A Performance Comparison of Communication APIs on Solaris and Windows Operating Systems

Sherali Zeadally and Jia Lu
High-Speed Networking Laboratory
Department of Computer Science
Wayne State University
Detroit, Michigan 48202

Abstract

Communication Application Programming Interfaces (APIs) constitute an important component of many network-based applications. They play a central role in the end-to-end performance ultimately delivered by networked applications. Most network architectures exploit the underlying networking APIs in their designs. In this paper, we conduct an empiricial performance evaluation on the PC platform of some of the most popular networking APIs which include: Winsock/BSD, Java, and RMI. To explore the impact of the underlying operating system and Java Virtual Machine (JVM) architecture, we conducted performance tests on two operating systems namely, Windows NT 4.0 and Solaris 8. We found that on both operating system platforms, Winsock and BSD sockets yield about 1.8 times better throughput than Java sockets, and Java sockets in turn yield twice the throughput of that obtained using Remote Method Invocation (RMI). We also obtained about 1.3 times higher latency overheads with Java compared to either Winsock or BSD as well as with RMI when compared to Java sockets on both Windows NT and Solaris operating systems. We hope that our results will be useful to application designers and developers in better optimimizing end-to-end application performance.

1 Introduction

One of the ultimate goals of designers and developers of network systems and applications is the delivery of optimal end-to-end performance to end-users. One of the components that plays a central role in the end-to-end performance delivered to end-user applications is the performance of communication APIs used in their deployment. Popular communication APIs used on both Windows and UNIX operating systems include: Winsock (on Windows Systems) [8], BSD sockets (UNIX platforms) [6], Java [11], and RMI [9].

BSD sockets are a generalized networking interface introduced in 4.1c BSD. The sockets feature is available with most current UNIX system releases. Sockets allow communication between two different processes on the same or different machines. WinSock was developed based on BSD sockets. Both of them follow the procedure paradigm. Windows Sockets (WinSock) is the network programming interface for Microsoft Windows. The WinSock 2 architecture allows for simultaneous support of multiple protocol stacks, interfaces, and service providers as well as new protocols such as Internet Protocol version 6 (IPv6). With the emergence of increasingly complex distributed computations and applications, Java emerged to provide a clean and type safe object oriented programming model. Java technology, developed by Sun Microsystems, is an object-oriented, platform-independent, multithreaded, programming environment. The Java socket API provides a simplified interface to native sockets such as BSD and Winsock 2. It hides much of the details involved in traditional socket programming [3, 7]. Java RMI is a Java based approach for distributed objects. Java RMI provides a simple mechanism for making method invocations on Java objects residing in different virtual machines. Java RMI hides the low level communication issues from the programmer, allowing the programmer to focus on the distributed algorithmic aspects. Objects can be transported across networks via RMI and an object can invoke the methods of another object in a different virtual machine via RMI, in the same way as methods on local objects are invoked.

This work presents a performance comparison of these

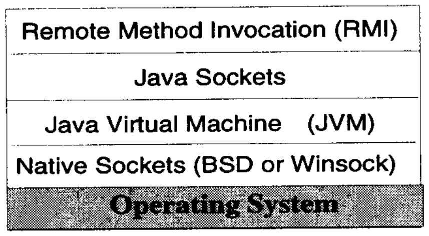
Remote Method Invocation (RMI)
Java Sockets
Java Virtual Machine (JVM)
Native Sockets (BSD or Winsock)
Operating System

Figure 1: Communication APIs and their dependence on each other.

communication APIs on two popular operating system platforms and we hope the results will provide insight to designers and application developers as to the overheads inherent when using these APIs in their applications and as a result save time in the search for performance optimizations of their applications. The rest of the paper is structured as follows. In the following section, we present the main contributions and benefits of this work. In section 3, we describe the measurement procedures and tests performed. Section 4 discusses our experimental results. Finally, in section 5, we summarise the main results and make some concluding remarks.

2 Contributions of this work

We have performed an extensive search in several venues (conferences, journals, and others) searching for publications that compare the different communication APIs. Surprisingly, we could not find any such results published. While many publications cover performance of each communication API alone, such as papers on BSD/Winsock [3] or Java [5, 2, 7] or RMI [1, 4], none of the papers give a systematic performance evaluation comparing the communication APIs side by side. This was one of the motivations for performing this work. Furthermore, while it is not surprising that in a layered system such as that shown in Figure 1 the overheads will increase from Winsock to RMI, we do not have any results published that actually show by *how much* the performance actually degrades when using one communication API compared to another. This work demonstrates empirically the extent to which performance worsens with different communication APIs. We also compare the communication APIs on two different operating systems running on the same Intel platform. In doing so, we are also able to compare the performance delivered by Java Virtual Machines (JVMs) running on Solaris x86 versus that running on Windows NT 4.0. Several technologies exist that use the underlying communication APIs such as Winsock 2, Java or RMI. Examples include plug and work architectures such as Jini [13, 10] or Universal Plug and Play (UPnP) [12] which use RMI and Java sockets respectively. Application designers deploying these technologies will have a better undertsanding of the performance of their applications if they understand the relative performance achieved by the different communication APIs. As a result, they can invest more time on other issues using the knowledge the performance results this work provides.

3 Measurement Procedures and Testbed Configuration

In this section, we present experimental tests performed on several network communication APIs. The main goal of these tests was to investigate the overheads introduced by the different APIs available today, and commonly used by network application designers and developers. We hope these results will give insight into the performance overheads associated with each of the communication APIs.

The experimental testbed used included the followings: a pair of Pentium III workstations with 500 MHz processor, 256 MB RAM, 40 GB hard disk were used in all tests. Each host was loaded with two operating systems: Windows NT 4.0 and Solaris. First, a set of tests was performed between the two hosts each running Windows NT 4.0. Second, another set of tests was performed between the two hosts each running the Solaris operating system (for the Intel platform). The two workstations were connected using 100 Mbits/s Ethernet via an Ethernet switch. All tests were executed on an unloaded Ethernet network using the TCP/IP protocol stack.

We conducted tests for various network Application Programming Interfaces (APIs) including: Winsock/BSD sockets, Java sockets, and RMI using throughput and round-trip latency as our performance metrics. They were measured as follows: *Throughput:* Average application to application throughput was measured by timing bulk data transfers over a sufficiently long period of time using test programs we have developed for a range of message sizes; *Round-Trip Time:*The test application on one host (one for each APIs used) echoes a message of a specified size to the peer application running on the remote host. Basically, the client machine sends an M-byte message to the server (timing starts) and waits to receive the M-byte message back; the interaction was repeated N times between client and server after which timing stops. From the N readings obtained, an average round-trip time for exchanging an M-byte message between the two workstations was calculated.

4 Experimental Results

4.1 Latency

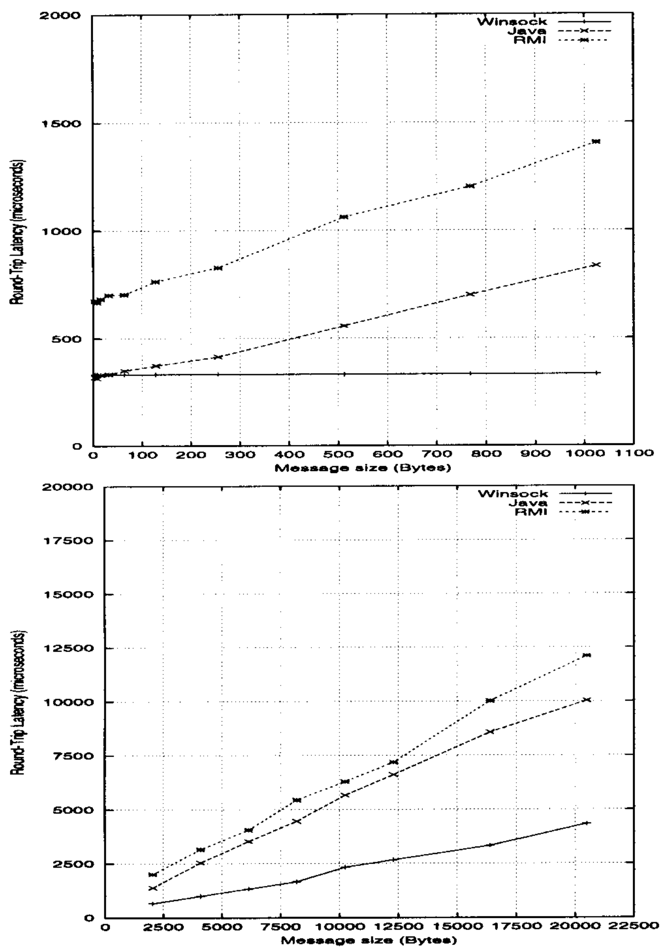

Figure 2: Round-trip latency on Windows NT 4.0 for small and large message sizes.

4.1.1 Latency on Windows NT 4.0

The latency results obtained with the different communications APIs on Windows NT 4.0 are shown in Figure 2.

From Figure 2 , we note that for the smallest message (1 byte), the least latency overhead incurred by RMI(663 microseconds) is about twice that incurred by Java sockets (305 microseconds). The minimal overheads incurred by Winsock and Java socket are almost the same (305 microseconds). We also note that the latency for Java sockets is about 4 times worse than Winsock. However, latency performance using RMI is only about 1.3 times higher than Java sockets.

With large message sizes, we observe from Figure 2 that latency using Java sockets is about 2.5 times higher than latency with Winsock. Latency performance with RMI for large messages is about 1.1 times higher than Java sockets.

4.1.2 Latency on Solaris

In the case of Solaris, for small message sizes (Figure 3), we note that for the smallest message size (1 byte), minimal latency incurred by RMI (897 microseconds) is about twice that incurred by Java sockets (407 microseconds). Minimal latency overhead for Java socket is almost twice that of BSD sockets (about 210 microseconds). But latency performance of RMI is about 1.3 times that of Java sockets.

With large messages on Solaris, Java sockets give a four-fold increase in latency compared to BSD sockets. Round-trip latency with RMI is about three times worse latency in contrast to their BSD counterpart.

4.2 Throughput on Windows NT and Solaris platforms

From Figure 4, throughput on Windows NT platform is more stable compared throughput results obtained on Solaris. On Windows NT, for message sizes greater than 4 KB, we observe that throughput for Winsock 2 is about 1.8 times the throughput of Java sockets and about 3.7 times the throughput obtained using RMI.

From Figure 4, throughput on Solaris is not as stable as Windows NT, particularly for small message sizes (less than 4 KB). Similar to Solaris, for message sizes greater than 4 KB, BSD socket throughput is almost 1.8 times the throughput of Java sockets and almost 3.4 times the throughput obtained with RMI.

It is interesting to note that on both Windows NT and Solaris platforms, the throughput performance of Java

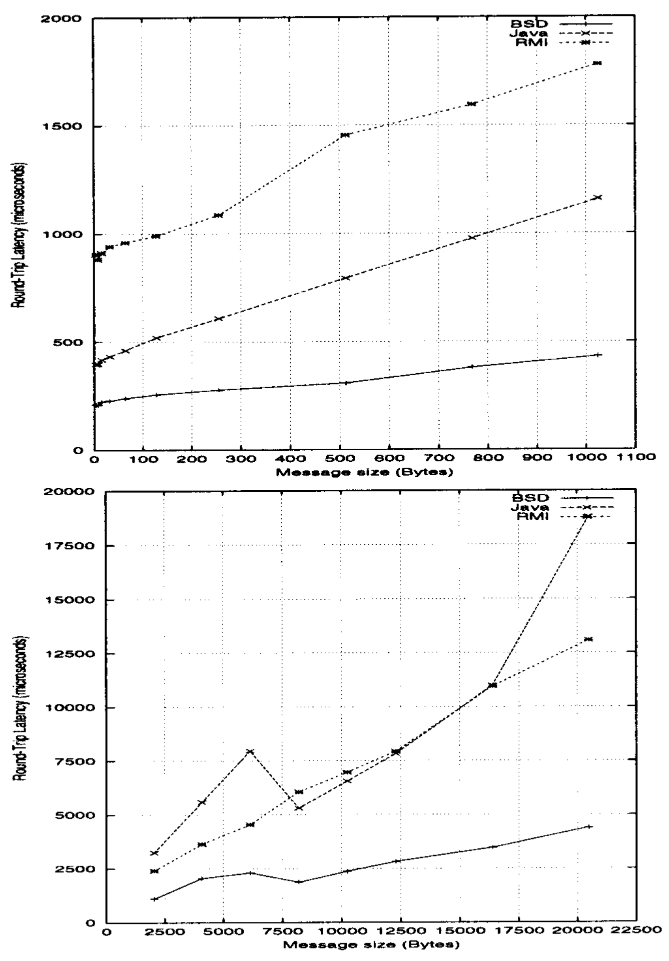

Figure 3: Round-trip latency on Solaris for small and large message sizes.

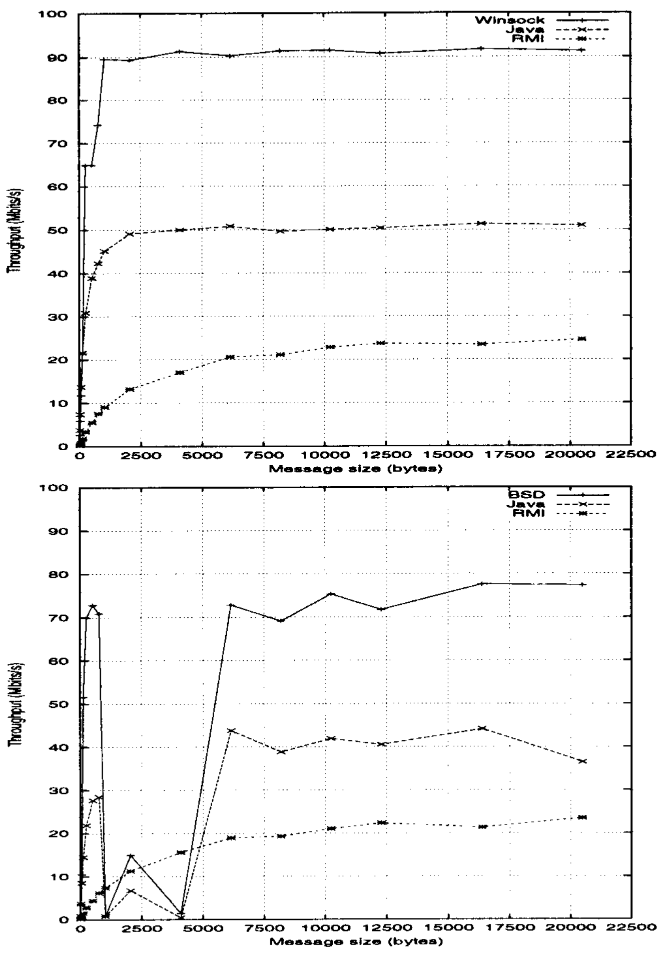

Figure 4: Throughput on NT and Solaris.

is almost *twice* that obtained with RMI.

5 Conclusions

In this work, we conducted a performance comparison of the most common communication APIs namely, Winsock, BSD Sockets, Java, and RMI. We summarize below our main performance results obtained on two operating systems namely, Windows NT 4.0 and Solaris on the Intel platform (using Solaris x86).

- The throughput performance obtained (on both Windows NT 4.0 and Solaris) over Java is *twice* that over RMI. Both Winsock (on Windows) and BSD Sockets (on Solaris) yield a throughput performance of about *1.8 times* higher than Java.
- In the case of the round-trip latency results, we found that RMI gives *1.3 times* higher latency than Java for small packets (less than 1 KB) on both Windows and Solaris. Moreover, we also obtained similar results (i.e. RMI latency is 1.3 times higher than that of Java) when we compare Java round-trip latency with Winsock and BSD Sockets round-trip latencies.
- Java socket and RMI are both more stable and have better performance on Windows NT than Solaris 8 x86 (on the Intel platform). This leads us to conclude that the JVM for Solaris (Intel version) is not as stable as the JVM on Windows NT.
- The performance delivered by BSD and Java sockets are not consistent and are unstable (for packet sizes between 1KB and 8 KB) on Solaris 8 x86 compared to Windows NT 4.0 which gave

consistent performance. We confirmed that Solaris 8 was responsible for the poor performance by repeating the same test on Solaris 8 running on the Sparc station (a SPARC Ultra 10) which yielded good performance. We infer from this result that the implementation of BSD on Solaris 8 x86 is not stable and since Java sockets use the underlying native socket (in this case BSD) implementation, they also give unstable performance.

Finally, as we mentioned early in the paper, we all know that APIs such as Java and RMI have several benefits associated with each of them including: platform independence, object oriented, code re-usability, simplicity, and so on. However, these benefits come at a cost. For instance, object serialization supported by RMI facilitates the development of distributed Java application by abstracting many of the networking issues, it is an important performance inhibitor [7] since the large overhead in RMI is cause by hierarchy of stream classes that copy date and call virtual methods [5]. This paper experimentally demonstrates to application developers and designers faced with a choice of communication APIs, the expected performance to be gained with each of them and the performance impact (e.g. by *how much* performance degrades) APIs such as Java and RMI have compared to native Winsock and BSD. Another major benefit of knowing the inherent performance that can be delivered by these APIs enable developers to invest their time in optimizing other parts of the application to achieve the high end-to-end performance objective.

6 Acknowledgements

This work was supported by a grant (EDUD-7824-000145-US) from Sun Microsystems Inc. (Palo Alto). The authors thank DV Sreenath for his suggestions on improving the quality of this paper.

References

[1] S. Ahuja and R. Quintao, "Performance Evaluation of Java RMI: A Distributed Object Architecture for Internet Based Application", in Proceedings of 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pages 565-569, 2000.

[2] F. Breg Constantine and D. Polychronopoulos, "Java Virtual Machine Support for Object Serialization", University of Illinois, Urbana-Champaign.

[3] C. Krintz and R. Wolski, "Using JavaNws to Compare C and Java TCP-Socket Performance", Journal of Concurrency and Practice and Experience, Vol. 13, No. 8-9, pages 815-839, 2001.

[4] C. Nester, M. Philippsen, and B. Haumacher, "A More Efficient RMI for Java", in Proceedings of ACM JavaGrande99 Conference, pages 152-159, San Francisco, June 1999.

[5] J. Maassen et al., "An efficient implementation of Javas Remote Method Invocation" in Proceedings of ACM Symposium on Principles and Practice of Parallel Programming , Atlanta, GA, May 1999.

[6] M. Mckusick, K. Bostic, M. Karels, and J. Quarterman, "The Design and Implementation of 4.4 BSD Operating System", Addison-Wesley, ISBN 020-1549794.

[7] R. Nieuwpoort et al., "Wide-Area Parallel Computing in Java", in Proceedings of ACM 1999 Java Grande Conference, pages 8-14, San Francisco, CA, June 1999.

[8] http://www.sockets.com/winsock2.htm.

[9] Sun Microsystems Inc., "JavaTM Remote Method Invocation (RMI) Specification" , Version 1.3 of JavaTM 2 SDK Sun Microsystem.

[10] Sun Micrfosystems, "JiniTM Architecture Specification", Version 1.0.1 Sun Microsystem, November, 1999.

[11] Sun Microsystems, "What is the Java Platform," http://java.sun.com/nav/whatis/, Sun, October 1999.

[12] UPnP Forum, http://www.upnp.org.

[13] J. Waldo, "The Jini Architecture for Network-Centric Computing", CACM, Vol. 42. No. 7, July 1999.