Performance Characteristics of a Java Object Request Broker

David Miron and Samuel Taylor

DSTO-TR-0696

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

Performance Characteristics of a Java Object Request Broker

David Miron and
*Samuel Taylor**

Information Technology Division
Electronics and Surveillance Research Laboratory

***Advanced Computational Systems**
Australian National University

DSTO-TR-0696

ABSTRACT

The efficiency of the Common Object Request Broker Architecture (CORBA) for the transfer of large files over a network is of particular interest to the Imagery Management and Dissemination Group (IMAD). The IMAD group will be using Java and CORBA for such transfers. This report studies the performance of a Java Object Request Broker (ORB) for the transfer of large files over such networks. This performance analysis is done using the Visigenics ORB Visibroker and involves the measurement of throughput and latency. These measurements are then compared with the results obtained when using socket to socket connections. The results show that the throughput of an ORB for large file transfer approaches that of sockets on low bandwidth networks. However on high bandwidth networks the throughput using the ORB is significantly less than that using sockets. It is also shown that the latency incurred by the ORB is much greater than that incurred using sockets.

RELEASE LIMITATION
Approved for public release

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

Published by

DSTO Electronics and Surveillance Research Laboratory
PO Box 1500
Salisbury South Australia 5108

Telephone: (08) 8259 5555
Fax: (08) 8259 6567

AR-010-589
June 1998

APPROVED FOR PUBLIC RELEASE

Performance Characteristics of a Java Object Request Broker

Executive Summary

The results of this report show that for Common Object Request Broker Architecture (CORBA) the transfer rate achievable for large files through the Object Request Broker (ORB) approaches that achievable through sockets over low speed networks. However, for high-speed networks the transfer rate through the ORB is somewhat poorer than those using sockets. Further the latency incurred by the ORB is significantly greater than the latency incurred using sockets. Therefore the sending of small frequent messages through the ORB should be avoided. It is also found that the Java "Just in Time Compiler" give some benefits in performance and large block sizes should be used for data transfer. Further the results show that the use of Java for the development of servers is not an issue, as it is the ORB that is impeding performance.

Authors

Mr. D. J. Miron
Information Technology Division

David is a Research Scientist employed in the Intelligence Systems Group. His current work is in the EXC3ITE project where he is working in CORBA. Davids interest are in Parallel and Distributed high performance computing. David has a PhD in Computer Science from the Australian National University. He also has a Masters of Applied Science from Monah University and a Bachelor of Applied Science 'Mathematics' from the Royal Melbourne Institute of Technology.

Mr. S. Taylor
Information Technology Division

Sam is a PhD student with the CRC for Advanced Computational Systems at the Australian National University. Sam worked in the IMAD project at DSTO Fernhill Canberra during a secondment from the Australian National University. His current research interests are in design patterns and visualisation

Contents

1. Introduction

The Common Object Request Broker Architecture (CORBA) [1] provides an environment for developing and deploying object-based distributed applications. There is an important class of distributed applications which require high bandwidth data transfer and only limited presentation layer services. Examples include scientific computations, image processing and multimedia streaming applications and the Experimental Command Communication Control and Intelligence Test Environment (EXC3ITE)[2].

CORBA objects are accessed through public interfaces specified in an Interface Definition Language (IDL)[1]. This is known as the Static Invocation Interface (SII). While this report focuses on the SII it is noted that CORBA also supports a Dynamic Invocation Interface (DII) which allows objects to discover and use interfaces dynamically, and does not use IDL.

The CORBA IDL allows developers to specify their interfaces using a rich set of data types and structures. An IDL compiler produces stub and skeleton classes which are used on the client and server hosts to marshall and unmarshall the rich IDL types for network transmission. These stub and skeleton classes provide presentation layer services for an ORB. Although the transparency of these presentation layer services is one of the more powerful aspects of CORBA, they can incur a significant performance overhead. Previous work has demonstrated that for such applications, the overhead of CORBA presentation layer services may unacceptably restrict the bandwidth of data transfer[3, 4].

Java[5] is a popular programming language for implementing portable, object-oriented applications. Java ORBs allow developers to write full-fledged CORBA applications in Java. Although the performance characteristics for C++ ORBs have already been examined[3, 6, 7], the performance of Java ORBs has not been considered in as much detail. This report investigates the suitability of Java ORBs for bulk data transfer over various network configurations and computer types.

The focus of this report is on transferring raw byte data using SII. This data is essentially typeless and should not require any presentation layer services. However, previous work has demonstrated that many ORBs actually perform presentation layer formatting of such data. The transfer of large volumes of typeless data represents a "best case" for an efficient ORB. It is anticipated that significantly poorer performance will result for richly typed data.

This report has essentially two aims. The first is to measure the performance of Java ORBs for bulk transfers of typeless data over a range of different networks. The second is to determine whether the Java Virtual Machine (JVM)[5] represents a bottleneck for bulk data transfer, and to measure the effects of JVM optimisations such as just-in-time (JIT) compilation.

The structure of this report is as follows. Section 2 describes the test environment used in our experiments. It includes details of the networks, hosts and ORB used as well as a brief description of the test application. Section 3 presents the results of our experiments and discusses some of the implications of these results. Section 4 identifies related work and suggests areas of future research. Finally in Section 5 our conclusions are presented.

2. Details of the Experiments

This section discusses the details of the experiments used for the collection of results.

2.1 Test Programs

The results presented in Section 3 were produced using a simple data transfer application. This application was designed to produce two sets of measurements:

1. Bulk transfer rates between applications for a range of transfer block sizes.
2. Application to application latency times for a minimal packet.

The application consists of separate client and server processes. The bulk transfer rates were calculated by transferring large amounts of data - typically tens or hundreds of megabytes - from server to client. Rather than attempt to transfer the entire amount in one go, the application transfers data as a series of smaller blocks. The results presented in Section 3 demonstrate that varying the size of these blocks has extremely significant effect on transfer rates.

The latency times were calculated by transmitting a minimal packet back and forth between client and server, until the packet made a specified number of trips. The total transmission time was then divided by the number of trips to produce an average latency time. It should be noted that the latency measure includes not only transmission cost, but also the cost of context switching packet transmission since packet transmission usually involves a context switch.

Two separate versions of the application were developed. The first used the basic Java socket primitive to transfer data, and essentially provides a peak transfer rate for Java applications on each network. The second implementation used VisiBroker for Java 3.0[8] - a Java ORB developed by Visigenic - to provide the communication mechanisms. VisiBroker is a pure Java ORB which runs on any platform that supports version 1.1.2 of the Java Developers Kit (JDK). Technically VisiBroker is not "100% Pure Java" because it includes an optional System Agent which is implemented in native code. The test application used only the portable elements of the ORB.

The VisiBroker ORB was chosen for two principal reasons. Firstly, VisiBroker for C++ was the basis of much earlier work into CORBA transfer rates[3, 9]. Secondly, the VisiBroker ORB is included in all Java-enabled versions of Netscape Navigator 4.0 and so can lay reasonable claim to being the most ubiquitous Java ORB.

2.2 Test Environments

The principal aim of our work was to measure the bulk data transfer performance of Java ORBs in a variety of machine and network environments. In keeping with this aim the test programs were run in three separate test environments, each designed to represent different levels of host and network performance. In the two lower performance environments, different machine architectures were used to provide a notion of client- and server-grade machines. However, although the machines were characterised as being either client-grade or server-grade, the bandwidth measurements were performed in both directions. Where applicable the tests were repeated with and without JIT compilation, to try to identify any improvement in performance.

In addition to the network tests in each environment, we performed *loopback* tests for each machine. In the loopback tests the client and server processes run on the same machine, and communicate through the localhost interface. Packets pass down through the TCP layer, are caught at the IP layer, and never reach the physical network. In effect, the client and server processes use the TCP/IP stack as an inter-process communication mechanism. The value of these tests is that the localhost interface represents an extremely high speed, low latency network. This approach, while providing a valuable indication of the peak transfer rates at the application level, is not without limitations. The limitations involve issues of buffer management within the TCP/IP stack which may unfairly affect localhost performance. Context switching would also appear to be an issue on uniprocessor machines. However, in practice the loopback tests incur no more context switching overhead than the network tests.

VisiBroker supports a number of optimised data transfer mechanisms which may be used when client and server applications are run on the same machine. For example, If client and server are run as separate threads within the same process VisiBroker uses the standard Java local method invocation mechanism. If client and server are run as separate processes on the same host, VisiBroker uses a shared memory buffer to transfer data between client and server.

It is anticipated that use of these mechanisms would provide extremely significant improvements in performance. However, for the purposes of the loopback tests they were disabled, since our aim was to determine the performance constraints of the actual network interface at the IP level.

3. Results

This section presents our experimental results, produced in three different test environments. For each environment loopback and network transfer rates are presented. The loopback results represent the best possible transfer rate for each machine, and provide a valuable contrast with the network rates. The latency times for each machine are also presented.

3.1 Transfer Rates over 10Mbps Ethernet

The first test environment was designed to represent a modest, low performance local area network. The physical network was a 10 megabits switched ethernet. Two different machine architectures were used to try to represent a typical client and server. The client-grade machine was a Dell OptiPlex GL+ 5100 desktop PC running Windows 95 on a 100MHz Pentium with 32 MB RAM. The server-grade machine was a Sun SparcStation 10 running Solaris 2.5 on dual 40MHz SuperSparc processors with 164MB RAM. Both machines used the Sun implementation of JDK1.1.4. The PC also used the Sun Java Performance Pack - an experimental JIT compiler which provides modest performance gains over the interpreted Java Virtual Machine (JVM). No JIT compiler was used on the SparcStation since none was supported at the time of testing.

Figure 1 presents the loopback transfer rates for the socket and ORB implementations of the test application running on the PC. Figure 2 presents the loopback transfer rates for the same application on the SparcStation. Note that the scale on the vertical axis of Figure 2 is a factor of 10 greater than that of Figure 1. This indicates that even in loopback mode the PC can sustain only a modest transfer rate. Clearly, on both machines the performance of the socket implementation was significantly higher than that of the ORB implementation, for all block sizes. The performance through the ORB was extremely poor for small block sizes, but improved for larger block sizes, with peak throughput achieved using blocks of 256K. On the low bandwidth PC the peak ORB performance was approximately 61% of the peak socket performance. However, on the higher bandwidth SparcStation the peak ORB performance was only 17% of the socket.

Figure 1 suggests that the effects of the JIT compiler used on the PC are slight. The socket code demonstrated a modest improvement in peak performance of approximately 7%. This is not unexpected as the socket implementation spends most of its time making system calls, and consequently the overhead of the interpreter is limited. For small block sizes the ORB implementation displayed a similar, modest improvement. However, for block sizes of 32K or more the performance of the

interpreter and the JIT compiler were similar. This suggests that performance of the JVM is not the significant bottleneck within the ORB.

Figure 3 presents the network transfer rates for the test application when data is sent from the PC to the SparcStation across the 10Mbps ethernet. Figure 4 presents the network transfer rates from the SparcStation to the PC. In both tests the JIT compiler was used on the PC. These two figures tell a similar story to the loopback results. Performance through the ORB was poor for small block sizes, but improved steadily and again peaked at 256K blocks. When the PC was the server the peak ORB rate was 82% of the peak socket rate and when the SparcStation was the server the peak ORB rate was 66% of the peak socket rate. However, transfer rates through the ORB are almost identical in either direction.

The principal result from these tests is that for low bandwidth networks, the performance of the ORB approaches that of raw sockets for appropriate block sizes. However, the SparcStation loopback measurements suggest that ORB performance falls away in higher bandwidth environments. The effects of JIT compilation are modest, and suggest that the JVM is not the principal bottleneck to ORB performance.

3.2 Transfer Rates over 100mbps Fast Ethernet

The second test environment was designed to represent a higher performance local area network. The physical network was a 100 megabits switched fast ethernet. Again, two different machine architectures were used which represent more powerful client and server machines. The client-grade machine was a Silicon Graphics O2, running IRIX 6.3 on a single 180MHz MIPS R5000 with 64 MB RAM. The server-grade machine was a Sun Enterprise E3000 running Solaris 2.5.1 on dual 250MHz UltraSparc processors with 512MB RAM. The E3000 ran the Sun implementation of the JDK1.1.4 using an interpreted JVM. The O2 used the Silicon Graphics implementation of the JDK1.1.2. This implementation of the JVM is novel because in addition to the standard interpreter it supports both a JIT compiler, and a native code translator. The native code translator is particularly interesting because it annotates each Java class file with optimised MIPS translations of the standard byte code. It does this by creating "fat" class files which contain two implementations of each method; one in portable JVM byte code and the other in optimised MIPS code. This approach promises significant performance gains over traditional JIT compilers, and is ideal for statically scoped class libraries such as an ORB. It is typically used in conjunction with a JIT compiler, which is used to optimise dynamically downloaded classes.

Figure 5 presents the loopback transfer rates on the O2 using the interpreted JVM, the JIT compiler and MIPS code translation. Figure 6 presents the loopback transfer rates of the E3000 using the interpreted JVM. Note that the scale of the vertical axis of Figure 6 is five times that of Figure 5. Once again the performance of the socket implementation was significantly better than that of the ORB implementation. On the

O2 the peak ORB performance was 22% of the peak socket performance and on the E3000 the ORB managed only 18% of the peak socket rate. As with the previous tests, the peak ORB rate was achieved with 256K blocks. The JVM optimisations supported on the O2 provided modest increases in the peak transfer rates of both implementations. Just-in-time compilation resulted in 7% better throughput for the socket implementation, and 11% better throughput for the ORB. MIPS translation was even more effective for the ORB, resulting in 16% better throughput than the interpreter.

Figure 7 presents the network transfer rates from the O2 to the E3000 for both implementations, while Figure 8 presents the rates from the E3000 to the O2. For transfers from the O2 to the E3000 the peak ORB performance was 51% that of the peak socket performance. For transfers from the E3000 to the O2 the ORB managed only 39% of the peak socket rate. This supports the result from the previous section, that ORB performance falls away on high bandwidth networks.

The most significant result to emerge from these tests is illustrated by the highly irregular shape of the ORB curve in Figure 7. Performance of the ORB was extremely sensitive to the block size, with small variations in the block size producing wildly different transfer rates. This behaviour is due to Nagle's algorithm[10] which is used by TCP stacks to buffer and aggregate very small packets. Nagle's algorithm is an effective mechanism to prevent the flooding of wide area networks with very small packets. However, when used on high-speed, low latency networks it can have a disastrous effect on throughput of some applications. For such applications, the only solution is to disable Nagle's algorithm.

Although the JDK1.1 Socket class provides the two new methods "java.net.Socket.getTcpNoDelay()" and "java.net.Socket.setTcpNoDelay()" to control Nagle's algorithm, the results presented in Figure 7 indicate that the sockets used by the ORB implementation are affected by Nagle's algorithm. SGI claims that their implementation of JDK correctly implements the new socket methods [18] which implies that the Visigenic ORB does not disable Nagle's algorithm on its sockets. The end result of this is that the performance of user level programs suffers tremendously.

3.3 Transfer Rates over 155Mbps ATM

The third test environment was designed to represent a very high performance network or a server cluster. The physical network was a 155 megabits ATM network switched through a Digital Gigaswitch/ATM. The client and server machines were both Digital AlphaStation 600s running Digital Unix 4.0c on 266MHz Alpha AXP-5 processors with 128MB RAM. Both machines used the Digital implementation of the JDK1.1.3 which includes a JIT-compiler and native thread support. While standard Java threads are typically provided at the application level the Digital JVM uses native Digital Unix threads which are provided through a hybrid user/kernel mechanism.

This gives a more robust threads architecture, which is of particular value to sophisticated server processes.

Figure 9 presents the loopback transfer rates for the AlphaStations using the standard interpreter and JIT compiler. Figure 10 presents the network transfer rates for the AlphaStations. As in the previous tests the socket implementation comprehensively outperformed the ORB implementation. Peak ORB throughput was only 21% of the peak socket throughput in loopback tests, and 32% in network tests. Figure 10 illustrates the effect of Nagle's algorithm on transfer rates. The effect is even more pronounced on a high speed network - with 8K blocks the ORB transfer rate was only 40K/sec. However, when Nagle's algorithm is disabled the ORB displays the same transfer characteristics seen on other networks.

Although the JIT compiler provided a modest gain for some block sizes, the overall effect was negligible. This should not be viewed as a failing of the JIT compiler. Rather, it indicates that on a strong workstation even an interpreted JVM is more than able to saturate a high speed network. For example, the socket implementation running on an interpreted JVM was able to maintain a transfer rate of 16393K/sec - which corresponds to approximately 96% of the theoretical peak performance of a 155Mbps ATM network.

3.4 Application Level Latency

In addition to the bulk data transfer bandwidth, we measured the application level latency times for each test environment. ORBs typically impose a number of software layers between a network primitive and an application. These layers are responsible for such things as demultiplexing objects and methods and unmarshalling parameters. Application level latency provides a measure of these overheads, and represents the time required for a minimal packet to pass through the ORB layers on both the client and server.

Figure 11 presents the latency times for the socket and ORB implementations of our test application in all the test configurations. The results in Figure 11 were produced using the most optimised JVM available on each platform. JIT compilation was used on the PC and the AlphaStations, MIPS translation was used on the O2, and the standard interpreter was used on the SparcStation and E3000. Not surprisingly the latency times of the ORB implementation are significantly higher than those of the socket implementation. On average the ORB was approximately 4.6 times slower than the socket. In relative terms the ORB performed better for the higher latency network connections than for the very low latency loopback tests, which suggests that network latency alone is not the bottleneck.

Figure 12 presents the network latency times for the ORB implementation using the various different virtual machines available on each platform. Figure 13 presents the

ORB loopback latency times for the three machines which supported optimised virtual machines. Both figures demonstrate that JIT compilation provided a modest improvement in latency times, and native code translation failed to provide anything more significant. The best case for an optimised virtual machine was for the native code translation used on the O2, which provided approximately 24% lower latency than the interpreter for both loopback and network tests.

The conclusion from these tests is that optimised JVMs provide only modest improvements in the latency times of ORBs. This result conflicts with earlier work into the latency times of Java ORBs[9], which concluded that JIT compilation can significantly reduce the latency incurred by an ORB. We attribute this difference to advances in the performance of both interpreted and JIT compiled Java Virtual Machines, and the relative costs of method invocation. Java method invocation is known to be expensive, since Java's inheritance semantics imply that Java methods are all equivalent to C++ pure virtual methods. There is little that either JIT compilers or native code translators can do to optimise method invocation. In effect it is as expensive to a JIT compiler as it is to an interpreter. Since ORB skeletons tend to comprise lengthy call chains the overhead of method invocation is a significant part of the ORB latency. Consequently, as the performance of interpreted JVMs has improved the scope for optimisation by JIT compilation has decreased.

3.5 Summary of Results

We have attempted to characterise the performance of Java ORBs in terms of bulk data transfer rates, and application level latency times. In so doing, we follow a pattern established in earlier work on the performance of C++ ORBs[3, 4]. Comparing our results with this previous work, leads us to conclude that the performance characteristics of Java ORBs are essentially the same as those of C++ ORBs. In both cases performance of the ORBs is acceptable over low bandwidth networks, but extremely poor over medium and high bandwidth networks.

Our tests with just-in-time compilers and native code translators lead us to conclude that they provide only modest improvements in the transfer rates and latency times. We attribute this to the fact that in both the socket and ORB code the performance of the Java Virtual Machine is not a bottleneck. Essentially the overhead of the interpreter is minor, and so there is only limited scope for optimisation through use of native code. There is certainly scope for optimising the ORB code, but this can only occur at the algorithmic level.

That the JVM is not a bottleneck is in itself a significant result. Java has been widely advocated as a language for developing CORBA clients, but few have advocated developing servers in Java. This result suggests that for applications where data transfer is the predominant limit, there is no reason not to develop both client and server processes in Java.

Transfer rates through the ORB were very sensitive to the size of the blocks transferred. Although it is likely that optimal block size is a function of the network and machines, in our tests 256K blocks consistently achieved peak performance. We conclude that in general large blocks perform better than small blocks, but that optimal block size should be determined empirically.

Sensitive as the ORB was to block size, it was even more sensitive to Nagle's algorithm. When Nagle's algorithm was used slight changes in block size had wildly different effects on ORB transfer rates. The JDK1.1 provides methods which may be used to disable Nagle's algorithm on a per-socket basis.

4. Related Work and Future Work

The results presented here are based on the performance of only one ORB. In future work we hope to perform a more comprehensive evaluation of Java ORBs, including OrbixWeb[11] and JacORB[12]. We would also like to measure the overhead incurred by an ORB when transferring rich data types. There are significant differences in the way CORBA IDL is mapped to Java and C++, which may affect the presentation layer overheads associated with transferring such data. Therefore it may be valuable to perform a direct comparison between C++ and Java ORBs for richly typed data. Since CORBA is only one of a number of distributed technologies available to Java developers we would like to evaluate the performance of Java RMI[13] and Java DCOM[14].

Significant work is being done to improve the performance of C++ ORBs. In particular there is a substantial push to develop real-time, high performance and parallel ORBs[15-17]. Much of this work is focused on optimising the IIOP protocol, performing more intelligent buffer management and providing more efficient method dispatch mechanisms. Clearly there is significant scope for applying this work to Java ORBs.

5. Conclusions

We have measured the data transfer rates, and latency times of a Java ORB in a range of different environments. We conclude that Java ORBs perform reasonably well over low-bandwidth networks, but poorly over medium and high bandwidth networks. This result supports earlier research into the performance of C++ ORBs, and suggests that commercial implementations of CORBA are not designed for high performance environments. We have identified that transfer rates for Java ORBs are sensitive to block size. More significantly we have identified that ORB transfer rates may be badly affected by Nagle's algorithm. Just-in-time compilers, and other optimised

implementations of the Java Virtual Machine, appear to provide only modest improvements in the transfer rates of Java ORBs. However, we believe that the ORB architecture, rather than the JVM is the principal bottleneck. We conclude that Java ORBs are performance competitive with C++ ORBs, but that in a high performance environment neither can compete with specialised socket code.

References

[1] OMG, *CORBA: The Common Object Request Broker: Architecture and Specification, Revision 2.0*: Object Modelling Group, 1995.

[2] Defence Science and Technology Organisation, "An Experimental C3I Capability and Techology Demonstrator: Phase One.," .

[3] A. Gokhale and D. C. Schmidt, "Measuring Performance of Communication Middleware on High-Speed Networks," in *SIGCOMM'96*. Stanford University, 1996.

[4] A. Gokhale and D. C. Schmidt, "Evaluating Latency and Scalability of CORBA over High-Speed ATM Networks," in *International Conference on Distributed Computing Systems*. Baltimore Maryland, 1997.

[5] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*: Addison Wesley, 1997.

[6] A. Gokhale and D. C. Schmidt, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks," in *GLOBECOM'96*. London, 1996.

[7] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *GLOBECOM'97*. Phoenix, AZ, 1997.

[8] VisiBroker, *VisBroker for Java: Reference Manual Version 3.0*: Visigenic Software Incorporated, 1997.

[9] R. Orfali and D. Harkey, *Client/Server Programming with Java and CORBA*: John Wiley and Sons Inc, 1997.

[10] W. R. Stevens, *Unix Network programming*: Prentice Hall, 1990.

[11] OrbixWeb, *OrbixWeb Reference Guide*: IONA Technologies Ltd, 1996.

[12] G. Brose, "JACORB: Implementation and Design of a Java ORB," presented at PDAIS'97, 1997.

[13] Java RMI, *RMI: Java Remote Method Invocation - Distributed Computing for Java*: Sun Microsystems, 1997.

[14] D. Box, *Understanding COM*: Addison Wesley, 1997.

[15] K. Keahey and D. Gannon, "PARDIS: A Parallel Approach to CORBA," presented at Proceedings of IEEE 6th International Symposium on High Performance Computing, Portland, OR, 1997.

[16] V. F. Wolfe, L. C. DiPippo, R. Ginis, M. Squarito, S. Wohlever, I. Zykh, and R. Johnston, "Real-Time CORBA," in *IEEE Real-Time Applications Symposium*, 1996.

[17] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design of the TAO Real-Time Object Request Broker," *Computer Communications Journal*, 1997.

[18] A. Vincent, Personal Communication, 1998.

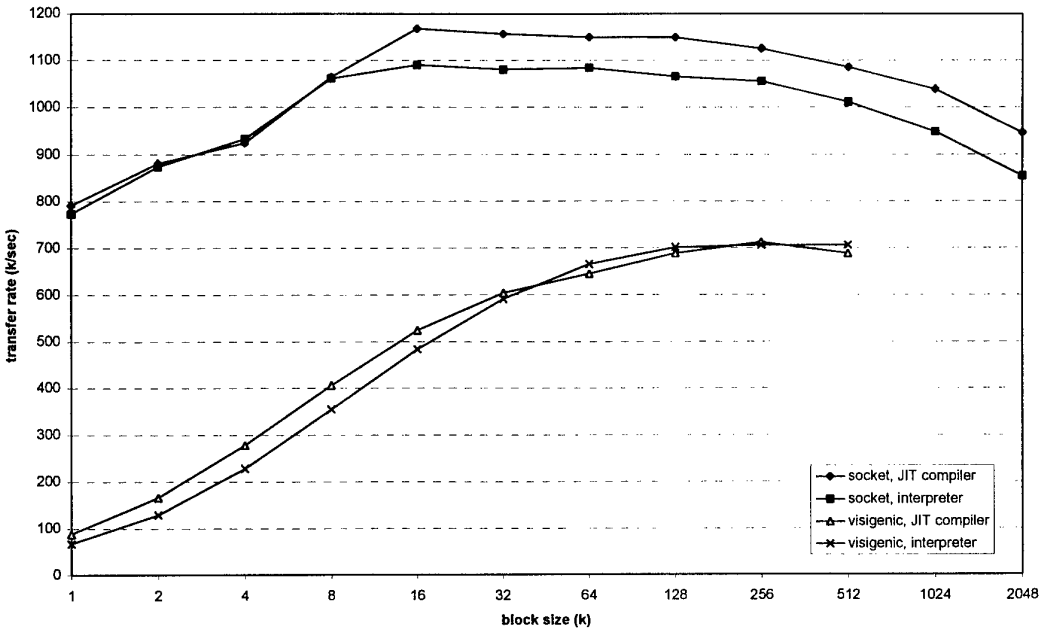

Figure 1: Loopback transfer rates for the PC

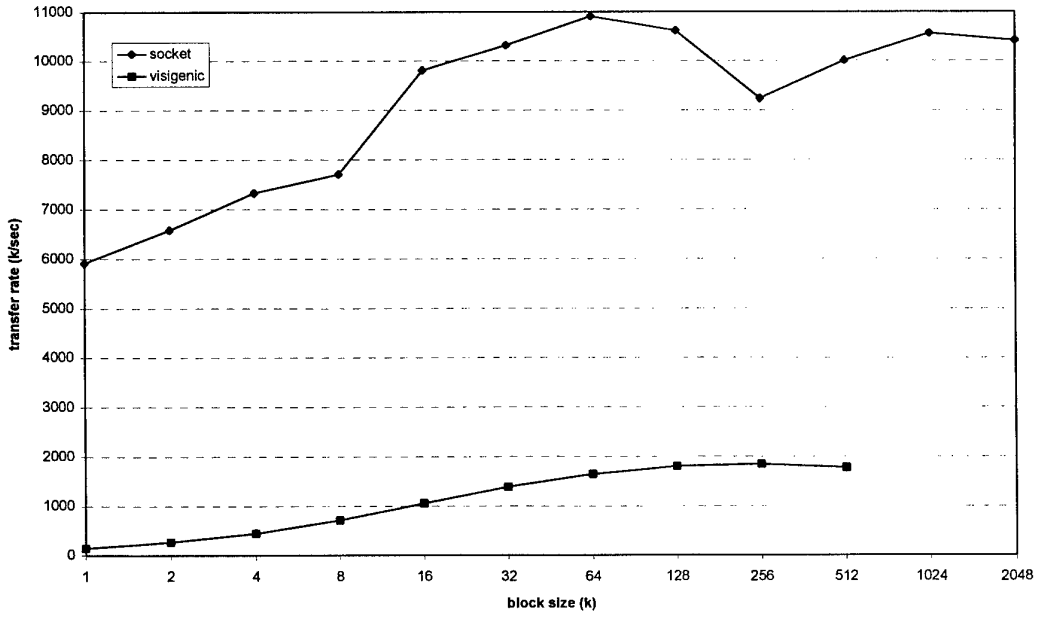

Figure 2: Loopback transfer rates for the Sun SparcStation 10

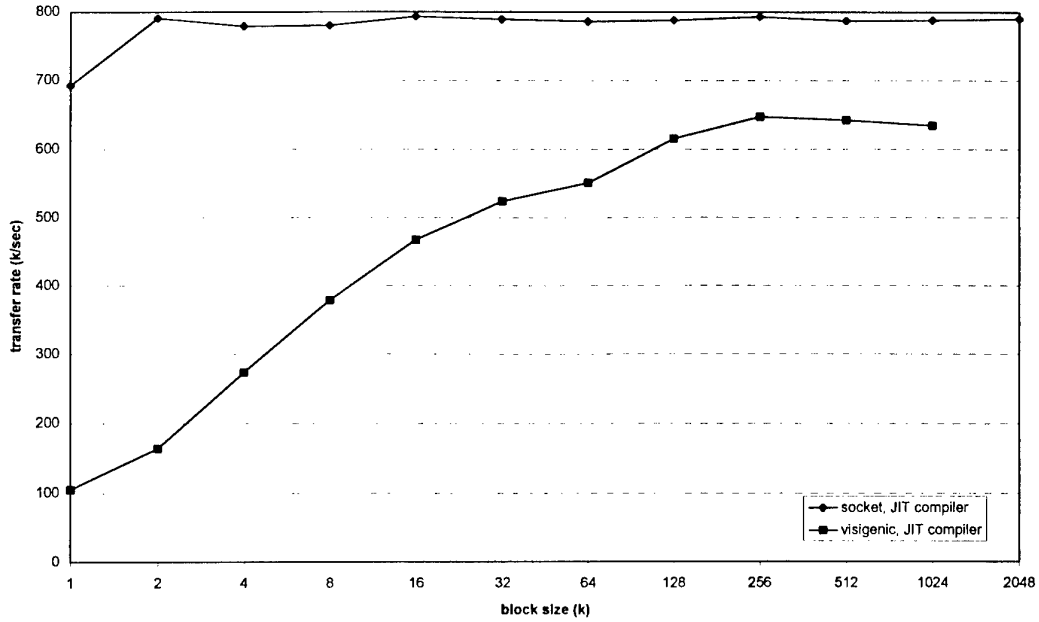

Figure 3: Network transfer rates from the PC to the SparcStation 10 over 10mbps ethernet

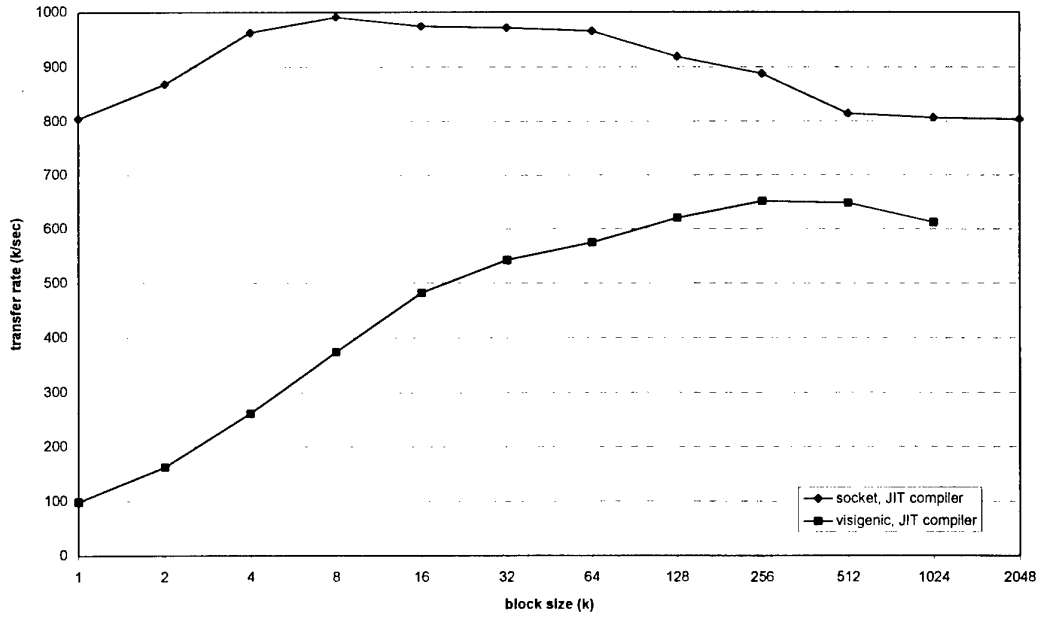

Figure 4: Network transfer rates from the SparcStation 10 to the PC over 10mbps ethernet

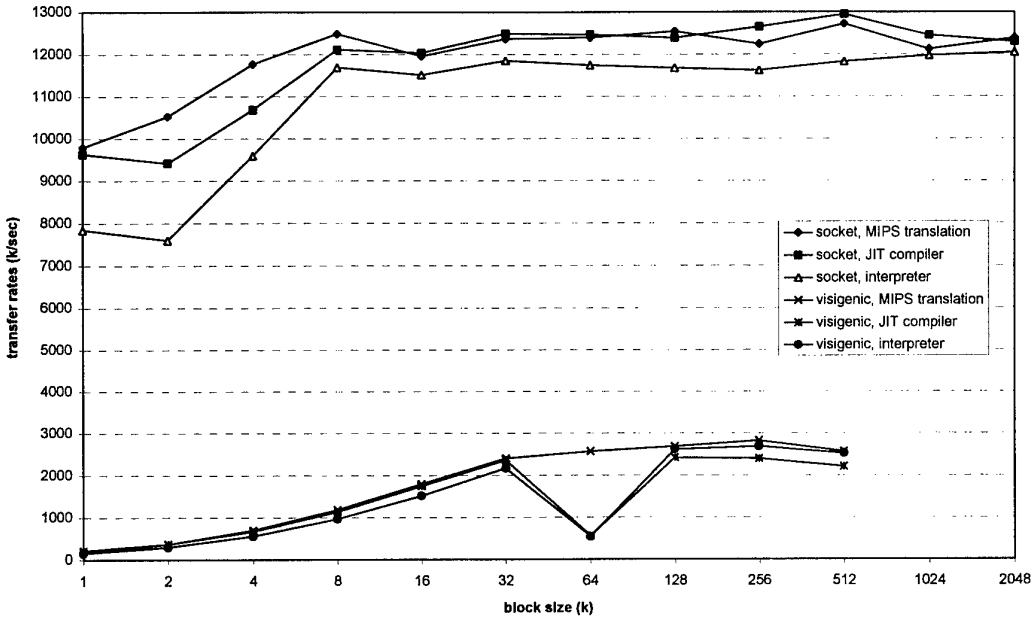

Figure 5: Loopback transfer rates for the Silicon Graphics O2

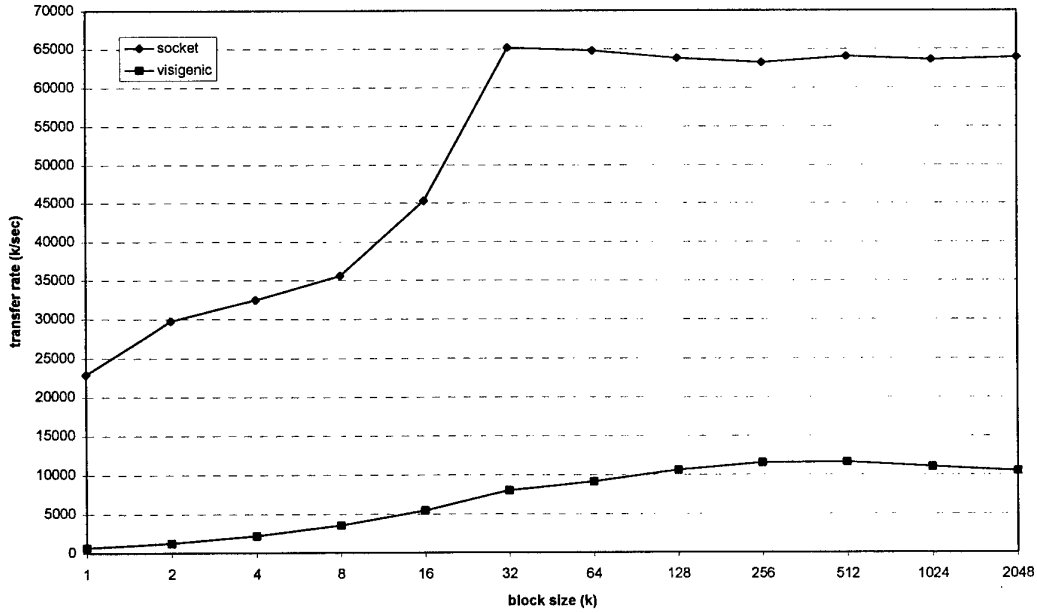

Figure 6: Loopback transfer rates for the Sun Enterprise E3000

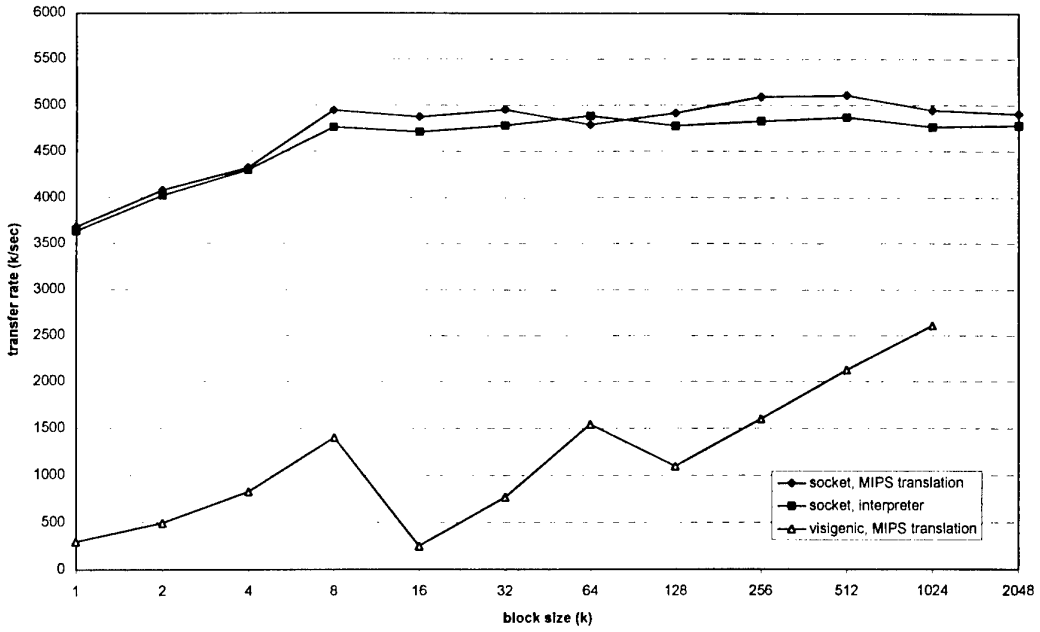

Figure 7: Network transfer rates from the SGI O2 to the E3000 over 100mbps Fast Ethernet.

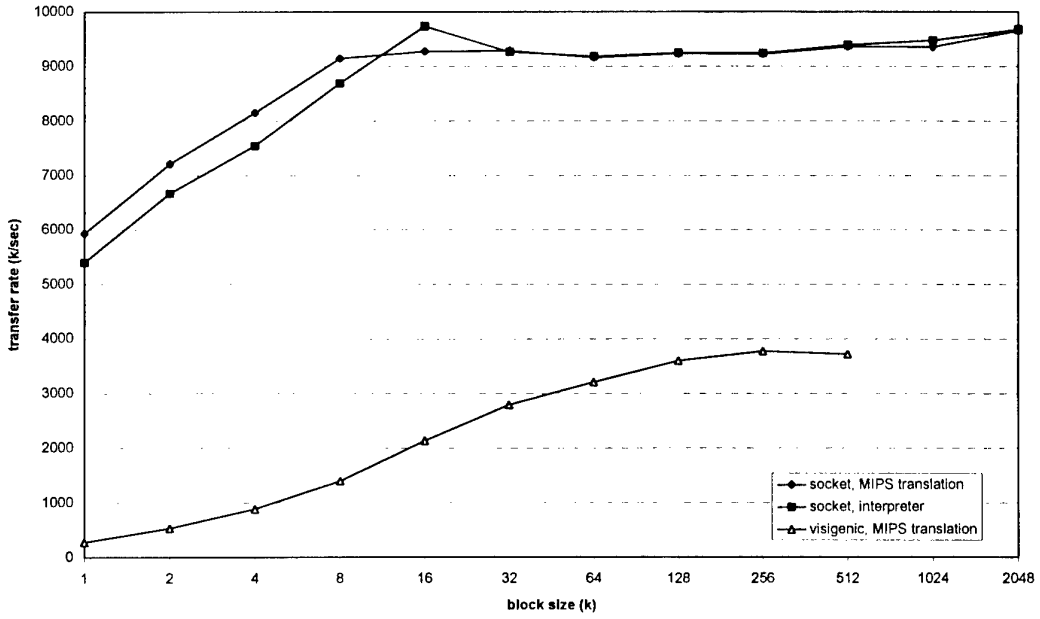

Figure 8: Network transfer rates from the E3000 to the SGI O2 over 100mbps Fast Ethernet.

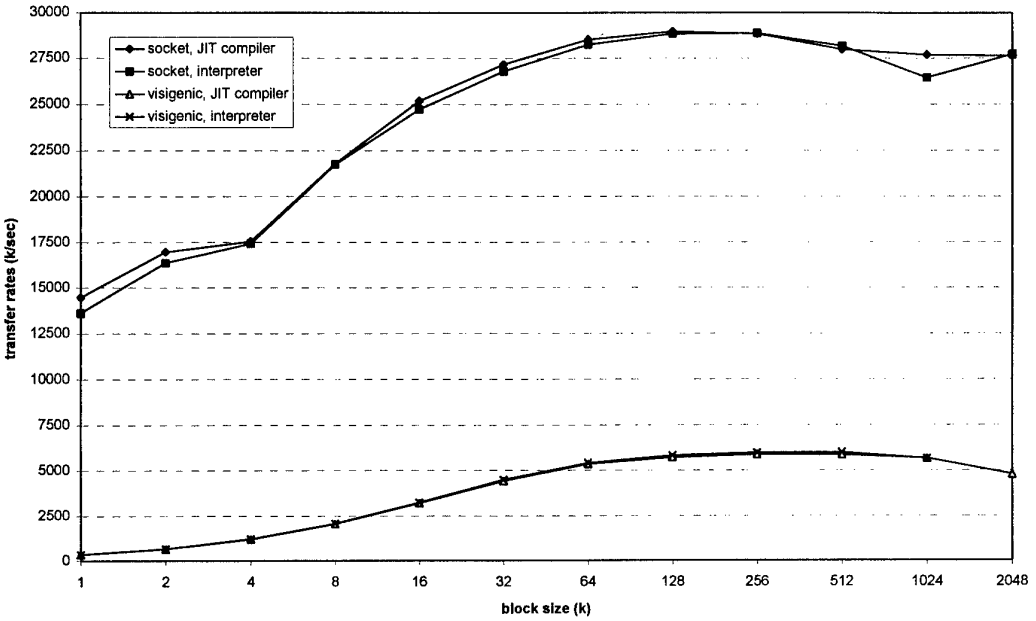

Figure 9: Loopback transfer rates for the Digital AlphaStation 600

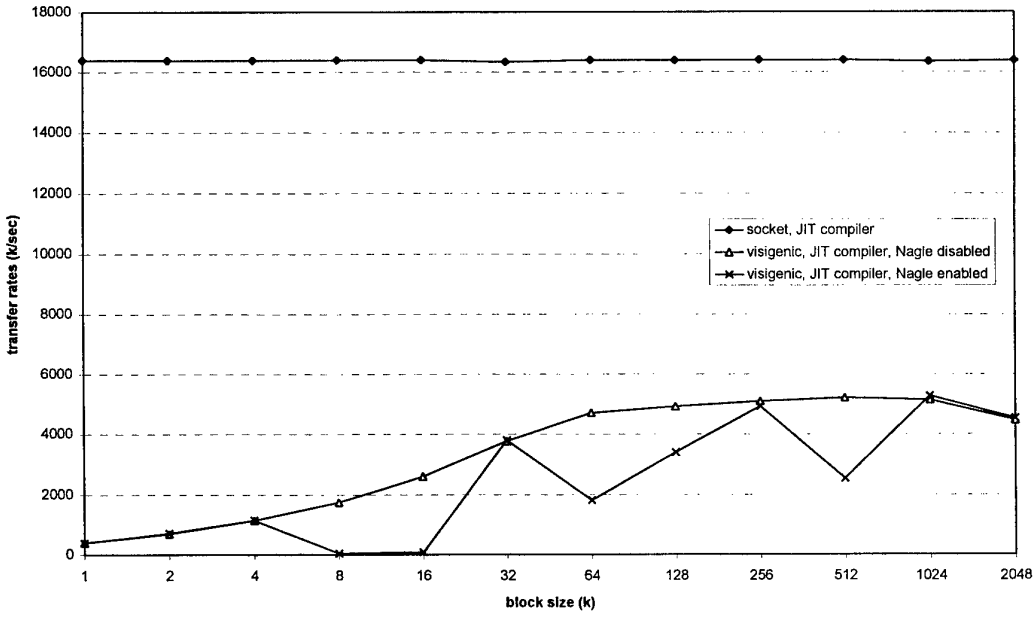

Figure 10: Network transfer rates between two Digital AlphaStations over 155mbps ATM

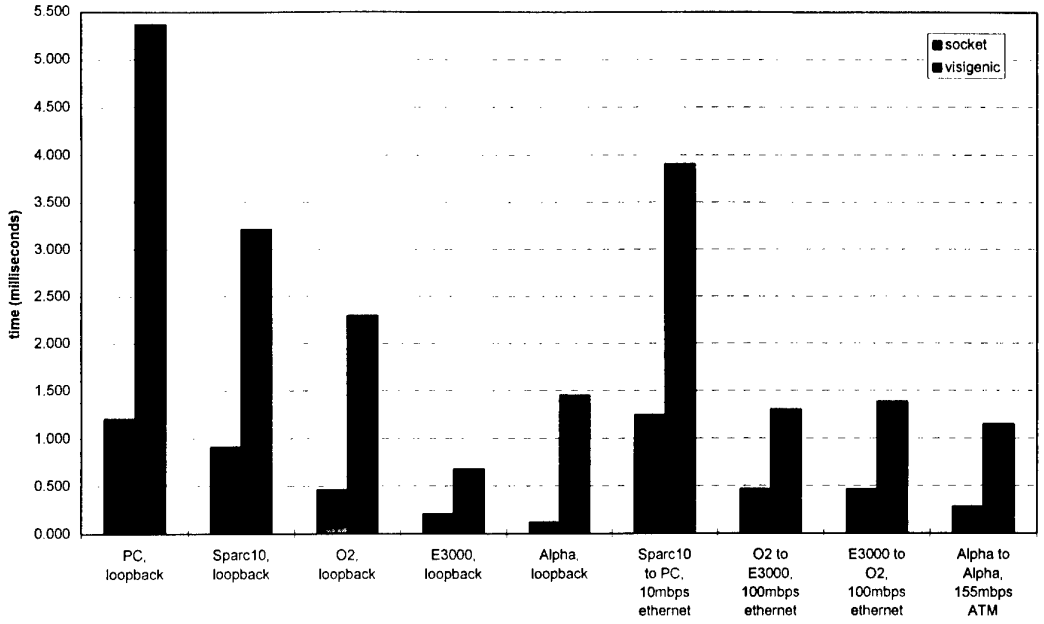

Figure 11: Latency times for the test machines and network configurations

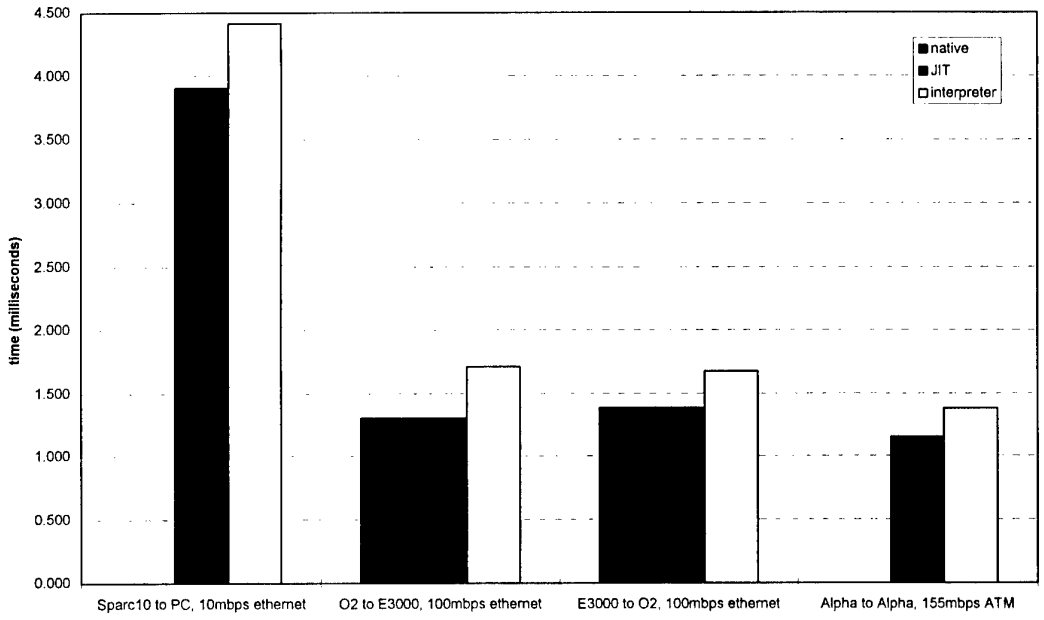

Figure 12: The effect of JVM optimisations on network latency for the ORB implementation

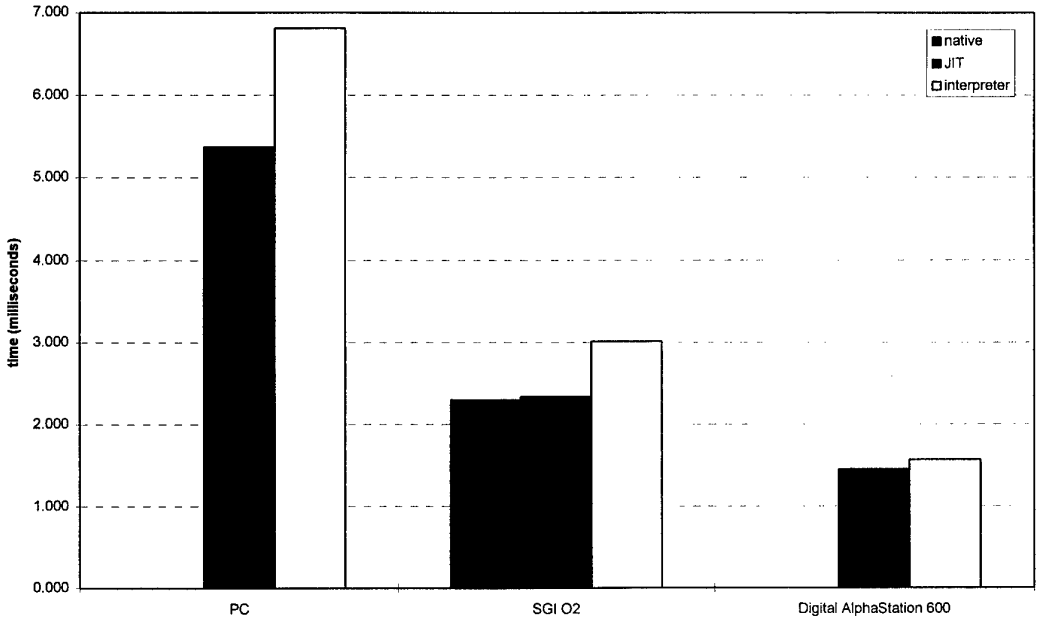

Figure 13 : The effect of JVM optimisations on loopback latency for the ORB implementation

DISTRIBUTION LIST

Performance Characteristics of a Java Object Request Broker
(DSTO-TR-0696)

Mr David Miron (DSTO)
Mr Sam Taylor (ACSys)

Number of Copies

AUSTRALIA

DEFENCE ORGANISATION

Task sponsor:

DGIO	1

S&T Program

Chief Defence Scientist)	
FAS Science Policy)	1 shared copy
AS Science Corporate Management)	
Director General Science Policy Development	1
Counsellor, Defence Science, London	Doc Control Sheet
Counsellor, Defence Science, Washington	Doc Control Sheet
Scientific Adviser to MRDC Thailand	Doc Control Sheet
Director General Scientific Advisers and Trials)	1 shared copy
Scientific Adviser - Policy and Command)	
Navy Scientific Adviser	1 copy of Doc Control Sheet and 1 distribution list
Scientific Adviser - Army	Doc Control Sheet and 1 distribution list
Air Force Scientific Adviser	1
Director Trials	1

Aeronautical & Maritime Research Laboratory

Director	1

Electronics and Surveillance Research Laboratory

Director	1
Chief Information Technology Division	1
Research Leader Command & Control and Intelligence Systems	1
Research Leader Military Computing Systems	1
Research Leader Command, Control and Communications	1
Executive Officer, Information Technology Division	Doc Control Sheet
Head, Information Architectures Group	1
Head, Information Warfare Studies Group	Doc Control Sheet
Head, Software Systems Engineering Group	Doc Control Sheet
Head, Year 2000 Project	Doc Control Sheet
Head, Trusted Computer Systems Group	Doc Control Sheet
Head, Advanced Computer Capabilities Group	Doc Control Sheet
Head, Computer Systems Architecture Group	Doc Control Sheet

Head, Systems Simulation and Assessment Group	Doc Control Sheet
Head, CCIS Interoperability Lab	Doc Control Sheet
Head Command Support Systems Group	1
Head, C3I Operational Analysis Group	Doc Control Sheet
Head Information Management and Fusion Group	1
Head, Human Systems Integration Group	Doc Control Sheet
Head, C2 Australian Theatre	1
Head, Intelligence Systems Group	1
Task Manager Dr Paul Whitbread	1
Authors: Mr David Miron (DSTO)	1
Mr Sam Taylor (ACSys)	1
Publications and Publicity Officer, ITD	1
DSTO Library and Archives	
Library Fishermens Bend	1
Library Maribyrnong	1
Library Salisbury	2
Australian Archives	1
Library, MOD, Pyrmont	Doc Control Sheet
Capability Development Division	
Director General Maritime Development	Doc Control Sheet
Director General Land Development	Doc Control Sheet
Director General C3I Development	Doc Control Sheet
Army	
ABCA Office, G-1-34, Russell Offices, Canberra	4
Intelligence Program	
DGSTA Defence Intelligence Organisation	1
Corporate Support Program (libraries)	
OIC TRS Defence Regional Library, Canberra	1
Officer in Charge, Document Exchange Centre (DEC)	Doc Cont Sheet & Distribution List
US Defence Technical Information Center,	2
UK Defence Research Information Centre,	2
Canada Defence Scientific Information Service,	1
NZ Defence Information Centre,	1
National Library of Australia,	1
Universities and Colleges	
Australian Defence Force Academy	1
Library	1
Head of Aerospace and Mechanical Engineering	1
Deakin University, Serials Section (M list)), Deakin University Library, Geelong, 3217	1
Senior Librarian, Hargrave Library, Monash University	1
Librarian, Flinders University	1

Other Organisations

NASA (Canberra)	1
AGPS	1
State Library of South Australia	1
Parliamentary Library, South Australia	1

OUTSIDE AUSTRALIA

Abstracting and Information Organisations

INSPEC: Acquisitions Section Institution of Electrical Engineers	1
Library, Chemical Abstracts Reference Service	1
Engineering Societies Library, US	1
Materials Information, Cambridge Scientific Abstracts	1
Documents Librarian, The Center for Research Libraries, US	1

Information Exchange Agreement Partners

Acquisitions Unit, Science Reference and Information Service, UK	1
Library - Exchange Desk, National Institute of Standards and Technology, US	1

SPARES 10

Total number of copies: 55

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)

2. TITLE

Performance Characteristics of a Java Object Request Broker

3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHOR(S)

David Miron and
Samuel Taylor

5. CORPORATE AUTHOR

Electronics and Surveillance Research Laboratory
PO Box 1500
Salisbury SA 5108

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO-TR-0696	AR-010-589	Technical Report	June 1998

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
N9505/15/108	DEF 95/074	DGIO	17	18

13. DOWNGRADING/DELIMITING INSTRUCTIONS

not applicable

14. RELEASE AUTHORITY

Chief, Information Technology Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE CENTRE, DIS NETWORK OFFICE, DEPT OF DEFENCE, CAMPBELL PARK OFFICES, CANBERRA ACT 2600

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CASUAL ANNOUNCEMENT Yes

18. DEFTEST DESCRIPTORS

CORBA (Computer architecture)
Object-oriented system architecture
Java
Data transfer

19. ABSTRACT

The efficiency of the Common Object Request Broker Architecture (CORBA) for the transfer of large files over a network is of particular interest to the Imagery Management and Dissemination Group (IMAD). The IMAD group will be using Java and CORBA for such transfers. This report studies the performance of a Java Object Request Broker (ORB) for the transfer of large files over such networks. This performance analysis is done using the Visigenics ORB Visibroker and involves the measurement of throughput and latency. These measurements are then compared with the results obtained when using socket to socket connections. The results show that the throughput of an ORB for large file transfer approaches that of sockets on low bandwidth networks. However on high bandwidth networks the throughput using the ORB is significantly less than that using sockets. It is also shown that the latency incurred by the ORB is much greater than that incurred using sockets.