

Comparison of Java Based Client-Server Computing APIs

Sanjay P. Ahuja Jayant Mishra Alex Gonzalez Albert Ritzhaupt Brandon Vega

Department of Computer and Information Sciences
University of North Florida
Jacksonville, FL 32224.
{sahuja, misj0001, rita0001}@unf.edu

ABSTRACT

Client-server computing architecture revolutionized the software field and continues to be widely used today. The advent of the Java programming language made it simpler to code clients and servers by encapsulating much of the programming complexity in the language. This paper evaluates three such Java client-server computing APIs quantitatively and qualitatively – Java Socket API, Java Datagram API and Java RMI API. Thus, this study encompasses TCP sockets, UDP datagrams and the standard RMI facility of Java.

General Terms

Performance, measurement.

Keywords

Sockets, Datagrams, Java Socket API, Java Datagram API, Java RMI.

1.0 INTRODUCTION

The client-server model started gaining acceptance and a wide following in the late 1980s. The client-server architecture is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability and scalability, as compared to centralized, time-sharing computing.

A client is defined as a requestor of services and a server is defined as a provider of services. A single machine can be either a client or a server based on whether it requests services, or provides the same. It can also be both a client and server, like in the case when it provides some services, and in order to do that, it needs to request another machine for some services [1].

A server process fulfills the client request by performing the task requested. It generally receives requests from client programs, executes database retrieval and updates, manages data integrity and dispatches responses to client requests. Sometimes server programs execute common or complex business logic. The server-based process "may" run on another machine on the network. The server process performs the back-end tasks that are common to similar applications. There are two types of servers – iterative and concurrent [2]. The iterative server handles the client's request itself whereas the concurrent server invokes another process to handle each client request.

Thus, the client/server is a computational architecture that involves client processes requesting services from server processes [3].

The focus of this paper is to evaluate and compare qualitatively and quantitatively the performance of three client-server computing APIs – Java Socket API, Java Datagram API and Java RMI API. It compares the underlying technologies behind each of these products, the ease of development and deployment of an application using these products, and their performance using various types of load on identical hardware, operating system and application infrastructure.

The remainder of this paper is organized as follows. Section 2, 3 & 4 discuss Java Socket API, Java Datagram API and Java RMI API respectively. Section 5 describes the project, in particular the application used to test these implementations in detail. Section 6 deals with the studies carried out and results. Section 7 deals with the conclusions by comparing the three technologies quantitatively & qualitatively in light of the results from the previous section.

2.0 JAVA SOCKET API

The original socket API was developed at Berkeley with the 4.1cBSD Unix release [2]. The Unix socket API was complex and involved a series of steps on both the client and server side. The Java Socket API simplified socket programming by encapsulating a lot of functionality in the API to make it easier for programming.

Here is the definition of sockets per the Sun Java Tutorial [4] –

“Sockets are a form of IPC (Inter-process communication) that provide communication between processes on a single system and between processes on different systems.”

The Java Socket API provides support for a TCP-oriented connection. TCP provides a reliable, point-to-point communication channel. Client-server applications on the Internet use TCP to communicate with each

other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection

A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes--Socket and ServerSocket – to implement the client and the server side of the connection.

Normally, a server runs on a specific computer and has a socket bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request. Once the connection is made, the client and server communicate by writing to or reading from their sockets.

The java.net package in the Java platform provides a class, Socket, that implements one side of a two-way connection between the Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from the client Java program. By using the java.net.Socket class instead of relying on native code, Java programs can communicate over the network in a platform-independent fashion. Additionally, java.net includes the ServerSocket class, which implements a socket that servers can use to listen for and accept connections to clients

3.0 JAVA DATAGRAM API

The UDP protocol provides a mode of network communication whereby applications send packets of data, called datagrams, to one another. A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed. The

DatagramPacket and DatagramSocket classes in the java.net package implement system-independent datagram communication using UDP.

Applications that communicate via datagrams send and receive completely independent packets of information. These clients and servers do not have and do not need a dedicated point-to-point channel. The delivery of datagrams to their destinations is not guaranteed. Nor is the order of their arrival.

Here is the definition of a datagram per the Sun Java Tutorial [4] –

“A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.”

4.0 JAVA RMI API

The Java Remote Method Invocation (RMI) system is Java’s native scheme for creating and using remote objects. It allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM [5]. RMI provides for remote communication between programs written in the Java programming language.

The stub/skeleton layer, the remote reference layer and the transport protocol layer comprise the basic remote-object communication facilities in RMI. The RMI API provides classes and methods that handle all of the underlying communication and parameter referencing requirements of accessing remote methods. It also handles the serialization of objects that are passed as arguments to methods of remote objects [5].

There are many advantages of using RMI from an architectural and programmer’s perspective. Since RMI is a Java native, integration of its remote object facilities into a Java application is seamless. It extends the internal garbage collection mechanisms of the standard JVM to provide distributed

garbage collection of remotely exported objects. RMI is also platform-independent (though not language independent). Also, entire objects can be migrated to a remote host via object serialization in RMI.

5.0 PROJECT DESCRIPTION

An application to compare the Client-Server computing APIs discussed above was implemented.

The application developed is very simple in nature. It simulates a FTP server. The server spawns a thread to service each client request. The client can READ and UPDATE data stored at the server. A data file was created on the server machine with real looking data detailing the person’s name and address as a string.

The client application when invoked presents the user the following simple menu:

1. GET
2. SEND
3. LIST
4. CD
5. QUIT

All these options closely mirror the FTP server. The GET option enables the user to get a file from the server to the client. The SEND option is similar to the FTP PUT command, and enables the user to send a file to the server. Both these options require the name of the file on which the GET/SEND operation is to be done. The LIST option is like the FTP DIR command, and enables the user to list the contents of the current directory on the server. The CD option enables the user to change the directory to a specified directory on the server.

Concurrent access from multiple clients was simulated in the project so that issues with concurrent access also can be evaluated. The number of clients using the applications was varied in order to model several users accessing the applications at one time. This was done to more accurately reflect real

world usage. On the server side, the Java synchronization technique was used to manage multiple connections using this tried-and-tested Java language feature.

The application measures the average performance for all the threads and prints it on the standard output. It tracks the time it takes for each request from the time it makes the request from the client machine to the time it receives the requested data from the server.

The following numbers of threads were used in the measurements –

1. 1 thread
2. 5 threads
3. 10 threads
4. 15 threads, and
5. 20 threads

The above measurements were taken for 2 sets – one for a light load (with less than 100 bytes of data), and the other for a heavy load (around 1 MB of data). The graphs from the heavy load measurement are shown in the Results section since they more accurately reflect the comparison as compared to the light load measurement. The configuration for the two machines used in this research is described in Table 1 below.

	Server	Client
Processor	Dual-CPU Pentium III 450MHz	Quad-CPU Hyperthreaded (emulates 8-CPU) Pentium 4 Xeon 1.5GHz
Memory	256 MB	8 GB
Hard Disk	18GB RAID-5 Array	263GB RAID-5 Array
Operating System	Linux SMP	Linux SMP
Software	Sun J2EE, Oracle Client, PHP	Sun J2EE, Oracle Client, PHP
Services	Apache, Tomcat,	Beowulf cluster manager, Apache,

	mySQL	Tomcat, mySQL, Oracle
--	-------	--------------------------

Table 1

6.0 RESULTS

As seen in figures 1 and 2, the Java TCP Sockets API implementation was slower than the UDP Datagram implementation due to the additional overhead associated with the TCP protocol. However, it was much more robust in terms of handling the multiple threads whereas the Datagram API implementation had issues with the same. The performance of the Java Sockets implementation deteriorated faster than that of the Java Datagram implementation as can be seen from the slope of the gradients in figures 1 and 2 for the implementation. The RMI implementation was also much slower than the datagram implementation, though it was close to the Sockets implementation for the SEND requests with more number of threads. The Datagram implementation's performance was better for both GET and SEND requests as the underlying protocol, UDP does not have all the rich set of features which TCP has, but which come with the added cost. That's the price the RMI and Sockets implementations had to pay.

The difference in the GET and SEND request processing times was due to the client performing better than the server due to better resources as shown in Table1.

Table 2 shows a qualitative comparison of the three APIs.

<i>A Qualitative Comparison</i>		
Sockets API	Datagrams	Java RMI
Reliable	Unreliable	Reliable
Connection set-up overhead	No connection set-up overhead	Some connection set-up overhead
Simple API	Moderately simple API	Simple API
Must encode and decode messages	Must encode and decode messages	Encodes and decodes messages
Does not require web server	Does not require web server	Requires a web server
		Tight coupling between client and server via the RMI interface file

Table 2

7.0 CONCLUSIONS

Since all the three client-server computing APIs were compared using the same strategy, most of the code written for one implementation was used for the others. This included the client interactions and the server side processing. The only part modified from implementation-to-implementation was the one where the computing API-specific processing was done to get data from and to the client and the server.

The socket API was easier to code than the datagram API as the underlying implementation took care of the additional overhead tasks, like ensuring the checksum is valid. RMI was a different beast in the sense that a lot of effort had to be devoted to set up the connection. Also with RMI, the connection between the server and client is tightly coupled, so if the interfaces change, it entails changes on both client and server side. However in the case of either the socket or the datagram API, the changes in interfaces translate to changes in the way data is passed from the server to the client

and vice-versa. The latter coupling is not that strong and is more flexible. Thus, for example, both the socket and the datagram API implementations can be easily modified to adopt an XML messaging protocol between the client and the server, whereas with RMI it will require a lot of changes on both sides.

REFERENCES

- [1] Coulouris, George and Dollimore, Jean and Kindberg, Tim, “*Distributed Systems – Concepts and Design*”, Addison Wesley, 2001
- [2] Stevens, W. Richard, “*UNIX Network Programming*”, Prentice-Hall, January 1999.
- [3] Client/Server Frequently Asked Questions
<http://www.faqs.org/faqs/client-server-faq/>
- [4] Sun’s Java WebSite
<http://java.sun.com/>
- [5] Heller, Philip & Roberts, Simon, “*Java2 sDeveloper’s Handbook*”, Sybex, 1999

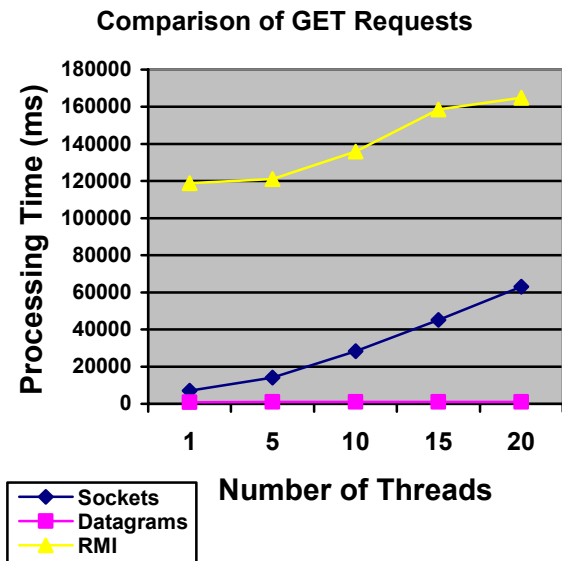


Figure 1

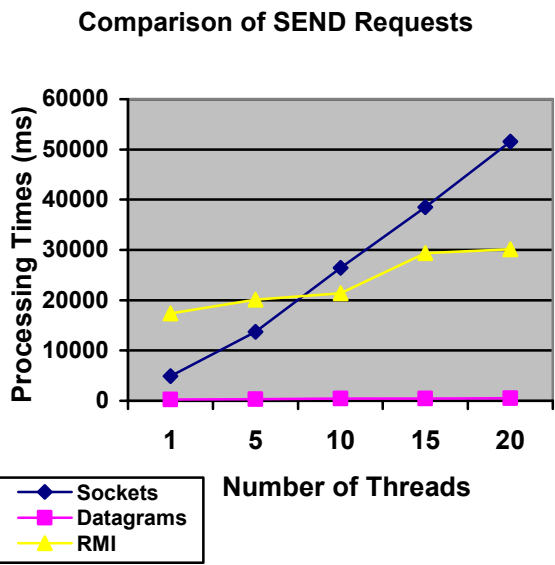


Figure 2