Performance evaluation of a Java-based networking Application Programming Interface (API)

S. Zeadally
Department of Computer Science, Wayne State University, Detroit, MI 48202, USA

Received 7 June 1999; received in revised form 29 May 2000
Communicated by F. Dehne

Abstract
Over the last few years, we have witnessed the emergence of many network-based multimedia applications, particularly those rich in audio, video, and images. Access to network services has become an integral component of these applications. In this paper, we present a Java-based networking Application Programming Interface (API), and compare the performance of our Java-based API with Microsoft Winsock 2 API over different protocols namely, TCP/IP, UDP/IP and native Asynchronous Transfer Mode (ATM). From the results obtained, we observe that the Java layer increases the overall user-to-user latency by about 120 microseconds compared to native Winsock 2 over ATM local area networks. We also found that jitter performance for continuous media applications, such as those involving digital video is highly non-deterministic. We obtained large variations in video jitter over Java, with maximum jitter in the range 6 to 12 milliseconds. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: API; ATM; Jitter; Latency; Performance evaluation

1. Motivation
Recent advances in enabling technologies such as networking, imaging, signal processing, databases, hardware and operating systems have created the opportunity for many types of multimedia applications ranging from telemedicine to distance learning. The use of multiple media (text, images, audio, video) is becoming increasingly common in emerging desktop applications. High-speed networks are also playing a fundamental role in the deployment of these applications. Advances in communications now enable the storage, transmission and retrieval of multimedia information both locally and remotely. An important factor which determines the successful deployment of these emerging network-based multimedia applications is the API used to develop them.

The Integrated Media Systems Center (IMSC) [5] at the University of Southern California is an engineering research center sponsored by the National Science Foundation. IMSC is currently conducting cross-disciplinary research in the area of multimedia and creative technologies in collaboration with industry participation. At IMSC, we found that when integrating efforts from various research groups (audio, database, computer interface, education), a significant amount of time is spent by application developers in these different groups on developing networking code needed to give networking access to their applications. Typical difficulties encountered during integra-
tion of projects from various groups include: the different operating systems used with different networking APIs (BSD socket on UNIX platforms, Winsock on Windows NT). These conventional APIs are also sometimes difficult to use by application developers with minimal network programming experience. For instance, the database or audio application developers spend a lot of time on network programming in order to perform common networking operations such as transmitting and receiving data from the underlying network. Since the most common, basic, networking operations include establishing network connections, transmit/receive data over the network and closing down connections, we therefore designed and implemented a Portable Networking API (henceforth called PNAPI) that provides these core functions to both UNIX and NT C/C++ applications developers. In order to provide platform independence to application developers, PNAPI exploits Java.

In this paper, the focus is on evaluating the performance of PNAPI in supporting networked multimedia applications. We report on performance results based on Quality of Service (QoS) parameters such as latency and jitter. In order to better assess the real performance benefits and limitations of PNAPI, we carefully conducted performance comparison experiments with other well-known, standard APIs such as Winsock 2 and Java. Our performance results demonstrate the impact that technologies such as Java has on the performance of the PNAPI implementation in the delivery of high end-to-end performance for continuous media applications such as those involving digital video. The remainder of this paper is organized as follows. Section 2 briefly presents the design and implementation of PNAPI. In Section 3, we present round-trip latency results. Section 4 compares video jitter performance over PNAPI and Microsoft Winsock 2 [11] networking API using different protocols over an ATM [14] local area network. Finally, in Section 5, we summarize our contributions and present future work.

2. The PNAPI design and implementation

A full description of the design and implementation of PNAPI is beyond the scope of this paper and has already been covered in [17]. However, in order to understand the impact of PNAPI and the various software layers including Winsock 2 and Java, on end-to-end application performance, we therefore present the architecture of PNAPI briefly in this section. In designing PNAPI, we were careful not to re-invent conventional networking APIs such as BSD sockets and Winsock 2 APIs on UNIX and Windows platforms, respectively. The main goals and motivations behind the design of PNAPI include:

- **Simple and easy to use**: this will speed up code development for application developers who will not have to delve into networking programming details (e.g., socket bind or QoS details in the case of QoS-based networks such as ATM in order to access network services).
- **Portable**: this allows PNAPI to be used by developers on both Windows NT and UNIX platforms. It is worth noting that Java architecture does offer Java socket classes which are simple to use and portable but these benefits are only available to Java application developers. However, most programmers nowadays still use C or C++ in their applications. So, we implemented PNAPI to provide a C-like interface over Java sockets. PNAPI provides a runtime library of networking functions and classes that can be easily used by both C/C++ and Java programmers in their applications.
- **QoS support**: Although networking APIs such as XTI (on UNIX systems) and Winsock 2 (Windows systems) currently do support QoS over ATM networks, the task of setting up native ATM network connections that exploit QoS on an end-to-end basis is complex due to the number of networking parameters needed. These can be difficult for someone who is not familiar with ATM. But with PNAPI providing a simple networking API with QoS support, the end-user simply performs QoS requests using parameters such as cell loss, latency, jitter, and network bandwidth. However, it is worthwhile noting that with current commercial ATM device drivers, only end-to-end network bandwidth reservation is possible via ATM switches.

Fig. 1 illustrates the software architecture of PNAPI over Java on Windows NT systems. Although PNAPI also works on UNIX systems, due to space limitations, in this paper we restrict our discussions to Windows NT systems only. A performance comparison of PNAPI on Windows NT and UNIX systems is presented in [16]. As mentioned previously, the
Fig. 1. This figure illustrates the PNAPI architecture over Java on Windows NT.

design and implementation of PNAPI is discussed in [17]. In this paper our main focus is on the performance evaluation of PNAPI in supporting low-latency, delay-sensitive, multimedia applications such as those involving continuous media such as audio and video. As shown in Fig. 1, PNAPI is implemented over Java which in turn invokes native Winsock 2 calls for networking functions on Windows NT. Our PNAPI implementation supports applications with and without QoS requirements. It is worthwhile noting that PNAPI supports QoS by exploiting Winsock 2 API (which allows QoS requirements to be specified for networks such as ATM) at a lower level. For IP-based networks, this is achieved via an resource ReSerVation Protocol (RSVP) [1] service provider and for ATM networks via the ATM service provider as shown in Fig. 1. These service providers are kernel-mode, layered drivers. The ATM service provider maps Winsock 2 calls to specific functions of the underlying ATM device driver. The RSVP service provider manages QoS requests on IP flows between IP applications and the underlying IP network. In the case of ATM, both the ATM service provider and the underlying ATM device driver are vendor specific. However, this does not matter for applications that use the Winsock 2 API. In the current PNAPI implementation we have extended Java to support QoS over ATM networks only (via native ATM). As a result, QoS support for multimedia applications using PNAPI is restricted only to native ATM applications. Other applications such as those running over TCP-UDP/IP use Classical IP [8] over ATM (Fig. 1). Thus, PNAPI allows applications with QoS requirements to coexist side by side with legacy TCP-UDP/IP applications. Work is under way to extend Java so that we can also provide QoS for IP applications running over non-ATM networks. The APIs in Table 1 remains unchanged to support RSVP since the user merely specifies another protocol parameter type (RSVP) when using the Server_Socket_Create and Client_Socket_Create API functions.

1 This is true only if different vendor service providers conform to the standard Winsock 2 Service Provider Interface (SPI) (Fig. 1).
Table 1
This table described the PNAPI functions implemented

<table>
<thead>
<tr>
<th>Java-based networking API functions</th>
<th>API description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNAPI_Init()</td>
<td>Initialize the Java virtual machine</td>
</tr>
<tr>
<td>Server_Socket_Create(char *address, int port, int backlog, int type)</td>
<td>Create a server socket using protocol type TCP, UDP, or ATM</td>
</tr>
<tr>
<td>Server_Socket_Transmit(int sock, char *ToAddr, int port, char *buf, int len)</td>
<td>Transmit data of size len bytes</td>
</tr>
<tr>
<td>Server_Socket_Receive(int sock, char *buf, int len)</td>
<td>Receive data of size len bytes</td>
</tr>
<tr>
<td>Server_Socket_Close(int sock)</td>
<td>Close server socket</td>
</tr>
<tr>
<td>Client_Socket_Create(char *address, int port, int type, QOS *qos)</td>
<td>Create a client socket using protocol type TCP, UDP or ATM with QoS</td>
</tr>
<tr>
<td>Client_Socket_Transmit(int sock, char *ToAddr, int port, char *buf, int len)</td>
<td>Transmit data of size len bytes</td>
</tr>
<tr>
<td>Client_Socket_Receive(int sock, char *buf, int len)</td>
<td>Receive data of size len bytes</td>
</tr>
<tr>
<td>Client_Socket_Close(int sock)</td>
<td>Close a client socket</td>
</tr>
</tbody>
</table>

3. Experimental testbed environment and measurement procedures

Our ATM local area network consists of Pentium II 400 Personal Computers (PCs) all of which are connected to an IBM 8265 ATM switch via multimode fiber. All performance tests were conducted between a pair of machines each of which was configured as follows: Windows NT 4.0 operating system, 256 MB of Random Access Memory (RAM), one 4 gigabyte disk drive, an ATM adapter from Efficient Networks Inc. which supports ATM connections up to 155.52 Mbits/s. Our ATM software and hardware support both Classical IP [8] and native ATM (over AAL5) protocols. It is worth noting that we use Visual Cafe 2.5a from Symantec Corporation [10] and Just-in-Time (JIT) [3,10] compilation in all Java experiments described below.

We are particularly interested in the overheads introduced by the software layers (PNAPI, Java, and Winsock 2) on QoS parameters such as round-trip user application latency and jitter. For our latency measurements, we used raw data and measured the variation of round-trip latency with message size over ATM. We repeated the experiment for three different user applications based on namely, Winsock 2, Java, and PNAPI as shown in Fig. 2.

Jitter (the variation in the delays experienced by packets transmitted over the network) is an important QoS parameter for continuous media applications. Real-time delivery of continuous media such as audio and video needs strict timing requirements to be met by the underlying network and end-system to ensure smooth delivery of audio and video to the end-user. It is imperative that jitter be kept within bounded
limits to ensure smooth audio and video delivery. Otherwise, we experience video flickering and audio distortions [6].

Recent works on Java performance have focused mainly on comparisons between Java runtime environments such as JIT compilation and static native compiled Java [7,2]. These research efforts have demonstrated far superior performance (usually by about an order of magnitude [3]) obtained with JIT and static native compilations compared to purely interpreted Java. Other performance studies on Java have used mainly traditional benchmarks such as Spec95 or CaffeineMark [9] (a series of tests that measure the speed of Java programs running in various hardware and software configurations). However, few results to date have been reported on Java performance for continuous media applications. In [2], jitter performance results are reported using a simulated MPEG video stream. Results were not satisfactory: Mark’s [2] results show that Java achieves only one third of the video frame rate (10 frames/second) of that obtained with a C++ runtime system and the video jitter obtained was also much lower in the latter case. However, given the number of assumptions made in these simulation experiments, it is hard to capture the fundamental interactions that take place in a real system. Jitter results reported in this work differ from those in [2] in that we demonstrate Java performance using a real multimedia application namely, real-time video conferencing. We used a video conferencing system we have designed and implemented at the University of Southern California. Our video conferencing system exploits low-cost technologies (e.g., video frame grabber, commodity PCs) to deliver live, uncompressed video at different resolutions over TCP/IP, UDP/IP and native ATM via PNAPI (and hence over Java and Winsock 2). A more detailed description of the system is given in [15]. In all the jitter performance tests conducted, we used a video frame size of 320 × 240 pixels (at 16-bit per pixel).

Using a continuous medium such as digital video in all our experiments allows us to verify claims about Java delivering low video frame rates and also enables us to experimentally explore Java performance in delivering live, uncompressed, digital video to desktop users.

We measured round-trip user latency and jitter as follows:

- **Latency**: We developed a program that echoes a message of a specified size between the two machines on the ATM network. 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. The end-to-end communication latency between user applications can be estimated as half the total round-trip time calculated for the M-byte message.

- **Jitter**: In all tests described below, we measured jitter as follows: we timestamp each incoming video frame. The **received interval** is the time between the receipt of an entire video frame (last packet in a frame) and the receipt of the next entire frame. We computed the received intervals for a sufficiently large number of video frames (i.e., by running the live video over a long period of time). The average received interval was then calculated. We then computed the jitter as the difference between the average received interval and each individual received interval between consecutive video frames. The mean value of all the jitter values for all video frames was then calculated. We also recorded minimum, maximum, and standard deviation jitter values.

4. Performance results

End-to-end application performance depends on many factors such as CPU speed, protocol stack, operating system, device driver, network interface, and the underlying network. In this section we discuss performance results obtained with PNAPI, Java, and Winsock 2 using protocols such as TCP-UDP/IP and native ATM. Our results demonstrate in particular the impact of PNAPI (and therefore Java) on QoS parameters such as round-trip latency and jitter which are important characteristics of low-latency, delay-sensitive multimedia applications.
4.1. Latency

To investigate the impact of latency on our PNAPI design, we performed the following three tests (as depicted in Fig. 2) on round-trip latencies for protocols such as TCP/IP, UDP/IP over ATM, and native ATM using Winsock 2, Java (which runs over Winsock 2), and PNAPI (which runs over Java and Winsock 2). The goal in these experiments is to determine how much overheads the different software layers add to end-to-end user latency and as a result enables us to investigate the performance costs incurred in achieving the simplicity and portability benefits of PNAPI and Java. We make the following observations from the graphs shown in Fig. 3:

- The user-to-user round-trip latency measured is composed of two parts: a fixed cost incurred by any message and is independent of message size used; the variable latency portion depends on the size of the message (since operating system overheads such as data copying will increase with larger messages) being delivered or received. The round-trip latencies for TCP/IP and UDP/IP over ATM for small messages (4–64 bytes) are around 330, 450, and 500 microseconds over Winsock 2, Java, and PNAPI, respectively. These are fixed latencies incurred by any message due to the software layers (e.g., Java, operating system, network interface) between the network and the application. As shown in Fig. 3, the overall round-trip latency increases almost linearly with increasing message sizes for all protocols over PNAPI.

- It is not surprising that for all graphs shown in Fig. 3, latency is highest over PNAPI (since a message traverses more software layers) and lowest for Winsock 2. However, it is interesting to note that round-trip latency over Java is very close to that on Winsock 2 and this is fairly consistent for TCP, UDP, and native ATM. We observe an average latency difference of about 120 microseconds between Winsock 2 and Java for the message sizes tested. The average latency difference observed between Winsock 2 and PNAPI is around 220 microseconds. The major difference in latency is due to the extra data copies incurred by both Java and PNAPI. When data is transferred between layers implemented in C/C++ and Java, a data copy is incurred. For instance, to transfer data between Winsock 2 (C/C++ code) and Java, the Java Virtual Machine (JVM) performs an explicit data copy via the Java Native Interface (JNI) [13]. Similarly, a user application (C/C++) exploiting PNAPI also makes another explicit data copy between PNAPI
Fig. 4. This figure illustrates the use of JNI between the different layers for a C/C++ application exploiting Java. The Java-based API used by Java applications is also depicted.

and Java. With Java and PNAPI, a network application incurs one and two additional copies respectively compared to Winsock 2 as illustrated in Fig. 5. These additional data copies explain the latency differences obtained over Java and PNAPI. Thus, the portability Java offers, the simplicity, QoS and platform independence that PNAPI provides are achieved at the expense of additional data copies and higher round-trip latency. The latencies over Java and PNAPI are 36.3 and 66%, respectively higher than over native Winsock 2 for the message sizes tested.

It is also worthwhile noting that from the graphs shown in Fig. 3, the difference in round-trip latencies between PNAPI and Java is greater than the difference between Winsock 2 and Java although as mentioned above there is only one extra data copy also between Java and PNAPI. In order to explain this result, one has to understand the use of JNI at different software layers. As illustrated in Fig. 4, JNI is used at two places: first, between PNAPI and Java; second, between Java and the implementation of native methods (e.g., Winsock 2). A plausible explanation which accounts for the higher difference in latency between PNAPI and Java than between Java and Winsock 2 is probably because in the former case, PNAPI invokes Java methods which result in the execution of “slower” interpreted bytecodes. This is in contrast to the use of JNI between JVM and Winsock 2 where JVM directly invokes natively compiled Winsock 2 libraries resulting in faster code execution.

• From Fig. 3(c), we observe higher latencies over native ATM than TCP or UDP. This is not quite what we would expect in theory: a lower latency over native ATM than TCP-UDP/IP since there are no protocols between the ATM device driver and the user application (Fig. 1). One feasible explanation for the unexpected higher latencies over native ATM is probably due to the overheads introduced by the ATM Service Provider. We plan further experiments in the future to confirm this interpretation.

4.2. Jitter

In our jitter experiments, we receive and display live, uncompressed video from the network using the two configurations shown in Fig. 5(a) and 5(b). For both configurations, live video is received over Winsock 2 and displayed by the C++ video application using native Win32 drawing routines. The average video frame rate observed was 22 frames per second (an inter-frame arrival time of 45.45 milliseconds). This is more than twice the frame rate (10 frames/second) obtained in [2] where we believe
that the low frame rate reported was due to software decompression rather than Java.

We used Win32 drawing routines to display the live video received from the network rather than Java drawing routines (e.g., those provided by the Java Media API[12,4]) for both configurations. This is because we are particularly interested in the impact that Java and PNAPI stacks have on end-to-end video jitter performance. This explains why in our test configurations we isolated any potential differences that may arise due to display performance by Java.

Based on the two configurations (Fig. 5(a) and 5(b)) used in our jitter experiments, we argue that the real difference in jitter performance is caused mainly by the Java layer depicted in Fig. 4 rather than the PNAPI layer. Our explanations for supporting this argument are as follows: in Fig. 4 we clearly show what we mean by the Java layer between the native method (Winsock 2 in this case) and the C/C++ application using PNAPI. As Fig. 4 illustrates, the Java layer includes both JNI interfaces and also part of the PNAPI layer since the principal C functions in PNAPI responsible for the transmission and reception of network data only invoke Java methods to achieve these tasks. Thus, the PNAPI layer is really a lightweight layer that simply provides a bridge between C/C++ programs and Java. In other words, the PNAPI functions that send and receive network data are just C-like wrappers around core Java methods that are invoked to perform data transfers between a C/C++ application and Java. It must be emphasized that these core Java methods used by PNAPI for sending and receiving network data are part of JNI and is the reason why a portion of the PNAPI layer is included as part of the Java layer shown in Fig. 4. In short, the bulk of code in PNAPI is really exploiting Java and is the reason why we argue that any difference in jitter performance obtained between the two configurations shown in Fig. 5 is really caused by the Java layer rather than the PNAPI.

The results for the configuration shown in Fig. 5(a) (over Winsock 2) are given in Table 2. From Table 2, jitter over UDP/IP is about two times better than over TCP/IP. We obtained lowest jitter over native ATM and is about two times better (i.e., lower) than over UDP/IP. Moreover, maximum jitter over native ATM is almost six times better than TCP/IP. Fig. 6 illustrates the variation of jitter for consecutive video frames over a period of time. It is clear from

<table>
<thead>
<tr>
<th>Winsock 2</th>
<th>TCP/IP</th>
<th>UDP/IP</th>
<th>Native ATM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum (microseconds)</td>
<td>0.14</td>
<td>0.14</td>
<td>1.04</td>
</tr>
<tr>
<td>Maximum (microseconds)</td>
<td>12317.86</td>
<td>3341.14</td>
<td>1878.04</td>
</tr>
<tr>
<td>Mean (microseconds)</td>
<td>554.51</td>
<td>266.92</td>
<td>148.26</td>
</tr>
<tr>
<td>Standard Deviation (microseconds)</td>
<td>1035.28</td>
<td>303.43</td>
<td>190.71</td>
</tr>
</tbody>
</table>

Fig. 6 that native ATM gives considerably better jitter performance than either TCP/IP or UDP/IP over ATM implementations.

The jitter results over PNAPI are given in Table 3 and Fig. 7. It is clear from Table 3 that the average jitter over all the protocols are higher than those obtained above with Winsock 2. The average jitter (554 microseconds) for native ATM in Table 7 is almost four times higher than that over Winsock 2 (148 microseconds)
Fig. 6. This figure shows the jitter experienced by consecutive video frames over TCP/IP, UDP/IP, native ATM using Winsock 2.

Maximum jitter values for UDP/IP and native ATM were much higher compared to TCP/IP: almost two and seven times higher for UDP/IP and native ATM, respectively over PNAPI compared to Winsock 2. Fig. 7 shows the large PNAPI jitter variations obtained over PNAPI compared to Winsock 2 for all the protocols tested. Thus, jitter for digital video is much higher over PNAPI (which uses Java) and is another performance degradation we incur as a result of the flexibility, simplicity, and portability of PNAPI and Java.
5. Conclusions and future work

We summarize the main contributions of this paper and our performance experiences with the PNAPI design and Java below:

- In this paper, we have discussed the design and performance of a portable networking API (PNAPI) based on Java. The implementation of PNAPI provides functionalities beyond those provided by the current Java networking API: PNAPI allows Java and non-Java-based programs (i.e., C/C++) to access networking services in a platform independent way and is easy to use by application developers. The simplicity of PNAPI will speed up code development by application developers (on UNIX and NT systems) in other areas (e.g., database, education) who do not have to worry about low-level networking programming details as is currently the case with conventional APIs such as Winsock 2 and BSD sockets. Another important feature of PNAPI is that it provides QoS support to end-user applications. However, at this stage, PNAPI only supports one QoS parameter namely, end-to-end network bandwidth reservation. PNAPI users specify the bandwidth required when a connection is set up and using ATM signaling to the switches and routers, the requested bandwidth is reserved along the entire path from the source to the destination. Other QoS parameters such as latency, jitter, and cell loss are not supported yet (e.g., commercial ATM network adapters and switches do not support them) but are planned in the future.
- We have explored the performance of PNAPI for protocols such as TCP-UDP/IP over ATM and native ATM. We showed the impact of PNAPI on application latency and jitter. PNAPI gives highest latency with the protocols tested compared to Winsock 2 due to additional data copies (two copies) incurred with Java. With PNAPI, two additional data copies are incurred between network and user applications in addition to the normal two data copies (network to kernel, kernel to user) with conventional APIs such as Winsock 2 and BSD. As mentioned previously, these extra data copies constitute partly the price we pay for the portability and simplicity provided by PNAPI and Java.
- We obtained higher round-trip latencies with native ATM compared to TCP/IP and UDP/IP over ATM. We speculate that this is probably due to the overheads introduced by the underlying ATM device driver and the ATM service provider. In addition, we found that fixed cost latency for user applications over TCP-UDP/IP and native ATM is still quite high (around 350 to 500 microseconds even for small messages). We obtained 36.3 and 66% higher latencies over Java and PNAPI, respectively compared to native Winsock 2 API. This paper systematically evaluates the common claim that “Java is slow at the networking layer”, and demonstrates that the claim is false. The latency results obtained show that the Java layer adds only about 120 microseconds delay (using a 400 MHz Pentium II machine) on top of native code.
- We explored experimentally the jitter performance of live, uncompressed video with PNAPI and Winsock 2 over ATM using PNAPI and Winsock 2 APIs. We obtained much higher video frame rates with Java compared to that was reported in [2]. However, we found that Java leads to increased jitter for digital video. For instance, over native ATM average jitter over PNAPI is almost four times higher than over Winsock 2. Maximum jitter values obtained were also much worse (e.g., about seven times higher over native ATM) over PNAPI compared to Winsock 2. We argue that the high jitter over PNAPI is due to the execution of the Java Virtual Machine and the extra data copies incurred between the user application and the network: one between Winsock 2 and Java, and another between Java and PNAPI.
- It is worthwhile noting that PNAPI also works on UNIX systems. However, in this paper, due to space limitations, performance results are restricted to Windows NT systems only. In [16], we present performance comparisons of PNAPI and Java on UNIX and NT systems. Another area we are also currently working on is the extension of PNAPI and Java classes to support QoS over non-ATM networks by exploiting protocol such as RSVP.

Finally, to conclude, we believe that PNAPI provides a simple, flexible, QoS-aware API suitable for networked applications including those with QoS requirements. Unlike other networking APIs such as Winsock 2 and BSD Sockets which are tied to the underlying operating system, PNAPI provides a platform independent API and achieves so by exploiting Java
(using optimizations such as JITs). However, as our experimental results demonstrate, these benefits come at a cost. As noted above, PNAPI incurs higher overheads (for instance data copying, since there are more software layers to traverse between the network and the application) which result in higher latencies and increased jitter mainly due to the Java layer. We believe that with better Java compilers, faster JVMs, JIT compilers, and possibly Java chips, PNAPI is likely to yield better performance results. However, how much impact PNAPI will have on application developers in the future remains to be seen although at IMSC we have already started to deploy PNAPI in a number of collaborative multimedia projects.

Acknowledgements

The author thanks Xuan Chen and Weiyou Cui for their help in all the experiments described in this work. The author also expresses his gratitude to the anonymous reviewers whose suggestions and comments greatly improved the quality of this paper. This work was supported by NSF grants EEC-9529152 (IMSC ERC) and MRI-9724567. We also gratefully acknowledge grant EDUD-7824-000145-US from Sun Microsystems Inc., Palo Alto, California. Equipment donations from the Intel Equipment Grant Program and the IBM SUR Grant are also acknowledged.

References


*Java chips will be able execute bytecodes directly without the need for interpreting or compiling.*


