8

Java in high performance networking applications

I. Azbel, A. Wynne, D. Cook, K. MacGregor
Department of Computer Science
University of Cape Town, South Africa
E-mail: {iazbel.awynne,dc,ken}@cs.uct.ac.za

Abstract

The acceptance of Java and its growth as a programming language is unparalleled. Since its inception, it has been used for a diverse range of applications. However the area that Java claims superiority over normal application development languages is that of networking. This paper describes the experiences gained in 8 man-years of development using Java in two networking applications. A detailed explanation of the experiences gained and the implementation issues are provided coupled with a discussion of problems encountered. A number of important issues have been discussed with regard to the performance of Java as well as its integration with C. The results indicate that Java can be used to implement high performance networking applications, however there are a number of issues with respect to the class libraries and the VM implementation which need to be addressed before it can be regarded as completely satisfactory for network applications.

Keywords

Java, video conferencing, Java C integration, tele-teaching, tele-medecine

1 INTRODUCTION

The use of Java as an application programming language has been widely publicised over the past two years. It is as a language for writing networked applications that Java distinguishes itself from standard application development

Systems Implementation 2000 R. Nigel Horspool (Ed.)

languages such as C and C++. Its security manager, the implementation with web browsers and the wide network class library all indicate that it should be the language of preference for networked applications. For this reason it was selected as the implementation language for two research projects being undertaken at the University of Cape Town (UCT), in the areas of tele-teaching and tele-medicine. In concept these applications can be regarded as similar, as they require collaborative communication between multiple networked systems.

The projects being undertaken commenced in February 1996 and were divided into two distinct phases. The first phase, undertaken during 1996, involved low bandwidth communications and was implemented purely in Java. This was a proof of concept of the maturity of Java as a network language and the development tools available for it. The second phase, undertaken during 1997, involved high bandwidth video-conferencing and required relatively high throughput necessitating the integration of Java and C.

The experiences discussed here cover both phases, but greater emphasis is given to the later experiences in high bandwidth applications using Java. In particular, the integration of Java and C on the client side; the performance of the client; the performance of the data distribution in Java (server-side issues); and a discussion of the results obtained. The server architecture will not be discussed, full details can be found in (Azbel and Wynne, 1997).

2 PROJECT SPECIFICATION

The project specification dictated that the workstation terminals should be low-cost (for widespread implementation), and the server architecture should allow for geographical dispersion of conference participants. For phase one of the project the network requirement was 64k wide area network links whereas for phase two relatively low bandwidth intra-network links (10 Mb/s Ethernet), and high bandwidth inter-network links (150 Mb/s ATM) were specified.

2.1 Hardware Platform

All experiments and measurements with regard to client-side capturing were made on a PC with 32MB RAM equipped with a Pentium 120 MHz processor, running Windows NT 4.0. The implementation uses both the Java JDK (version 1.1.4) and Visual C++ (V5.0) languages.

The choice of the video camera for the implementation was carefully considered. After evaluating several cameras on the market, it was decided that the Colour Quickcam from Connectix Corporation, was best suited to this project. It is comparatively cheap owing to the lack of dependency on a video capture board. The camera's design incorporates on-chip proprietary image compression (VIDEC) that consistently achieves between 3:1 and 4:1 video compression.

2.2 Low Bandwidth Communication

The first phase of the project implemented modules of the tele-teaching application. The concept was that a tutor situated at a central point could communicate with geographically separated remote students. Among the components implemented were: chat sessions either between the tutor and an individual student or between the tutor and entire group; shared whiteboards either between the tutor and an individual student or between the tutor and entire group; and a shared file viewer where the tutor could move the cursor to a line in the file and all the students cursors moved to the same line. The shared file viewer, together with the chat session, proved extremely useful in illustrating common program errors.

The entire system was written in Java, and runs in the Netscape browser. Native method calls were used to access local files for the file viewer.

This system was used by the computer graphics group at UCT to conduct tutorial sessions with students at other universities in South Africa. The results were extremely positive and the system proved that Java was indeed a language that could be used to implement network, and specifically Internet, applications. In addition, the performance of Java on a distribution server was adequate to achieve multi-user communication and specifically collaborative learning.

2.3 High bandwidth Communication

The application areas involved in the second phase not only required low bandwidth collaborative communication, but needed high bandwidth communication and, particularly, in the tele-medicine applications require real-time video conferencing. This ability for one or more remote collaborators to share common video and audio data would permit greater inter-personal contact and benefit both the medical diagnostic and tutorial learning systems. To achieve this, the client side of the system required the integration of Java to C as both the camera and audio system had C drivers. In addition as the requirements were for real-time audio and video, both the client side and the server side had to have a high performance. The methods of achieving this integration and the performance achieved are described in the following sections.

3 INTEGRATING JAVA AND C

The specification of this project demanded that both the client and the server be implemented in Java. One of the key issues of a Java implementation is the fact that Java abstracts itself away from hardware. In order to communicate with hardware (in this case, the video camera and microphone), Java allows for *native methods* that are written in C. The interface to these native methods varies in the differing implementations of Java. (V1.0 has the Native method Interface (NMI) and V1.1 has the Java Native Interface (JNI).)

In this implementation, media conferencing is achieved through capturing media data (sound and graphics) in C, transferring the data to Java, distributing it over a network in Java, and playing it back in C (see Figure 1).

In order to capture media data, a *C call-back loop* is initiated. This means that a single Java thread sets up a C loop that captures both sound and image data simultaneously. When a unit of media is captured (either a 4KB sound buffer or a frame of video data), the C program calls a predefined media transmission function. These functions then copy the captured data to a Java array and call back into Java to distribute it.

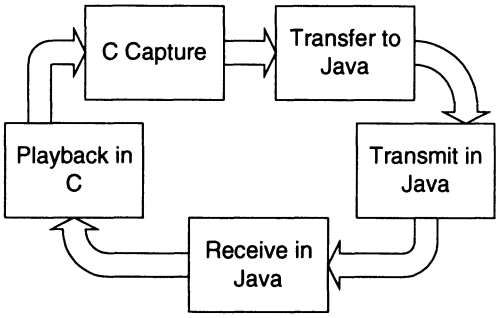

Figure 1 The Java media capture cycle

4 EVALUATION CRITERIA OF NATIVE METHOD INTERFACE

In order to evaluate which native method interface to implement, a number of key issues that arose from the needs of this project were:

- *Performance of call-backs.* As video conferencing is a real-time application, the interface must have a fast method of calling from C back to Java.
- *Interfacing to Java arrays.* The captured data had to be transferred to Java arrays in order to be distributed over a network in Java.

4.1 The Java Native Interface

The JNI is the latest version of a native method interface. It was first implemented in Java 1.1, and presents a totally different way of interfacing Java and C, from the earlier NMI. A major advantage of this interface is that it allows automated transfer of data between C and Java. In addition it is generally more stable, and affords the developer more control in interfacing Java and C.

A major problem with this interface (and the reason for not using it) was its method for calling back from C into Java. As the call-back function is invoked from C when the device drivers finish capturing a media unit, it means that the C code is not explicitly called from Java, but rather the other way around. The JNI

does not handle this scenario particularly well since it has provision for making a global reference to the class block (class identifier) only.

When a Java call-back is made from C, a classblock and methodblock must be supplied to a call-back function. These identify the desired class (and desired method in that class) to call. A global classblock reference means the returned classblock can be used at any time, at any place in the code. The problem is that one can't do the same for a methodblock. This means that in the specific implementation, the video capture and audio capture call-back functions (in C), had to repeatedly look up the methodblock in order to call back into Java. This incurred a substantial performance overhead for every frame (and sound buffer).

4.2 The Native Method Interface

The native methods interface selected for this project was the NMI (JDK1.02) for the performance reasons given above. Whilst the interfacing to Java arrays was somewhat less elegant than the JNI, the degree of optimisation possible warranted this approach. This was achieved by finding the classblocks and methodblocks only once, upon initialisation. Methodblocks were determined by searching the classblock's methodtable, until a match on a hash of the method name and signature was found. This reduced significantly the overhead in the call-back procedure and resulted in a highly optimised Java call-back routine.

There is an undocumented change in the way JDK1.1.4 handles the NMI, as opposed to the JDK1.0.2. The classblock returned when a *FindClass* method is called has a handle to it in JDK1.1.4 (thus one must "unhand" it), whereas there is no handle in JDK1.0.2.

It should also be noted too, that very little data exists on optimising the NMI. However a published optimisation method (Descarte 1996) was adopted, and its reliability improved in finding method blocks for all types of functions (static and instance functions).

5 INTEGRATION PROBLEMS

When writing to a Java array in C, and then calling back from C to Java (to transmit the data in this case), the Java data member must be declared static for its value to be read by Java afterwards. This seems to be a contradiction, as native methods cannot access static Java data members (by definition).

This problem was circumvented by wrapping the call-back Java class in another class. This parent class instantiates a static object of this native class. The C code was then amended, to call back to the wrapper class, rather than the native class. This allowed the wrapper Java class to access the captured data and distribute it. The fact that the media capture class is statically instantiated is immaterial, as only one capture object is ever instantiated for each workstation.

5.1 Breaking the Paradigm

As stated previously, the project is implemented on an Intel-based PC platform for cost reasons. On this platform, the native method library is manifested in the form of a DLL file (Dynamically Linked Library). The methods contained therein are all static to the classes to which they pertain. This means that when many threads use the same methods, all the threads share the local variables in this method. This goes directly against the Object Oriented approach of Java. Every instance of the class using the native method should have its own set of data, and it should be transparent to the other threads (as is the case with non-native methods).

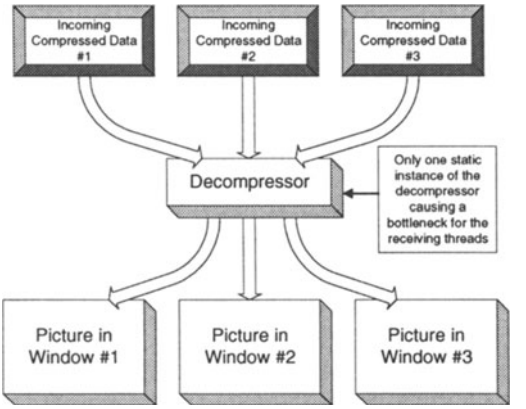

Figure 2 *The static decompressor problem*

This problem manifests itself when the threads that are receiving compressed video data are competing for the use of the decompressor in the native method (see Figure 2). To get around this problem, C semaphores were used, thereby forcing other threads to block until the semaphore cleared. This caused a bottleneck in the semaphored code between decompression threads.

Conceptually an ideal solution to this would be to have multiple copies of the DLL in memory. That is, a copy for each thread that needs to store data in the DLL variables. This would conform to object oriented programming techniques where all objects are self-contained.

6 ADVANTAGES OF USING JAVA

Java has some distinct advantages with regards to development time when compared to C. The diversity of the JDK APIs helped significantly in data distribution. Data distribution was easily facilitated through the Java Networking API. To provide additional functionality, this set of classes was extended to include such features as logging and exception handling.

Another major advantage of writing the client shell in Java is the threading capability of the Java Virtual Machine. The capturing, distributing and playing back routines are all separated into threads. This allows the individual thread priorities of the various sections of the program to be explicitly set (thus audio playback can be given preference over video playback).

For performance reasons, the picture playback eventually had to be coded in C. It was felt that the additional functionality afforded by the Java AWT (Abstract Windows Toolkit) should be maintained in the image receiving windows. To facilitate this, the drawing routine was fooled into drawing directly onto a Java window. This means that normal Java AWT components can be added to the image receive window, thereby adhering to the Java structure. Hence all user-initiated events can still be handled with the ease of Java.

7 JAVA CLIENT PERFORMANCE

In a video conferencing system, client performance counts for a great deal of the functionality of the system. One of the major accusations levelled at Java is its lack of performance. This project shows that the bottleneck of the system lies with the image capturing speed (camera dependant) and not with its distribution or playback. Various aspects of the Java/C integration impact on the client-side performance: the colour models, hardware architecture problems and array access.

7.1 Java Colour Models

In Java, the colour models are used to determine the pixel colour data from raw byte arrays. The models are structured for 8 and 24 bit colour only. In order to display a 16 bit colour image (the default colour depth of the cameras), each 16 bit pixel had to be aligned (in C) to 32 bit boundaries, thus doubling the space needed for the image. In order to accomplish this, a loop had to be iterated for every frame. The result was a marked lag before image playback. The loop was directly related to the image size, and thus slowed larger images proportionately.

7.2 Architecture Problems

Sun Microsystems developed the Java language and specification. Sun hardware is big-endian (van der Linden, 1997), alluding to the convention for memory ordering. When a Java picture playback method was attempted, the resultant image was inverted and upside down.

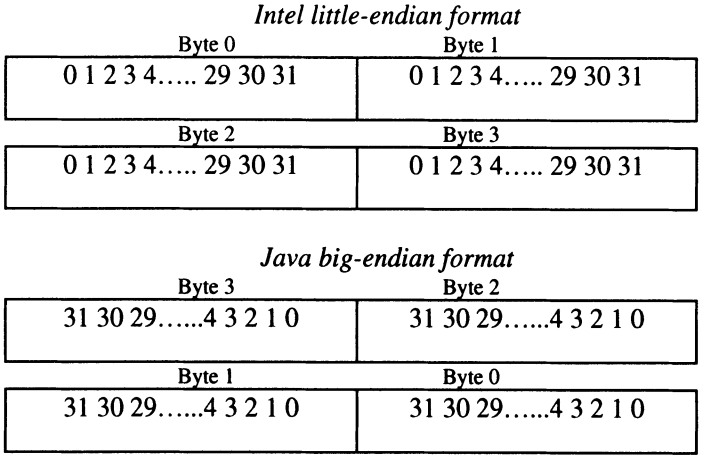

Figure 3: *The different ways of storing pixel data*

It was deduced that this was a result of the little-endian architecture of the Intel-based (see Figure 3) workstation using Java's big-endian image reconstruction methods. The image redrawing was also unacceptably slow (regardless of image orientation).

To get around the image orientation problem, the 32-bit spacing loop (see *Colour Models* above) was modified to reverse the image as well. This achieved the desired result, but the overall image playback performance was slow.

7.3 JIT Performance Issues

Near the beginning of the project development, Sun announced the release of the Windows Performance Pack. This manifested in a JIT (Just in Time) compiler that ran at run-time, compiling segments of code on-the-fly to increase the overall performance of the Java code. The result was an increase in the execution speed by a large margin. This seems to be the way to make Java's performance competitive: a super-optimised JIT compiler for each platform. Image capture speed did not increase, but image playback saw a marked performance increase.

7.4 Accessing Arrays

Another major issue, when integrating Java and C for a project such as this, is accessing and copying of arrays. It is vital that this be well implemented, because byte arrays form the containers of all captured and received image and audio data.

This issue mainly pertains to the efficient operation of the videoconference capture cycle (see Figure 1). Ideally, once the video and audio data has been captured, a Java array should be made to point to the captured data for UDP datagram construction. Unfortunately, Java has no concept of pointers, and thus the captured data needed to be copied to the Java array for every frame. This was

implemented using a highly optimised array-copying function. Despite the entire frame array being copied to the Java array, this still proved to be faster than the capturing rate, as there was no noticeable degradation in frame rate.

7.5 Client Performance Tests

In order to understand how much load the client is handling, 6 tests were done and their results analysed. These 6 tests took the form of a video conference with 2 people, then 3 people, each being tested at low quality video, medium quality video and high quality video. A client preview window has a maximum frame rate of 19.4 fps – this would be the maximum theoretical frame rate attainable from a workstation. The average resulting frame rates over a period of time were recorded. Figures 4, 5, 6 and 7 display all the results.

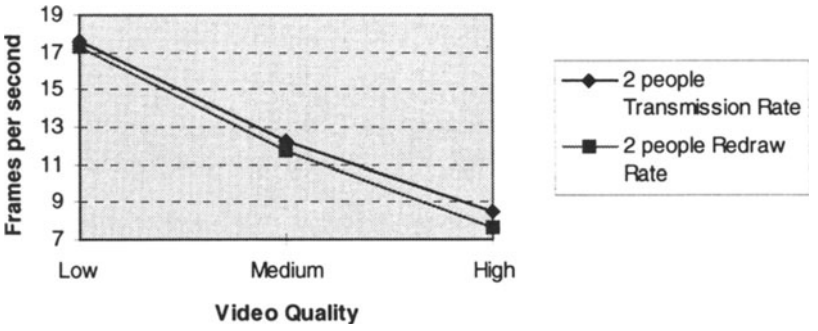

Figure 4 *Sending and redrawing rates in a 2 person conference.*

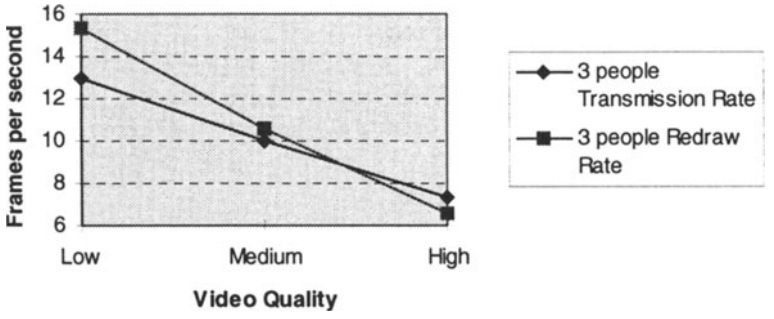

Figure 5 *Sending and redrawing rates in a 3 person conference.*

Figures 4 and 5 show how the transmission and redraw rates change during a 2 and 3 person conference. The difference in frame rate between low quality and high quality video is substantial, enough to make the difference between a usable

and unusable conference. It was decided that, at around 7 frames per second, a conferencing system cannot be regarded as 'usable'. It is therefore noted immediately that video conferencing on mid-range PCs, such as the ones utilised here for testing, cannot be successful using high quality video images. Frame rates between 16 and 17 frames per second seem reasonable and it was felt that this performance is acceptable for interaction between people.

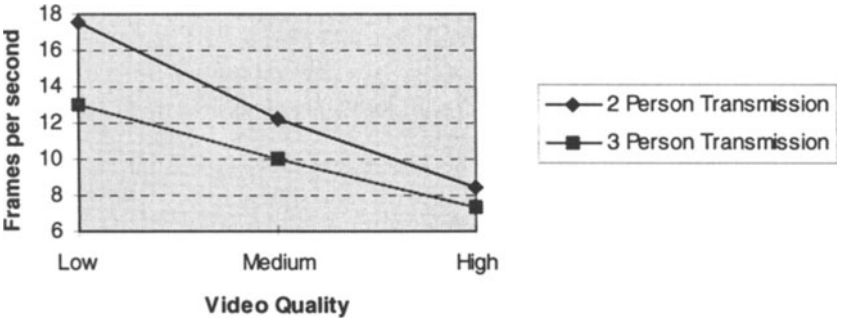

Figure 6 *Comparison between the sending rates in a 2 and 3 person conference*

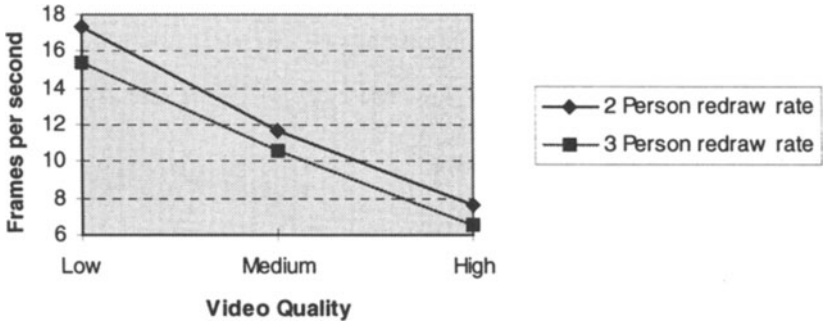

Figure 7 *Comparison between the redraw rates in a 2 and 3 person conference*

In Figure 7, the difference in the redraw rates seems to remain stable throughout the entire quality range, but Figure 6 shows us that the image transmission rate differences are substantial, particularly in the low quality range where call-backs from C to Java happen more frequently.

8 DATA DISTRIBUTION IN JAVA

Two major experiments were done on Java to test its data distribution potential. One was a test that simply involved sending of data, while the other involved creation, allocation and distribution of data. Both tests made use of multicast data distribution techniques (Mbone 97; Byne et al., 1997).

It was noted that, when viewing the bytecode of a compiled Java class that uses datagrams, a security check is done at every sending and receiving routine. It is felt that this is reasonable when small traffic is expected, but for larger traffic, such as produced in video conferencing, this is extremely inefficient. Because of this inefficiency, the Sun Java Development Kit (JDK) was edited to remove all security calls done at sending and receiving routines of datagrams. For example the send method in the MulticastSocket class in the network class library was amended as follows:

```
public synchronized void send(DatagramPacket p, byte ttl)
        throws IOException {

// Security manager makes sure that the multicast address
// is allowed one and that the ttl used is less than the
// allowed maxttl.
/*SecurityManager security = System.getSecurityManager();
  if (security != null) {
    if (p.getAddress().isMulticastAddress()) {
      security.checkMulticast(p.getAddress(), ttl);
    } else {
   security.checkConnect(p.getAddress().getHostAddress(),
      p.getPort());
    }
  }
*/
  byte dttl = getTTL();
  if (ttl != dttl) {
       impl.setTTL(ttl); // set the ttl
  }
  impl.send(p);      // call the datagram method to send
  if (ttl != dttl) {
    impl.setTTL(dttl); // set it back to default
  }
}
}
```

With this performance issue in mind, these two major tests, were each split into two subsets; one with all datagram security code included in the classes, and one test in which all security code was removed from the datagram classes.

One further enhancement was made before running the tests. All bytecodes produced by the test program were analysed and edited to make them as efficient as possible. This was done with the Java decompiler *javap* that is bundled with

Sun's JDK. As many stack operations as possible were removed by making as many variables as possible local inside loops. It was felt that while this might be a very small adjustment, it was necessary to get the greatest performance possible from the test.

Test 1: Creation, Allocation and distribution

This test involved allocating memory (with a *new* statement), looping though the byte array that was just allocated and assigning dummy data to each element and finally sending the data in a datagram packet. With this test, Java's ability to produce its own data and send it is tested, instead of having it rely on external intervention (such as native C calls) to create and assign data to datagram packets.

Test 2: Distribution

In this test, it is assumed that a byte array already exists and simply needs to be sent using a multicast. This test is an important test to see how Java may perform as a media distribution tool, for example in a server architecture, where it would be responsible for redistribution of multimedia to remote destinations (Azbel and Wynne, 1997).

Test results

The results from tests conducted are displayed in Figure 8.

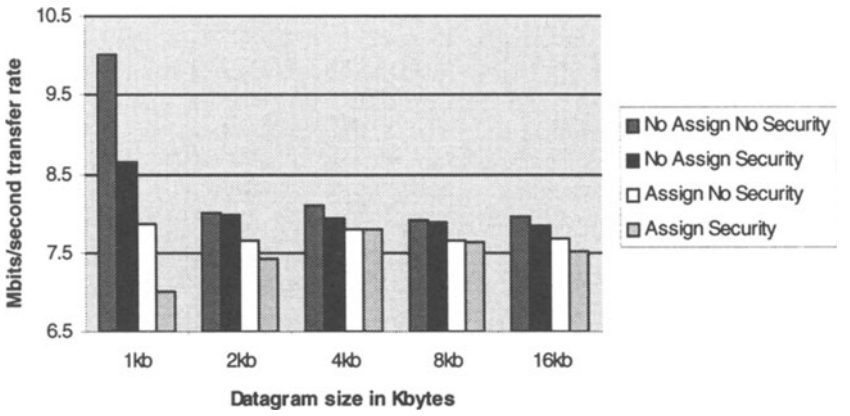

Figure 8 *Java media distribution test results*

Before any discussion of these test results, it should be noted that the results for 1Kbyte packets are not accurate. In fact, the maximum bandwidth for a 10 Mbit line is approximately 8.8 Mbits/s. It is also not known why such varied speeds

were attained between the four tests — Java seems to be very erratic for small datagram sizes, sending the datagrams in bursts, rather than in a constant flow.

Because of all the enhancements to the test program, it is believed that the above results are the maximum throughput of Java without the use of a JIT compiler. No tests were done using a JIT compiler because this would not be symbolic of the performance of Java itself, but rather a test of the performance of the platform and computer on which it is being tested. It is thought, however, that Java will not have any difficulty in sending data at the full 8.8 Mbits/s rate when run through a JIT compiler.

One can immediately see how both assigning data in a loop and the security mechanisms in Java slow down the performance of the data distribution. As expected, with no security and no assignment routines, Java performed optimally, but was gradually slowed as security routines were added and finally assignment loops were integrated.

We can, however, note that the degradation of speed in the four tests is very large for smaller packet sizes, and seems to stabilise for larger packet sizes. Java seems to perform relatively well at 4 Kbyte and 8 Kbyte packet sizes, with all four tests being within 0.3 Mbits/s of each other.

An important observation that should be noted is the performance gain attained by removing all security calls in the sending and receiving functions in the *DatagramSocket* and *MulticastSocket* classes. For high volumes of data, as in Video conferencing, such a performance gain could be critical to the general performance of the system, and therefore critical to the extent of its usage.

The tests described above are to test different uses of Java. The first test (creation, allocation and distribution) tests Java's ability to create its own data and sent it. It is, however, not pertinent to this project because all arrays are allocated in C with native methods. It does, however, give us an indication of the abilities of Java as a utiliser of high bandwidth networks. In this case, Java does not rate very highly. It should be noted that in a real environment, multiple workstations that are using Java clients could be transmitting at the above rates and therefore utilise the given bandwidth to capacity.

The second test (distribution only) is more pertinent for the primary use of Java, both on the client side, where datagrams are sent only (C is used to fill the packets), and at the server side where datagrams are distributed only. At no stage does Java do any type of data allocation or assignment. Here, Java performs relatively well, having the ability to redistribute data at more than 8 Mbits/s. This indicates that Java is capable of data redistribution on a small scale.

9 CONCLUSION

In conclusion the experience over the past two years have shown that network applications can be easily written in Java. Where pure Java applications with relatively low bandwidth requirements are required, Java provides acceptable performance. Even when high performance collaborative networking is required

Java can still be of use for data distribution. However certain issues require to be addressed.

The overhead of the security manager calls in UDP communications needs to be investigated. The concept of joining a session with an initial call to the security manager, and then only transmitting data without incurring the overhead of a call to the security manager at each send should be implemented in the class libraries. This concept of a trusted client needs to be introduced. The need of every implementation to modify, and distribute, differing copies of the base class libraries will lead to many non-standard implementations of what should be standard class libraries.

In addition, the Java VM lacks the ability to thread processes on multiple processors on multiprocessor machines and therefore limits itself in terms of performance when redistributing multiple streams of data, as is necessary in video conferencing. Future implementations of the VM should make effective use of multiprocessor architectures.

Considering client-side performance, Java can be regarded as having sufficient performance to capture, distribute and playback image and sound data in real-time. It was found that faster processors significantly improve performance.

Experience shows that a pure C implementation would be better (no conversion or integration problems), but the increase in performance attained would not be great enough to warrant the extra implementation effort.

10 REFERENCES

Azbel, I., and Wynne, A. (1997). Impact of Network Bandwidth on Video and Audio conferencing. *Proceedings of Teletraffic '97 Conference.*

Descarte, A. (1996). URL: http://www.hermetica.com.

van der Linden, P., (1997). Java Programmers FAQ 97.

JavaSoft (1996). *The Java Developer's Kit 1.02.*
URL: http://www.javasoft.com/products/jdk/1.0.2/.

JavaSoft (1997). *The Java Developer's Kit 1.1.4.*
URL: http://www.javasoft.com/products/jdk/1.1/.

Mbone (1997). URL: http://www.mbone.com.

Byne, M. , Courtny, A., Felton, E., Hurst, L., and O'Sullivan, B. (1997). An Introduction to IP Multicast.
URL: http://ganges.cs.tcd.ie/4ba2/multicast/index.html.