Implementation and Performance of QoS-aware Java Applications over ATM Networks

S. ZEADALLY

High-Speed Networking Laboratory, Department of Computer Science, Wayne State University, Detroit, MI 48202, USA
Email: zeadally@cs.wayne.edu

In recent years, we have witnessed the emergence of different types of multimedia applications, particularly those involving continuous media such as digital audio and video. Improvements in hardware, software and networking technologies are enabling a wide deployment of these applications, each with its own Quality of Service (QoS) requirements. Along with these technological advances, Java has emerged as a powerful platform for the development of many desktop applications. In this paper, we first present a review of popular QoS-aware standard networking APIs available to developers on both UNIX and Windows systems. We then discuss the design and implementation extensions we made to the Java architecture to support QoS over native Asynchronous Transfer Mode (ATM) networks. Our implementation offers Java application developers the ability to specify their QoS requirements over ATM networks and simultaneously takes advantage of platform independence provided by Java. In addition, we discuss practical performance experiences obtained with our design in supporting digital video applications running over ATM networks.

Received 16 March 2000; revised 17 July 2000

1. MOTIVATION

Over the past few years, networks such as ATM [1], capable of providing Quality of Service (QoS) guarantees have become available. In order to fully exploit the QoS characteristics of such networks, it is important for networked applications to be able to specify their QoS requirements. Networked applications can only do so via some communication Application Programming Interface (API) that supports QoS (also known as QoS-aware communication APIs) [2].

ATM has emerged as one of the most compelling high-speed networking technologies due to the benefits it provides in terms of speed, scalability, isochronous and QoS functionalities. A distinctive feature of ATM is its ability to transport different data types with different QoS requirements in a unified way on a single platform. Moreover, the isochronous characteristics of ATM enable the support of real-time multimedia applications (e.g. video conferencing, distant learning, teleworking) and other time-sensitive applications. The standard cell format used by ATM provides seamless integration of Local Area Networks (LANs) and Wide Area Networks (WANs). All these benefits of ATM coupled with the decreasing costs of ATM switching hardware and network interface adapters are pushing towards the adoption of ATM in both LAN and WAN environments.

Java’s current popularity because of its platform independence on a wide range of platforms (e.g. Windows, OS/2, Linux) has led to its deployment in a wide range of computing environments. Some of the commonly cited benefits of Java include: its object-oriented language allows modular design and re-use of objects; the fact that it is architecture independent which allows cross-platform portability without recompilation or porting; and its support for dynamic loading of components over local and wide area networks. Two basic types of Java programs are around—applets and applications. Applets are class files written in Java and compiled using the Java compiler. Java applets are normally called in an HTML script by, for example, a Web page by a Web browser. This causes the applet to be downloaded from a remote server and executed locally to provide dynamic functions such as animations.

Our goal in this work is to combine the QoS benefits of networks such as ATM with the advantages of platform-independence that Java provides. This will enable the deployment of platform-independent, QoS-aware applications which only need to be written once and reused on different platforms without recompiling or porting. In order to achieve this objective, we need to implement applications using Java; second, we need to extend the Java networking API to support QoS. This requires extensions to be made to Java in order to support QoS provided by low-level native APIs (e.g. Winsock 2, XTI) supported by the underlying operating system. In this work, we demonstrate that by combining Java and ATM technology
supports the underlying QoS needed by applications), we enable the deployment of platform-independent, QoS-aware applications.

The rest of the paper is organized as follows. In Section 2, we review two most popular QoS-aware standard networking APIs available today. Section 3 discusses the implementation extensions to Java to support QoS using native ATM. In Section 4, we present and discuss performance results of a Java-based, multimedia application that runs over different communication protocols. Finally, in Section 5, we summarize our contributions and make some concluding remarks.

2. A REVIEW OF QoS-AWARE NETWORKING APIs

QoS support for multimedia applications has received a lot of attention recently. It is widely accepted that, to achieve true end-to-end QoS, it is necessary to provide support at all levels including network, end-system (network interface, operating system) and application. The different levels of QoS support required have led to different types of QoS architectures and implementations [3] in recent years. Although these architectures demonstrate the notion of QoS support to different extents, these implementations often use proprietary APIs. As a result, it becomes difficult to port applications developed for these systems to other similar architectures and networks. It is imperative to have QoS-aware APIs in order to successfully deploy portable QoS-based applications. A few promising QoS APIs have emerged recently and they show the possibility of accelerating the deployment of future QoS-aware multimedia applications. Standard communication APIs which support QoS include native ATM and the Resource ReSerVation Protocol (RSVP) [4]. RSVP is a proposed Internet standard defined by the Internet Engineering Task Force (IETF). It is a resource reservation set-up protocol designed for providing integrated services on the Internet. It provides support for reservation of resources for simplex, multicast and unicast flows. An application uses the RSVP protocol to identify a data flow and to request a specific QoS from the network. Under RSVP, resource reservation requests are made by data from the receivers. Moreover, with RSVP, reservation requests are handled separately from the data transfer. For a given RSVP session, a sender describes its traffic characteristics using a traffic description (called Tspec) and actual reservations initiated by the receiver are made using flow specifications which specify the desired QoS and filter specifications which specify which packets are to receive the requested QoS.

2.1. Windows systems

The networking API available on Windows systems is Winsock 2 which specifies a generic QoS interface that is independent of the underlying QoS service provider (e.g. ATM, RSVP, ST-2) and is useful for writing applications that do not depend on particular QoS features of the underlying service provider (Figure 1). Winsock 2 calls are mapped to the underlying device driver calls via a kernel-mode service provider which is a kernel-mode layered driver between Winsock 2 and the underlying device driver. In the case of native ATM, an ATM service provider is needed. For RSVP-based applications, an RSVP service provider is needed.

2.1.1. Native ATM

To really exploit the QoS characteristics of native ATM, we need to implement ATM-aware applications (henceforth referred to as native ATM applications) that access native ATM networking features in a platform-independent way. Winsock 2 API specifies how native ATM applications can be implemented on Windows systems and takes advantage
of features such as QoS guarantees and point-to-multipoint connections. The Winsock 2 API is compatible with ATM Forum’s adopted ATM API semantic model description [5], which describes native ATM services independently of any programming language and operating system to the application developer at the user side of the ATM User Network Interface (UNI) [6].

The QoS structure used by Winsock 2 has three elements namely SendingFlowspec, ReceivingFlowspec and Provider Specific fields. The SendingFlowSpec and ReceivingFlowSpec comes from the flow specification (‘flow spec’) proposed in [7]. Flow specs describe a set of characteristics about unidirectional flows through the network. Flow specs indicate what level of service is required and provide a feedback mechanism for applications to use in adapting to changes in networking conditions. The flow specs used by Winsock 2 include QoS characteristics such as source traffic description, latency (delay and delay variation) and service type (best-effort, controlled-load and guaranteed service). The ProviderSpecific field has two elements, a buffer and a length field. This buffer allows the flow spec to be extended to include provider specific parameters. For instance, the protocol-specific QoS structure used by ATM is contained in the ProviderSpecific buffer field of the Winsock 2 QoS structure. The ATMP protocol specific QoS structure is a concatenation of Q.2931 Information Element described in [8, 9]. It is worth noting that if both flow specs and the ATM-specific QoS structure are specified, the value in the latter takes precedence over the flow specs. With Winsock 2, an application establishes its QoS requirements at connection set-up time, and they can be explicitly modified later using the appropriate system calls.

2.1.2. RSVP

As mentioned above, Winsock 2 also supports the RSVP protocol as illustrated in Figure 1. An application uses a socket in the conventional way (using the protocol identifier IPPROTO_JP) for transferring data but all resource reservation transactions must be carried out on a separate, non-blocking RSVP socket (created using the protocol identifier IPPROTO_RSVSP). With Winsock 2, the RSVP API consists of a set of RSVP-specific control codes (called dwIoControlCode parameter) that are used in specific system calls on RSVP sockets. Examples of such dwIoControlCode include RSVP_REGISTER which allows a sender or receiver to create an RSVP session, RSVP_SENDER (allowing a sender to specify its traffic descriptions) and RSVP_RESERVE (allowing a receiver to initiate resource reservations), and other calls which are given in [7].

2.2. UNIX systems

2.2.1. Native ATM

Native ATM API deployment on UNIX platforms has been slower than Windows systems. However, recent work by X/Open Consortium led to ATM extensions made to the X/Open [10] Transport Interface (XTI) network programming interface, which is a standard part of X/Open compliant UNIX systems from companies such as HP, Sun, IBM, SCO and others. ATM connection attributes can be negotiated for a connection using network transport calls such as t_connect, t_accept, t_listen and so on. The XTI interface, which is also used by MacOS has been extended to provide a native ATM interface on Macintosh computers by Lin et al. [11]. Native ATM extensions made to XSockets (the X/Open version of BSD sockets) are defined in [12].

2.2.2. RSVP

The only published API (called RAPI [13]) for RSVP on UNIX systems is that defined by ISI [14] and Sun Microsystems Inc. which is used in their RSVP implementations. Unfortunately, the RSVP implementation is only currently available on SunOS and Solaris on Sun architectures and as a result this limits the scope for wide deployment on other UNIX platforms such as HP-UX, Linux, Digital UNIX and others. A Java application exploiting QoS-aware communication APIs over native ATM and RSVP uses the software architecture illustrated in Figure 1 on UNIX platforms.

3. QoS IMPLEMENTATION FOR JAVA APPLICATIONS OVER NATIVE ATM

In order for Java applications to exploit ATM, the underlying platform must support ATM communications technology. This requires ATM API, protocol and device driver support. On UNIX systems, API extensions have been made to XTI sockets to support native ATM and on Windows systems (e.g. Windows NT), native ATM support is provided via the standard Winsock 2 API (Figure 1).

Java applications can be provided with ATM transport capability using the most common solution adopted today, namely an IP-over-ATM solution which includes implementations such as Classical IP (CLIP) LAN Emulation (LANE) and Multi-Protocol over ATM (MPOA). The major benefit of an IP-over-ATM solution is that Java applications based on TCP/IP or UDP/IP protocols run unmodified over such implementations and existing APIs for protocols such as TCP/IP and UDP/IP can be used to access the underlying ATM network via the IP protocol stack. However, the IP-over-ATM approach really provides user applications (Java applications based on TCP/IP and UDP/IP protocols) access only to ATM features such as high bandwidth and low delay. However, these approaches do not allow explicit QoS requirements to be specified by end-user applications.

To give Java applications full access to ATM capabilities (including native ATM support), we have extended the Java networking (socket) classes to include support for native ATM based on ATM Forum’s native ATM Services Specification [5] which Winsock 2 conforms to. We have implemented two Java socket classes
Operating System Platform (UNIX or NT)

Standard TCP-UDP/IP Application

QoS-aware Java Application

Standard Java Socket
(ServerSocket, Socket (TCP)
DatagramSocket (UDP))

Extended Java Socket (with QoS)
(ATMServerSocket, ATMSocket)

Native ATM QoS layer
(NativeATM.DLL, QoS.DLL)

BSD UNIX / Winsock 2 Sockets

Operating System Platform (UNIX or NT)

FIGURE 2. Java extensions to support native ATM.

(namely, ATMServerSocket and ATMSocket) over native ATM as shown in Figure 2. These classes allow user applications to exploit QoS over ATM networks. It is worthwhile pointing out that, like most commercial ATM device drivers available today, our underlying ATM device driver currently only supports one QoS parameter, namely network bandwidth reservation. Furthermore, to keep our implementation of native ATM support in Java modular, we encapsulate all native ATM calls in two Dynamic Linked Libraries (DLLs) (NativeATM.DLL and QoS.DLL). These DLLs map native ATM requests in Java to the underlying native socket implementation (e.g. Winsock 2). The DLLs are loaded by the Java classes during runtime execution. The current implementation runs on Windows NT, but work is underway to implement equivalent functions on UNIX platforms in the future. When a QoS-aware Java application makes ATM QoS requests, the ATMServerSocket and ATMSocket classes are invoked and use the Java Native Interface (JNI) to exchange protocol information and data with the underlying native ATM network implementations (i.e. Winsock 2 in this case). Standard Java-based networking applications (i.e. those using TCP/IP or UDP/IP and without QoS requirements) run side-by-side Java QoS-aware applications by exploiting standard Java networking sockets as illustrated in Figure 2. Further implementation details on our Java implementation can also be found in [16].

Our approach gives Java applications direct access to ATM’s QoS features, such as network bandwidth reservations. This approach exploits socket implementations, such as BSD Sockets and Winsock 2, and enables application developers access to native ATM features, such as QoS reservations and native ATM multicasting. A similar approach has also been presented briefly in [17] but no actual implementation was discussed.

4. PERFORMANCE OF JAVA OVER ATM

To evaluate the performance of Java networking APIs along with our implementation of native ATM support in Java, we measured the jitter performance of live, uncompressed, digital video delivered to the desktop using Java. We repeated our tests for protocols including TCP/IP over ATM, UDP/IP over ATM and native ATM. All tests were performed using the Java networking API (i.e. Java sockets). The jitter reported is the end-to-end jitter observed over an unloaded ATM network.

Our configuration uses a pair of Pentium II 400 MHz Personal Computers (PCs) connected to an IBM 8265 ATM switch via multimode fiber. The ATM local area network used in our experiments supports connections of up to 155.52 Mbits s⁻¹. Each end-system was configured as follows: Windows NT 4.0 operating system/Red Hat Linux 5.1 (kernel version 2.1.126), 256 MB of random access memory (RAM), one 4 Gbyte disk drive and an ATM adapter from Efficient Networks Inc. Our ATM software and hardware support both CLIP and native ATM (over AAL5) protocols. On Windows NT, we use the Visual Cafe 2.5a [18] Just-In-Time (JIT) compiler, while on Linux we use JDK 1.2 [19] also with the JIT option enabled.

In all tests described below, we used a real-time video conferencing system [20] which delivers a 320 × 240 pixels (at 16 bit per pixel) video (uncompressed) frame size. We intentionally chose to use uncompressed, digital video in our tests to avoid any jitter that could be introduced by compression and decompression algorithms. We measured jitter as follows. We time stamped 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
Table 1. Frame rates of live video transmitted and received over an ATM LAN using Java sockets on Windows NT 4.0 and Linux operating systems.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>With JIT (frames s⁻¹)</th>
<th>Without JIT (frames s⁻¹)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows NT 4.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCP/IP</td>
<td>17.4</td>
<td>4.4</td>
</tr>
<tr>
<td>UDP/IP</td>
<td>17.9</td>
<td>4.2</td>
</tr>
<tr>
<td>Native ATM</td>
<td>18.1</td>
<td>4.2</td>
</tr>
<tr>
<td>Linux</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCP/IP</td>
<td>14.6</td>
<td>4.2</td>
</tr>
<tr>
<td>UDP/IP</td>
<td>14.7</td>
<td>4.2</td>
</tr>
</tbody>
</table>

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 and maximum jitter values.

To evaluate the impact of JIT on Java applications running over protocols such as TCP/IP, UDP/IP and native ATM, we conducted performance tests which measure the frame rate of live, uncompressed video for each protocol with and without JIT.

From the results given in Table 1, we make the following observations.

- Using JIT gives a fourfold improvement in overall performance of live video for all the protocols.
- Native ATM gives the highest performance (frame rates) compared to TCP/IP and UDP/IP over ATM.
- To evaluate the impact of JVM and JDK environments on UNIX and NT platforms, we conducted performance tests on Windows NT 4.0 and Linux operating systems. We obtained a fourfold increase when using JIT on Windows NT. On Linux, we obtained an improvement of 3.5 times in the frame rate when JIT is enabled.
- An interesting result from Table 1 is that Java performance on Windows NT 4.0 is better than on the Linux platform. Based on this result, we believe that the JVM implementation on Windows NT 4.0 is more efficient than the JVM implementation on Linux platforms.

Figure 3 illustrates the variation of jitter for consecutive video frames for TCP/IP over ATM, UDP/IP over ATM and native ATM. The jitter results are also given in Table 2. It is clear from the jitter fluctuations shown in Figure 3 that without JIT there is increased jitter between consecutive video frames on the Windows NT 4.0 platform. Another interesting observation is that our native ATM extensions to Java to support native ATM services yield a twofold improvement (i.e. native ATM jitter of 380 μs) in jitter performance compared to TCP/IP (i.e. 692 μs) and UDP/IP (667 μs). We also note from Table 2 that, when using JIT, the mean jitter is about four to five times better (lower) compared to when JIT is not used, and this holds true for all protocols tested in Table 2. As far as maximum jitter is concerned, we note that although the use of JIT also improves the worst case jitter performance, the maximum jitter values obtained with our video conferencing application are still quite high (in the range 25–48 ms from Table 2). Based on these results, we argue that in order to support real-time services (especially for delay-sensitive applications such as those involving digital audio and video) with guaranteed QoS running over Java platforms, the JVM and Java API need further improvements to deliver better performance.

To evaluate the impact of the underlying operating system on Java applications, we repeated our measurements for TCP/IP and UDP/IP over ATM using the Solaris operating system with JDK 1.2 installed. We investigated the impact on jitter performance of digital video with the JIT option enabled and disabled. It was not possible to carry out similar tests using native ATM on the Linux platform because the underlying efficient ATM device driver used in all our tests does not support Linux. As a result, only tests for TCP/IP and UDP/IP over ATM are reported for Linux. As mentioned above, when JIT is enabled, the frame rate obtained is about 3.5 times better compared to when JIT is disabled. However, in the case of jitter, we obtained radically different results from those obtained on Windows NT. As Table 3 and Figure 4 show, the mean jitter obtained on Linux running JDK 1.2 is actually worse (i.e. almost an order of magnitude higher) when JIT is enabled. Moreover, the maximum jitter values (actually in the range 35–44 ms—Table 3) obtained for both TCP/IP and UDP/IP are also unaffected by JIT, in contrast to the results obtained above with the JVM running in a Windows NT environment (enabling JIT yields better jitter performance). It is surprising that with JIT enabled on Linux we actually obtained a worse performance.

We speculate that one explanation for this result is that the JDK 1.2 package used on Linux was just released at the time these tests were performed and therefore the JDK implementation was not as mature and with as high a performance as the JVM used on the Windows NT platform. In the future, we plan to conduct similar tests as described above on the Solaris operating system to explore the performance of the Java networking API for protocols such as TCP/IP and UDP/IP. This will allow us to confirm whether the performance anomaly observed with JIT is due to the underlying operating system itself or due to an immature implementation of the JVM on UNIX platforms.

5. CONCLUSIONS

ATM networks offer great promise in providing flexible and efficient support for multimedia applications. QoS support is often cited as one of ATM’s great benefits. Unfortunately, in the last few years we have not really witnessed any

1Engineers at Sun Microsystems Inc. confirmed the view that the JDK package was just released and was an experimental version.
FIGURE 3. TCP/IP over ATM, UDP/IP over ATM, and native ATM over Java on Windows NT 4.0.

TABLE 2. Jitter observed at the receiver over Java on Windows NT 4.0.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Mean (µs)</th>
<th>Maximum (µs)</th>
<th>Standard deviation (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP (with JIT)</td>
<td>692.49</td>
<td>48,342</td>
<td>2810.71</td>
</tr>
<tr>
<td>TCP/IP (without JIT)</td>
<td>2509.33</td>
<td>183,255.19</td>
<td>6793.18</td>
</tr>
<tr>
<td>UDP/IP (with JIT)</td>
<td>667</td>
<td>51,063.35</td>
<td>3012.44</td>
</tr>
<tr>
<td>UDP/IP (without JIT)</td>
<td>2453.16</td>
<td>41,186.17</td>
<td>1866.72</td>
</tr>
<tr>
<td>Native ATM (with JIT)</td>
<td>380.8</td>
<td>25,323.13</td>
<td>1555.91</td>
</tr>
<tr>
<td>Native ATM (without JIT)</td>
<td>1574.33</td>
<td>64,992.58</td>
<td>2708.93</td>
</tr>
</tbody>
</table>
TABLE 3. Jitter observed at the receiver over Java on Linux.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Mean (µs)</th>
<th>Maximum (µs)</th>
<th>Standard deviation (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP (with JIT)</td>
<td>3850.9</td>
<td>35,348.3</td>
<td>2497.7</td>
</tr>
<tr>
<td>TCP/IP (without JIT)</td>
<td>416</td>
<td>35,226.6</td>
<td>1951.6</td>
</tr>
<tr>
<td>UDP/IP (with JIT)</td>
<td>5276</td>
<td>44,449.3</td>
<td>2953.7</td>
</tr>
<tr>
<td>UDP/IP (without JIT)</td>
<td>412.9</td>
<td>34,649.9</td>
<td>2055.3</td>
</tr>
</tbody>
</table>

FIGURE 4. TCP/IP over ATM and UDP/IP over ATM using Java on Linux.

real, wide deployment of QoS-aware applications, with the exception of some experimental prototype implementations [3, 21, 22] and architectures, most of which tend to be based on proprietary APIs or operating systems. The slow deployment of QoS-aware applications was partly because of the lack of standard APIs that provide QoS support.

We summarize the main contributions of this paper as follows:

- We have presented a review of current QoS-aware APIs on both Windows NT and UNIX platforms. Moreover, we have also discussed QoS-aware APIs that work over ATM and non-ATM networks. The review discussed details of how to implement QoS-aware applications. We hope that the review given in the early part of this paper will help application developers in the QoS API communication support area, and allow them to develop and implement future QoS-aware applications on either UNIX or NT platforms.
- The major goal in this work is to provide a framework for developing Java-based, QoS-aware applications. We have implemented extensions to Java to enable platform independence for native ATM applications running on Windows NT. We have restricted our implementation to ATM networks in this paper, but another project in our laboratory is aimed at addressing the issue of QoS support using protocols such as RSVP over heterogeneous networks. We conducted a series of performance tests with an actual video conferencing application delivering live, uncompressed video to the desktop.
- Our performance results reveal a fourfold improvement in the performance (frame rate) of live, digital video when using the JIT option on NT. The improvement
factor with the JIT option enabled on Linux was also close to that obtained on Windows NT. However, in the case of video jitter, we obtained a worse performance when JIT was enabled on Linux. We speculate that this is probably due to the JVM and JDK being immature implementations on the UNIX platform (for the Linux port). We plan to conduct further testing on the Solaris operating system to confirm if the poor jitter performance obtained on Linux was due to the UNIX operating system or because of the implementation of JVM and JDK 1.2 themselves. Our results have also demonstrated the superior performance of native ATM (as would be expected since there are less protocol overheads and fewer intermediate layers involved) over other protocols such as TCP/IP and UDP/IP running over ATM.

We believe that by exploiting Java and ATM technologies, it is possible to deploy a wide range of applications which not only have QoS capabilities, but also enable platform independence. As mentioned above, future work is currently addressing QoS support in Java for non-ATM networks (particularly IP-based networks using protocols such as RSVP).

ACKNOWLEDGEMENTS

This work has been supported by grants from NSF (EEC-9529152) and Sun Microsystems Inc. (Palo Alto) (grant EDUD-7824-000145-US). The author thanks Xuan Chen for his help in all the experiments reported in this paper. Finally, the anonymous reviewer’s comments are also gratefully acknowledged, since they helped produce a paper which is better than it would otherwise have been.

REFERENCES