INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600
DESIGNING AND BUILDING PROTOCOLS
FOR WEB APPLICATIONS

by
June Sup Lee

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Science)

August 1999

Copyright 1999 June Sup Lee
DEDICATION

To my wife and son, Hewon Han & Peter Jihoon Lee
To my parents, Wonshik & Jungsook Lee
ABSTRACT

This thesis is about three topics, (i) software architectures, (ii) building application-level protocols, and (iii) transforming legacy software onto the web. In part (i) I describe an architectural style in which the computation is divided between the client and server. I call this style the 3-tier Web architecture. I provide a formal specification for the 3-tier Web architecture. One important aspect of this formal specification is how it assists in identifying various elements to consider when implementing an instance of this architecture. I present two instances of the 3-tier Web architecture - Web-COCOMO and Web-WinWin, which I have designed at the Center for Software Engineering at USC. While designing and building these example Web applications, I discovered that building the application-level protocol is one of the key issues in building Web applications. In part (ii) I focus down within the 3-tier Web architecture on the question of creating application-level protocols. I design and build a tool which helps a programmer generate such a protocol using RPC elements. The tool is novel in several ways, including the fact that it generates a complete and consistent protocol which is also efficient. In part (iii) of this thesis I apply the 3-tier Web architecture and my protocol tool to the problem of transforming legacy software systems to the Web. I identify the key elements in the process and show how the architecture and my tool can be successfully used.
# TABLE OF CONTENTS

1.0 Introduction ................................................................................................................. 1
  1.1 Legacy Software Systems ............................................................................................ 2
  1.2 Applying Web Technology to Legacy Software Systems ............................................. 2
  1.3 What This Thesis All About ......................................................................................... 5

2.0 Web-based Software Architectures .................................................................................. 7
  2.1 Server-side Architectural Style ..................................................................................... 7
    2.1.1 CGI ........................................................................................................................ 8
    2.1.2 Servlets .................................................................................................................. 10
    2.1.3 Server APIs ............................................................................................................ 11
  2.2 Client-side Architectural Style ..................................................................................... 11
  2.3 The 3-tier Web Architecture ......................................................................................... 13
    2.3.1 Overview of the 3-tier Web Architecture ............................................................... 14
  2.4 Examples of transforming legacy code into Web ......................................................... 16
    2.4.1 Application to COCOMO ....................................................................................... 16
    2.4.2 The Protocol of Web-COCOMO ......................................................................... 17
    2.4.3 Analysis of the Web-COCOMO Protocol .............................................................. 20
    2.4.4 Application to WinWin ........................................................................................... 21
    2.4.5 The protocol of Web-WINWIN ............................................................................. 22
    2.4.6 Analysis of the Web-WinWin Protocol ................................................................. 26
  2.5 Formal Definition of the 3-tier Web Architecture ......................................................... 28

3.0 Communication between Client and Server ................................................................... 35
  3.1 Customized protocol - Message Passing ................................................................... 35
  3.2 Remote Procedure Call (RPC) ..................................................................................... 35
  3.3 Remote Method Invocation (RMI) .............................................................................. 36
    3.3.1 Java Native Interface (JNI) ................................................................................... 38
  3.4 CORBA and IIOP ....................................................................................................... 39
    3.4.1 CORBA (Common Object Request Broker Architecture) .................................... 39
    3.4.2 IIOP (Internet Inter-ORB Protocol) ..................................................................... 40
  3.5 Applying Communication Protocols to the 3-tier Web Architecture ......................... 41

4.0 A Tool for Building Application Level Protocols ......................................................... 44
  4.1 Developing Application-level Protocols ....................................................................... 44
  4.2 Incremental RPC tool ................................................................................................. 44
    4.2.1 An example of using the IRPC tool ...................................................................... 49
    4.2.2 Advantages of using the IRPC tool ...................................................................... 52
  4.3 Meta Protocol Specification ......................................................................................... 54
    4.3.1 Message Formats ................................................................................................. 54
    4.3.2 IRPC Data Types ................................................................................................... 57
    4.3.3 An example protocol generated by the IRPC tool ............................................... 59
    4.3.4 Implementation Details ....................................................................................... 60
5.0 Evaluating the Generated Protocols
5.1 The Methodology
5.2 The Boxplot
5.3 Response Time Analysis
5.4 Statistical Decisions and Tests of Hypotheses of the Response Time
5.5 RMI
5.6 CORBA/IIOP
5.7 The IRPC protocol
5.8 Comparison of Protocols on a Fast Network
5.9 Comparison of Protocols on a Slow Network
5.10 Overall Conclusions of Protocol Efficiencies

6.0 Creating an Application Server and Client from a Legacy Program
6.1 The Basic Ideas
6.1.1 Generating the Application Server
6.1.2 Generating the Client
6.2 An Example of Building Web Applications Using the IRPC Tool
6.2.1 Generating Binding Routines
6.2.2 Generating the Protocol Using the IRPC Tool
6.2.3 Generating the User Interface
6.2.4 Deploying Files
6.2.5 Revising the Protocol Using the IRPC Tool
6.2.6 The Summary

7.0 Conclusions

8.0 References

9.0 Appendix A: Test Results

10.0 Appendix B: Sample Program Files
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Server-side architecture</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Client-side architecture</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>The 3-tier Web Architecture</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>A closer look at the 3-tier Web architecture</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>an instance (COCOMO) of the 3-tier Web Architecture</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>an instance (WINWIN) of the 3-tier Web architecture</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>Example Code of Web-WinWin</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>Picture of the 3-tier Web Architecture</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>The architecture of the RMI system</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>An application of Java and C using JNI</td>
<td>39</td>
</tr>
<tr>
<td>11</td>
<td>Simplified 3-tier Web Architecture</td>
<td>41</td>
</tr>
<tr>
<td>12</td>
<td>IRPC Main Window</td>
<td>46</td>
</tr>
<tr>
<td>13</td>
<td>Procedure Window</td>
<td>47</td>
</tr>
<tr>
<td>14</td>
<td>An example routine in the IRPC tool</td>
<td>50</td>
</tr>
<tr>
<td>15</td>
<td>An example client code generated by the IRPC tool</td>
<td>51</td>
</tr>
<tr>
<td>16</td>
<td>An example server code generated by the IRPC tool</td>
<td>52</td>
</tr>
<tr>
<td>17</td>
<td>Boxplot of time length</td>
<td>64</td>
</tr>
<tr>
<td>18</td>
<td>Boxplot of RMI protocol in milliseconds on 10 Mbps LAN</td>
<td>74</td>
</tr>
<tr>
<td>19</td>
<td>Boxplot of RMI protocol in milliseconds on 33.6 Kbps network</td>
<td>75</td>
</tr>
<tr>
<td>20</td>
<td>Boxplot of CORBA/IIOP protocol in milliseconds on 10 Mbps LAN</td>
<td>78</td>
</tr>
<tr>
<td>21</td>
<td>Boxplot of CORBA/IIOP protocol in milliseconds on 33.6 Kbps network</td>
<td>79</td>
</tr>
<tr>
<td>22</td>
<td>Boxplot of the IRPC protocol in milliseconds on 10 Mbps LAN</td>
<td>82</td>
</tr>
<tr>
<td>23</td>
<td>Boxplot of the IRPC Protocol in milliseconds on 33.6 Kbps Network</td>
<td>83</td>
</tr>
<tr>
<td>24</td>
<td>Splitting legacy code</td>
<td>89</td>
</tr>
<tr>
<td>25</td>
<td>Sample C Code (sample.c)</td>
<td>92</td>
</tr>
<tr>
<td>26</td>
<td>The Sample Routine in the IRPC Tool</td>
<td>95</td>
</tr>
<tr>
<td>27</td>
<td>Sample User Interface</td>
<td>96</td>
</tr>
<tr>
<td>28</td>
<td>Sample Event Handling Routine</td>
<td>96</td>
</tr>
<tr>
<td>29</td>
<td>Revised Sample C Code (sample.c)</td>
<td>99</td>
</tr>
<tr>
<td>30</td>
<td>Revised Event Handling Routing</td>
<td>100</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 1. Time for Operations on sunset.usc.edu ................................................................. 13
Table 2. Box Instances for the 3-tier Web Architecture ......................................................... 30
Table 3. Line Instances for the 3-tier Web Architecture ......................................................... 30
Table 4. Port Instances for the 3-tier Web Architecture ......................................................... 31
Table 5. Possible combinations of Application server, Client, application-level protocol ............ 42
Table 6. Statistics of RMI protocol in milliseconds on 10 Mbps LAN ..................................... 72
Table 7. Statistics of RMI protocol in milliseconds on 33.6 Kbps network .............................. 72
Table 8. Statistics of RMI test result in milliseconds .............................................................. 73
Table 9. Statistics of CORBA/IIOP protocol in milliseconds on 10 Mbps LAN ....................... 75
Table 10. Statistics of CORBA/IIOP protocol in milliseconds on 33.6 Kbps network ............. 76
Table 11. Statistics of CORBA/IIOP test result in milliseconds ............................................ 77
Table 12. Statistics of the IRPC protocol in milliseconds on 10 Mbps LAN ............................ 79
Table 13. Statistics of the IRPC in milliseconds on 33.6Kbps network .................................. 80
Table 14. Statistics of the IRPC protocol test result in milliseconds ....................................... 81
Table 15. Comparison of Protocols in milliseconds on 10 Mbps LAN .................................. 84
Table 16. Comparison of Protocols in milliseconds on 33.6 Kbps network ............................ 86
Table 17. First Experiment of Communication Speed ............................................................ 108
Table 18. Second Experiment of Communication Speed ......................................................... 109
Table 19. Third Experiment of Communication Speed .......................................................... 110
Table 20. First Experiment of Marshaling Speed ..................................................................... 111
Table 21. Second Experiment of Marshaling Speed ............................................................... 112
Table 22. Third Experiment of Marshaling Speed ................................................................... 113
1.0 Introduction

The Internet is a global digital infrastructure that connects millions of computers and tens of millions of people. It contains within those computers a wide variety of information, much of which can be browsed, searched and transferred to a machine connected to it. Information on the Internet comes in many forms, including text, graphics, video, and sound. Some information is displayed, some must be searched.

The World Wide Web (or Web) is an Internet-wide hypermedia information retrieval system that provides universal access to a large universe of resources. It has brought the Internet into the public consciousness. According to the MERIT Network Information Center [3], during April 1995 the Web became the largest byte and packet mover on the Internet.

The Web was first designed to provide access to documents in an unidirectional way [4]. But the Web has become a more interactive client/server application platform. Now, programming the Web is a rapidly growing phenomena. Web applications (Web-based systems) can take advantage of the benefits of the Web such as platform independence, wide area access, and handling of hypertext and multimedia. Most documents which display on a Web browser can be displayed on another Web browser on another machine. A developer can easily distribute an application to millions of clients by placing it on a public Web site. A user can access the latest version of an application simply by clicking on an anchor in a Web page.
1.1 Legacy Software Systems

Legacy software systems, by definition, have been in operation for years. Such systems usually suffer from old age in several respects. First, the original developers are not no longer with the company, and little or no documentation exists to describe the internal workings of the software. Second, maintenance of the legacy software systems must continue. Especially, when the underlying hardware changes or when the operating system is upgraded, legacy software systems must also be adjusted. Third, the perennial issue of bug fixes may arise. As a result, system maintenance can occupy many programmers full time, often leaving no time or money to enhance or totally rewrite the application. The development of Web technology (browsers, servers, and the like) is having a major impact far beyond the world of Web sites. In particular, flourishing corporate intranets make use of Web technology within the confines of their organizations [19].

1.2 Applying Web Technology to Legacy Software Systems

If we transform legacy software systems to the Web, we can take advantage of Web properties such as platform independence, wide area access, instant publication of the software and so on. Once we transform a legacy software system to the Web in a corporate intranet environment, we need not worry about a client machine running out-of-date software. All the people are using the same version of the software, therefore, we can upgrade the software by simply replacing the old software on a server machine with new
The user interface of legacy software systems can also be transformed into a modern graphical user interface.

But there is no straightforward way to transform legacy software systems to the Web. One possible way is reprogramming legacy software systems in Java [7]. But building a system in Java is a highly controversial step from several perspectives. Most important is that a Java version necessitates an entire rewrite of the software, and any major development effort has risks associated with it. A Java-based development has the additional disadvantage of dealing with immature software, such as the Java Virtual Machine. So a positive decision to move ahead with a Java rewrite requires serious justification.

[19] shows the three basic steps to follow when migrating legacy software systems to a Web client/server environment. The first step is to thoroughly understand the legacy software system. This has two facets: understanding what the software does and understanding how it does it. Moving from code to design documentation remains an art, but a necessary first step. If this were easy, legacy systems wouldn’t be so problematic. But it is a very hard problem.

The second step is specifying the system architecture for Web software systems. For hosting to the Web, this usually means determining what will be executed on the client side, what will be executed on the server side, and what communication between client and server will occur. The 3-tier Web architecture, presented later, aids in this process.
On the client side, the main issue is the user interface. There are two competing alternatives: HTML forms [5] or Java. HTML forms represent a reliable and efficient mechanism for capturing user input, but they provide limited capabilities for creating user interfaces. With Java’s Abstract Windowing Toolkit, we have a much more powerful mechanism for creating the interface we want, but at the expense of performance.

A second major client side issue is the amount of computation to be performed. This is intimately related to the amount of information to be moved from the server to the client and how long that will take.

The main issue on the server side is how to create the application server from the legacy system. Usually we must extract the old user interface code; this can be easy or difficult depending on the original architecture of the software.

Another issue about determining the client and server side programs is how the distributed application is split between the client and the server. The fat server model places more function on the server. The fat client model does the reverse [18]. This is intimately related to the amount of information to be moved from the server to the client and how long that will take.

Once we’ve determined the client and server side programs, we must decide on the actual communication protocol between them. A handcrafted program will use a socket library and will transfer information between client and server. Handcrafted protocols can be flexible and efficient but are cumbersome and error prone.

Rather than a handcrafted protocol, we may want to use one of several competing protocols for managing distributed objects. CORBA’s IIOP [21] and Sun’s
RMI [22] are two such protocols that help automate communication between client and server.

The third step is testing and tuning the software system. Once we’ve converted the software, traditional issues about its architecture will arise. Performance, reliability, and security will be three of the biggest problems.

1.3 What This Thesis All About

In this thesis, I focus on the architectural issues of Web-based systems and especially on where the computation should be done. I first discuss the “standard” Web software architecture styles and discuss their pros and cons. I then describe a hybrid software architecture style in which the computation is split into both the client and the server, which I call the 3-tier Web architecture. This architecture is valuable for several reasons. One is that I provide a formal specification for the architecture. Thus it is possible to transform the abstract architecture into a specific implementation assuring correctness and consistency as the process continues. Secondly, the architecture identifies key characteristics that must be considered (e.g. optimized) to attain efficiency of implementation.

In the second part of this thesis I focus down on the question of building the application protocol for the 3-tier Web architecture. Of all of the elements involved in transforming an application onto the Web, this is the single most complicated step. As such, I have designed and built a tool to help the programmer create this protocol. I call the tool IRPC for Incremental Remote Procedure Call. Since the success of Web-based
software systems following this architecture style is highly dependent on the communication protocol between the client and the server, this tool will play an important role for developers. I evaluate the tool on the basis of its utility to produce fully correct and efficient protocols.

In the last part of this thesis I apply the material developed in the first two sections to the application of transforming legacy software systems onto the Web.
2.0 Web-based Software Architectures

In this chapter, I describe three different architectural styles. First, I describe server-side and client-side architectural styles which are widely used to build Web-based software. I describe the pros and cons of these architectures. I then present an architectural style in which the computation is divided between the server and the client.

2.1 Server-side Architectural Style

Figure 1 shows the architecture of server-side Web applications. After a user on the client-side (browser-side) fetches a Web document which contains an anchor invoking a server-side program (called a script), (s)he sends a request by clicking on an anchor (1, 2). The Web server gets the request and invokes the script (3). The script is now executed on the server-side and may read/write the data source (4). Then the script usually generates a Web page containing the result and the Web server sends the Uniform Resource Locator (URL) [5] of the Web page to the client (5, 6).
The Common Gateway Interface (CGI) is the earliest method for executing a server-side script. More recently, the CGI method is being replaced by other mechanisms such as Web server APIs and Java servlets. Here, I briefly introduce those mechanisms one by one.

2.1.1 CGI

The Common Gateway Interface is a standard for interfacing external applications with information servers, such as Web servers. A plain Hypertext Markup Language (HTML) document, that the Web daemon retrieves is static, which means it exists in a constant state: a text file that doesn't change. A CGI program, on the other hand, is executed in real-time, so that it can output dynamic information [5].

There are many challenges that need to be addressed before an instance of this architecture using CGI can lead to a successful implementation. The first problem is the fact that CGI computation is stateless. For every invocation of a CGI program, a new instance of a program must be started. This causes a serious problem in that the program can not reflect the change of the state of the client. There are ways to work around the statelessness of CGI programs. One method is to use hidden fields within a form to maintain state on the client side. So, the program stores the state of the process in the forms it sends back to the client instead of storing it in its own memory. But this solution increases the network traffic between the client and the server because the data in the hidden fields is transferred back and forth for every data transfer between the client and the server, and the data is getting bigger and bigger as these steps go on. Therefore,
applying this solution to complicated software systems is a real work of art. Another method is to store the client's state in a relatively small file (called a cookie) on the client's computer. But this method also increases the network traffic because the server doesn't keep the state information of the client and asks the state information to be transferred from the client.

The second problem is that the user interface created in HTML is very limited. It is very hard to implement a good user interface using <FORM> elements provided by HTML.

The third problem is related to the server performance in this architecture using CGI. All computation is done on the server-side. The Web server needs to create a new process to invoke a CGI script and for each CGI script being executed, there is a corresponding Web daemon process to manage it. These features don't create serious problems if a server-side program does a simple job in a short period of time. But in case of using the server-side architecture for a complicated application which lasts relatively long periods, these features may cause the server machine to be flooded with processes and make the server machine become a bottleneck.

There are also some other problems related to data transfers. All data must pass across the internet/intranet via stateless HTTP protocol. This means that there is not much room to localize the action to diminish the frequency of data transfers.
Recently the Common Gateway Interface (CGI) method is being replaced by other mechanisms. Here I briefly introduce those mechanisms, Java servlets and a Web server’s API.

2.1.2 Servlets

Servlets are modules that extend request/response-oriented servers, such as Java-enabled Web servers [27]. Multi-threaded servlets are loaded and initialized in Java-enabled Web servers, which then handle client requests by exchanging HTTP messages with clients. Since servlets are running in Java-enabled Web servers, and are multi-threaded, there is no need of invoking a new process for every servlet call. But on the client side, the user interface is still limited because <FORM> elements are used.

Servlets and clients are sending messages to each other via HTTP protocol. This means that servlets need to take care of the statelessness of the HTTP protocol. The servlet API provides two ways to track client state - Session tracking and Cookies. Session tracking is a mechanism that servlets use to maintain state about a series of requests from the same user across some period of time. Cookies are a mechanism that a servlet uses to have clients hold a small amount of state information associated with the user. Servlets can use the information in a cookie as the user enters a site, as the user navigates around a site, or both. These two ways are better organized than using hidden fields within a form in CGI. But the state-related data still needs to be transferred between the servlet and clients because the servlet does not keep the state of clients in either of two ways.
2.1.3 Server APIs

Though most Web servers have APIs, here I specifically discuss the Web Application Interface (WAI) API of the Netscape Enterprise Web server. WAI API can replace the CGI. WAI is a CORBA-based programming interface that defines object interfaces to the HTTP request/response data and server information [28]. Using WAI, a developer can write a Web application in C, C++, or Java that accepts an HTTP request from a client, processes it, and returns a response to the client.

WAI modules have similar properties as servlets. WAI modules are embedded in the enterprise Web server, and are multi-threaded. Therefore, there's no need of invoking a new process for every request from a client. But <FORM> elements are also used for the user interface on the client side, which are limited. WAI modules also use the HTTP protocol to communicate with clients. WAI modules keep track of client state by using cookies.

Whether the mechanism is CGI, API, or servlets, all the methods are abstractly defined by the Server-Side Architecture of Figure 1.

2.2 Client-side Architectural Style

Java is an object-oriented programming language developed by Sun Microsystems [6]. Java applets are programs written in Java that are embedded in Web
pages to run on the client-side in an applet-enabled Web browser. For more information about Java and Java applets, see [7].

Figure 2 shows an architecture for client-side Web applications using Java applets. A user sends a request by clicking on an anchor in a Web page (1, 2). The Web server gets the request and finds the appropriate Web document which contains a Java applet (3). The Web server sends the applet to the browser (4, 5). The browser receives the applet, loads it and runs it.

There are several problems associated with this strategy.

1. In the client-side architectural style, the time to download an applet for a "large" program can take minutes or longer because the applet is composed of Java class files and all the class files must be downloaded via internet/intranet to the browser one by one. All initial data must also be downloaded to the browser before the applet is loaded.

2. The Java virtual machine must be started on the client side. This can often take a long time. The client sees this by looking towards the bottom of the browser and noticing the phrase "Starting Java".
3. Another problem is that all computation is done on the client-side. Java is an interpreted language and is slow.

Table 1 shows the result of running a simple program written in Java against the same program written in C. The test program was running on sunset.usc.edu which is an ultra sparc 1 with 128 mega bytes of memory. The test program checks the time for incrementing an integer from 0 to 10000 and checks the time for comparing two strings, “this is a string” and “this is a string”, for 1000 times. One can see that the Java version is more than 10 times slower than the same program written in C.

<table>
<thead>
<tr>
<th></th>
<th>Incrementing an integer from 0 to 10000</th>
<th>Comparing strings 1000 times</th>
</tr>
</thead>
<tbody>
<tr>
<td>In C</td>
<td>In Java</td>
<td>In C</td>
</tr>
<tr>
<td>1 ms</td>
<td>14 ms</td>
<td>1 ms</td>
</tr>
</tbody>
</table>

In conclusion, client-side applications must be carefully designed to avoid these pitfalls. Given the tremendous variety of client computers, a general rule is to avoid client-side computation as much as possible.

2.3 The 3-tier Web Architecture

The 3-tier Web architecture is a software specification whose goal is to incorporate all of the major elements of web applications. By doing so it makes clear to implementors what factors must be considered for their application. The 3-tier Web architecture splits an application into 2 parts, a user interface and an application server. The user interface is a Java applet which runs on the client machine and communicates
with the application server via the application-level protocol. A detailed explanation of the application-level protocol is presented further on in this thesis. The value of the 3-tier Web architecture is that it lets one explore alternate ways to divide an application so it will run efficiently on the Web. These applications may either be legacy programs or new programs. The user interface on the client-side does not do any major share of the execution of the application, but only executes user interface actions which necessarily must run on the client-side. All the other computation is done by the application server on the server-side. The application server is in charge of data processing, communication with data source(s), and is also in charge of managing the application’s data structures.

![Diagram of the 3-tier Web Architecture](image)

**Figure 3. The 3-tier Web Architecture**

### 2.3.1 Overview of the 3-tier Web Architecture

Figure 3 shows the 3-tier Web architecture and explains how a Web application following the 3-tier Web architecture is initially started. First a user sends a request to a Web server by clicking on an anchor in a Web page (1, 2). The Web server finds a Web document which contains a Java applet that implements a remote user interface (3). The
Web server sends the applet to the client (4, 5). After the applet-enabled browser loads and runs the applet, the applet sends a request to the Web server asking for a TCP/IP connection with an application server on the Web server machine (6, 7). The Name server initiates the application server (8). And a connection is made between the application server and the Client (9, 10).

After the connection between the application server and the client is made, these two sides communicate with each other via the application-level protocol. The Web server and name server processes do not intervene anymore. The 3-tier Web architecture makes use of the Web server and the name server only for the initialization of the Web applications.

Figure 4. A closer look at the 3-tier Web architecture
Figure 4 shows two instances of the 3-tier Web architecture after the connection is made. The application server does the data processing, calculation and data Input/Output by request of the client via the application-level protocol. The client gets input from the user, sends a request to the application server, receives the result from the application server and displays the result.

The main requirement of the 3-tier Web architecture is that an application have a user interface that can be separated from the rest of the application. The data source can be a file system, a DBMS, multiple DBMS's or another application. The detailed relationship between the data source and the application server is determined by the type of the application.

2.4 Examples of transforming legacy code into Web

One advantage of the 3-tier Web architecture is that it provides a clear way to transform legacy programs into Web applications. The transformation is done in such a way that the application developer splits the legacy code into two parts, the application server and the client. At the Center for Software Engineering at USC we have already built two applications using the 3-tier Web architecture as our guide. These are briefly described.

2.4.1 Application to COCOMO

COCOMO (Constructive Cost Model) is a screen-oriented, interactive software package that assists in budgetary planning and schedule estimation of a software
project prior to any work beginning [8]. COCOMO is a standalone tool. The data source of COCOMO is a local file system. In Figure 5, you see Web-COCOMO application server which is the same as the ordinary COCOMO application except that the user interface part is detached. Web-COCOMO also saves the data into the application server’s local file system. In this architecture, we don’t need the name server because the Web server performs the function of the name server. Web-COCOMO is available at http://sunset.usc.edu/javacocomo/cocomo_client/cocomo.html.

![Diagram of 3-tier Web Architecture](image)

**Figure 5. an instance (COCOMO) of the 3-tier Web Architecture**

### 2.4.2 The Protocol of Web-COCOMO

In this section, I describe the customized protocol used by Web-COCOMO. This protocol is an application-level message passing protocol which works on top of TCP/IP. There are two message formats - Request and Response. A client initiates the Request and the server initiates the Response. The BNF for request and response messages is given below.

- **Request Message**

  Request = Request-Header CD Request-Argument CEL
Request-Header = (DO | GET | SET) CD #_of_Args

Request-Argument = do-args | get-args | set-args

do-args = INIT | NEW | COMPUTE | ACCEPT | RESET | ADD

| CREATE CD user_name CD passwd
| CHECK CD user_name CD passwd
| LOAD CD <proj_name|model_name|calib_name>
| SAVE CD <proj_name|model_name|calib|name>
| DELETE CD <mod_index|proj_index>
| CALCULATE (CD <arg>)*
| (COPY|PASTE) CD mod_index

set-args = PROJ_NAME CD proj_name

| PROJ_EFFORT CD proj_index CD proj_effort
| PROJ_SCHEDULE CD proj_index CD proj_schedule
| SCALE_FACTOR (CD <arg>)*
| SCED CD sced_val CD inter_sced
| (PRODUCT | PLATFORM | PERSONNEL | PROJECT | UDF)
(CD <dirver_value> CD <inter_value>)*
| SCALEFACTOR (CD <dirver_value> CD <inter_value>)*
| EQUATION CD effort-entry-value CD schedule-entry-value
| MOD_LINE CD mod_index (CD <arg>)*
| MOD_EDSI_BRAK CD mod_index CD brak
| MOD_EDSI_FLAG CD mod_index CD flag
| MOD_EDSI_FP (CD <arg>)*
| MOD_RATE CD mod_index CD rate
| MOD_EAF (CD <arg>)*

get-args = PROJECTS | PRJ_MOD_NUM | MODELS | PROJ_NAME
| CALIBRATIONS | SCALEFACTOR
| EQUATION | PROJ_REPORT | PROJ_TOTAL | HELPS
| RISK_LEVEL CD mod_index
| (MOD_LINE | MOD_EAF | MOD_EDSI_FLAG) CD mod_index
| (MOD_FP_LANG | MOD_EDSI_AAF | MOD_EDSI_FP)

CD mod_index

| PROJ_LINE CD proj_index
| (MOD_OVERALL | MOD_PLAN_RPT | MOD_DESIGN_RPT | MOD_INTE_RPT) CD mod_index
| (PRJ_OVERALL | PRJ_PLAN_RPT | PRJ_DESIGN_RPT | PRJ_INTE_RPT) CD proj_index
| (CAL_NAME | CAL_DATE | CAL_EFF | CAL_SCED)

CD proj_index

• Response Message

Response = Response-Header CD Response-Argument

Response-Header = Error_Status CD No_of_Args

Error_Status = 0 | 1 | 2

0: Success
1: General Error
2: Format Error

Response-Argument = PROJ_NAME | MOD_LINE | PROJ_TOTAL
| MOD_EAF | MOD_EDSI_FLAG | MOD_FP_LANG
| MOD_RISK | EQUATION | CAL_NAME
| CAL_PROJ_LINE | DEL_MOD | DEL_PROJ
| PROJECTS | MODELS | CALIBRATIONS
| OPEN_PROJ | PARAMETERS

- Miscellaneous

CD(COCOMO_DELIMITER) = "!/#%"

CEL(COCOMO_END_LINE) = "@$^"

2.4.3 Analysis of the Web-COCOMO Protocol

The Web-COCOMO protocol is designed based on the HTTP protocol. All data is transferred in the form of strings between the client and the server. In Web-COCOMO, most of the data types are integers and floats. Therefore, the conversion from strings to integers or floats and vice versa slows down the communication speed. The type conversion between strings and integers, strings and floats takes a relatively long time, especially on the Java client side.

Handcrafted protocols tend to be implemented inconsistently. The Web-COCOMO protocol was designed by 4 students in the “Software Engineering Projects” class at USC in the spring semester in 1996. The BNF in the previous section was the part
of their report. But the protocol specified by the BNF doesn’t match the actual protocol. For example, according to the BNF, the request message to set the current project name should be “SET CD 2 CD PROJ_NAME CD proj_name CEL”. But the actual message in Web-COCOMO is “COCOMO_SET_PRJ_NAME CD 1 CD proj_name CEL”. The student who implemented the protocol ignored the BNF and implemented it on his/her own way. Furthermore, the BNF is not complete. There is no way to get the current project name by the BNF, which is necessary in Web-COCOMO.

Type checking is very important for reliable protocol implementation. Usually, the implementation of the protocol checks data types of the input items and sends error messages in case of finding wrong data types. In the Web-COCOMO protocol which is handcrafted, it is the responsibility of the programmer to implement this process. But in some cases, the implementation of the Web-COCOMO protocol does not check the data types, which often results in the program crashing.

From this example it becomes clear that a tool that maintains consistency between protocol messages and documentation, and that automatically checks type is needed.

2.4.4 Application to WinWin

WinWin is a computer program that aids in the capture, negotiation, and coordination of requirements for a large system [9]. Viewed architecturally, WinWin consists of a set of data files shared among multiple WinWin processes. Those files are managed by a data-server process. Figure 6 shows how the Sparcstation version of
WinWin was transformed into Web-WinWin, while continuing to work with the Sparcstation version. Web-WinWin contains the same way of accessing the data sources and the same way of communicating with other WinWin processes. Web-WinWin application makes maximal use of WinWin legacy code. Only the user interface and a message protocol needed creation. In this architecture, we don't need the name server because the Web server performs the function of the name server. We have built a preliminary version of Web-WinWin and it is at http://sunset.usc.edu/JWinWin/WinApplet.html.

![Diagram](image)

**Figure 6. an instance (WINWIN) of the 3-tier Web architecture**

### 2.4.5 The protocol of Web-WINWIN

In this section, I describe the customized protocol used by Web-WINWIN. This protocol is an application-level message passing protocol which works on top of TCP/IP. There are also two message formats - Request and Response. A client initiates the
Request and the server initiates the Response. The BNF for requests and responses is given below. For example, if the request message is “GET SP WWTP/1.0 SP ARTF_LIST SP AGRE EOR”, the corresponding response message is “WWTP/1.0 SP 200 CRLF ARTFLIST CRLF 3 CRLF junelee-AGRE-1 CRLF joolee-AGRE-2 CRLF junelee-AGRE-2 CRLF CRLF EOT”.

- **Request Message**

  Request = Request-Header SP Request-Argument EOR

  Request-Header = (DO | GET | SET) SP Version#

  Version# = WWTP/1.0

  Request-Argument = do-args | get-args | set-args

  do-args = ADDUSER SP password SP user-name SP role SP title SP position SP organization

  | CHECK

  | CREATE SP project-name SP password SP role SP title SP position SP organization

  | DELETE

  | DELUSER SP user-name SP password

  | EXIT

  | OPEN SP project-name SP password

  | UPDATE
| TAX_ADD_NODE SP index
| TAX_DELETE_NODE SP index
| TAX_EDIT_NODE SP index SP tax_item
| TAX_MOVE_NODE SP index SP direction
| TAX_MODE

get-args = ARTF_ID SP artifact-id ( all | field-name )

| ARTF_LIST SP ( artifact-type | MESS )
| FACTS
| LOPTION SP artifact-type
| POPTION
| PROJECTS
| TAXONOMY
| TAXRELARTF SP index-of-taxonomy-list
| USERDATA SP user-name
| USERS

artifact-type = TERM | WINC | ISSU | OPTN | AGRE

field-name =  Cdate | Rdate | Name | Body | Priority | Status | State

| Taxonomy | RefTo | RefBy | Attachment | Desensitized
| ArtfSet | Comments | VData | VPolicy | VState

set-args = ARTF_ID SP artifact-id SP field-name \n
   num-of-lines \n value \n
| CREATE_ARTF SP artifact-type
| DELETE_ARTF SP artifact-id
| LOPTION SP value
| POPTION SP value
| USERDATA SP user-name SP password SP role SP
title SP position SP organization

- **Response Message**

Response = Response-Header CRLF Response-Argument EOT

Response-Header = Version# SP Status-Code

   ( line-of-Error-Msg CRLF Error-Msg CRLF | 0 CRLF )

Status-Code = 200 | 300 | 400

200 : Success

300 : Bad Request

400 : Server Error

Response-Argument = response-args CRLF

   ( line-of-Response-Msg CRLF Response-Msg CRLF CRLF | CRLF )

response-args = TAX_RELAX | ARTFLIST | GET_ARTF | SET_ARTF

   | GETLOPTN | SETLOPTN | CREATPRJ | PRJ_FACT

   | USERLIST | MOD_USER | USERDATA | CREAT_AR

   | DEL_ARTF | PROJECTS | GETPOPTN | SETPOPTN

   | DEL_PROJ | DEL_USER | OPEN_PRJ | UPDT_PRJ

   | EXPT_PRJ | CHEK_PRJ | TAX_LIST | ADD_USER

   | TAX_ADD | TAX_DEL | TAX_MODE | TAX_EDIT
2.4.6 Analysis of the Web-WinWin Protocol

The Web-WinWin protocol is also designed based on the HTTP protocol. All data is transferred in the form of strings between the client and the server. In Web-WinWin, this doesn’t cause any performance degradation because most of the WinWin data types are strings.

The Web-WinWin protocol was designed by 3 students in the “Software Engineering Projects” class at USC in the spring semester in 1996. The protocol specified by the BNF in the previous section matches the actual protocol. In both the BNF and the actual protocol, there are 29 request messages and corresponding 29 response messages excluding one error message.
In Web-WinWin, protocol related code is mixed with other code, which makes it very hard to read and modify the code. Figure 7 shows example code of Web-WinWin which runs on the server-side. This routine receives a string as an input and edits a WinWin taxonomy item with it. And the routine sends the resulting taxonomy line to the client. As shown in figure 7, protocol related code (388 - 391, 421-422, 425) is mixed with
other code (405 - 407) which actually deals with application specific data structure management.

The implementation of the Web-WinWin protocol checks data types in messages inconsistently. Usually, it checks data types and sends error messages in case of finding wrong data types. But in some cases, it fails to do so, which results in the program crashes.

2.5 Formal Definition of the 3-tier Web Architecture

To provide a formal specification of the 3-tier Web architecture I adopt the notation first introduced by Horowitz in [26]. In this paper, an architecture is composed of three basic entities: box, line and port. Each entity has a set of attributes and methods, in effect an abstract data type. Attributes are merely names and associated data types. Methods are abstract programs stated in a procedural style of language such as Java.
Figure 8 shows the picture of the formal definition of the 3-tier Web architecture. This picture has following boxes, lines and ports:

- **Boxes**: client, Web server, name server, application server, Web browser
- **Lines**: the connection between the client and the application server, the connection between the Web server and the Web browser, the connection between the client and the name server, process activating of the application server by the name server, process spawning of the client by the Web browser
- **Ports**: the port for the Web server, the port for the name server, the port for the application server, the port for the client, the port for the Web browser
Table 2, Table 3, and Table 4 show the attributes of boxes, lines, and ports of the 3-tier Web architecture.

### Table 2. Box Instances for the 3-tier Web Architecture

<table>
<thead>
<tr>
<th>Box</th>
<th>Box</th>
<th>Box</th>
<th>Box</th>
<th>Box</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:Process</td>
<td>Type:Process</td>
<td>Type:Process</td>
<td>Type:Process</td>
<td>Type:Process</td>
</tr>
<tr>
<td>ID:1</td>
<td>ID:2</td>
<td>ID:3</td>
<td>ID:4</td>
<td>ID:5</td>
</tr>
<tr>
<td>Name:Client</td>
<td>Name:Application Server</td>
<td>Name:Web Browser</td>
<td>Name:Web Server</td>
<td>Name:Name Server</td>
</tr>
<tr>
<td>Shape:solid</td>
<td>Shape:solid</td>
<td>Shape:solid</td>
<td>Shape:solid</td>
<td>Shape:solid</td>
</tr>
<tr>
<td>PortList:11</td>
<td>PortList:12</td>
<td>PortList:13</td>
<td>PortList:14</td>
<td>PortList:15</td>
</tr>
</tbody>
</table>

### Table 3. Line Instances for the 3-tier Web Architecture

<table>
<thead>
<tr>
<th>Line</th>
<th>Line</th>
<th>Line</th>
<th>Line</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:pipe</td>
<td>Type:spawn</td>
<td>Type:pipe</td>
<td>Type:spawn</td>
<td>Type:pipe</td>
</tr>
<tr>
<td>ID:6</td>
<td>ID:7</td>
<td>ID:8</td>
<td>ID:9</td>
<td>ID:10</td>
</tr>
<tr>
<td>Name:</td>
<td>Name:</td>
<td>Name:</td>
<td>Name:</td>
<td>Name:</td>
</tr>
<tr>
<td>Shape:double arrow</td>
<td>Shape:arrow</td>
<td>Shape:double arrow</td>
<td>Shape:arrow</td>
<td>Shape:double arrow</td>
</tr>
<tr>
<td>Direction:inout</td>
<td>Direction:in</td>
<td>Direction:inout</td>
<td>Direction:in</td>
<td>Direction:inout</td>
</tr>
<tr>
<td>BoxList:</td>
<td>BoxList:3,1</td>
<td>BoxList:</td>
<td>BoxList:5,2</td>
<td>BoxList:</td>
</tr>
<tr>
<td>PortList:13,14</td>
<td>PortList:</td>
<td>PortList:11,15</td>
<td>PortList:</td>
<td>PortList:11,12</td>
</tr>
</tbody>
</table>
The semantics of this architecture is defined using a notation similar to C or Java. Though "higher level" specification languages exist such as Z [34], ACME [32][33], this notation has two major advantages: it is more easily understood by the average computer scientist, and it permits simulation.

The methods for client, Web server and Web browser, name server and application server follow. The first segment of code verifies that an instance architecture is structurally well formed. That is followed by nine, so-called, dynamic properties.

/* Initial Structural Property Checking */

/* initializing each box */
Process ns = null, ws = null, wb = null, cl = null, as = null;

for (bx = first("Process"); bx != null; bx = bx.next()) {
/* Each process has only one port */
if (bx.portlist.length() != 1)
    Error ("error - too few or too many ports");

/* Initially, there are three processes -
Web server, name server and Web browser */
if (bx.name == "Web Server")
    if (ws == null)
        ws = bx; //setting the Web server Process
    else
        Error ("error - Too many Web Servers");
if (bx.name == "Web Browser")
    if (wb == null)
        wb = bx; //setting the Web browser Process
    else
        Error ("error - Too many Web Browsers");
if (bx.name == "Name Server")
    if (ns == null)
        ns = bx; //setting the name server Process
    else
        Error ("error - Too many Name Servers");
}

if (ws == null) Error ("error - No Web Servers");
if (wb == null) Error ("error - No Web Browsers");
if (ns == null) Error ("error - No Name Servers");

/ * Web Server is waiting for requests from the Web browser */
ws.wait();

/ * End of Initial Structural Property Checking */

/ * Numbers in parentheses refer to objects in figure 8 */
/ * Dynamic Property 1: The Web browser (3) sends request to the
  Web server (4) on line (6) */
wb.connect(ws.address, ws.portlist);
wb.wait();
ws.accept(wb.address, wb.portlist);
w.
wait();
wb.send(ws.address, ws.portlist, wb.getPage());
w.
wait();

/ * Dynamic Property 2: The Web server (4) returns applet to the
  Web browser (3) on line (6) */
ws.receive(url);
ws.send(wb.address, wb.portlist, ws.getPage(url));
w.
wait();

/ * Dynamic Property 3: The Web browser (3) spawns the client
  (1) on line (7) */
w.
receive(page);
c1 = wb.spawn("Process", ws.address, ws.portlist, page);
w.
wait();

/ * Dynamic Property 4: The client (1) requests the Name server
  (5) the port number of the application server (2) on line
  (8) */
cl.portlist = cl.create_port();
cl.send(ns.address, ns.portlist, "Get-Server-Port");
cl.wait();

/* Dynamic Property 5: The Name server (4) activates the
application server (2) on line (9) */
ns.receive("Activate-Process");
as = ns.activate("Process");
ns.wait();
as.portlist = as.activate_port();
as.send(ns.address, ns.portlist, as.portlist);
as.wait();
ns.receive(as.portlist);
ns.wait();

/* Dynamic Property 6: The Name server (5) returns the port
number (12) on line (8) */
ns.receive("Server-Port");
ns.send(cl.address, cl.portlist, as.portlist);
ns.wait();

/* Dynamic Property 7: The client (1) requests connection to
the application server (2) on line (10) */
cl.receive(as.portlist);
cl.connect(as.address, as.portlist);
cl.wait();

/* Dynamic Property 8: The application server (2) accepts the
request from the client (1) on line (10) */
as.accept(cl.address, cl.portlist);
as.wait();

/* Dynamic Property 9:
Loop until done
The client (1) sends requests to the application server (2)
on line (10)
The application server (2) responds to the client (1) on
line (10)
*/
while(true) {
  cl.send(as.address, as.portlist, request-message);
  cl.wait();
as.receive(request-message);
as.send(cl.address, cl.portlist, response-message);
as.wait();
  cl.receive(response-message);
  cl.wait();
}

/* primitive routines which are generally found in a standard C
libraries */
wait() suspends the routine while the input queue is empty.
send() writes the data in the local buffer into the output queue.
receive() reads the input queue and saves the data into the local buffer.
connect() writes the "connection setup message" into the output queue.
accept() reads the "connection setup message" from the input queue and writes the acknowledgement into the output queue.
spawn() creates a new process.
activate() creates a new process if there are no processes available, otherwise just returns the id of the available process.
activate_port() creates a new port if there are no ports available, otherwise just returns the number of the available port.
3.0 Communication between Client and Server

In this section, I discuss how to design and build the protocol between client and server. These protocols are application level and assume TCP/IP as the underlying technology.

3.1 Customized protocol - Message Passing

Traditional message passing provides two communication primitives - send and receive. The client sends a request message to the server. The server receives the message, handles the request and sends the reply message to the client. On either direction of message passing, the receiving machine buffers the message until the recipient issues a receive. Message passing is powerful and flexible but requires the programmer to provide "low-level" details such as converting data between client and server formats, handling machine and communication line errors, determining the address of the server machine, composing/decomposing messages from/into data structures according to the message passing protocol and keeping track of which routine to invoke in case of receiving a certain message.

3.2 Remote Procedure Call (RPC)

RPC allows a program on the client to invoke a procedure on the server using the standard procedure call mechanism. The most common implementation [1] uses stub procedures which are responsible for the interpretation of arguments and results of RPC.
calls. A client that makes a remote procedure call is actually calling a client stub. The client stub composes a message with the procedure name and parameters and sends the message to the server machine. A server stub receives the message, decomposes the message into the procedure name and parameters and invokes the procedure. The server stub waits for the procedure to finish, composes a message with the result and sends the message to the client stub. The client stub decomposes the message into the result and returns the result to the client.

3.3 Remote Method Invocation (RMI)

Java Remote Method Invocation (RMI) is a distributed object model for the Java language. RMI allows programmers to write distributed objects using Java [22][23]. A local object can invoke the method of a remote object just like it invokes the method of a local object.

Figure 9. The architecture of the RMI system
As in Figure 9, the RMI system consists of three layers. First, the stub/skeleton layer is the interface between the application and the rest of the RMI system. Second, the remote reference layer deals with the lower-level transport interface. Third, the transport layer is responsible for connection setup, connection management, and keeping track of and dispatching to remote objects (the target of remote calls) residing in the transport’s address space [22].

The stub/skeleton layer is the interface between the application layer and the rest of the RMI system. The stub/skeleton layer transmits data to the remote reference layer via the abstraction of marshal streams. Marshal streams employ a mechanism called object serialization which flattens a local Java object’s state into a serial stream that the object’s state can then be passed as a parameter inside a message [24]. A stub for a remote object is the client side proxy for the remote object. A skeleton for a remote object is a server side entity that contains a method which dispatches calls to the actual remote object implementation. When a local object calls a remote method, the stub marshals arguments to a marshal stream, informs the remote reference layer that the call should be invoked. Then the skeleton receives the marshal stream from the remote reference layer, unmarshals arguments from the marshal stream, makes the up-call to the actual remote object implementation, and marshals the return value or an exception onto the marshal stream. Then the stub receives the return value or exception from the remote reference layer, unmarshals it, and informs the remote reference layer that the call is complete.

The advantage of the RMI system is that the RMI system deals with communication related details, and the programmer doesn’t need to worry about it. Once,
the programmer generates a remote interface, the RMI compiler generates stubs and skeletons from it. But the RMI system provides only a Java-to-Java solution. This is a major limitation for using RMI systems for Web-based software systems because there are many existing applications which were not written in Java.

There are ways to get around this. We can use a legacy wrapper which connects Java code and legacy code together. This legacy wrapper can be Java Native Interface (JNI) or JDBC. We can also use IDL-based protocols such as Internet Inter-ORB protocol (IIOP) for the communication between the server and the client instead of using RMI’s protocol which is based on Java object serialization. But in this case, we lose the advantages of having Java on both sides such as the use of mobile code.

3.3.1 Java Native Interface (JNI)

The Java Native Interface (JNI) is a native programming interface which allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly [25].
JNI serves as the glue between Java and native applications. Figure 10 shows how the JNI ties the C side of an application to the Java side. JNI’s Invocation API allows programmers to embed the Java Virtual Machine into native applications. A Java program can call C routines and C libraries through JNI. JNI also enables C functions to create, inspect and update Java objects. C functions can also catch and throw exceptions and have these exceptions handled on the Java side.

3.4 CORBA and IIOP

3.4.1 CORBA (Common Object Request Broker Architecture)

The Common Object Request Broker Architecture (CORBA) is a specification for creating, distributing, and managing distributed program objects across a network. CORBA was developed by a consortium called the Object Management Group (OMG), which currently includes over 500 member companies.
The Object Request Broker (ORB) is an essential concept in CORBA. Given a network of clients and servers on different computers, ORB support means that a client program, which is treated as an object, can request services from a server program or from another object without having to understand where the server is in the network or what the interface to the server program actually is.

CORBA uses Interface Definition Language (IDL) to specify object interfaces and definitions. The IDL is purely declarative without containing any implementation details. IDL provides operating system and programming language independent interfaces to all the services and components that reside on a CORBA [19][20][21].

CORBA objects communicate with each other through Internet Inter-ORB Protocol (IIOP) which is implemented in ORBs.

3.4.2 IIOP (Internet Inter-ORB Protocol)

The General Inter-ORB Protocol (GIOP) element specifies a standard transfer syntax (low level data representation) and a set of message formats for communications between ORBs. The GIOP is built for ORB to ORB interactions and is designed to work directly over any connection-oriented transport protocol.

The GIOP specification consists of the Common Data Representation (CDR) definition, GIOP message formats, and GIOP Transport Assumptions. CDR definition shows the format in which the GIOP represents OMG IDL data types in an octet stream. GIOP message formats support eight message types - Request, Reply, CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError, and Fragment. The GIOP
describes general assumptions concerning any network transport layer that may be used to transfer GIOP messages. It also describes how connections may be managed, and constraints on GIOP message ordering.

The GIOP can be mapped onto a number of different transports, and specifies the protocol elements that are common to all such mappings. The GIOP by itself does not provide complete interoperability. The Internet Inter-ORB Protocol (IIOP) element specifies how GIOP messages are exchanged using TCP/IP connections. The IIOP specifies a standardized interoperability protocol for the Internet [21].

3.5 Applying Communication Protocols to the 3-tier Web Architecture

Figure 11 shows the simplified 3-tier Web architecture which doesn’t include binding details.

![Figure 11. Simplified 3-tier Web Architecture](image-url)
We can deploy multiple sets of technologies onto the tier one (client), tier two (application server) of the 3-tier Web architecture and use different protocols between them. For example, one possible set consists of a CGI script for the application server, HTML forms for the client, and the HTTP protocol for the application-level protocol. Table 5 shows six possible implementation combinations.

<table>
<thead>
<tr>
<th>Application Server</th>
<th>application-level Protocol</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CGI Script</td>
<td>HTTP</td>
<td>HTML forms</td>
</tr>
<tr>
<td>2 Java Servlet</td>
<td>HTTP</td>
<td>HTML forms</td>
</tr>
<tr>
<td>3 Web Server API</td>
<td>HTTP</td>
<td>HTML forms</td>
</tr>
<tr>
<td>4 Java Application Server</td>
<td>RMI/Object Serialization</td>
<td>Java Applet</td>
</tr>
<tr>
<td>5 C++/Java Application Server</td>
<td>CORBA/IOP</td>
<td>Java Applet</td>
</tr>
<tr>
<td>6 Application Server in any Language</td>
<td>Handcrafted Protocol</td>
<td>Java Applet</td>
</tr>
</tbody>
</table>

Currently, the top three rows in Table 5 are the most widely used to implement Web applications. But there are problems in these matches. The biggest problem is the lack of the user interface versatility on the client. HTML forms don’t provide “pop-up” windows or menus. HTML forms also don’t allow drawing lines, rectangles, circles and so on. In contrast, the bottom three rows use Java applets as clients. Therefore, we can take advantage of the versatile Java AWT (Abstract Windowing Toolkit) to implement the user interface.

The biggest restriction of the row 4 of Table 5 is that the use of RMI is a Java-to-Java solution. We can get around this restriction by using JNI on the server side at the expense of the performance.
The fifth row also has multiple problems. One of the problems is that the developer has to choose one Object Request Broker (ORB) to use out of multiple ORBs on the market. If the developer chooses an ORB which does not last a long period of time, (s)he would have a lot of trouble in replacing the ORB with another ORB. Another problem of the fifth row is that the client needs to have the client-side ORB installed. To get around this problem, some client-side ORBs are downloaded to the browser along with the client applet files and loaded together on the client machine. For example, the client-side ORB of Visibroker [30] is written in Java and is downloaded to the client machine along with client files. But this scheme causes much longer downloading time especially on a slow network. I will talk more about this in section 5.1.

The sixth row doesn't have those restrictions mentioned above. But the biggest problem it has is that the developer has to worry about the communication details. The protocol generation process is error prone and the resulting protocols can be inconsistent.

In the next chapter, I will introduce a tool which lets the developer take advantage of the benefits of the sixth row without worrying about the communication details.
4.0 A Tool for Building Application Level Protocols

The 3-tier Web architecture relies upon the development of a protocol between the application server and the client. In this section I consider how best to develop this protocol.

4.1 Developing Application-level Protocols

In the 3-tier Web architecture, the protocol should be tailored in such a way that it takes into account special features of Web applications such as low bandwidth concerns. In order to facilitate the definition of this protocol, and the constant change that must occur until an acceptable protocol is arrived at, I have developed a tool. This tool, when used on an application based upon the 3-tier Web software architectural style assists a programmer in creating and modifying an application-level protocol with ease.

4.2 Incremental RPC tool

The programmer first generates the application server procedures that are to be called by the clients. Then the programmer runs the Incremental RPC (IRPC) tool. The programmer enters the procedure name, the number of parameters, the parameter types, and the return value type to the IRPC tool. The IRPC tool generates two files of source codes, one for the server and the other for the client. The client code includes the client stub and the server code includes the server skeleton and a dispatching procedure. The procedure name in the client stub is the same as the procedure name on the server so that
the programmer writes client code to call the remote procedure as if the procedure is on
the client side. The programmer doesn’t need to create the interface modules and also
doesn’t need to use the stub compilers. The IRPC tool has a graphical user interface to get
inputs from application programmers. The IRPC tool also follows the 3-tier Web
architecture so that it can be placed on a Web site. The programmer uses the IRPC tool
incrementally, developing and testing protocols. Figure 12 and Figure 13 show the main
user interface for the IRPC tool. Currently, the IRPC tool generates Java client code and C
server code.

Figure 12 shows the main window of the IRPC tool. This main window
provides three different views selected by the “View” menu. In the “Procedures” view, the
window shows the list of functions which have been implemented. In the “Client Code”
view, the window shows the client stub code so far implemented. And in the “Server
Code” view, the window shows the server stub code so far implemented.
Screen 1 of Figure 12 is an example of the IRPC tool's main window showing a list of all the procedures. The application programmer can modify a selected procedure, enter a new procedure, delete a selected procedure, and sort procedures by name or by the procedure-id generated by the tool. The application developer can generate and see the client code and the server code by clicking the "Client Code" command and "Server Code" command respectively in the "View" menu. Screen 2 of Figure 12 shows the IRPC tool's main window when viewing the client code created by the IRPC tool.
Screen 1 of Figure 13 shows the IRPC tool's procedure window. The application developer can modify the name of the procedure, the description which is used as comments in the server and client code, and the return type for this procedure. Parameter names are decided by the IRPC tool. The number of parameters is updated automatically by adding/deleting a parameter to this procedure. The IRPC tool generates two types of messages - blocked and nonblocked. If a client sends a blocked message to the application server, the client is blocked until it receives the response from the application server. If a client sends a nonblocked message, the client is not waiting for the response of the server. Screen 2 of Figure 13 shows a different view of the IRPC tool's procedure window. The application developer can modify the "Type" field of a parameter,
and the “Set By” field of a parameter. If a parameter is “set by client”, the client stub sends
the contents of this parameter created by the client to the server. If a parameter is “set by
server”, the client allocates a space which is filled by the client stub with the data received
from the server. If a parameter is “set by both”, the client stub sends the contents of this
parameter created by the client to the server and the client stub fills the parameter with the
data received from the server. The application developer can add a parameter by setting
the “Mode” field to “Add” and clicking “Apply” button. In Screen 2 of Figure 13, the
application developer modifies the first parameter such that the parameter is a string and is
“set by client”.

The procedure-id in the “ID” field is generated automatically by the tool and is
used to discriminate one procedure from another. The procedure-id is included in a
message which the server stub and the client stub generate and is used by the dispatching
procedure created by the tool.

Since a long message is transferred by multiple packets, there may be a case
that the recipient reads the incomplete message and flushes the input buffer before the
remainder of the message arrives. To avoid this happening, the code generated by the
IRPC tool checks if the end of message character is received. If it is not received, the
recipient shows the partial result along with the error message.

The 3-tier Web architecture is a software architecture running on the Web
which is not secure. And the communication line between the client and the application
server is based on the TCP/IP which has security problems [12]. Much work has been
done to provide the secure communication [10][11][13][14][17], and people want to
standardize the way of supporting security on Internet [2][15]. For more information about the Internet security, see [16]. My standard message protocol can generate the protocol on top of the Secure Sockets Layer (SSL V3.0) to provide the secure communication between the application server and the client.

4.2.1 An example of using the IRPC tool

This section shows an example of an application developer who generates the client stub and the server skeleton using the IRPC tool.

The application developer has a routine which (s)he wants to use as a server routine. The routine gets a string id as an input, sets a string list and returns the size of the string list.

```c
/* This routine gets id as input and
   sets an array of artifact set as output and
   returns the number of artifact set */
int get_artf_set(id, artfset)
char *id;
char ***artfset;
{

    int size, i;

    size = get_artfset_size(id);
    *artfset = (char **) calloc(size, sizeof(char *));

    for(i=0; i<size; i++)
        (*artfset)[i] = (char *) strdup(g_artfset(id, i));
    return size;
}
```
The application developer creates a routine using the IRPC tool. As in Figure 14, the name of the routine is the same as the name of the server routine. This routine returns int value and has two parameters. When the client calls this routine, the client is blocked until it receives the response from the server. The ID field is generated by the IRPC tool and used as the procedure identifier in the message. The first parameter is a string set by the client and the second parameter is a string array set by the server. The number of the string array is passed to the client by the return value which is int.

Once the application developer enters a routine into the IRPC tool, (s)he can create the client stub by clicking the "Client Code" in the "View" menu in the main window and clicking the "Generate Client Code" button.
Figure 15. An example client code generated by the IRPC tool

Figure 15 shows an example client stub code generated by the IRPC tool. This code marshals the first parameter which is a string set by the client, and generates the message with version ("IRPC/1.0"), procedure id ("0"), marshaled parameter, and the end of message character (EoT) delimited by space. EoT and space are defined as "0" and "3" respectively by default but the application programmer can change them.

Since the routine (get_artf_set) is called in a blocked way, the client stub calls send_N_rec_bytes() which blocks the client until it receives the response from the server. And it checks the flag to see if the server has processed the routine successfully, gets the string array out of the message from the server, and returns the number of the string array.

To create the server skeleton, the application developer clicks the “Server Code” in the “View” menu in the main window, clicks the “Generate Server Code” button.
Figure 16 shows an example server code generated by the IRPC tool. After the server skeleton receives a message from the client, it checks the version ("IRPC/1.0"), the procedure id ("0"), and calls the appropriate routine which is stub_get_artf_set() in this case. The server skeleton gets the first parameter out of the message from the client, and actually calls the server routine with the parameter.

The server skeleton needs to return the second parameter and the return value to the client. To do this, it marshals the return value and the second parameter, generates the message, and sends it to the client.

4.2.2 Advantages of using the IRPC tool

The biggest advantage of the IRPC tool is that it provides a user interface for the programmer to easily add/modify/delete/list/sort the client stubs and server skeletons.
There is no interface language and no stub compiler intervention. It is very easy to use, helps the programmer develop protocols incrementally, and optimizes the data transfer time by controlling the size of data in one data transfer. For example, an application programmer can merge multiple small data transfers into one bigger data transfer by generating a super procedure on the server which calls the appropriate multiple procedures. And then, the application programmer generates the merged protocol easily by using the IRPC tool. By doing this, the application programmer can minimize the data transfer time.

Another advantage is that the IRPC tool generates simple source code that application programmers can easily understand. Traditionally, only client/server codes call stubs/skeletons and there is no backward direction procedure invocation. Therefore, it is not possible for application programmers to change the contents of stubs/skeletons. But my method gives application programmers flexibility to insert routines into stubs, which call client/server procedures and to change the code generated by the IRPC tool. Application programmers can use their own encryption and authentication algorithms. They can even change the message protocol to add additional fields or delete existing fields in the message format generated by the IRPC tool.

As we see in section 2.4.3 and section 2.4.6, handcrafted protocols tend to be inconsistent. But the IRPC tool generates consistent protocols between stubs and skeletons. The tool does the consistent type checking of parameters and return values of each protocol message.
The IRPC tool generates protocol documentation to automate the documenting process. This also helps programmers and program maintainers understand what is exactly transferred between the server and the client.

The protocol generated by the IRPC tool is also efficient. In the next chapter I describe the evaluation procedure and present the results of my performance experiments.

4.3 Meta Protocol Specification

In this section, I describe the meta protocol specification generated by the IRPC tool. The IRPC protocol is the application-level RPC protocol which works on top of TCP/IP. The main goals of the IRPC protocol are simplicity, efficiency, and flexibility.

The IRPC protocol is intended to be as simple as possible, while meeting other goals. The IRPC tool generates the source code of client stubs and server skeletons which users can understand and modify. Therefore, simplicity is the best approach to ensure a variety of uses. For simplicity, all the messages are transferred between the client and the server as strings. This eliminates the complexity of handling different byte ordering of different machines, and different data type representations of different machines.

4.3.1 Message Formats

There are only two message formats - Request and Response. A client initiates the Request and the server initiates the Response.

- Request Message

  Request = Request-Header SP Request-Argument EoT
Request-Header = Version# SP Proc-ID

Version# = IRPC/1.0

Proc-ID = int

Request-Argument = PARS

PARS = [Parameter] [SP Parameter]*

Parameter = Primitive-Element | Array-Element

Primitive-Element = int | char | string | float | double | boolean

Array-Element = int array | string array | float array | double array |

    boolean array

int = [+ | -](0-9) [0-9]*

cchar = (0-9) | (A-Z) | (a-z) | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / | : | ; | < |

    = | > | ? | @ | \ | | | \ | _ | ` | { | | } | ~ | Space

string = [char]*

float = [+ | -](0-9)[0-9]*.[0-9]* | [+ | -](0-9)[0-9]*.[0-9]*e[+ | -](0-9)[0-9]* |

    [+ | -](0-9)[0-9]*.[0-9]*E[+ | -](0-9)[0-9]*

double = [+ | -](0-9)[0-9]*.[0-9]* | [+ | -](0-9)[0-9]*.[0-9]*e[+ | -](0-9)[0-9]* |

    [+ | -](0-9)[0-9]*.[0-9]*E[+ | -](0-9)[0-9]*

boolean = true | false

int array = SIZE SP Integers

SIZE = int

Integers = [int] | [int SP]*int

string array = SIZE SP Strings
Strings = [string] | [string SP]*string

float array = SIZE SP Floats

Floats = [float] | [float SP]*float

double array = SIZE SP Doubles

Doubles = [double] | [double SP]*double

boolean array = SIZE SP Booleans

Booleans = [boolean] | [boolean SP]*boolean

* Response Message

Response = Response-Header SP Response-Argument EoR

Response-Header = Version# SP Status-Code

Status-Code = 200 | 300 | 400

200 : Success

300 : Bad Request

400 : Server Error

Response-Argument = [Return-Value SP]PARS | Error-Message

Return-Value = Parameter

Error-Message = string

EoT = \0

EoR = \4

SP = \3

Space = " "

56
4.3.2 IRPC Data Types

IRPC protocol supports primitive data types such as int, char, string, float, double, and boolean. It also supports arrays of primitive data types such as int array, string array, float array, double array, and boolean array.

- **int** - int is transferred as a sequence of numeric characters such as “123” and stored as a four byte integer on the client machine and stored as the natural size of integer on the server machine.

- **char** - char is transferred as a single printable character such as “c” and stored as a one byte character on the client machine and stored as the natural size of character on the server machine.

- **string** - string is transferred as a sequence of printable characters such as “string1” and stored as a sequence of character bytes on the client and server machine.

- **float** - float is transferred as a sequence of numeric characters and a dot (“.”) such as “123.45” or a sequence of numeric characters, a dot, and an exponential character (“e” or “E”) such as “1.234e12”. Float is stored as a four byte single precision number on the client machine and the natural size of single precision number on the server machine.

- **double** - double is transferred as a sequence of numeric characters and a dot (“.”) such as “123.45” or a sequence of numeric characters, a dot, and an exponential character (“e” or “E”) such as “1.234e12”. Double is stored
as an eight byte double precision number on the client machine and the natural size of double precision number on the server machine.

- **boolean** - boolean is transferred as a sequence of characters “true” or “false” and stored as a boolean value on the client machine and as an integer on the server machine.

- **int array** - int array is transferred as a sequence of numeric characters which has the size information of the int array, followed by SP and a sequence of integers delimited by SP, where each integer is a sequence of numeric characters. An example of an int array is “3 SP 123 SP 456 SP 789”.

- **string array** - string array is transferred as a sequence of numeric characters which has the size information of the string array, followed by SP and a sequence of strings delimited by SP, where each string is a sequence of printable characters. An example is “3 SP string1 SP string2 SP string3”.

- **float array** - float array is transferred as a sequence of numeric characters which has the size information of the float array, followed by SP and a sequence of floats delimited by SP, where each float is a sequence of numeric characters and a dot (“.”) such as “123.45” or a sequence of numeric characters, a dot, and an exponential character (“e” or “E”) such as “1.234e12”. An example is “3 SP 12.3 SP 45.8e10 SP -7.89E-3”.
• **double array** - double array is transferred as a sequence of numeric characters which has the size information of the double array, followed by SP and a sequence of doubles delimited by SP, where each double is a sequence of numeric characters and a dot (",") such as "123.45" or a sequence of numeric characters, a dot, and an exponential character ("e" or "E") such as "1.234e12". An example is "3 SP 12.3 SP 45.8e10 SP -7.89E-3".

• **boolean array** - boolean array is transferred as a sequence of numeric characters which has the size information of the boolean array, followed by SP and a sequence of booleans delimited by SP, where each boolean is a sequence of characters "true" or "false". An example is "3 SP true SP false SP true".

4.3.3 **An example protocol generated by the IRPC tool**

In section 4.2.1, I showed an example stub/skeleton code which deals with communication details of get_artf_set(). This section shows the message formats for this routine.

• Request Message:

```
IRPC/1.0 SP 0 SP junelee-WINC-1 EoT
```

The request message has the version ("IRPC/1.0"), the procedure id ("0"), and the first parameter of the routine which is a string ("junelee-WINC-1"). These are delimited by SP, and followed by the end-of-transmission character (EoT).
- **Response Message:**

```
IRPC/1.0 SP 200 SP 3 SP 3 SP joolee-ISSU-1 SP horowitz-OPTN-3
SP junelee-AGRE-4 EoR
```

The response message has the version ("IRPC/1.0"), the flag ("200" - meaning success), return value ("3"), the size of the second parameter ("3"), and three strings which are the second parameter of the routine. These are also delimited by SP and followed by the end-of-record character (EoR).

### 4.3.4 Implementation Details

The IRPC tool also follows the client/server architectural style in which the Java client runs on the Web browser. Therefore, the tool is composed of the application server and the client communicating with each other via the IRPC protocol. As in most of the Web applications which follow client/server architectural style, the client provides the user interface and the application server does the calculation, which is reading/writing files, maintaining the data structures, and generating client stub routines in Java and server skeleton routines in C. Currently, the application server is written in C and running on Solaris and Windows NT. The client is written in Java and running on Netscape browsers 3.0 and above on Solaris, Sun OS and Windows 95/98/NT. The client is also running on Internet Explorer 3.0 on Windows 95/98/NT.
5.0 Evaluating the Generated Protocols

In this chapter I will investigate the differences in the efficiency of protocols implemented by RMI, CORBA or IRPC. The efficiency of a protocol implementation is determined by how compact the messages are and how fast the data structure is transformed into a message protocol (marshaling) and how fast the message is transformed into the data structure (unmarshaling). Marshaling is the process of flattening objects or data structures into a serial stream. Unmarshaling is the process of transforming a flattened serial stream back into objects or data structures.

5.1 The Methodology

I generated and ran test programs on four types of data structures, string, int array, string array, and float array, which represent typical primitive data structures in programming languages. The size of each data type was chosen as follows: 512 bytes of string data, an array of 64 integers, an array of 64 strings, and an array of 64 floats. This client program checked the response time between the client and the server, which included the time for marshaling data on the client, the time for sending the message from the client to the server, the time for dispatching the message to the appropriate routine on the server, the time for unmarshaling the message on the server, the time for marshaling data on the server, the time for sending the message from the server to the client, and the time for unmarshaling the message on the client. The server part of the program simply
returned the same data it had received from the client so that it affected the communication time as little as possible.

Here is the Java program that implements this test procedure.

```java
/* Sending and receiving String of 512 bytes */
void String_test() {
    String sending = ""
    String receiving = ""
    for(int i=0; i<51; i++)
        sending += "0123456789"
        sending += "01"
    int init = System.currentTimeMillis();
    send_and_receive(sending, receiving);
    int end = System.currentTimeMillis();
    System.out.println("TIME (String): " + (end-init));
}

/* Sending and receiving int array of 64 elements */
void int_array_test() {
    int[] sending = new int[64];
    int[] receiving = new int[64];
    for(int i=0; i<64; i++)
        sending[i] = i;
    int init = System.currentTimeMillis();
    send_and_receive(sending, receiving);
    int end = System.currentTimeMillis();
    System.out.println("TIME (int array): " + (end-init));
}

/* Sending and receiving String array of 64 elements */
void String_array_test() {
    String sending[] = new String[64];
    String receiving[] = new String[64];
    for(int i=0; i<64; i++)
        sending[i] = "" + i;
    int init = System.currentTimeMillis();
    send_and_receive(sending, receiving);
```
```java
int end = System.currentTimeMillis();
System.out.println("TIME (String array): " + (end-init));
}

/* Sending and receiving float array of 64 elements */
void float_array_test() {

  float[] sending = new float[64];
  float[] receiving = new float[64];

  for(int i=0; i<64; i++)
    sending[i] = (float) i;

  int init = System.currentTimeMillis();
  send_and_receive(sending, receiving);
  int end = System.currentTimeMillis();
  System.out.println("TIME (float array): " + (end-init));
}

I ran the above routines using RMI, CORBA/IIOP, and IRPC to implement "send_and_receive(sending, receiving)". I compare the results in the following subsections. The server program was running on sunset.usc.edu which is an ultra sparc 1 with 128 Mbytes of memory. The client program was running on mugu.usc.edu which is an ultra sparc 5 with 128 Mbytes of memory. The client machine is connected to the server machine via 10Mbps LAN. Another client machine is a 120 MHz PC with 16 Mbytes of memory which is connected to the Internet via 33.6 Kbps modem. I also checked the "start up" time which includes time for downloading Java class files and time for loading Java Applet on the browser.

I generated three test programs for testing RMI, CORBA/IIOP, IRPC protocols respectively. Each of those three programs ran the test on four data types 100 times. I ran each test program 10 times on the fast LAN and 5 times on the slow network. I selected six resulting sets based on lowest standard deviation. These resulting sets include the results
of the RMI protocol on the fast LAN and on the slow network, the results of the CORBA/IIOP protocol on the fast LAN and on the slow network, and the results of the IRPC protocol on the fast LAN and on the slow network.

I used the RMI protocol from Sun Microsystem’s jdk 1.1.2 [7], CORBA/IIOP protocol from Inprise’s Visibroker for Java and Visibroker for C++ version 3.2 [30]. I used Netscape 4.5 as the browser on the client side.

5.2 The Boxplot

![Boxplot of time length](image)

I used boxplots to visualize the distribution of time length [29]. As in Figure 17, a boxplot consists of a box with a horizontal line through it, and two vertical lines extending from the upper and lower boundaries of the box. The line through the box marks the location of the sample median relative to the vertical axis of the plot. The lower edge of the box marks the location of the first quartile $q_{25}$ and the upper edge marks the third
quartile $q_{.75}$. The box contains 50 percent of the observations. The width of the box contains no relevant information.

The line extending down from the box terminates at the data point which is closest to, but larger than

$$L = q_{.75} - 1.5(q_{.75} - q_{.25})$$

data points with values smaller than $L$ are called lower outer values. Similarly, the line extending up from the box terminates at the data point being the closest to, but smaller than

$$U = q_{.75} + 1.5(q_{.75} - q_{.25})$$

data points with values larger than $U$ are called upper outer values.

The boxplot in Figure 17 has lower outer values but no upper outer values. This leads to the impression that the data is skewed toward the smaller values of length.

### 5.3 Response Time Analysis

In this subsection, I decompose the response time into the communication time and the marshaling/unmarshaling time. I also suggest a way to approximate the communication time and the marshaling/unmarshaling time out of the test result. This response time analysis will help explain the overhead of communication and the overhead of marshaling/unmarshaling of the RMI, CORBA/IIOP, and IRPC protocols.

*Response time* is the period of time from the time that a client issues a request to the time that the client receives the response and is ready to process a new command. The response time includes the time for marshaling the client message, the time for
sending the client message to the application server, the time for dispatching the message to the appropriate application server routine, the time for unmarshaling the message on the application server, the time for processing the requested routine on the application server, the time for marshaling the server message, the time for sending the server message to the client, and the time for unmarshaling the server message on the client.

In the example in section 5.1, the application server directly returns the input without processing it. So I consider the time for processing the requested routine to be zero. I divide the response time as the time for communication ($t_c$) and the time for handling messages ($t_m$). The time for communication ($t_c$) includes the time for sending client message to the application server and the time for sending the server message to the client. The time for handling messages ($t_m$) includes the time for marshaling/unmarshaling the message on the client, the time for marshaling/unmarshaling the message on the application server, and the time for dispatching the message to the appropriate application server routine. Then the response time ($t_r$) is

$$t_r = t_c + t_m$$

In the example, I run the same test on the fast speed LAN and on the slow speed network. Let $t_{r1}$ be the response time on the fast LAN, $t_{c1}$ be the communication time on the fast LAN, and $t_{m1}$ be the marshaling/unmarshaling time on client 1 (sparc), then

$$t_{r1} = t_{c1} + t_{m1}$$
Also let $t_{r2}$ be the response time on the slow network, $t_{c2}$ be the communication time on the slow network, and $t_{m2}$ be the marshaling/unmarshaling time on client 2 (pc), then

$$t_{r2} = t_{c2} + t_{m2}$$

There are several major factors that affect the response time:

1. the size of the data to be transferred
2. the size of the problem that is executed on the application server
3. the network bandwidth between the application server and the client
4. the raw performance of the server machine
5. the raw performance of the client machine
6. the workload of the server machine
7. the workload of the client machine
8. the network traffic between the client and the server machines

Factors 1, 2, and 4 are the same on both tests so I ignore those factors. Factors 3 and 8 affect $t_{c1}$ and $t_{c2}$. Factors 5, 6, and 7 affect $t_{m1}$ and $t_{m2}$. I try to minimize the effect of factors 6, 7, and 8 by running both tests under similar environment as possible.
5.4 Statistical Decisions and Tests of Hypotheses of the Response Time

An hypothesis is a statement about the probability law of a random variable. The procedure employed to test an hypothesis consists of obtaining a random sample from the probability distribution about which the hypothesis has been formulated, computing some appropriate statistic, and then rejecting or failing to reject the stated hypothesis [31].

The hypothesis to be tested is called the null hypothesis denoted $H_0$ and associated with $H_0$ is an alternative hypothesis denoted $H_1$. Rejecting $H_0$ is equivalent to not rejecting $H_1$ and vice versa. The mean value of the population is denoted $\mu$ and the mean value of the null hypothesis is denoted $\mu_0$. In this analysis, I ran an experiment to get the mean value of the null hypothesis. Then I set the null hypothesis as follows:

$H_0: \mu = \mu_0$

$H_1: \mu \neq \mu_0$ where $\mu_0$ is the mean value from the result of the experiment

Then I ran another experiment which had a bigger data set than the previous one and I verified the null hypothesis with the result. To verify the null hypothesis, we need to compute the normalized test statistic $Z_0$ which is

$$Z_0 = \frac{(X-bar - \mu_0)}{(\sigma / \sqrt{n})}$$

where $\sigma$ is the standard deviation of the second experimental sample, $n$ is the number of data set of the second experimental sample, $X-bar$ is the mean value of the second experimental sample. I reject the null hypothesis if $|Z_0| > Z_{\alpha/2}$ and fail to reject the null hypothesis otherwise, where $Z_{\alpha/2}$ is 1.96 [31].
**Proposition 1:** there exists a constant $c_c$ which satisfies $t_{c2} = c_c \cdot t_{c1}$

**Justification:** I ran an experiment where the client sent messages to the application server and the application server sent those messages back to the client without doing anything else so that the response time only reflected $t_{c1}$ and $t_{c2}$. I ran the experiment 3 times with various sizes of messages ranging from 200 bytes of message to 1500 bytes of message by the increment of 50 bytes. For each message size, the test program looped for 30 times and generated the average. From the result of the experiment 1, as in table 17 of appendix 9.0, I get the average of $t_{c2}/t_{c1}$, which is 19.38. I set the number as the null hypothesis of average. Therefore, the null hypothesis is

$$H_0: \mu = 19.38$$

According to [31], I verify the null hypothesis with the result of the remaining two tests which is in table 18 and table 19 of appendix 9.0.

$$Z_0 = \frac{(\overline{X}-\mu_0)}{\sigma/\sqrt{n}}\text{, where } \sigma = 1.620305, \; n = 54, \; \overline{X} = 19.758879$$

$$Z_0 = 1.7183085$$

$$|Z_0| < 1.96$$

Therefore we can accept the null hypothesis with confidence coefficient of 95%. This means that the average ratio of $t_{c2}/t_{c1}$ in the population can be considered as 19.38. Therefore, $c_c$ is

$$c_c = 19.38 \quad \text{(EQ 1)}$$
I used string concatenation to represent the approximation of the marshaling/unmarshaling time. The marshaling process is the process of flattening data structures into a byte stream which is the process of adding data elements to the stream multiple times. I used the string type data elements to simplify the test experiment.

**Proposition 2:** There exists a constant \( c_m \) which satisfies \( t_{m2} = c_m \times t_{m1} \)

**Justification:** I ran an experiment where the client processes string concatenation multiple times. I ran the experiment 3 times with various number of concatenation ranging from 300 times of string concatenation to 560 times of string concatenation by the increment of 10 times of string concatenation. For each number of string concatenation, the test program looped for 30 times and generated the average. From the result of the experiment 1, as in table 20 of appendix 9.0, I get the average of \( t_{m2}/t_{m1} \), which is 2.01. I set the number as the null hypothesis of average. Therefore, the null hypothesis is

\[ H_0: \mu = 2.01 \]

According to [31], I verify the null hypothesis with the result of remaining two tests which is in table 21 and table 22 of appendix 9.0.

\[ Z_0 = (\overline{X} - \mu_0) / \left( \sigma / \sqrt{n} \right), \text{ where } \sigma = 0.138849, n = 54, \overline{X} = 1.997298723 \]

\[ Z_0 = -0.67 \]

\[ |Z_0| < 1.96 \]
Therefore we can accept the null hypothesis with confidence coefficient of 95%. This means that the average ratio of \( t_{m2}/t_{m1} \) in the population can be considered as 2.01. Therefore, \( c_m \) is

\[
\frac{c_m}{2.01} \tag{EQ 2}
\]

Let \( c_t \) be the ratio \( t_{c2}/t_{c1} \). Then

\[
t_{c2} = c_c * t_{c1} + c_m * t_{m1} \\
= c_t * t_{c1} + c_t * t_{m1} \\
(c_t - c_m) * t_{m1} = (c_c - c_t) * t_{c1}
\]

Let \( \{(c_c - c_t)/(c_t - c_m)\} \) be \( \alpha \), where \( c_c > c_t \) and \( c_t > c_m \)

\[
\alpha = 0 \text{ where } c_c \leq c_t, \quad \alpha = \text{ where } c_t \leq c_m
\]

\[
t_{m1} = \alpha * t_{c1}, \text{ therefore}
\]

\[
t_{c1} = t_{t1} / (\alpha + 1) \tag{EQ 3} \\
t_{m1} = \alpha * t_{t1} / (\alpha + 1) \tag{EQ 4} \\
t_{c2} = c_c * t_{c1} / (\alpha + 1) \tag{EQ 5} \\
t_{m2} = c_m * \alpha * t_{t1} / (\alpha + 1) \tag{EQ 6}
\]

### 5.5 RMI

Table 6 and table 7 show the result of RMI tests on 10Mbps LAN and 33.6Kbps network respectively. From these tables we can estimate the communication times \((t_{c1}, t_{c2})\) and the marshaling/unmarshaling times \((t_{m1}, t_{m2})\) for four message types.
<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Std. Dev.</th>
<th>Minimum</th>
<th>Median</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>53.29</td>
<td>6.4874</td>
<td>48</td>
<td>50</td>
<td>75</td>
</tr>
<tr>
<td>Int Array</td>
<td>16.22</td>
<td>6.6933</td>
<td>12</td>
<td>14</td>
<td>45</td>
</tr>
<tr>
<td>String Array</td>
<td>53.56</td>
<td>7.5067</td>
<td>48</td>
<td>50</td>
<td>87</td>
</tr>
<tr>
<td>Float Array</td>
<td>15.95</td>
<td>6.2172</td>
<td>13</td>
<td>14</td>
<td>51</td>
</tr>
</tbody>
</table>

Table 6. Statistics of RMI protocol in milliseconds on 10 Mbps LAN

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Std. Dev.</th>
<th>Minimum</th>
<th>Median</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>632.2</td>
<td>28.305</td>
<td>580</td>
<td>640</td>
<td>750</td>
</tr>
<tr>
<td>Int Array</td>
<td>255.3</td>
<td>34.829</td>
<td>210</td>
<td>270</td>
<td>390</td>
</tr>
<tr>
<td>String Array</td>
<td>425.7</td>
<td>71.17</td>
<td>380</td>
<td>440</td>
<td>1040</td>
</tr>
<tr>
<td>Float Array</td>
<td>257.9</td>
<td>59.003</td>
<td>210</td>
<td>270</td>
<td>660</td>
</tr>
</tbody>
</table>

Table 7. Statistics of RMI protocol in milliseconds on 33.6 Kbps network

**string messages**: From table 6 and table 7, we have $c_t = 632.2/53.29 = 11.86$, $\alpha = 0.7635$, $t_{c1} = 30.22$ by (EQ 3), $t_{m1} = 23.07$ by (EQ 4), $t_{c2} = 585.66$ by (EQ 5), $t_{m2} = 46.37$ by (EQ 6).

**int array messages**: From table 6 and table 7, we have $c_t = 255.3/16.22 = 15.74$, $\alpha = 0.2651$, $t_{c1} = 12.82$ by (EQ 3), $t_{m1} = 3.40$ by (EQ 4), $t_{c2} = 248.45$ by (EQ 5), $t_{m2} = 6.83$ by (EQ 6).

**string array messages**: From table 6 and table 7, we have $c_t = 425.7/53.56 = 7.95$, $\alpha = 1.9242$, $t_{c1} = 18.32$ by (EQ 3), $t_{m1} = 35.25$ by (EQ 4), $t_{c2} = 355.04$ by (EQ 5), $t_{m2} = 70.85$ by (EQ 6).
**float array messages:** From table 6 and table 7, we have $c_t = \frac{257.9}{15.95} = 16.17$, $\alpha = 0.2267$, $t_{c1} = 13.00$ by (EQ 3), $t_{m1} = 2.95$ by (EQ 4), $t_{c2} = 251.94$ by (EQ 5), $t_{m2} = 5.93$ by (EQ 6).

table 8 shows the statistics of RMI test result where $t_{r1}/t_{r2}$ is the response time, $t_{c1}/t_{c2}$ is the communication time, $t_{m1}/t_{m2}$ is the marshaling/unmarshaling time on the $LAN/slow$ network.

<table>
<thead>
<tr>
<th></th>
<th>$t_{r1}$</th>
<th>$t_{c1}$</th>
<th>$t_{m1}$</th>
<th>$t_{r2}$</th>
<th>$t_{c2}$</th>
<th>$t_{m2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>53.29</td>
<td>30.22</td>
<td>23.07</td>
<td>632.2</td>
<td>585.66</td>
<td>46.37</td>
</tr>
<tr>
<td>Int Array</td>
<td>16.22</td>
<td>12.82</td>
<td>3.40</td>
<td>255.3</td>
<td>248.45</td>
<td>6.83</td>
</tr>
<tr>
<td>String Array</td>
<td>53.56</td>
<td>18.32</td>
<td>35.25</td>
<td>425.7</td>
<td>355.04</td>
<td>70.85</td>
</tr>
<tr>
<td>Float Array</td>
<td>15.95</td>
<td>13.00</td>
<td>2.95</td>
<td>257.9</td>
<td>251.94</td>
<td>5.93</td>
</tr>
</tbody>
</table>

Table 8. Statistics of RMI test result in milliseconds

As in table 8, marshaling/unmarshaling speed of string or string array is much slower than the marshaling/unmarshaling speed of int array or float array. The communication speed of string or string array is slower than the communication speed of int array or float array. Int array and float array have almost the same communication time and marshaling/unmarshaling time with each other.

These properties can explain Figure 18 and Figure 19. In Figure 18 and Figure 19, the response time of string or string array is bigger than the response time of int array or float array. The ratio of the response time of string array to int array in Figure 18 is more than 3, but the ratio of the response time of string array to int array in Figure 19 is less than 2 because on the slower network, communication time has more effect on the
response time than the marshaling/unmarshaling time. Both pictures show that there are a few upper outer values which are probably from the context switching on client/server machines and/or from the retransmitting of underlying TCP/IP protocol.

Figure 18. Boxplot of RMI protocol in milliseconds on 10 Mbps LAN
5.6 CORBA/IIOP

Table 9 and table 10 show the result of CORBA/IIOP tests on 10Mbps LAN and 33.6Kbps network respectively. From these tables we can estimate the communication times \( (t_{c1}, t_{c2}) \) and the marshaling/unmarshaling times \( (t_{m1}, t_{m2}) \) for four message types.

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Std. Dev.</th>
<th>Minimum</th>
<th>Median</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>15.15</td>
<td>8.7078</td>
<td>12</td>
<td>12</td>
<td>73</td>
</tr>
<tr>
<td>Int Array</td>
<td>14.42</td>
<td>8.0229</td>
<td>11</td>
<td>12</td>
<td>72</td>
</tr>
<tr>
<td>String Array</td>
<td>23.27</td>
<td>14.341</td>
<td>18</td>
<td>18</td>
<td>125</td>
</tr>
<tr>
<td>Float Array</td>
<td>14.69</td>
<td>8.4623</td>
<td>12</td>
<td>12</td>
<td>74</td>
</tr>
</tbody>
</table>

Table 9. Statistics of CORBA/IIOP protocol in milliseconds on 10 Mbps LAN
<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Std. Dev.</th>
<th>Minimum</th>
<th>Median</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>309.4</td>
<td>16.319</td>
<td>260</td>
<td>310</td>
<td>320</td>
</tr>
<tr>
<td>Int Array</td>
<td>230.9</td>
<td>24.867</td>
<td>210</td>
<td>220</td>
<td>330</td>
</tr>
<tr>
<td>String Array</td>
<td>314.5</td>
<td>24.095</td>
<td>270</td>
<td>330</td>
<td>330</td>
</tr>
<tr>
<td>Float Array</td>
<td>229</td>
<td>20.076</td>
<td>220</td>
<td>220</td>
<td>280</td>
</tr>
</tbody>
</table>

Table 10. Statistics of CORBA/IIOP protocol in milliseconds on 33.6 Kbps network

**string messages:** From table 9 and table 10, we have $c_t = 309.4/15.15 = 20.42$, $\alpha = 0.0$, $t_{c1} = 15.15$ by (EQ 3), $t_{m1} = 0.0$ by (EQ 4), $t_{c2} = 309.4$ by (EQ 5), $t_{m2} = 0.0$ by (EQ 6).

**int array messages:** From table 9 and table 10, we have $c_t = 230.9/14.42 = 16.01$, $\alpha = 0.2407$, $t_{c1} = 11.62$ by (EQ 3), $t_{m1} = 2.80$ by (EQ 4), $t_{c2} = 225.20$ by (EQ 5), $t_{m2} = 5.63$ by (EQ 6).

**string array messages:** From table 9 and table 10, we have $c_t = 314.5/23.27 = 13.52$, $\alpha = 0.5091$, $t_{c1} = 15.42$ by (EQ 3), $t_{m1} = 7.85$ by (EQ 4), $t_{c2} = 298.84$ by (EQ 5), $t_{m2} = 15.78$ by (EQ 6).

**float array messages:** From table 9 and table 10, we have $c_t = 229/14.69 = 15.59$, $\alpha = 0.2791$, $t_{c1} = 11.48$ by (EQ 3), $t_{m1} = 3.20$ by (EQ 4), $t_{c2} = 222.48$ by (EQ 5), $t_{m2} = 6.43$ by (EQ 6).
table 11 shows the statistics of CORBA/IIOP test result where $t_{r1}/t_{r2}$ is the response time, $t_{c1}/t_{c2}$ is the communication time, $t_{m1}/t_{m2}$ is the marshaling/unmarshaling time on the \textit{LAN/slow network}.

<table>
<thead>
<tr>
<th></th>
<th>$t_{r1}$</th>
<th>$t_{c1}$</th>
<th>$t_{m1}$</th>
<th>$t_{r2}$</th>
<th>$t_{c2}$</th>
<th>$t_{m2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>15.15</td>
<td>15.15</td>
<td>0.0</td>
<td>309.4</td>
<td>309.40</td>
<td>0.0</td>
</tr>
<tr>
<td>Int Array</td>
<td>14.42</td>
<td>11.62</td>
<td>2.80</td>
<td>230.9</td>
<td>225.20</td>
<td>5.63</td>
</tr>
<tr>
<td>String Array</td>
<td>23.27</td>
<td>15.42</td>
<td>7.85</td>
<td>314.5</td>
<td>298.84</td>
<td>15.78</td>
</tr>
<tr>
<td>Float Array</td>
<td>14.69</td>
<td>11.48</td>
<td>3.20</td>
<td>229</td>
<td>222.48</td>
<td>6.43</td>
</tr>
</tbody>
</table>

Table 11. Statistics of CORBA/IIOP test result in milliseconds

As in table 11, marshaling/unmarshaling speed of string array is more than two times slower than the marshaling/unmarshaling speed of int array or float array. The marshaling/unmarshaling speed of string is the fastest. The communication speed of string or string array is slower than the communication speed of int array or float array. Int array and float array have almost the same communication time and marshaling/unmarshaling time with each other.

These properties can explain Figure 20 and Figure 21. In Figure 20 and Figure 21, the response time of string array is a little bit bigger than the rest. Both pictures show that there are a few upper outer values which are probably from the context switching on client/server machines and/or from the retransmitting of underlying TCP/IP protocol.
Figure 20. Boxplot of CORBA/IiOP protocol in milliseconds on 10 Mbps LAN
5.7 The IRPC protocol

Table 12 and Table 13 show the result of the IRPC protocol tests on 10Mbps LAN and 33.6Kbps network respectively. From these tables we can estimate the communication times ($t_c_1$, $t_c_2$) and the marshaling/unmarshaling times ($t_m_1$, $t_m_2$) for four message types.

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Std. Dev.</th>
<th>Minimum</th>
<th>Median</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>21.58</td>
<td>9.0701</td>
<td>16</td>
<td>18</td>
<td>53</td>
</tr>
</tbody>
</table>

Table 12. Statistics of the IRPC protocol in milliseconds on 10 Mbps LAN
Table 12. Statistics of the IRPC protocol in milliseconds on 10 Mbps LAN

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Std. Dev.</th>
<th>Minimum</th>
<th>Median</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int Array</td>
<td>28.35</td>
<td>9.9233</td>
<td>22</td>
<td>23</td>
<td>50</td>
</tr>
<tr>
<td>String Array</td>
<td>21.51</td>
<td>8.4883</td>
<td>17</td>
<td>18</td>
<td>48</td>
</tr>
<tr>
<td>Float Array</td>
<td>39.87</td>
<td>10.657</td>
<td>33</td>
<td>34</td>
<td>63</td>
</tr>
</tbody>
</table>

Table 13. Statistics of the IRPC in milliseconds on 33.6Kbps network

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Std. Dev.</th>
<th>Minimum</th>
<th>Median</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>292.2</td>
<td>32.709</td>
<td>220</td>
<td>280</td>
<td>380</td>
</tr>
<tr>
<td>Int Array</td>
<td>216.1</td>
<td>36.733</td>
<td>160</td>
<td>220</td>
<td>280</td>
</tr>
<tr>
<td>String Array</td>
<td>211.5</td>
<td>40.636</td>
<td>160</td>
<td>220</td>
<td>280</td>
</tr>
<tr>
<td>Float Array</td>
<td>311.9</td>
<td>38.421</td>
<td>270</td>
<td>330</td>
<td>390</td>
</tr>
</tbody>
</table>

**string messages**: From table 12 and table 13, we have $c_t = 292.26/21.58 = 13.54$, $\alpha = 0.5065$, $t_{cl} = 14.32$ by (EQ 3), $t_{m1} = 7.25$ by (EQ 4), $t_{c2} = 277.52$ by (EQ 5), $t_{m2} = 14.57$ by (EQ 6).

**int array messages**: From table 12 and table 13, we have $c_t = 216.1/28.35 = 7.62$, $\alpha = 2.0963$, $t_{cl} = 9.16$ by (EQ 3), $t_{m1} = 19.20$ by (EQ 4), $t_{c2} = 177.52$ by (EQ 5), $t_{m2} = 38.59$ by (EQ 6).

**string array messages**: From table 12 and table 13, we have $c_t = 211.5/21.51 = 9.83$, $\alpha = 1.2212$, $t_{cl} = 9.68$ by (EQ 3), $t_{m1} = 11.82$ by (EQ 4), $t_{c2} = 187.60$ by (EQ 5), $t_{m2} = 23.76$ by (EQ 6).
**float array messages:** From table 12 and table 13, we have \( c_1 = \frac{311.9}{39.87} = 7.82, \alpha = 1.9897, t_{c1} = 13.34 \text{ by (EQ 3)}, t_{m1} = 26.54 \text{ by (EQ 4)}, t_{c2} = 258.53 \text{ by (EQ 5)}, t_{m2} = 53.35 \text{ by (EQ 6)}.\)

Table 14 shows the statistics of CORBA/IiOP test result where \( t_{r1}/t_{r2} \) is the response time, \( t_{c1}/t_{c2} \) is the communication time, \( t_{m1}/t_{m2} \) is the marshaling/unmarshaling time on the *LAN/slow network.*

<table>
<thead>
<tr>
<th></th>
<th>( t_{r1} )</th>
<th>( t_{c1} )</th>
<th>( t_{m1} )</th>
<th>( t_{r2} )</th>
<th>( t_{c2} )</th>
<th>( t_{m2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>21.58</td>
<td>14.32</td>
<td>7.25</td>
<td>292.2</td>
<td>277.52</td>
<td>14.57</td>
</tr>
<tr>
<td>Int Array</td>
<td>28.35</td>
<td>9.16</td>
<td>19.20</td>
<td>216.1</td>
<td>177.52</td>
<td>38.59</td>
</tr>
<tr>
<td>String Array</td>
<td>21.51</td>
<td>9.68</td>
<td>11.82</td>
<td>211.5</td>
<td>187.60</td>
<td>23.76</td>
</tr>
<tr>
<td>Float Array</td>
<td>39.87</td>
<td>13.34</td>
<td>26.54</td>
<td>311.9</td>
<td>258.53</td>
<td>53.35</td>
</tr>
</tbody>
</table>

Table 14. Statistics of the IRPC protocol test result in milliseconds

As in table 14, marshaling/unmarshaling speed of string or string array is faster than the marshaling/unmarshaling speed of int array or float array because int/float messages need type conversion for marshaling/unmarshaling. The communication speed of string array is almost the same as the communication speed of int array because both messages have the same length in the IRPC protocol. String has the fastest marshaling/unmarshaling time and slowest communication time. Float array has the slowest marshaling/unmarshaling time and second slowest communication time.

These properties can explain Figure 22 and Figure 23. In Figure 22, string has the fast response time because the marshaling/unmarshaling time has more effect on the fast network. But in Figure 23, the response time of string is slow because the
communication time has more effect on the slow network. The ratio of the response time of int array to string array in Figure 22 is about 1.3, but the ratio of the response time of int array to string array in Figure 23 is about 1.0 because on the slower network, communication time has more effect on the response time than the marshaling/unmarshaling time. In Figure 22 and Figure 23, float array shows the worst response time because both the communication time and the marshaling/unmarshaling time is slow. Both pictures show that there are a few upper outer values which are probably from the context switching on client/server machines and/or from the retransmitting of underlying TCP/IP protocol.

Figure 22. Boxplot of the IRPC protocol in milliseconds on 10 Mbps LAN
5.8 Comparison of Protocols on a Fast Network

Table 15 shows the comparison of protocols on the fast LAN.

Start up time: The start up time is the time for downloading Java class files and loading those files on the client machine. The start up time for RMI and the IRPC protocol is fast because only small number of test files are downloaded. But the start up time for CORBA is very slow because the client-side ORB needs to be transferred from the server machine to the client machine along with test files.
<table>
<thead>
<tr>
<th></th>
<th>Start up</th>
<th>String</th>
<th>Int Array</th>
<th>String Array</th>
<th>Float Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>2830</td>
<td>53.29</td>
<td>16.22</td>
<td>53.56</td>
<td>15.95</td>
</tr>
<tr>
<td>CORBA/TIOP</td>
<td>19540</td>
<td>15.15</td>
<td>14.42</td>
<td>23.2</td>
<td>14.69</td>
</tr>
<tr>
<td>IRPC</td>
<td>1720</td>
<td>21.58</td>
<td>28.35</td>
<td>21.51</td>
<td>39.87</td>
</tr>
</tbody>
</table>

Table 15. Comparison of Protocols in milliseconds on 10 Mbps LAN

**String**: CORBA shows the best response time because of the fastest marshaling/unmarshaling time (0 msec). The IRPC protocol has the fastest communication time (14.32 msec) but the overhead of the marshaling/unmarshaling (7.25 msec) is higher than that of CORBA. RMI shows the worst result in both the communication time (30.22 msec) and the marshaling/unmarshaling time (23.07 msec).

**Int array**: CORBA and RMI show the fast response time similar to each other. These two protocols show the similar communication times (11.62 msec for CORBA and 12.82 msec for RMI) and the similar marshaling/unmarshaling times (2.8 msec for CORBA and 3.4 msec for RMI). The IRPC protocol shows the fastest communication time (9.16 msec) but shows the slowest marshaling/unmarshaling time (19.2 msec) because of the type conversion between the string data type and integer data type.

**String array**: The IRPC protocol shows the best response time because of the fastest communication time (9.68 msec). CORBA shows the similar response time. RMI shows the worst response time because of the worst marshaling/unmarshaling time (35.25 msec).

**Float array**: CORBA and RMI show the fast response time similar to each other. These two protocols show the similar communication times (11.48 msec for
CORBA and 13.0 msec for RMI) and the similar marshaling/unmarshaling times (3.2 msec for CORBA and 2.95 msec for RMI). The IRPC protocol shows the worst marshaling/unmarshaling time (26.54 msec) because of the type conversion between the string data type and float data type.

5.9 Comparison of Protocols on a Slow Network

Table 16 shows the comparison of protocols on the slow network.

**Start up time**: The start up time for CORBA is very slow. This is because of downloading the client-side ORB on the slow network. RMI and the IRPC protocol show acceptable start up times.

**String**: The IRPC protocol shows the best response time because of the fastest communication time (277.52 msec). CORBA shows the fastest marshaling/unmarshaling time (0 msec). But the communication time (309.4 msec) is slower than the communication time for the IRPC protocol, and this communication time has more effect on the response time on the slow network. RMI shows the worst communication time (585.66 msec) and the worst marshaling/unmarshaling time (46.37 msec).

**Int array**: The IRPC protocol shows the best response time because of the fastest communication time (177.52 msec). CORBA and RMI have faster marshaling/unmarshaling times (5.63 msec for CORBA and 6.83 msec for RMI) than the IRPC protocol (38.59 msec) but the communication times (225.2 msec for CORBA and 248.45 msec for RMI) are slower than the IRPC protocol.
**String array:** The IRPC protocol shows the best response time because of the fastest communication time (187.6 msec). CORBA shows the fastest marshaling/unmarshaling time (15.78 msec). But the communication time (298.84 msec) is slower than the communication time for the IRPC protocol, and this communication time has more effect on the response time on the slow network. RMI shows the worst communication time (355.04 msec) and the worst marshaling/unmarshaling time (70.85 msec).

**Float array:** CORBA and RMI show the fast response time similar to each other. These two protocols show the similar communication times (222.48 msec for CORBA and 251.94 msec for RMI) and the similar marshaling/unmarshaling times (6.43 msec for CORBA and 5.93 msec for RMI). The communication time for the IRPC protocol (258.53 msec) is similar to the communication time for RMI. But the IRPC protocol shows the worst response time because of the worst marshaling/unmarshaling time (53.35 msec).

<table>
<thead>
<tr>
<th></th>
<th>Start up</th>
<th>String</th>
<th>int array</th>
<th>String array</th>
<th>float array</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>15650</td>
<td>632.2</td>
<td>255.3</td>
<td>425.</td>
<td>257.9</td>
</tr>
<tr>
<td>CORBA/IIOP</td>
<td>171290</td>
<td>309.4</td>
<td>238.6</td>
<td>322.2</td>
<td>237.7</td>
</tr>
<tr>
<td>IRPC</td>
<td>14590</td>
<td>292.2</td>
<td>216.1</td>
<td>211.5</td>
<td>311.9</td>
</tr>
</tbody>
</table>

Table 16. Comparison of Protocols in milliseconds on 33.6 Kbps network
5.10 Overall Conclusions of Protocol Efficiencies

CORBA/IIOP shows the best results on the fast LAN environment except for the long delay of start up time. But on a slow network, the start up time delay is too big to be ignored. Therefore, on a slow network, it is not good to use CORBA/IIOP between tier 1 and tier 2. On a fast network, it is good to use CORBA/IIOP between tier 1 and tier 2.

RMI shows good results for handling int array and float array on the fast LAN. But RMI shows worst results for handling strings and string arrays on both the fast LAN and the slow network. RMI also shows bad results for handling all data types on the slow network. In this experiment, I used the application server written in Java to directly use RMI to communicate with the client. But if we want to use application servers written in other languages such as C or C++, we need to use a wrapper which connects Java code and C/C++ code together. This wrapper can use the Java Native Interface (JNI) but this will introduce more delay. Therefore, it is good to use RMI only when the application server is written in Java and the client and the server are connected with each other via the fast network and the data types transferred between the client and the server are mostly composed of integers and floating point numbers.

The IRPC protocol shows worse results than CORBA/IIOP on the fast LAN but shows the best results on the slow network except for handling float array. The IRPC protocol also shows the smallest start up time which implies that the IRPC protocol is the lightest of all. Therefore, it is good to use the IRPC protocol on the slow network and also good to use the IRPC protocol for handling strings and string arrays on the fast network.
6.0 Creating an Application Server and Client from a Legacy Program

We can take advantage of Web properties such as platform independence, wide area access, instant publication of the software and so on by transforming legacy software systems to the Web. But there is no straightforward way to transform legacy software systems to the Web. In this chapter, I will introduce basic ideas of migrating legacy software systems to the 3-tier Web architecture and I will elaborate the process of generating protocols between the client (tier 1) and the application server (tier 2) by using the IRPC tool.

6.1 The Basic Ideas

When converting legacy code, there are many challenges. The language may no longer have a processor (e.g. COBOL, ADA). The database may be out of date (e.g. Dbase, Paradox). The software may assume a lot about the underlying operating system. In this chapter I ignore all of these problems and focus on a single, but crucial concern, namely the process of splitting the legacy code into an application server and the client. Figure 24 shows the main functional components and whether they are placed in the application server or the client. The first step of splitting legacy code is to understand the legacy code. This step is sometimes very hard but is necessary.

One assumption I make is that the code has a user interface which gets events from the user. A user interface can be built in many ways. It may be an ASCII interface (i.e. not-graphical). The code may write directly to the screen, and receive keyboard
events. But my primary focus is on legacy programs which have a graphical user interfaces (GUI). GUIs are created using a graphical toolkit such as Motif/X-Windows (UNIX), MFC (Windows), or AWT (JAVA). In the case of a GUI, it is common to have a main event loop which waits for the user's event and sends the event to the appropriate routine, user interface management routines which create and manage the user interface, and event handling routines which receive the event from the main loop and do the appropriate operation.

Figure 24. Splitting legacy code

Figure 24 shows the relationship between the legacy code and the Web software which is composed of an application server and client. In the middle is the legacy code which has a GUI. We split the legacy code to generate an application server and client which are in the left and right of the legacy code. Both the application server and the client have binding routines. These two routines work together to establish a connection.
between the application server and the client. The server skeleton on the application server and the client stub on the client are in charge of marshaling/unmarshaling messages. These two routines are generated by the IRPC tool and hides the communication specific details from Web software developers. The dispatching routine of the application server waits for the message from the client and sends the message to the appropriate server skeleton routine. The main loop routine, the user interface management routine, and the event handling routine do the same job as those routines in legacy code. But we need to translate the user interface code into Java if the legacy code was not written in Java.

The difficulty of legacy code transformation to the Web largely depends on the language in which the legacy code was written. Currently the IRPC tool generates server-side skeletons written in C and client-side stubs written in Java. Therefore, the IRPC tool supports the transformation of legacy programs written in C to the Web. But the IRPC tool could support more languages in the future. The IRPC tool can be extended to support all the languages which have socket libraries.

6.1.1 Generating the Application Server

The application server is directly generated from the legacy program with some manual modifications and tool support. The first step for generating the application server code is to get rid of the user interface code from the legacy code. The application server reacts on the request of the client instead of reacting on the request of the user event. Therefore, the application server doesn’t need the user interface capability but needs the communication capability with the client. If the software is designed well, the
user interface part is easily detachable from the rest of the software. If the software is not well designed, it may be very hard to detach the user interface part from the legacy code, but this process is necessary.

After getting rid of the user interface code, we can directly use other parts of the legacy code such as data structure creation/management, data processing, calculation, input/output, communication with other programs and so on. The application server can run in the same environment as that of legacy code. It doesn’t matter if legacy code is running standalone or running cooperatively with other programs.

We need to generate routines for the communication with the client, such as the dispatching routine, the binding routine and the server skeleton. The dispatching routine and the server skeleton routine are generated by the IRPC tool. These two routines can be compiled and linked with the rest of the legacy code. There are multiple ways to implement the binding mechanism. I presented an example of binding mechanism in appendix 10.0 and I will discuss it in the next section.

6.1.2 Generating the Client

If the user interface of legacy code is written in Java, we can directly use it for the client. But in most cases, we need to re-create windows corresponding to the original user interface. After building the code to generate the user interface, we also need to generate event handling routines to deal with the user events, which is similar to the original event handling routines because the event handling routines in the client calls the same functions as the original event handling routines.
We need to generate a data structure for the client to hold application data, which follows the data structure of the original code, and generate a client data structure to manage the status of the client. And the application developer also needs to determine the policy of data receiving from the application server. The client can receive bulk data in advance or it can receive data only when needed.

6.2 An Example of Building Web Applications Using the IRPC Tool

In this section, I will show the procedures of building Web applications using the IRPC protocol. Figure 25 shows the sample C code that I will transform to the Web. The function `displayString()` receives an integer index and returns a string according to the index. We have this function after we got rid of the user interface including the main loop.

```
#include <stdio.h>

char *str[] = {"selected",
               "<row 1,",
               "<row 2,",
               "<row 3,",
               "column 1> ",
               "column 2> ",
               "column 3> "};

char *displayString(index)
int index;
{
    return str[index];
}
```

Figure 25. Sample C Code (sample.c)
6.2.1 Generating Binding Routines

Both the client and the application server have binding routines which establish a connection between the client and the application server. I presented an example of the binding mechanism in appendix 10.0. The server-side binding routines are in `vmain.c` and the client-side binding routines are in `winsoc.java` and `ConThread.java` in appendix 10.0.

The example binding mechanism follows the binding mechanism of the 3-tier Web architecture. Here are steps of the binding mechanism after the client is downloaded from the Web server and is loaded on the virtual machine:

Step 1. The client requests the Web server the port number of the application server. In this example, I used Web server as a name server.

Step 2. The Web server creates an application server process and the application server creates the port and saves the port number in a file.

Step 3. The Web server sends the file to the client.

Step 4. The client requests a connection to the application server.

Step 5. The application server accepts the request and the connection is established.

Application servers can be created before clients request or can be created upon requests from clients. In the first case, the name server can keep the information about application servers such as port numbers and give them to clients when requests come from clients. But this mechanism increases the complexity because the name server should keep track of the information of all the application servers. RMI uses this mechanism in which the name server (rmiregistry) keeps track of the remote references of
all the application servers. CORBA also uses this mechanism where the osagent works as a name server.

I used the second mechanism where application servers are created upon requests from clients. This simplifies the name server mechanism because the name server only needs to create the application server. But in this case, the name server can not send the port number of the application server to the client. In my example, the application server writes its port number to a file and the Web server sends it to the client.
6.2.2 Generating the Protocol Using the IRPC Tool

To generate the protocol, we need to enter the information about the sample routine (*displayStringO*). As in Figure 26, *displayStringO* receives an integer as a parameter, and returns a string.

Figure 26. The Sample Routine in the IRPC Tool

After we enter the information about the routine, the IRPC tool generates the server-side skeleton (*ServerSkel.c*) and the client-side stub (*ClientStub.java*).
6.2.3 Generating the User Interface

Figure 27 shows the sample user interface written in Java. When a user clicks on a button, the appropriate row column pair is displayed in the text field. For example, if the user clicks on the button numbered 8, 

"<row3,column2> selected" is displayed.

![Sample User Interface]

Figure 27. Sample User Interface

User interface code is also in charge of event handling which receives the user's events and handles them. For example, when a user clicks on the button numbered 8, the event handling routine calls the legacy routine three times to generate the string 

"<row3,column2> selected".

```java
...
else if (label.equals("8")) {
    result = ClientStub.displayString(3) + "<row 3,"
    ClientStub.displayString(5) + "column 2> 
    ClientStub.displayString(0); //"selected"
    tf.setText(result); //"<row 3, column 2> selected"
}
...
```

Figure 28. Sample Event Handling Routine
6.2.4 Deploying Files

In appendix 10.0, there are four types of files for generating the sample Web application - files for generating the application server, files for generating the client, an HTML file and a script file for activating the application server.

The application server is composed of three files - main.c, sample.c, and ServerSkel.c. main.c is the main routine which has the main loop. The main loop is waiting for a message from the client, and sends the message to the dispatching routine. main.c also takes care of the server-side binding mechanism. Sample.c has the legacy routine (displayStringO). ServerSkel.c is in charge of marshaling and unmarshaling of messages and dispatching the incoming messages to the appropriate routines. These three files are compiled and linked together to form the application server.

The client is composed of seven files - IRPCTest.java, winsoc.java, ConThread.java, TestMain.java, ErrorMsg.java, Asep.java, and ClientStub.java. IRPCTest.java is the applet file which is in charge of initializing, starting, stopping and running the client. Winsoc.java and ConThread.java is in charge of client-side binding mechanism. ClientStub.java takes care of marshaling and unmarshaling of messages. TestMain.java creates and manages the user interface. ErrorMsg.java and Asep.java are optional. They implement the error message window.

IRPCTest.html contains the applet class and provides parameters for the applet running. This file should be deployed in a place which is accessible by Web browsers.

There are many possible ways that a name server activates the application server upon the request of the client. Usually the name server receives a connection
request from a client, and activates an application server by choosing one in the server pool or by creating one. But in this example, I didn’t use separate name server. I used the Web server as the name server. The client requests a connection to the Web server and the Web server invokes the CGI script which is irpcexam. Then irpcexam invokes the application server. Therefore irpcexam should be deployed in the same directory as specified in parameters of the IRPCTest.html file. For example, if the html script is

```html
<APPLET CODE="IRPCTest.class" WIDTH=400 HEIGHT=100>
  <PARAM NAME=HOST_NAME VALUE="sunset.usc.edu">
  <PARAM NAME=HOST_PORT VALUE="8080">
  <PARAM NAME=SCRIPT VALUE="../cgi-bin/irpcexam">
</APPLET>
```

then the script irpcexam should be deployed to be accessed by the url of http://sunset.usc.edu:8080/../cgi-bin/irpcexam.

### 6.2.5 Revising the Protocol Using the IRPC Tool

The IRPC tool helps the programmer develop protocols incrementally, and optimizes the data transfer time by controlling the size of data transfers into one bigger data transfer. As in Figure 28, the event handling routine calls the remote procedure three times on one button click. And each call results in small number of data transfer. We can improve the response time by merging three small data transfers into one bigger data transfer.
Figure 29 shows the revised sample.c file which has a new function 

\textit{displaySelection()} which calls the legacy function \textit{(displayString())} three times.

\begin{verbatim}
#include <stdio.h>

char *str[] = {"selected",
               "<row 1,",
               "<row 2,",
               "<row 3,",
               "column 1> ",
               "column 2> ",
               "column 3> "};

char result[100];

char *displayString(index)
int index;
{
    return str[index];
}

char *displaySelection(ind1, ind2, ind3)
int ind1, ind2, ind3;
{
    strcpy(result, displayString(ind1));
    strcat(result, displayString(ind2));
    strcat(result, displayString(ind3));
    return result;

Figure 29. Revised Sample C Code (sample.c)

\end{verbatim}

After modifying the \textit{sample.c} file, we need to re-run the IRPC tool to generate the protocol implementing the new function. This process is easily done by modifying the procedure name and parameters in the IRPC tool. We also need to modify the event handling routine. As in Figure 30, the event handling routine only calls the remote function once on one button click.
else if (label.equals("8")) {
    tf.setText(ClientStub.displaySelection(3, 5, 0));
}

Figure 30. Revised Event Handling Routing

6.2.6 The Summary

The difficulty of transforming legacy programs into Web applications comes from issues such as how to detach the user interface from the legacy program, how to translate the user interface written in legacy language into Java, and how to generate communication protocols between the application server and the client. I concentrated on the third issue of generating protocols and in this example, I showed a way of generating protocols between the application server and the client incrementally so that the resulting 3-tier architecture would be acceptable.

In this example, the application server supports only one client. The application server can be extended to support multiple clients. One way to do this is that the application server have multiple sockets and each socket supports one client. We can still use the IRPC tool for generating the protocol. We can even use the same binding routines to connect the application server with multiple clients. But this causes other issues such as how many clients the application server can support without degrading performance, how to cluster application servers to produce best performance, how to keep
track of the state of each client on the application server. These are current issues being researched, and these are beyond the scope of this thesis.
7.0 Conclusions

For obvious reasons many applications are being ported to the Web. In this thesis, I researched on three topics about Web applications - Web software architectures, application-level protocols between application servers and clients, and legacy code transformation into the Web. The following list concludes this research by summarizing the key contributions.

- Presenting the 3-tier Web architecture for Web-based software

  I focused on architectural styles that are suited to applications on the World Wide Web. I presented a style which I call the 3-tier Web architecture. This architecture calls for splitting an application into two parts and dividing the computation between the client and the server. It graphically points out areas where optimizations can be made to avoid Web peculiarities, such as low bandwidth connections and weak client machines.

- Abstract description of a 3-tier Web architecture

  I presented an abstract description of a 3-tier Web architecture in section 2.5. In this abstract description, I represented the entities of 3-tier Web applications, attributes of those entities, and the semantics of the architecture. I used graphical notation to show connectivity and communication and a language similar to Java to define semantics. This description helps computer scientists easily understand the architecture. The description also permits simulation of actions of entities.
• Comparing existing protocols for using as an application-level protocol

After studying my architecture, I identified the generation of protocols between the client and the application server as a major topic of study. I reviewed existing protocol strategies such as using HTTP, CORBA/IIOP, RMI, or handcrafted protocols. I looked into six possible combinations of the application server, application-level protocol, and client which can be deployed to implement the 3-tier Web architecture but all of them had limitations as I described in section 3.5.

• Designing and building the IRPC tool

Therefore, I designed and built a tool, IRPC, which assists the programmer in defining his application-level protocols which avoid those limitations mentioned above. The tool's main leverage is that the programmer can make successive refinements with marginal effort, while maintaining the completeness and consistency, and preserving the efficiency of the protocol.

• Quantifying the performance of application-level protocols

I devised experiments to quantify the performance of application-level protocols. CORBA/IIOP showed the best results on the fast network except for the long delay of start up time. But on the slow network, this start up time delay is too big to be ignored. RMI showed good results for handling integers and floats but it showed the worst results for handling strings. It would have showed even worse results if I had used a non-Java application server and JNI. The IRPC protocol showed comparable results on the fast network and showed the best results on the slow network except for handing
floating point numbers. The IRPC protocol also showed the best start up time delay which implies that the IRPC protocol is the lightest of all.

- Providing a process for transforming Legacy code to the Web

Finally, I provided a process for transforming legacy software to the 3-tier Web applications. This process uses the legacy program as much as possible by splitting legacy software into an application server and client. There are three key issues in the process and I concentrated on one of the key issues, which is generating communication protocols between the application server and the client. I used the IRPC tool to generate protocols and showed an example of deploying the application server and the client on the Web.
8.0 References


[34] J. Spivey. The Z Notation. Prentice Hall. 1992.
### 9.0 Appendix A: Test Results

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>LAN Connection (milliseconds)</th>
<th>Modem Connection (Milliseconds)</th>
<th>Modem/LAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>10.06667</td>
<td>171.3333</td>
<td>17.01986665</td>
</tr>
<tr>
<td>250</td>
<td>9.2</td>
<td>167.3333</td>
<td>18.18840543</td>
</tr>
<tr>
<td>300</td>
<td>8.366677</td>
<td>191.6667</td>
<td>22.90836684</td>
</tr>
<tr>
<td>350</td>
<td>9.466677</td>
<td>192</td>
<td>20.28169007</td>
</tr>
<tr>
<td>400</td>
<td>11.76667</td>
<td>210</td>
<td>17.84702499</td>
</tr>
<tr>
<td>450</td>
<td>10.66667</td>
<td>230.3333</td>
<td>21.59374901</td>
</tr>
<tr>
<td>500</td>
<td>12.53333</td>
<td>238.3333</td>
<td>19.01595769</td>
</tr>
<tr>
<td>550</td>
<td>13.36667</td>
<td>266.3333</td>
<td>19.92518629</td>
</tr>
<tr>
<td>600</td>
<td>12.93333</td>
<td>272.3333</td>
<td>21.05670132</td>
</tr>
<tr>
<td>650</td>
<td>14.96667</td>
<td>296.3333</td>
<td>19.7995539</td>
</tr>
<tr>
<td>700</td>
<td>14.66667</td>
<td>316.3333</td>
<td>21.56818111</td>
</tr>
<tr>
<td>750</td>
<td>16.86667</td>
<td>321.3333</td>
<td>19.05138283</td>
</tr>
<tr>
<td>800</td>
<td>19.06667</td>
<td>337</td>
<td>17.67482487</td>
</tr>
<tr>
<td>850</td>
<td>18.2</td>
<td>348</td>
<td>19.12087912</td>
</tr>
<tr>
<td>900</td>
<td>19.13333</td>
<td>366</td>
<td>19.12892019</td>
</tr>
<tr>
<td>950</td>
<td>19.26667</td>
<td>375</td>
<td>19.46366748</td>
</tr>
<tr>
<td>1000</td>
<td>20.26667</td>
<td>386.3333</td>
<td>19.06249992</td>
</tr>
<tr>
<td>1050</td>
<td>19.53333</td>
<td>410.3333</td>
<td>21.00682613</td>
</tr>
<tr>
<td>1100</td>
<td>21.23333</td>
<td>414.3333</td>
<td>19.51334395</td>
</tr>
<tr>
<td>1150</td>
<td>22.46667</td>
<td>425.6667</td>
<td>18.9465874</td>
</tr>
<tr>
<td>1200</td>
<td>24.6</td>
<td>449.3333</td>
<td>18.26558252</td>
</tr>
<tr>
<td>1250</td>
<td>24.8</td>
<td>468.3333</td>
<td>18.88440847</td>
</tr>
<tr>
<td>1300</td>
<td>25.7</td>
<td>462.6667</td>
<td>18.00259416</td>
</tr>
<tr>
<td>1350</td>
<td>26.03333</td>
<td>476.3333</td>
<td>18.29705516</td>
</tr>
<tr>
<td>1400</td>
<td>26.1</td>
<td>505.6667</td>
<td>19.37420192</td>
</tr>
<tr>
<td>1450</td>
<td>29.2</td>
<td>501.6667</td>
<td>17.18036541</td>
</tr>
<tr>
<td>1500</td>
<td>30.4</td>
<td>637.3333</td>
<td>20.96491217</td>
</tr>
</tbody>
</table>

Table 17. First Experiment of Communication Speed
<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>LAN Connection (milliseconds)</th>
<th>Modem Connection (Milliseconds)</th>
<th>Modem/LAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>9.333333</td>
<td>158.6667</td>
<td>17.00000042</td>
</tr>
<tr>
<td>250</td>
<td>8.066667</td>
<td>182.3333</td>
<td>22.60330528</td>
</tr>
<tr>
<td>300</td>
<td>9.566667</td>
<td>204.3333</td>
<td>21.35888459</td>
</tr>
<tr>
<td>350</td>
<td>11.03333</td>
<td>201</td>
<td>18.21752321</td>
</tr>
<tr>
<td>400</td>
<td>9.933333</td>
<td>221.3333</td>
<td>22.28187893</td>
</tr>
<tr>
<td>450</td>
<td>10.13333</td>
<td>223</td>
<td>22.00657967</td>
</tr>
<tr>
<td>500</td>
<td>13.16667</td>
<td>240.6667</td>
<td>18.2784808</td>
</tr>
<tr>
<td>550</td>
<td>11.76667</td>
<td>254.6667</td>
<td>21.64305916</td>
</tr>
<tr>
<td>600</td>
<td>14.2</td>
<td>283.3333</td>
<td>19.95305141</td>
</tr>
<tr>
<td>650</td>
<td>13.23333</td>
<td>285</td>
<td>21.53652447</td>
</tr>
<tr>
<td>700</td>
<td>16</td>
<td>313.6667</td>
<td>19.60416688</td>
</tr>
<tr>
<td>750</td>
<td>16.8</td>
<td>324</td>
<td>19.28571429</td>
</tr>
<tr>
<td>800</td>
<td>17</td>
<td>331.6667</td>
<td>19.50980412</td>
</tr>
<tr>
<td>850</td>
<td>18.66667</td>
<td>347.3333</td>
<td>18.60714235</td>
</tr>
<tr>
<td>900</td>
<td>18.73333</td>
<td>360.3333</td>
<td>19.23487561</td>
</tr>
<tr>
<td>950</td>
<td>19.1</td>
<td>405.6667</td>
<td>21.23909267</td>
</tr>
<tr>
<td>1000</td>
<td>20.36667</td>
<td>369</td>
<td>18.11783931</td>
</tr>
<tr>
<td>1050</td>
<td>20.93333</td>
<td>408.6667</td>
<td>19.52229346</td>
</tr>
<tr>
<td>1100</td>
<td>21.46667</td>
<td>424</td>
<td>19.7515249</td>
</tr>
<tr>
<td>1150</td>
<td>20.5</td>
<td>427</td>
<td>20.82926829</td>
</tr>
<tr>
<td>1200</td>
<td>24.26667</td>
<td>440</td>
<td>18.13186788</td>
</tr>
<tr>
<td>1250</td>
<td>24.03333</td>
<td>472</td>
<td>19.63939001</td>
</tr>
<tr>
<td>1300</td>
<td>26.26667</td>
<td>501.6667</td>
<td>19.09898466</td>
</tr>
<tr>
<td>1350</td>
<td>25.36667</td>
<td>466.3333</td>
<td>18.38370528</td>
</tr>
<tr>
<td>1400</td>
<td>25.83333</td>
<td>499.3333</td>
<td>19.32903238</td>
</tr>
<tr>
<td>1450</td>
<td>27.83333</td>
<td>504.3333</td>
<td>18.11976058</td>
</tr>
<tr>
<td>1500</td>
<td>30.43333</td>
<td>661</td>
<td>21.71960593</td>
</tr>
</tbody>
</table>

Table 18. Second Experiment of Communication Speed
<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>LAN Connection (milliseconds)</th>
<th>Modem Connection (Milliseconds)</th>
<th>Modem/LAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>9.666667</td>
<td>169</td>
<td>17.4827586</td>
</tr>
<tr>
<td>250</td>
<td>9.3</td>
<td>177.6667</td>
<td>19.103943</td>
</tr>
<tr>
<td>300</td>
<td>8.666667</td>
<td>194.6667</td>
<td>22.4615388</td>
</tr>
<tr>
<td>350</td>
<td>11.33333</td>
<td>191.3333</td>
<td>16.8823531</td>
</tr>
<tr>
<td>400</td>
<td>10.06667</td>
<td>216</td>
<td>21.4569529</td>
</tr>
<tr>
<td>450</td>
<td>11</td>
<td>225.3333</td>
<td>20.4848482</td>
</tr>
<tr>
<td>500</td>
<td>13.43333</td>
<td>281.6667</td>
<td>20.9677427</td>
</tr>
<tr>
<td>550</td>
<td>12.1</td>
<td>284.3333</td>
<td>23.4986223</td>
</tr>
<tr>
<td>600</td>
<td>14.4</td>
<td>284.3333</td>
<td>19.7453701</td>
</tr>
<tr>
<td>650</td>
<td>13.13333</td>
<td>291</td>
<td>22.157361</td>
</tr>
<tr>
<td>700</td>
<td>15.9</td>
<td>319.6667</td>
<td>20.104822</td>
</tr>
<tr>
<td>750</td>
<td>15.86667</td>
<td>316.3333</td>
<td>19.9369742</td>
</tr>
<tr>
<td>800</td>
<td>18</td>
<td>343.6667</td>
<td>19.0925928</td>
</tr>
<tr>
<td>850</td>
<td>18.46667</td>
<td>336.6667</td>
<td>18.2310468</td>
</tr>
<tr>
<td>900</td>
<td>19.63333</td>
<td>415.3333</td>
<td>21.1544993</td>
</tr>
<tr>
<td>950</td>
<td>22.63333</td>
<td>368.6667</td>
<td>16.2886602</td>
</tr>
<tr>
<td>1000</td>
<td>19.16667</td>
<td>380.3333</td>
<td>19.8434777</td>
</tr>
<tr>
<td>1050</td>
<td>21.66667</td>
<td>392</td>
<td>18.0923074</td>
</tr>
<tr>
<td>1100</td>
<td>21.1</td>
<td>435</td>
<td>20.6161137</td>
</tr>
<tr>
<td>1150</td>
<td>20.9</td>
<td>420.3333</td>
<td>20.1116426</td>
</tr>
<tr>
<td>1200</td>
<td>22.73333</td>
<td>476.3333</td>
<td>20.9530793</td>
</tr>
<tr>
<td>1250</td>
<td>23.93333</td>
<td>466.6667</td>
<td>19.4986077</td>
</tr>
<tr>
<td>1300</td>
<td>27</td>
<td>488.6667</td>
<td>18.0987656</td>
</tr>
<tr>
<td>1350</td>
<td>25.63333</td>
<td>477.6667</td>
<td>18.6345907</td>
</tr>
<tr>
<td>1400</td>
<td>26.03333</td>
<td>489</td>
<td>18.783611</td>
</tr>
<tr>
<td>1450</td>
<td>30.23333</td>
<td>520.3333</td>
<td>17.2105844</td>
</tr>
<tr>
<td>1500</td>
<td>29.23333</td>
<td>616.3333</td>
<td>21.0832384</td>
</tr>
</tbody>
</table>

Table 19. Third Experiment of Communication Speed
<table>
<thead>
<tr>
<th>Number of Operations</th>
<th>Marshaling Speed1 on Client 1 (milliseconds)</th>
<th>Marshaling Speed2 on Client 2 (Milliseconds)</th>
<th>Speed2/Speed1</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>100.23</td>
<td>154</td>
<td>1.536466128</td>
</tr>
<tr>
<td>310</td>
<td>91.9</td>
<td>205.33</td>
<td>2.234276387</td>
</tr>
<tr>
<td>320</td>
<td>107.07</td>
<td>203.33</td>
<td>1.899038013</td>
</tr>
<tr>
<td>330</td>
<td>108.8</td>
<td>201.33</td>
<td>1.850459559</td>
</tr>
<tr>
<td>340</td>
<td>123.7</td>
<td>216.33</td>
<td>1.747838733</td>
</tr>
<tr>
<td>350</td>
<td>117.57</td>
<td>251.33</td>
<td>2.137705197</td>
</tr>
<tr>
<td>360</td>
<td>130.67</td>
<td>251.67</td>
<td>1.925996786</td>
</tr>
<tr>
<td>370</td>
<td>134.13</td>
<td>264.33</td>
<td>1.97070067</td>
</tr>
<tr>
<td>380</td>
<td>149.37</td>
<td>266.33</td>
<td>1.783022026</td>
</tr>
<tr>
<td>390</td>
<td>152.4</td>
<td>308</td>
<td>2.020997375</td>
</tr>
<tr>
<td>400</td>
<td>154.43</td>
<td>311.33</td>
<td>2.015994302</td>
</tr>
<tr>
<td>410</td>
<td>172.07</td>
<td>318.33</td>
<td>1.850002906</td>
</tr>
<tr>
<td>420</td>
<td>174.47</td>
<td>356.67</td>
<td>2.044305611</td>
</tr>
<tr>
<td>430</td>
<td>176.97</td>
<td>365</td>
<td>2.062496468</td>
</tr>
<tr>
<td>440</td>
<td>194.67</td>
<td>370</td>
<td>1.900652386</td>
</tr>
<tr>
<td>450</td>
<td>197.13</td>
<td>407.33</td>
<td>2.066301425</td>
</tr>
<tr>
<td>460</td>
<td>214.67</td>
<td>421.67</td>
<td>1.964270741</td>
</tr>
<tr>
<td>470</td>
<td>216.77</td>
<td>459.33</td>
<td>2.118974028</td>
</tr>
<tr>
<td>480</td>
<td>222.03</td>
<td>470</td>
<td>2.116831059</td>
</tr>
<tr>
<td>490</td>
<td>237.6</td>
<td>506</td>
<td>2.12962963</td>
</tr>
<tr>
<td>500</td>
<td>246.83</td>
<td>516.33</td>
<td>2.091844589</td>
</tr>
<tr>
<td>510</td>
<td>257.8</td>
<td>563.67</td>
<td>2.186462374</td>
</tr>
<tr>
<td>520</td>
<td>270.83</td>
<td>557</td>
<td>2.056640697</td>
</tr>
<tr>
<td>530</td>
<td>277.47</td>
<td>610</td>
<td>2.198435867</td>
</tr>
<tr>
<td>540</td>
<td>290.27</td>
<td>608.33</td>
<td>2.09573845</td>
</tr>
<tr>
<td>550</td>
<td>299.43</td>
<td>659.33</td>
<td>2.201950372</td>
</tr>
<tr>
<td>560</td>
<td>315.2</td>
<td>668.67</td>
<td>2.121414975</td>
</tr>
</tbody>
</table>

Table 20. First Experiment of Marshaling Speed
<table>
<thead>
<tr>
<th>Number of Operations</th>
<th>Marshaling Speed1 on Client 1 (milliseconds)</th>
<th>Marshaling Speed2 on Client 2 (milliseconds)</th>
<th>Speed2/Speed1</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>101.4</td>
<td>158.33</td>
<td>1.561439842</td>
</tr>
<tr>
<td>310</td>
<td>94.467</td>
<td>197.33</td>
<td>2.088777597</td>
</tr>
<tr>
<td>320</td>
<td>107.57</td>
<td>199</td>
<td>1.849958167</td>
</tr>
<tr>
<td>330</td>
<td>120.8</td>
<td>210.33</td>
<td>1.741142384</td>
</tr>
<tr>
<td>340</td>
<td>117.2</td>
<td>204.67</td>
<td>1.746331058</td>
</tr>
<tr>
<td>350</td>
<td>130.7</td>
<td>255.33</td>
<td>1.953557766</td>
</tr>
<tr>
<td>360</td>
<td>131.17</td>
<td>261.67</td>
<td>1.994892125</td>
</tr>
<tr>
<td>370</td>
<td>142.67</td>
<td>258.33</td>
<td>1.810681993</td>
</tr>
<tr>
<td>380</td>
<td>146.2</td>
<td>313.33</td>
<td>2.143160055</td>
</tr>
<tr>
<td>390</td>
<td>156.33</td>
<td>264.67</td>
<td>1.693021173</td>
</tr>
<tr>
<td>400</td>
<td>169.6</td>
<td>315</td>
<td>1.857311321</td>
</tr>
<tr>
<td>410</td>
<td>173.13</td>
<td>363</td>
<td>2.096690348</td>
</tr>
<tr>
<td>420</td>
<td>178.2</td>
<td>354.67</td>
<td>1.990291807</td>
</tr>
<tr>
<td>430</td>
<td>194</td>
<td>368.33</td>
<td>1.898608247</td>
</tr>
<tr>
<td>440</td>
<td>201.87</td>
<td>416.67</td>
<td>2.064051122</td>
</tr>
<tr>
<td>450</td>
<td>214.13</td>
<td>421.33</td>
<td>1.967636483</td>
</tr>
<tr>
<td>460</td>
<td>217.87</td>
<td>427</td>
<td>1.959884335</td>
</tr>
<tr>
<td>470</td>
<td>226</td>
<td>460</td>
<td>2.03539823</td>
</tr>
<tr>
<td>480</td>
<td>238.67</td>
<td>481</td>
<td>2.015334981</td>
</tr>
<tr>
<td>490</td>
<td>252.67</td>
<td>499.33</td>
<td>1.976214034</td>
</tr>
<tr>
<td>500</td>
<td>259.1</td>
<td>520.67</td>
<td>2.00932999</td>
</tr>
<tr>
<td>510</td>
<td>278.57</td>
<td>562.33</td>
<td>2.018630865</td>
</tr>
<tr>
<td>520</td>
<td>283.53</td>
<td>561.67</td>
<td>1.980989666</td>
</tr>
<tr>
<td>530</td>
<td>299</td>
<td>613.67</td>
<td>2.052408027</td>
</tr>
<tr>
<td>540</td>
<td>312.87</td>
<td>610.67</td>
<td>1.95183303</td>
</tr>
<tr>
<td>550</td>
<td>319.4</td>
<td>657.33</td>
<td>2.058015028</td>
</tr>
<tr>
<td>560</td>
<td>337.6</td>
<td>717.33</td>
<td>2.124792654</td>
</tr>
</tbody>
</table>

Table 21. Second Experiment of Marshaling Speed
<table>
<thead>
<tr>
<th>Number of Operations</th>
<th>Marshaling Speed 1 on Client 1 (milliseconds)</th>
<th>Marshaling Speed 2 on Client 2 (milliseconds)</th>
<th>Speed2/Speed1</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>89</td>
<td>157.33</td>
<td>1.76775281</td>
</tr>
<tr>
<td>310</td>
<td>103.3</td>
<td>198</td>
<td>1.91674734</td>
</tr>
<tr>
<td>320</td>
<td>107.73</td>
<td>206.33</td>
<td>1.91525109</td>
</tr>
<tr>
<td>330</td>
<td>109.93</td>
<td>202.67</td>
<td>1.84362776</td>
</tr>
<tr>
<td>340</td>
<td>111.03</td>
<td>211</td>
<td>1.90038728</td>
</tr>
<tr>
<td>350</td>
<td>128.03</td>
<td>255</td>
<td>1.99172069</td>
</tr>
<tr>
<td>360</td>
<td>130.27</td>
<td>257.33</td>
<td>1.97535887</td>
</tr>
<tr>
<td>370</td>
<td>132.53</td>
<td>264</td>
<td>1.99200181</td>
</tr>
<tr>
<td>380</td>
<td>136.7</td>
<td>265.33</td>
<td>1.94096562</td>
</tr>
<tr>
<td>390</td>
<td>150</td>
<td>307.67</td>
<td>2.05113333</td>
</tr>
<tr>
<td>400</td>
<td>153.67</td>
<td>315.33</td>
<td>2.05199453</td>
</tr>
<tr>
<td>410</td>
<td>168.77</td>
<td>318.67</td>
<td>1.88819103</td>
</tr>
<tr>
<td>420</td>
<td>165.2</td>
<td>358.33</td>
<td>2.1690678</td>
</tr>
<tr>
<td>430</td>
<td>183.53</td>
<td>373.33</td>
<td>2.03416335</td>
</tr>
<tr>
<td>440</td>
<td>185.6</td>
<td>366.33</td>
<td>1.97376078</td>
</tr>
<tr>
<td>450</td>
<td>196.8</td>
<td>413.33</td>
<td>2.10025407</td>
</tr>
<tr>
<td>460</td>
<td>208.3</td>
<td>425.67</td>
<td>2.04354297</td>
</tr>
<tr>
<td>470</td>
<td>210.1</td>
<td>457.33</td>
<td>2.17672537</td>
</tr>
<tr>
<td>480</td>
<td>225.43</td>
<td>473.67</td>
<td>2.1011844</td>
</tr>
<tr>
<td>490</td>
<td>230.1</td>
<td>507.67</td>
<td>2.20630161</td>
</tr>
<tr>
<td>500</td>
<td>244.8</td>
<td>519.67</td>
<td>2.12283497</td>
</tr>
<tr>
<td>510</td>
<td>253.23</td>
<td>560.67</td>
<td>2.21407416</td>
</tr>
<tr>
<td>520</td>
<td>263.43</td>
<td>569.33</td>
<td>2.1612193</td>
</tr>
<tr>
<td>530</td>
<td>276.37</td>
<td>609.33</td>
<td>2.20476173</td>
</tr>
<tr>
<td>540</td>
<td>288.13</td>
<td>622.33</td>
<td>2.1598931</td>
</tr>
<tr>
<td>550</td>
<td>297.97</td>
<td>660.33</td>
<td>2.21609558</td>
</tr>
<tr>
<td>560</td>
<td>324.67</td>
<td>680</td>
<td>2.09443435</td>
</tr>
</tbody>
</table>

Table 22. Third Experiment of Marshaling Speed
10.0 Appendix B: Sample Program Files

1. vmain.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys-param.h>
#include <sys/time.h>
#include <sys/types.h>
#include <signal.h>
#include <malloc.h>
#include <netdb.h>
#include <assert.h>

extern int errno;

#define DBG 0
#define STR_PORT 6811
#define HOST_NAME 40

#define SEC 1
#define MINUTE * SEC * 60
#define DATELENGTH 25
#define BUFSIZE 8192

int vmain_port;
int vmain_fd;
int stream_fd;

int stream_port;
char fname[100];
char user_name[20];
int sock;
char *get_current_user(), *get_current_dateNtime();
int process_in_str();
void set_start_dir(), restore_start_dir();
int local_str_socket_addr();
char start_dir[MAXPATHLEN+1];

/* Initialize Vmain */
void init_vmain ()
{
    FILE *fi;

    sprintf(fname, "%s\%s", user_name, ".tmp");

    fi = fopen(fname, "w");
    stream_fd = socket (AF_INET, SOCK_STREAM, 0);
    if (stream_fd == -1) {
        exit (-1);
    }
}```
stream_port = local_str_socket_addr (stream_fd);
listen(stream_fd, 5);

fprintf(fi, "%d

u
strearrtportl
f close(fi);

/* main loop for vmain */
void loop_vmain()
{
  int i, j;
  char buf[BUFSIZE];
  int k;
  struct sockaddr_in isa;
  struct timeval timeout;
  fd_set rd_fds;

  k = sizeof isa;
  sock = accept(stream_fd, &isa, &k);
  if (sock < 0)
    {
      fprintf(stderr, "%s %s
accept
W
.getd~~t,u~ero

.exit(-1);
    }
  remove(fname);

  j = 0;
timeout.tv_sec = 30 MINUTE;
timeout.tv_usec = 0;

  while (1)[
    (void) memset(buf, 0, BUFSIZE);

    FD_ZERO (&rd_fds);
    FD_SET (sock, &rd_fds);

    k = select(FD_SETSIZE, &rd_fds, (fd_set *) NULL, (fd_set *) NULL, &timeout);

    if (k > 0)
      {i=read(sock, buf, BUFSIZE);
       if (i > 0)
         process_in_str(buf);
       j = 0;
timeout.tv_sec = 30 MINUTE;
timeout.tv_usec = 0;
     }
    else if (i == 0)
      {
        if (j <= 50)
          j++;
        continue;
      }
    else
      {
        fprintf(stderr, "%s %s
self termination
n.
get_current_user(),
get_current_dateTime());
        close(sock);
        exit (1);
      }
  }
}
else{
  fprintf(stderr, "%s %s
error %d
n.
get_current_user(),
get_current_dateTime(),errno);

115
void main(argc, argv)
int argc;
char **argv;
{
    sprintf(user_name, "%s", argv[1]);
    set_start_dir();
}

void set_start_dir()
{
    getcwd (start_dir, MAXPATHLEN);
    chdir ("./htdocs");
}

void restore_start_dir()
{
    chdir (start_dir);
    chdir ("./htdocs");
}

void write (wfd, str, len)
int wfd;
char *str;
int len;
{
    int k;
    fd_set wr_fds;
    int i;

    FD_ZERO (&wr_fds);
    FD_SET (wfd, &wr_fds);

    k = select(FD_SETSIZE, (fd_set *) NULL, &wr_fds, (fd_set *) NULL,
               (struct timeval *) NULL);

    if (k > 0 ) {
        write(wfd, str, len);
    }
Get the socket address with port from local host */
int local_str_socket_addr (fd)
int fd;
{
    struct sockaddr_in *sin;
    struct hostent *hp;
    char host[HOST_NAME];
    int port, temp;

    if (gethostname(host, HOST_NAME) == -1) {
        fprintf(stderr, "Lib: gethostname
get-current-user()
get-current-dateTime());
        exit(-1);
    }

    sin = (struct sockaddr_in *)malloc (sizeof(struct sockaddr_in));
    if (sin == NULL) {
        fprintf(stderr, "Lib: malloc
exit(-1);
    }
    hp = gethostbyname(host);
    bzero ((char *) sin, sizeof (struct sockaddr_in));
    sin->sin_addr.s_addr = htonl(INADDR_ANY);
    sin->sin_family = hp->h_addrtype;
    assert(sin->sin_family = AF_INET);

    port = STR_PORT;
    do {
        port++;
        sin->sin_port = htons(port);
        temp = bind (fd, sin, sizeof (struct sockaddr_in));
    } while (temp == -1 && errno == EADDRINUSE);
    if (temp == -1) {
        fprintf(stderr, "Lib: bind
get-current-user()
get-current-dateTime(),errno);
        exit (-1);
    }
    return (port);
}

char *get_current_dateTime ()
{
    static char time_str[DATELENGTH];
    struct timeval tp;
    struct tm *tm;
    time(&tp);
    tm = localtime(&tp.tv_sec);
    strftime(time_str, DATELENGTH, "%D %R", tm);

    return time_str;
}

char *get_current_user() { return user_name; }
2. IRPCTest.java

```java
import java.awt.*;
import java.net.*;
import java.io.*;

public class IRPCTest extends java.applet.Applet implements Runnable {

    public Thread runner;
    public static Button button = null;
    public static TextField tf = null;
    boolean click = false;

    public static ErrorMsg errMsg = null;
    public static TestMain testmain = null;
    public static String TitleString = null;
    public static IRPCTest MyApplet = null;
    public static boolean debug = false;
    public static String hostName = null;
    public static String scriptNo = null;
    public static String Username = null;
    public static int hostPort = 0;

    public void init() {
        MyApplet = this;
        if (getParameter("DEBUG").equalsIgnoreCase("TRUE"))
            debug = true;
        hostName = new String(getParameter("HOST_NAME"));
        printf("Host Name: "+ IRPCTest.hostName);
        hostPort = Integer.valueOf(getParameter("HOST_PORT")).intValue();
        printf("Host Port Number: " + IRPCTest.hostPort);
        scriptNo = new String(getParameter("SCRIPT"));
        printf("Script File Name: " + scriptNo);

        add(new Label("UserName:"));
        add(tf = new TextField(20));
        add(button = new Button("Start IRPC Test");
    }

    public void stop() {
    }

    public void start() {
        if (testmain == null)
            testmain = new TestMain();
        if (errMsg == null)
            errMsg = new ErrorMsg(testmain);

        if (runner == null) {
            runner = new Thread(this);
            runner.start();
        }
    }

    public void destroy()
    {
    }

    public synchronized void run() {
```
while ( runner != null ) {
    while ( click == false ) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    click = false;
    ClientStub.soc = new winsoc(Username);
    printf("winsoc created");
    TitleString = "IRPC Example";
    testmain.setTitle(TitleString);
    testmain.pack();
    testmain.show();
}

public synchronized boolean action(Event evt, Object arg) {
    String label = (String)arg;
    if (evt.target instanceof Button) {
        if (label.equals("Start IRPC Test")) {
            Username = tf.getText();
            Username = Username.trim();
            if (Username.length() <= 0) {
                err_msg.showErrorMsg("No valid username");
            } else {
                click = true;
                button.disable();
                tf.disable();
                notify();
            }
        }
    }
    return false;
}

public static void printf(String in) {
    if(IRPCTest.debug)
        System.out.println(in);
}

3. winsoc.java

import java.awt.*;
import java.net.*;
import java.io.*;
import java.lang.*;
import java.lang.*;
import java.applet.*;

public class winsoc {


int myport;
ServerSocket s;
Socket ss;
InputStream is;
OutputStream os;
InetAddress iadr[];

int remote_port;

URL ul;
URLConnection conn = null;
DataInputStream data = null;
String port_file;
String line;
StringBuffer buf = new StringBuffer();

public static int SSize = 0;
public static int RSize = 0;
public static String space = " ";
public static String EOT = "\0";
public static String EOR = "\4";
public static String EoF = "\n";
public static char EoRC = '\4';
public static char EoTC = '\0';
public static char EoFC = ' ';
public static String version = "JRPC/1.0";
public static int MAX.ARR_SIZE = 8196;

long init, end;

public winsoc (String username)
{
    String temp;
    int i = 0;
    temp = new String(IRPCTest.script + "?" + username);
    IRPCTest.printf("Invoking cgi-script "+ temp);

    // call thread to invoke cgi-script
    ConThread t = new ConThread(IRPCTest.host_port,temp,IRPCTest.host_name);
    IRPCTest.printf("conthread started: "+ IRPCTest.host_port +
                   IRPCTest.host_name + temp);
    t.start();

    for (i=1, remote_port = 0; (i < 15) & & (remote_port == 0);i++) {
        try {
            IRPCTest.MyApplet.runner.sleep(500 * i);
        } catch (InterruptedException ire) {
            System.out.println("interrupted Exception OK ");
        }
    }

    try {
        conn = null;
        ul = null;
        data = null;

        ul = new URL("http", IRPCTest.host_name, IRPCTest.host_port,
                      "/" + username + ".tmp");
        conn = ul.openConnection();
        conn.setUseCaches (false);
conn.connect();
data = new DataInputStream(new BufferedInputStream(
    conn.getInputStream()));
while ((line = data.readLine()) != null) {
    buf.append(line + "\n");
}

remote_port = Integer.parseInt(buf.toString().trim());

try {
    System.out.println("Remote Port Number : " + remote_port);
    if (remote_port == 0) {
        System.out.println("Connection Failure!!");
    }
    try {
        // Get connected
        ss = new Socket(IRPCTest.host_name, remote_port);
        IRPCTest.printf("Connection Established");
    } catch (IOException IOe) {
        System.out.println("I/O error occurs when waiting for a connection.");
    }
    try {
        os = ss.getOutputStream();
        is = ss.getInputStream();
        // stop thread
        t.stop();
    } catch (IOException IOe) {
        System.out.println("I/O error occurs when creating I/O stream.");
    }
}

public void WriteSocket(String str) {
    int leng = str.length();
    byte[] buffer = new byte[leng];

    buffer = str.getBytes();
    try {
        os.write(buffer, 0, leng);
    } catch (IOException IOe) {
        System.out.println("I/O Exception at WriteSocket");
    }
}

public String ReadSocket()
{  
    int size = 0, old_size = 0;
    byte[] buffer = new byte[MAX_ARR_SIZE];
    String result;
    String final_res;

    try {
        RSize = 0;
        size = is.read(buffer, 0, MAX_ARR_SIZE);
        if (size == -1) {
            System.out.println("test1 server not responding");
        }
        final_res = new String(buffer, 0, size);

        RSize = size;

        while (buffer[size - 1] != EoRC) {
            size = is.read(buffer, 0, MAX_ARR_SIZE);
            result = new String(buffer, 0, size);
            final_res += result;
        }
        return final_res;
    } catch (IOException IOe) {
        System.out.println("IOException");
        return null;
    }
}

public synchronized String send_N_rec_bytes (String str) {
    int size, i;
    String result;

    WriteSocket(str);
    IRPCTest.printf("sending: " + ' ' + str + '\n');
    result = ReadSocket();

    if (result == null) {
        System.out.println("test2 server not responding");
        System.out.println("Do you want to establish a new connection?" + '\n' +
                "Unsaved data will be lost." + '\n' + ' ' +
                "OK-Reconnect  Cancel-Exit");
        }
    IRPCTest.printf("received: " + ' ' + result + '\n");
    return result;
}

public synchronized void send_bytes (String str) {
    WriteSocket(str);
    IRPCTest.printf("sending: " + ' ' + str + '\n");
    return;
}

public void closeInputStream() throws IOException {
    is.close();
}
public void closeOutputStream() throws IOException {
    os.close();
}

public void closeSocket() throws IOException {
    ss.close();
}

public void Close() {
    try {
        closeInputStream();
        closeOutputStream();
        closeSocket();
    } catch (IOException IOe) {
        System.out.println("IO exception");
    }
}

public InputStream inputStream() {
    return is;
}

public OutputStream OutputStream() {
    return os;
}

4. ConThread.java

import java.net.*;
import java.io.*;
import java.lang.*;
import java.applet.*;

class ConThread extends Thread {
    URL ul;
    URLConnection conn = null;
    String script;
    String host;
    int port;

    ConThread ( int p, String s, String h) {
        script = s;
        host = h;
        port = p;
    }

    public void run() {
        try {
            if ( port == 80 )
                ul = new URL("http://" + host + "/" + script );
            else
                ul = new URL("http://" + host + ":*" + port + "/" + script );
            conn = ul.openConnection();
            conn.setUseCaches (false);
            conn.connect();
            conn.getContent();
        }
5. TestMain.java

import java.awt.*;

public class TestMain extends Frame {
    Panel ButtonArea=null;
    TextField tf=null;
    Button b1=null, b2=null, b3=null;
    Button b4=null, b5=null, b6=null;
    Button b7=null, b8=null, b9=null;

    TestMain() {
        ButtonArea = new Panel();
        tf = new TextField(20);

        ButtonArea.setLayout(new GridLayout(3,3));
        ButtonArea.add(b1 = new Button("1");
        ButtonArea.add(b2 = new Button("2");
        ButtonArea.add(b3 = new Button("3");
        ButtonArea.add(b4 = new Button("4");
        ButtonArea.add(b5 = new Button("5");
        ButtonArea.add(b6 = new Button("6");
        ButtonArea.add(b7 = new Button("7");
        ButtonArea.add(b8 = new Button("8");
        ButtonArea.add(b9 = new Button("9");

        setLayout(new BorderLayout());

        add("Center", ButtonArea);
        add("South", tf);
    }

    public boolean actionEvent(Event evt, Object arg) {
        String label = (String)arg;

        if (evt.target instanceof Button) {
            if (label.equals("1")) {
                tf.setText("1");
                tf.setText(ClientStub.displaySelection(1, 4, 0));
            } else if (label.equals("2")) {
                tf.setText("2");
                tf.setText(ClientStub.displaySelection(1, 5, 0));
            } else if (label.equals("3")) {
                tf.setText("3");
                tf.setText(ClientStub.displaySelection(1, 6, 0));
            } else if (label.equals("4")) {

6. Asep.java

```java
import java.awt.Panel;
import java.awt.Graphics;
import java.awt.Rectangle;
import java.awt.Color;

public class Asep extends Panel {
    public Rectangle rect = new Rectangle(0,0,0);
    private int AsepHeight=7;
    private final int LongEnough=1500;

    Asep() {
    }

    public void paint(Graphics g) {
        rect.width = LongEnough;
        rect.height = AsepHeight;
        setSize(rect.width, rect.height);
        g.setColor(Color.gray);
        g.drawLine(0, 3, rect.width, 3);
        g.setColor(Color.black);
        g.drawLine(0, 4, rect.width, 4);
        g.setColor(Color.white);
        g.drawLine(0, 5, rect.width, 5);
    }
}
```

7. IRPCTest.html

```
<HTML>
```

125
8. irpcexam

```bash
#!/usr/bin/csh -f
/usr/cse01/junelee/IRPC/irpcexam/bin/vmain $*
```

9. ErrorMsg.java

```java
import java.awt.*;

public class ErrorMsg extends Dialog {
    Label msg;
    Button cancel;
    Panel p1 = new Panel();
    int no=0;

    ErrorMsg(Frame parent) {
        super(parent, "Error Message", true);

        Panel pan = new Panel();
        Panel p2 = new Panel();
        Panel p21 = new Panel();

        setLayout(new BorderLayout());
        pan.setLayout(new BorderLayout());
        p1.setLayout(new BorderLayout());
        p2.setLayout(new BorderLayout());

        msg = new Label("*");
        cancel = new Button(" OK ");
        p21.add(cancel);

        p1.add("Center", msg);
        p2.add("North", new AsEp());
        p2.add("South", p21);
        pan.add("Center", p1);
        pan.add("South", p2);
        add("Center", pan);
    }

    public void showErrorMsg(String mess) {
        Label labels[] = new Label[30];
        String msg[] = new String[30];
        int last=0;
        int leng = mess.trim().length();
```
if(leng <= 1) return;

for(int i=0; i<leng; i++) {
    if(mess.charAt(i) == '\n') {
        msgs[no++] = new String(mess.substring(last, i));
        if(i+1 < leng)
            last = i+1;
    }
}
msgs[no++] = new String(mess.substring(last, leng));

for(int i=0; i<no; i++) {
    if (msgs[i] == null) {
        return;
    }    
    if(msgs[i].length() <= 0)
        msgs[i] = "";
    labels[i] = new Label(msgs[i]);
}
pl.removeAll();
pl.setLayout(new GridLayout(no, 1));
for(int i=0; i<no; i++)
    pl.add(labels[i]);

pack();
show();
}

public boolean action(Event evt, Object arg) {
    if ( ((String)arg).equals(" OK ") ) {
        hide();
        return true;
    }
}

public void Set_Message(String ititle, String imsg) {

    this.setTitle(ititle);
    msg.setText(imsg);
}

public void Hide() {
    pl.removeAll();
    pl.setLayout(new GridLayout(1, 1));
    pl.add(new Label("x"));
    no = 0;
    hide();
}

10. sample.c
#include <stdio.h>

char *str[] = {"selected", "<row 1>", "<row 2>", "<row 3>", "column 1>", "column 2>", "column 3>"};

char result[100];

char *displayString(index)
int index;
{
    return str[index];
}

char *displaySelection(ind1, ind2, ind3)
int ind1, ind2, ind3;
{
    strcpy(result, displayString(ind1));
    strcat(result, displayString(ind2));
    strcat(result, displayString(ind3));
    return result;
}

11. ServerSkel.c

#define S_Size(a) (sizeof(a)/sizeof(char*))
#define Arr_Size(a) ((S_Size(a)==1)?((int)(*(a-2))-1)/sizeof(char*)):((S_Size(a)))
#define MAX_ARR_SIZE 8196
#define DEBUG 1

#ifndef OK
#define OK 200
#endif

#ifndef BAD_REQ
#define BAD_REQ 300
#endif

#ifndef UNAUTH
#define UNAUTH 301
#endif

#ifndef FORBIDDEN
#define FORBIDDEN 302
#endif

#ifndef NOT_FOUND
#define NOT_FOUND 303
#endif

#ifndef SRV_ERR
#define SRV_ERR 400
#endif

#ifndef CR
#define CR \015
#endif

#ifndef LF
#define LF \012
#endif

#ifndef SP
#define SP \040
#endif

#ifndef EOT
#define EOT \004
#endif
```c
#define BUF_SIZE 2048
#define clear_results(a,b) {clear_result_func(a,b); a=NULL;}

#define SPC '\3'
#define SPS "\3"

char CRLF[3] = { LF, '\0'};
char *Success_str = "IRPC/1.0 200";
char *version = "IRPC/1.0";

int RtnValue=0;
int PrvValue=0;

extern int sock;
char *TmpRes[10000];
int NTmpRes = 0;
int CurIndex = 0;
char *ERRMSG;
int ERRMSGFLAG = 0;
int error_num;
int err_flag = 0;
void send_error();
void send_success();
int process_in_str();
void process_cmd();
int get_proc_no();

int ERRORMSG(msg)
char *msg;
{
    ERRMSG = (char *)_strdup(msg);
    ERRMSGFLAG = 1;
    return 0;
}

int clear_result_func( results, size )
char** results,
int size;
{
    int i;

    if ( results == NULL || size == 0 ) return 1;

    for (i=0; i<size; i++)
        if (results[i] != NULL) {
            free( results[i] );
            results[i] = NULL;
        }

    free(results);
    results = NULL;

    return 1;
}

int get_proc_no (str)
char *str;
{
```
char *buffer, *buff, *buf, scanstr[300];
int numread, no_proc;

/* 8: # of chars in "IRPC/1.0" */
buffer = (char *) calloc(strlen(str) - 8, sizeof(char));
sprintf(buffer, "IRPC/1.0%c%d%c%s", SPC, SPC);
numread = sscanf(str, scanstr, &no_proc, buffer);
buf = strchr(str+9, SPC);
buff = (char *) (buf+1);
NoTmpRes = gen_str_arr(buffer);
CurIndex = 0;
return no_proc;
}

int gen_str_arr(str)
char *str;
{
    int length = strlen(str);
    int i, inx=0, old_inx=0, no=0, t_inx=0;
    char tmp[MAX_ARR_SIZE];
    for(i=0; i<NoTmpRes; i++)
        if(TmpRes[i])
            free(TmpRes[i]);
    for(i=0; i<length; i++) {
        if( *(str+i) != SPC && *(str+i) != 0)
            tmp[t_inx++] = *(str+i);
        else {
            tmp[t_inx] = 0;
            TmpRes[no++] = strdup(tmp);
            t_inx = 0;
        }
    }
    return no;
}

int process_in_str(in_str)
char *in_str;
{
    int no_proc = get_proc_no(in_str);
    process_cmd(no_proc);
}

void send_success (str)
char *str;
{
    char *buffer;
    int i;
    if(str) {
        buffer = (char *) calloc(strlen(str)+20, sizeof(char));
        sprintf(buffer, "%s%c%d%c%s%c", version, SPC, 200, SPC, str, EOT);
    } else {
        buffer = (char *) calloc(20, sizeof(char));
        sprintf(buffer, "%s%c%d%c", version, SPC, 200, EOT);
    }
    Write (sock, buffer, strlen(buffer));
    free(buffer);
void send_error (errcode, errmsg)
int errcode;
char *errmsg;
{
char *buffer;
buffer = (char *) calloc(strlen(errmsg)+20, sizeof(char));
sprintf (buffer, "%s%c%d%c%s%c", version, SPC, errcode, SPC, errmsg, EOT);
Write (sock, buffer, strlen(buffer));
free(buffer);
}

int get_int()
{
return atoi(TmpRes[CurIndex++]);
}

int get_int_list(iist)
int **iist;
{
int i, size = atoi(TmpRes[CurIndex++]);

*iist = (int *) calloc(size, sizeof(int));
for(i=0; i<size; i++)
(*iist)[i] = atoi(TmpRes[CurIndex++]);
return size;
}

char get_char()
{
return *TmpRes[CurIndex++];
}

char *get_String()
{
return (char *) strdup(TmpRes[CurIndex++]);
}

int get_String_list(strlist)
char ***strlist;
{
int i, size = atoi(TmpRes[CurIndex++]);

*strlist = (char **) calloc(size, sizeof(char *));
for(i=0; i<size; i++)
(*strlist)[i] = (char *) strdup(TmpRes[CurIndex++]);
return size;
}

float get_float()
{
return (float) atof(TmpRes[CurIndex++]);
}

int get_float_list(flist)
float **flist;
{
int i, size = atoi(TmpRes[CurIndex++]);

*flist = (float *) calloc(size, sizeof(float));
}
for(i=0; i<size; i++)
    (*f)[i] = (float) atof(TmpRes[CurIndex++]);
    return size;
}

double get_double()
{
    return (double) atof(TmpRes[CurIndex++]);
}

int get_double_list(dlist)
    double **dlist;
{
    int i, size = atoi(TmpRes[CurIndex++]);

    *dlist = (double *) calloc(size, sizeof(double));
    for(i=0; i<size; i++)
        (*dlist)[i] = (double) atof(TmpRes[CurIndex++]);
        return size;
}

int get_boolean()
{
    int res;

    if(*TmpRes[CurIndex] == 'F' || *TmpRes[CurIndex] == 'f' ||
        *TmpRes[CurIndex] == '0')
        res = 0;
    else
        res = 1;
    CurIndex++;
    return res;
}

int get_boolean_list(blist)
    int **blist;
{
    int i, size = atoi(TmpRes[CurIndex++]);

    *blist = (int *) calloc(size, sizeof(int));
    for(i=0; i<size; i++)
        (*blist)[i] = get_boolean();
        return size;
}

char *marshal_int(ip)
    int ip;
{
    char tmp[20];

    sprintf(tmp, "%d", ip);
    return (char *) strdup(tmp);
}

char *marshal_int_list(ilist, asize)
    int *ilist;
    int asize;
{
    int i, j, size;
    int length=0;
    char *sending, **str, tmp[100];
if (asize==0) size = RtnValue;
else if (asize==1) size = PrvValue;
else return 0;

str = (char **) calloc(size, sizeof(char *));
for (i=0; i<size; i++) {
    sprintf(tmp, "%d", iList[i]);
    str[i] = (char *) strdup(tmp);
    length = length + strlen(str[i]) + 1;
}

sprintf(tmp, "%s%d", ", size);
length = length + strlen(tmp) + 1;

sending = (char *) calloc(length, sizeof(char *));
sprintf(sending, ",", tmp);

for (i=0; i<size; i++)
    sprintf(sending, ",%c%c", sending, SPC, str[i]);

clear_results(str, size);

return sending;
}

char *marshal_char(chr)
char chr;
{
    char tmp[3];

    sprintf(tmp, "%c", chr);
    return (char *) strdup(tmp);
}

char *marshal_String(str)
char *str;
{
    return str;
}

char *marshal_String_list(strlist, asize)
char **strlist;
int asize;
{
    int i, j, size = 3;
    int length=0;
    char *sending, **str, tmp[100];

    if (asize==0) size = RtnValue;
    else if (asize==1) size = PrvValue;
    else return 0;

    str = (char **) calloc(size, sizeof(char *));
    for (i=0; i<size; i++) {
        str[i] = (char *) strdup(strlist[i]);
        length = length + strlen(str[i]) + 1;
    }

    sprintf(tmp, "%s%d", ", size);
    length = length + strlen(tmp) + 1;
sending = (char *) malloc(length, sizeof(char));
sprintf(sending, "%s", tmp);

for(i=0; i<size; i++)
    sprintf(sending, "%s%c%s", sending, SPC, str[i]);

clear_results(str, size);

return sending;
}

cchar *marshal_float.fp
float fp;
{
    char tmp[20];

    sprintf(tmp, "%f", fp);
    return (char *) strdup(tmp);
}

cchar *marshal_float_list(flist, asize)
float *flist;
int asize;
{
    int i, j, size = 3;
    int length=0;
    char *sending, **str, tmp[100];

    if(asize==0) size = RtnValue;
    else if(asize==1) size = PrvValue;
    else return 0;

    str = (char **) calloc(size, sizeof(char *));
    for(i=0; i<size; i++)
    {
        sprintf(tmp, "%f", flist[i]);
        str[i] = (char *) strdup(tmp);
        length = length + strlen(str[i]) + 1;
    }

    sprintf(tmp, "%s%d", "," , size);
    length = length + strlen(tmp) + 1;

    sending = (char *) malloc(length, sizeof(char));
    sprintf(sending, "%s", tmp);

    for(i=0; i<size; i++)
        sprintf(sending, "%s%c%s", sending, SPC, str[i]);

    clear_results(str, size);

    return sending;
}

cchar *marshal_double(dp)
double dp;
{
    char tmp[20];

    sprintf(tmp, "%f", (float) dp);
    return (char *) strdup(tmp);
char *marshal_double_list(dlist, asize)
double *dlist;
int asize;
{
  int i, j, size = 3;
  int length=0;
  char *sending, **str, tmp[100];

  if(asize==0) size = RtnValue;
  else if(asize==1) size = PrvValue;
  else return 0;

  str = (char **) calloc(size, sizeof(char *));
  for(i=0; i<size; i++) {
    sprintf(tmp, "%f", (float) dlist[i]);
    str[i] = (char *) strdup(tmp);
    length = length + strlen(str[i]) + 1;
  }

  sprintf(tmp, "%s%d", "", size);
  length = length + strlen(tmp) + 1;

  sending = (char *) calloc(length, sizeof(char));
  sprintf(sending, "%s", tmp);
  for(i=0; i<size; i++)
    sprintf(sending, "%s%c%s", sending, SPC, str[i]);

  clear_results(str, size);

  return sending;
}

char *marshal_boolean(bp)
int bp;
{
  if(bp)
    return (char *) strdup("true");
  else
    return (char *) strdup("false");
}

char *marshal_boolean_list(blist, asize)
int *blist;
int asize;
{
  int i, j, size = 3;
  int length=0;
  char *sending, **str, tmp[100];

  if(asize==0) size = RtnValue;
  else if(asize==1) size = PrvValue;
  else return 0;

  str = (char **) calloc(size, sizeof(char *));
  for(i=0; i<size; i++) {
    str[i] = marshal_boolean(blist[i]);
    length = length + strlen(str[i]) + 1;
```c
}
sprintf(tmp, "%s%d", ", size);
length = length + strlen(tmp) + 1;

sending = (char *) calloc(length, sizeof(char));
sprintf(sending, "%s", tmp);
for (i=0; i<size; i++)
    sprintf(sending, "%s%c%s", sending, SPC, str[i]);

clear_results(str, size);

return sending;
}

void process_cmd (no_proc)
int no_proc;
{
    switch (no_proc) {
    case 1:
        skel_displaySelection();
        break;
    default:
        break;
    }
}

/* Start of Skeleton Function Definitions */

int skel_displaySelection(){
    int p1;
    int p2;
    int p3;
    char* rtn;
    char* displaySelection();
    int size;
    char *p[BUF_SIZE], *sending;
    err_flag = 0;
    p1 = get_int();
    if (err_flag) return -1;
    p2 = get_int();
    if (err_flag) return -1;
    p3 = get_int();
    if (err_flag) return -1;
    rtn = displaySelection(p1, p2, p3);
    p[0] = marshal_string(rtn);
    if (err_flag) return -1;
    sending = (char *) calloc(strlen(p[0]) + 2, sizeof(char));
    sprintf(sending, "%s", p[0]);
    send_success(sending);
    return 1;
}
import java.lang.*;

public class ClientStub {

    public static int SSize = 0;
    public static int RSize = 0;
    public static String space = "\3";
    public static char spaceC = '\3';
    public static String EoT = "\0";
    public static String EoR = "\004";
    public static String EoF = "\003";
    public static char EoTC = '\0';
    public static char EoRC = '\004';
    public static char EoFC = '\0';
    public static String version = "IRPC/1.0";
    static byte result[];
    static String result_str;
    public static int OK=1;
    public static int CurIndex=0;
    public static winsoc soc;
    static int[] arr_size = new int[BUFSIZE];
    static int arr_index = 0;
    static boolean err_flag = false;

    public static String displaySelection(int p1, int p2, int p3) {
        String res = null;
        arr_index = 0;
        String p[] = new String[BUFSIZE];
        err_flag = false;

        p[0] = marshal_par(p1);
        if(err_flag) return null;
        p[1] = marshal_par(p2);
        if(err_flag) return null;
        p[2] = marshal_par(p3);
        if(err_flag) return null;


        result_str = soc.send_N_rec_bytes (str);
        OK = get_flag (result_str);
        if (OK == -1) return null;

        res = get_string(result_str);
        if(err_flag) return null;
        arr_index = 0;
        return res;
    }

    public static int get_arr_size() {
        return arr_size[arr_index++];
    }
}
public static int get_int(String str)
{
    int old_inx=CurIndex, no=0;
    for (int i=CurIndex; i<str.length(); i++)
    {
        if (str.charAt(i) == ' ') || str.charAt(i) == 'EoRC') {
            CurIndex = i+1;
            return Integer.valueOf(str.substring(old_inx, i)).intValue();
        }
    }
    return -1;
}

public static int get_int_list(String str, int intlist[])
{
    int old_inx=CurIndex, size=0, CurIndexSave=CurIndex;
    for(int i=CurIndex; i<str.length(); i++)
    {
        if (str.charAt(i) == 'EoRC') {
            return -1;
        }
        if (str.charAt(i) == ' '){
            if (i+1 == str.length()) {
                break;
            }
            size = Integer.valueOf(str.substring(old_inx, i)).intValue();
            old_inx = i+1;
            CurIndex = i+1;
        }
    }
    return size;
}

public static char get_char(String str)
{
    int old_inx=CurIndex, no=0;
    for (int i=CurIndex; i<str.length(); i++)
    {
        if (str.charAt(i) == 'EoRC') {
            intlist[no++] = Integer.valueOf(str.substring(old_inx, i)).intValue();
            break;
        }
        if (str.charAt(i) == ' '){
            intlist[no++] = Integer.valueOf(str.substring(old_inx, i)).intValue();
            old_inx = i+1;
            CurIndex = i+1;
        }
    }
    return ' ';
}

public static String get_string(String str)
{
    int old_inx=CurIndex, no=0;
}
for (int i=CurIndex; i<str.length(); i++) {
    if (str.charAt(i) == space || str.charAt(i) == EoRC) {
        CurIndex = i+1;
        return str.substring(old_inx, i);
    }
}
return null;

public static int get_String_list(String str, String strlist[]) {
    int old_inx=CurIndex, size=0, CurIndexSave=CurIndex;
    for(int i=CurIndex; i<str.length(); i++) {
        if (str.charAt(i) == EoRC) {
            return -1;
        }
        if (str.charAt(i) == space) {
            size = Integer.valueOf(str.substring(old_inx, i)).intValue();
            old_inx = i+1;
            CurIndex = i+1;
            break;
        }
    }
    for (int i=CurIndex, no=0; i<str.length() && no<size; i++) {
        if (str.charAt(i) == EoRC) {
            strlist[no++] = str.substring(old_inx, i);
            break;
        }
        if (str.charAt(i) == space) {
            strlist[no++] = str.substring(old_inx, i);
            old_inx = i+1;
            CurIndex = i+1;
        }
    }
    return size;
}

public static float get_float(String str) {
    int old_inx=CurIndex, no=0;
    for (int i=CurIndex; i<str.length(); i++) {
        if (str.charAt(i) == EoRC || str.charAt(i) == EoRC) {
            CurIndex = i+1;
            return Float.valueOf(str.substring(old_inx, i)).floatValue();
        }
    }
    return (float) -1.0;
}

public static int get_float_list(String str, float floatlist[]) {
    int old_inx=CurIndex, size=0, CurIndexSave=CurIndex;
    for(int i=CurIndex; i<str.length(); i++) {
        if (str.charAt(i) == EoRC) {
            return -1;
        }
if (str.charAt(i) == spaceC) {
    size = Integer.valueOf(str.substring(old_inx, i)).intValue();
    old_inx = i+1;
    CurIndex = i+1;
    break;
}
}

for (int i=CurIndex, no=0; i<str.length() && no<size; i++) {
    if (str.charAt(i) == EoRC) {
        floatlist[no++] = Float.valueOf(str.substring(old_inx, i)).floatValue();
        break;
    }
    if (str.charAt(i) == spaceC) {
        floatlist[no++] = Float.valueOf(str.substring(old_inx, i)).floatValue();
        old_inx = i+1;
        CurIndex = i+1;
    }
}
return size;
}

public static double get_double(String str) {
    int old_inx=CurIndex, no=0;
    for (int i=CurIndex; i<str.length(); i++) {
        if (str.charAt(i) == EoRC || str.charAt(i) == EoRC) {
            CurIndex = i+1;
            return Double.valueOf(str.substring(old_inx, i)).doubleValue();
        }
    }
    return -1.0;
}

public static int get_double_list(String str, double dlist[]) {
    int old_inx=CurIndex, size=0, CurIndexSave=CurIndex;
    for (int i=CurIndex; i<str.length(); i++) {
        if (str.charAt(i) == EoRC) {
            return -1;
        }
        if (str.charAt(i) == spaceC) {
            size = Integer.valueOf(str.substring(old_inx, i)).intValue();
            old_inx = i+1;
            CurIndex = i+1;
            break;
        }
    }
    for (int i=CurIndex, no=0; i<str.length() && no<size; i++) {
        if (str.charAt(i) == EoRC) {
            dlist[no++] = Double.valueOf(str.substring(old_inx, i)).doubleValue();
            break;
        }
        if (str.charAt(i) == spaceC) {
            dlist[no++] = Double.valueOf(str.substring(old_inx, i)).doubleValue();
            old_inx = i+1;
            CurIndex = i+1;
        }
    }
}
public static boolean get Boolean(String str)
{
    int old_inx=CurIndex, no=0;
    for (int i=CurIndex; i<str.length(); i++) {
        if(str.charAt(i) == spaceC || str.charAt(i) == EoRC) {
            CurIndex = i+1;
            return Boolean.valueOf(str.substring(old_inx, i)).booleanValue();
        }
    }
    return false;
}

public static int get Boolean_list(String str, boolean blist[])
{
    int old_inx=CurIndex, size=0, CurIndexSave=CurIndex;
    for(int i=CurIndex; i<str.length(); i++) {
        if(str.charAt(i) == EoRC) {
            return -1;
        }
        if(str.charAt(i) == spaceC) {
            size = Integer.valueOf(str.substring(old_inx, i)).intValue();
            old_inx = i+1;
            CurIndex = i+1;
            break;
        }
    }
    for (int i=CurIndex, no=0; i<str.length() & no<size; i++) {
        if(str.charAt(i) == EoRC) {
            blist[no++] = Boolean.valueOf(str.substring(old_inx, i)).booleanValue();
            break;
        }
        if(str.charAt(i) == spaceC) {
            blist[no++] = Boolean.valueOf(str.substring(old_inx, i)).booleanValue();
            old_inx = i+1;
            CurIndex = i+1;
        }
    }
    return size;
}

public static byte[] String_to_byte (String str) {
    int i, len = str.length();
    byte b_array[] = new byte[len];
    for (i=0; i<len; i++) {
        b_array[i] = (byte) str.charAt(i);
    }
    return b_array;
}

public static String byte_to_string (byte b_array[]) {
    String dest = new String (b_array, 0, b_array.length);
    return dest;
}
public static void clear_bytes(byte b_array[], int length) {
    int i;
    for (i=0;i<length;i++)
        b_array[i] = 0;
}

public static int get_flag(String res_str) {
    int old_inx=0, inx=0, tmp_inx;
    int i, status;
    StringErrMsg;
    CurIndex = 0;

    if (res_str.indexOf("IRPC/1.0") == -1) {
        IRPCTest.err_msg.showErrorMsg("Message Format Error: Illegal Version");
        return -1;
    }

    inx += 8; // string length of "IRPC/1.0"
    old_inx = inx;
    if ((inx = res_str.indexOf (space)) == -1) {
        IRPCTest.err_msg.showErrorMsg("Message Format Error: Illegal Message Header");
        return -1;
    }
    if ((tmp_inx = res_str.indexOf (space, inx+1)) == -1 &&
        (tmp_inx = res_str.indexOf (EoR, inx+1)) == -1) {
        IRPCTest.err_msg.showErrorMsg("Message Format Error: Illegal Message Header");
        return -1;
    }
    status = Integer.valueOf(res_str.substring(inx+1,tmp_inx)).intValue();
    CurIndex = tmp_inx+1;
    if (status == 200) //OK
        return tmp_inx+1;
    else {
        old_inx = inx;
        if ((inx = res_str.indexOf (space, tmp_inx)) == -1) {
            IRPCTest.err_msg.showErrorMsg("Message Format Error: Illegal Message Status");
            return -1;
        }
        if ((tmp_inx = res_str.indexOf (EoT, inx+1)) == -1) {
            IRPCTest.err_msg.showErrorMsg("Message Format Error: Illegal Message Ending");
            return -1;
        }
       ErrMsg = new String(res_str.substring(inx+1,tmp_inx));
        if (ErrMsg.trim().length() > 0)
            IRPCTest.err_msg.showErrorMsg(ErrMsg.trim());
        return -1;
    }
}

public static String marshal_par(int par)
{
public static String marshal_par(char par) {
    return "" + par;
}

public static String marshal_par(String par) {
    return par;
}

public static String marshal_par(float par) {
    return "" + par;
}

public static String marshal_par(double par) {
    return "" + par;
}

public static String marshal_par(boolean par) {
    return "" + par;
}

public static String marshal_par(int par[]) {
    int leng = par.length;
    String result = "";
    for(int i=0; i<leng; i++) {
        result = result.concat(space);
        result = result.concat(Integer.toString(par[i]));
    }
    result = leng + result;
    return result;
}

public static String marshal_par(String par[]) {
    int leng = par.length;
    int size = leng;
    String result = "";
    for(int i=0; i<leng; i++) {
        if(par[i] != null) {
            result = result + space + par[i];
        } else {
            size = i;
            break;
        }
    }
    result = size + result;
    return result;
}
public static String marshal_par(float par[]) {
    int leng = par.length;
    String result = "";
    for(int i=0; i<leng; i++) {
        result = result.concat(" ");
        result = result.concat(Float.toString(par[i]));
    }
    result = leng + result;
    return result;
}

public static String marshal_par(double par[]) {
    int leng = par.length;
    String result = "";
    for(int i=0; i<leng; i++) {
        result = result.concat(" ");
        result = result.concat(Double.toString(par[i]));
    }
    result = leng + result;
    return result;
}

public static String marshal_par(boolean par[]) {
    int leng = par.length;
    String result = "";
    for(int i=0; i<leng; i++) {
        result = result + " "+ par[i];
    }
    result = leng + result;
    return result;
}
}