implementation undoubtedly added some overhead and slightly inflated the results in Table 1. The fixed overhead is likely to be small, but further testing would be necessary to determine exactly how much. Nevertheless, when reading the results in Table 1, it should be kept in mind that the two cases are not performing identical operations.

## 5.3 Do the Tests Represent Best-Case Performances?

Can the implementations of the message-passing frameworks discussed here be improved? At the time of this writing, we investigated this question only for the XML-RPC implementation; thus, our discussion here is limited to that case, but we feel confident that improvements could also be made to the homegrown-RPC implementation.

As mentioned above, the test results already reflect one optimization on the XML-RPC implementation, namely to package arrays of `double`'s as byte streams and thereby reduce the time required to parse the arrays by the XML parser. We also experimented with different XML parsers and chose the one that provided the fastest performance (see Appendix A).

The results in Table 1 show that empty messages take almost as long to exchange as messages containing actual data. This implies that most of the differences in run times between the XML-RPC and homegrown-RPC cases are due to fixed overhead costs in XML-RPC. Visual inspection of the XML-RPC implementation leads us to believe that a significant number of additional operations could in fact be streamlined. Brief profiling showed that it performs a substantial number of string operations; an obvious area to improve is reducing the use of the Java `String` data type. The XML-RPC implementation also uses threading; another possible area to investigate is to handle threading differently.

It is difficult to estimate the impact that such optimizations may have. Let us make some rough estimates and assume that further optimizations on the XML-RPC framework would improve performance by 10%–25%. Table 3 shows the run times that could be expected in that case.

## 5.4 What Other Factors Should Be Considered?

Some additional issues are worth mentioning in the context of deciding on an approach for SBW.

### 5.4.1 Standardization

XML-RPC has an established following among a number of software developers world-wide; it is even described and used in several chapters of a book on Java and XML (McLaughlin, 2000). Even Microsoft is using a more elaborate version of XML-RPC, SOAP, as the cornerstone of its .NET strategy. Although SBW will hide the message format and protocol behind libraries used by SBW developers, widespread support and

| Test | Mean Run Times (ms) 10% | 25% |
|---|---|---|
| short array of `double`'s | 1.4 | 1.2 |
| short character string | 1.2 | 1.0 |
| long array of `double`'s | 10.2 | 8.5 |
| long character string | 8.0 | 6.7 |
| empty message | 1.1 | 0.9 |

**Table 3:** *Run times expected for the XML-RPC version, under the assumption that the implementation can be optimized enough to produce 10%–25% improvements over the results shown in Table 1.*

the availability of books describing this message-passing approach lends it a certain cachet that a unique homegrown-RPC approach lacks.

#### 5.4.2 Support for Required Data Types

The data types made available in the homegrown-RPC implementation were chosen (1) to support the types of data that we expected would be needed by systems biology simulators, and (2) to provide a reasonable balance between flexibility and conciseness. We knew that data types such as IEEE floating point would be important for simulation software. It is therefore important to note that XML-RPC does not specify the use of IEEE floating point numbers: XML-RPC's floating point type is `double`, whatever that may be in a given programming language, and when written out in XML, the double is expressed as a sequence of ASCII characters (e.g., `"234.255"`). As a consequence, there is no formalized way of expressing overflow and NaN. Thus, using XML-RPC as the basis of the SBW communications framework would require that we add mechanisms to support IEEE floating-point quantities (perhaps by manually encoding numbers using byte arrays). This would likely have no impact on performance, but it would call into question the logic of using XML-RPC when it does not support the basic data types needed by our application.

#### 5.4.3 Availability of Implementations

XML-RPC implementations are currently available for a wide variety of languages, including Python, Delphi, Java, C, C++, AppleScript, Guile, Tcl, and Perl. This suggests it would be easier to implement support for new languages in SBW if we used XML-RPC, because a starting point for the message-passing layer would likely be available. (But see the next subsection.)

#### 5.4.4 Quality of Existing Implementations

Although many XML-RPC implementations are available, our informal survey of the available C/C++ and Delphi implementations shows that they were either immature or designed in an awkward-to-use manner. We believe that we would have to implement our own XML-RPC libraries for at least C/C++, and possibly other languages. This negates one of the primary motivations for considering XML-RPC, namely that we would be spared the effort of implementing a communications framework by using off-the-shelf software.

#### 5.4.5 Firewall Transparency

Because XML-RPC uses HTTP as its transport protocol, it is able to cross most network firewall without special requirements because most firewalls already allow HTTP traffic to pass through. Crossing a firewall using the homegrown-RPC method would require the firewall to allow network traffic on a nonstandard port. Most site administrators, especially in corporate network environments, are reluctant to introduces special-case holes in their network security systems.

Unfortunately, it turns out that for SBW, using XML-RPC would not make it easier to cross network firewalls. The problem is described in the following subsection. Thus, for our application, neither the homegrown-RPC nor the XML-RPC approaches provide an immediate solution.

### 5.4.6 Bidirectional Communications

The broker architecture of SBW requires bidirectional communications between modules. One module may call on another (via the broker as intermediary), and two modules may exchange information with each other. In all cases, any module can serve as the initiator. An important question, then, is whether a given communications framework cleanly supports this kind of full-duplex connectivity.

One of our realizations during the testing process was that XML-RPC does not support bidirectional connections. The root of the problem is the HTTP protocol, which is oriented towards client-server applications in which an agent initiates a connection to a server listening on a designated TCP/IP port. The implication of using XML-RPC for SBW is that if modules A and B needed to invoke operations on each other simultaneously, module A would have to initiate a connection to B *and* B would have to initiate a *separate* connection on module A. Although this would not be impossible, it has two important implications: (1) every module would need to be assigned a unique TCP/IP port on a given machine, so that it could listen for requests directed at it; and (2) the firewall transparency implied by using HTTP would be lost, because modules would have to use ports other than the standard HTTP port and communications could not all take place through a single connection through a firewall as in normal HTTP.

It appears the only work-around to this would be to violate the HTTP protocol and somehow tunnel a bidirectional data stream over a single HTTP connection. However, such an implementation would not adhere to the XML-RPC specification and would not be compatible with existing XML-RPC implementations.

# 6 Conclusions

The tests reported here show the homegrown-RPC system to be more efficient than the XML-RPC system tested. It is *possible* that the performance of the XML-RPC implementation can be improved; however, it will never be as fast as the homegrown-RPC scheme because XML-RPC involves a more verbose data encoding and a more costly parsing/writing approach. On the other hand, for the average run times expected for SBW modules, the time required to exchange a message using either scheme is relatively small, and the relative difference between the schemes is even smaller: roughly 3%.

But there are other considerations. Although XML-RPC has certain advantages as a more established approach that is documented and recognized on the Internet, XML-RPC as a standard lacks support for certain data types that are important for the Systems Biology Workbench. Further, although existing implementations are available for a variety of programming languages, the quality of some of the implementations has not impressed us. If we cannot count on being able to use existing implementations, we are not saved any work compared to implementing our own messaging/RPC scheme. From the standpoint of software developers who wish to use SBW, the existence of XML-RPC implementations may be a moot point: most developers will use our SBW libraries and these will hide the underlying message-passing system behind an API. Few developers will care about the details of the messaging scheme.

A more significant concern is that XML-RPC is not oriented towards bidirectional message exchanges (as discussed in Section 5.4.6). At best, this means that the theoretical firewall transparency of using XML-RPC would be lost in our application; but in fact, this particular issue raises the more fundamental question of whether XML-RPC has an architecture suitable for the needs of SBW.

Given that SOAP (Box et al., 2000) and XML-RPC are fundamentally quite similar, we believe most of the objections raised above to using XML-RPC would apply to SOAP as well.

For these reasons, we have decided to continue using our homegrown-RPC scheme as the basis for the message-passing infrastructure in SBW, rather than XML-RPC or its cousin SOAP. If it proves necessary, there is always the option of implementing multiple messaging protocols inside the SBW broker and libraries, and translating between them internally.

# Appendix

## A XML Parser Implementations Tested

A number of free XML parsers written in Java were available at the time of this writing. We tested the following alternatives:

- Ælfred version 1.1 (Microstar Software Ltd., 1998).
- Crimson version 1.1 (Apache Software Foundation, 2001a).
- MinML version 1.0 (Wilson, 2001).
- OpenXML version 1.2 (OpenXML.org, 2001).
- XP version 0.5 (Clark, 1998).
- Xerces version 1.3.1 (Apache Software Foundation, 2001b).

Informal experiments using the test suite described in this document revealed that MinML gave the fastest performance (i.e., the shortest message transmission times). We therefore used MinML in the tests discussed in Sections 3–5.

## B The Testbed Code

Shown below is the Java code for the test driver used in the timing tests presented in this report.

```
/*
** classname   : TestBed.java
** Description : Test client
** Author(s)   : Andrew Finney, Michael Hucka
** Organization: Caltech ERATO Kitano, California Institute of Technology
** Created     : 2001-03-27
** Revision    : $Id: TestBed.java,v 1.1 2001/03/29 19:42:33 mhucka Exp $
** $Source: /cvs/sysbio/src/tests/xml-rpc/TestBed.java,v $
**
** Copyright (C) 2001 California Institute of Technology, and Japan
** Science and Technology Corporation.
*/

/**
 * @author Andrew Finney, Michael Hucka
 * @author $Author: mhucka $
 * @version $version$
 **/
public class TestBed
{
    private static double[] getFilledArray(int x)
    {
        double[] result = new double[x];
        int i = 0 ;

        while (i != x)
        {
            double v = i ;

            result[i] = v + v / 100 ;

            i++ ;
        }

        return result ;
    }

    private static String getFilledString(int x)
```

```
    {
        String alphabet = "abcdefghijklmnopqrstuvwxyz";
        StringBuffer buffer = new StringBuffer(x);
        int i = 0 ;

        while (i < x)
        {
            buffer.insert(i, alphabet);
            i += alphabet.length() ;
        }

        buffer.setLength(x);

        return buffer.toString();
    }


    public static void main(String[] args)
    {
        SimpleSend client = new SimpleSendClient();
        int i = 0 ;

// Test sending empty messages.

System.out.println("Sending nothing.");
long emptyCount = 10000;
        long time = System.currentTimeMillis();
        while (i != emptyCount)
        {
            client.send();
            i++ ;
        }
        long emptyTime = System.currentTimeMillis() - time ;
        System.out.println(emptyCount + " sends of nothing took: "
   + emptyTime);

// Test sending short array of doubles.

System.out.println("Sending 1000 double[10]'s.");
long shortArrayCount = 1000;
        double[] array = getFilledArray(10);
        i = 0 ;
        time = System.currentTimeMillis();
        while (i != shortArrayCount)
        {
            client.send(array);
            i++ ;
        }
        long shortArrayTime = System.currentTimeMillis() - time ;
        System.out.println(shortArrayCount + " sends of double[10] took: "
   + shortArrayTime);

// Test sending large array of doubles.

System.out.println("Sending 100 double[1000]'s.");
long longArrayCount = 100;
        array = getFilledArray(1000);
        i = 0 ;
        time = System.currentTimeMillis();
        while (i != longArrayCount)
        {
            client.send(array);
            i++ ;
        }
        long longArrayTime = System.currentTimeMillis() - time ;
        System.out.println(longArrayCount + " sends of double[1000] took: "
   + longArrayTime);

// Test sending short strings.
```

```
System.out.println("Sending 1000 strings of 100 chars.");
long shortStringCount = 1000;
        String string = getFilledString(100);
        i = 0 ;
        time = System.currentTimeMillis();
        while (i != shortStringCount)
        {
            client.send(string);
            i++ ;
        }
        long shortStringTime = System.currentTimeMillis() - time ;
        System.out.println(shortStringCount + " sends of 100 char string took: "
   + shortStringTime);

// Test sending long strings.

System.out.println("Sending 100 strings of 10000 chars.");
long longStringCount = 100;
        string = getFilledString(10000);
        i = 0 ;
        time = System.currentTimeMillis();
        while (i != longStringCount)
        {
            client.send(string);
            i++ ;
        }
        long longStringTime = System.currentTimeMillis() - time ;
        System.out.println(longStringCount + " sends of 10000 char string took: "
   + longStringTime);

// Print stats.

        System.out.println("Times: ");
System.out.println("  Total elapsed time = "
   + (emptyTime + longStringTime +
      shortStringTime + longArrayTime + shortArrayTime));
        System.out.println("  Mean time for empty        = "
   + (double) emptyTime/emptyCount);
System.out.println("  Mean time for short array  = "
   + (double) shortArrayTime/shortArrayCount);
        System.out.println("  Mean time for long array   = "
   + (double) longArrayTime/longArrayCount);
        System.out.println("  Mean time for short string = "
   + (double) shortStringTime/shortStringCount);
        System.out.println("  Mean time for long string  = "
   + (double) longStringTime/longStringCount);
    }
}
```

## References


Apache Software Foundation (2001a). Crimson version 1.1. Available via the World Wide Web at `http://xml.apache.org/crimson/index.html`.

Apache Software Foundation (2001b). Xerces version 1.3.1. Available via the World Wide Web at `http://xml.apache.org/xerces-j/index.html`.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. (2000). Simple Object Access Protocol (SOAP) 1.1: W3C note 08 May 2000. Available via the World Wide Web at `http://www.w3.org/TR/SOAP/`.

Clark, J. (1998). XP version 0.5. Available via the World Wide Web at `ftp://ftp.jclark.com/pub/xml/`.

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, MA, USA.

Janssen, B., Spreitzer, M., Larner, D., and Jacobi, C. (1999). ILU 2.0beta1 reference manual. Available via the World Wide Web at `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

McLaughlin, B. (2000). *Java and XML*. O'Reilly & Associates.

Mendes, P. (1997). Biochemistry by numbers: Simulation of biochemical pathways with Gepasi 3. *Trends in Biochemical Sciences*, 22:361–363.

Mendes, P. (2001). Gepasi 3.21. Available via the World Wide Web at `http://www.gepasi.org`.

Microstar Software Ltd. (1998). Ælfred version 1.1: Microstar's Java-based XML parser. Available via the World Wide Web at `http://www.microstar.com/XML/`.

OMG (2001). *CORBA*. Specification documents available via the World Wide Web at `http://www.omg.org`.

Open Software Foundation (1993). *OSF DCE Application Development Guide*. Prentice-Hall.

OpenXML.org (2001). OpenXML version 1.2. Available via the World Wide Web at `http://www.openxml.org`.

Sauro, H. M. (2000). Jarnac: A system for interactive metabolic analysis. In Hofmeyr, J.-H. S., Rohwer, J. M., and Snoep, J. L., editors, *Animating the Cellular Map: Proceedings of the 9th International Meeting on BioThermoKinetics*. Stellenbosch University Press. ISBN 0-7972-0776-7.

Sauro, H. M. and Fell, D. A. (1991). SCAMP: A metabolic simulator and control analysis program. *Mathl. Comput. Modelling*, 15:15–28.

Seetharaman, K. (1998). The CORBA connection. *Communications of the ACM*, 41(10):34–36.

Steflik, D. and Sridharan, P. (2000). *Advanced Java Networking*. Prentice-Hall.

Sun Microsystems (1987). XDR: External data representation standard (RFC 1014). Internet Request for Comments 1014, Sun Microsystems, Inc. Available via the World Wide Web at `http://www.faqs.org/rfcs/rfc1014.html`.

Sun Microsystems (1998). Java™ object serialization specification. Technical report. Available via the World Wide Web at `http://java.sun.com/products/jdk/1.2/docs/guide/serialization/`.

Sun Microsystems (2001). Java Development Kit 1.3. Available via the World Wide Web at `http://java.sun.com/j2se/1.3/`.

UserLand Inc. (2001). XML-RPC home page. Available via the World Wide Web at `http://www.xml-rpc.com/`.

Vinoski, S. (1997). CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communication*.

Wilson, J. (2001). MinML version 1.0. Available via the World Wide Web at `http://www.wilson.co.uk/`.

Winer, D. (2001). XML-RPC specification. Available via the World Wide Web at `http://www.xmlrpc.com/spec/`.