Using Java to stream audio over ATM

Stephen Malloch (1)
Stephen Arnold (1)
Torsten Pflicke (2)

(1) Department of Music, University of Glasgow, Scotland, UK.
malloce@music.gla.ac.uk, s.arnold@music.gla.ac.uk, http://www.music.gla.ac.uk/
(2) formerly Dept. of Music, University of Glasgow. Now with Point4 Consulting Ltd., Kingston-upon-
Thames, UK. torsten@point4.com, http://www.point4.com

Abstract

University of Glasgow, in conjunction with four other Scottish higher education establishments, has recently made a substantial investment in ATM technology for the purpose of researching and developing real-time multimedia applications. This paper focuses on one such application, a high-quality audio streamer developed as part of the *NetMuse* project, which is implemented in Java with native methods in C. It examines the interface between Java and the network, comments on the unsuitability of the TCP protocol for real-time audio and video data, and details how to implement an interface with Java which omits the TCP layer to make full use of the ATM network.

1 Introduction

The Department of Music in the University of Glasgow, in conjunction with a range of other Scottish higher education establishments [1], has recently made a substantial investment in ATM technology in order to research and develop real-time multimedia applications [2].

*NetMuse* [3] [4], which is one of a number of projects at Glasgow which demonstrate aspects of networked high-quality multimedia, focuses on the development of educational materials, based on the World Wide Web model, for distribution to all the project partners via the ATM-based Scottish Metropolitan Area Networks (MANs) [1]. These pedagogical materials, covering aspects of music history, performance, composition and research, are presented as a distributed resource suitable for distance learning. They consist of text, graphics and CD quality audio (16bit, 44.1kHz), and have been implemented in HTML, Perl and Java. [5]

A core tool is a specially written audio streaming application, the *NetMuse* Audio Player, which was written in Java and C and which is used as a helper application to Netscape. This was authored by Torsten Pflicke.

This paper focuses on how the *NetMuse* audio player interfaces with the network, comments on the limitation of TCP for streaming audio, and then discusses in some detail how, in this case, Java can be made to bypass the TCP/IP layers and go straight to the ATM layer. The limitations of

the solution will be discussed and some conclusions drawn.

2 ATM Networking and Java

ATM networking was selected since this type of networking offers not only the necessary bandwidth, but also guaranteed quality-of-service (QoS) mechanisms, which are essential for the distribution of high-quality audio and video in real-time. The QoS features of ATM allow applications to request a specified amount of bandwidth which, if available, is guaranteed for the duration of the connection. This means that for time-based materials such as audio and video glitches and stutters can be eliminated by requesting an adequate amount of bandwidth.

2.1 TCP/IP over ATM

Manufacturers of ATM hardware realised early on that support of the Internet Protocol (IP) would be an important factor in the acceptance of ATM, and developed ways of transparently integrating IP [6]. Conventional IP routing software, combined with the driver software from Fore Systems [7] enables TCP packets to be directed over the ATM network interface of a workstation. The driver creates an ATM connection to the required host for the duration of the communication, allowing TCP applications to run transparently over the ATM layer.
2.2 TCP support in Java

The Java Developer's Kit (JDK) from Sun [8] includes built-in support for TCP and UDP networking through the java.net package of Sun's Java API. The JDK, or any implementation of the Java Virtual Machine (VM) and Sun's Java API, connects to the socket interface of the respective platform in question through native methods. These native methods make the appropriate operating system (OS) calls to supply the functionality of the socket connection (see top half of Fig 1a).

![Diagram of network support for NetMuse audio player (client and server) showing how the TCP layer is bypassed.]

The NetMuse audio streaming application was developed using the existing Java socket structure to make the connections between client and server, i.e. using the built in TCP/IP support which was then directed over the ATM. (Fig. 1a.) In order to play the audio data received, further native methods were written for each client type (IRIX, Win95 and MacOS) to pass the data to the relevant OS audio API calls.

2.3 Limitations of TCP

TCP is commonly known as a 'reliable' protocol, because it guarantees to deliver every packet accurately. However, there is no guarantee of delivery time inherent in this protocol: packets may arrive out of order, and dropped packets are re-transmitted causing further latency and delays. This makes TCP unsuitable as far as streaming high-quality audio is concerned, where absolute fidelity is required, even though with a video signal a few dropped frames may be imperceptible [9]. Moreover, performance is significantly less when running TCP/IP over ATM compared with Fore API over ATM [10].

To make proper use of the bandwidth and QoS available through ATM, and to guarantee the quality of the audio received, the NetMuse Audio Player had to be made to work not just over 'TCP/IP-over-ATM', but over 'real' ATM. The question then arose: how can Java be used to stream audio over ATM without using TCP/IP?

3 Java over ATM

3.1 The Java / Network Interface

According to Sun/Javasoft themselves [8] the only networking support built in to Java - through the package java.net - is for TCP and UDP over IP. To be more specific about Java network support, the classes in the java.net package provide a system independent interface to a TCP or UDP implementation. The implementation itself will always be system dependent and is supported through the use of native methods.

The Java API includes a 'factory' mechanism to allow programmers to implement network sockets in their own way. While this is primarily intended for differing security implementations such as firewalls etc. it also allows, with some additional programming, the implementation of a socket which can bypass the TCP/IP layers and access the Network Interface Card (NIC) directly. (Fig 1b)

The constructors Socket() and ServerSocket() in the Java API both test if a socket 'factory' (i.e. an instance of a class which implements the SocketImplFactory interface) has been created. If it has not, then the default implementation of the socket is used, class PlainSocketImpl, which contains native methods that make the OS specific network system calls.

By looking carefully at the Java side of this connection (abstract class SocketImpl, and default implementation PlainSocket), it can be observed that as long as the native part presents data to the Java side in the format that it expects, there is no
real requirement that the implementation of the network socket connection must use TCP. Instead, provided a suitable API is available, software system calls direct to the ATM hardware can be made from the native methods, allowing full use of the bandwidth and quality of service that ATM provides, rather than using the TCP/IP layers over the ATM network.

Fortunately, just such an API exists for, at least, Fore Systems NICs. (The API used here was for IRIX 5.3: a beta version of the API for Win32, which integrates with Winsock2, is also being trialled.) This API provides programmer access to the familiar Berkeley style socket commands such as open(), bind(), connect() etc. Though in this case they are atm_open(), atm_bind() etc.

Some new Java classes were obviously required in order to implement an ATM socket connection. These are outlined in Fig 2, above. The shaded classes are ones that were written during this project to allow direct ATM access. Others are from the package java.net, and origination of the rest is cited in the Figure (e.g. java.io). The class ATMSocketImpl which implements the abstract class SocketImpl, declares the native methods where the ATM socket system calls are made.

Neat use of the object model of Java was made in the classes ATMSocketInputStream and ATMSocketOutputStream. Inheriting the io stream properties of the classes above them, and differing from the regular socket io stream classes only in that their native methods make ATM system calls rather than conventional calls, allowed the use of the existing java.io package for reading and writing at the Java level. In theory, after the initial set-up phase where ATM is specified by the programmer, using an ATM socket is transparent; identical from a programmer’s point of view to using a regular socket.

Figure 2. Components of the java.net package used in the NetMuse audio player. (Shaded boxes indicate code written to support ATM.)
3.3 The loose threads

In practice, however, the situation is not quite so perfect. Although the socket interface to Java and the Fore Systems API calls match quite closely, they do not match up exactly. This is because the API interaction is in fact at the Data Link Layer, although Transport Layer semantics are used [10]. For example, the ATM API call atm_bind() also contains the parameter for the number of connections to accept, which on the Java side comes in the socketListen method. The ATMSocket implementation then, calls atm_bind() from the socketListen method rather than the socketBind method.

A more serious mismatch comes with the Java side available() method. This method is designed to return the number of bytes which are available to be read without blocking. With TCP this can be implemented using the appropriate ioctl() system call, however, with native ATM there is no direct equivalent to this [11]. The available() method remains unimplemented for the ATM/Java socket, something of which the programmer needs to be aware. This reduces the general purpose use of the ATM/Java socket slightly but the NetMuse audio application functions satisfactorily without the use of this method.

An interesting comparison can be found in Fore's API for the 32bit Windows platform. This API, though still in the beta stage, sits transparently within the OS Winsock2 layer. Hence, since the java.net package is implemented already for the existing Winsock (e.g. in Sun's Win32 version of the JDK), then on the Win32 platform the ATM/Java socket should match the API exactly and make a better interface between Java and the ATM level.

4 Conclusion

Software has been written for SGI Indy/Challenge client and server machines to allow the NetMuse audio player to access the IP/ATM level directly. Versions of this software for Win32 is under development using the beta API from FORE. As yet there is no equivalent API for the Apple Machine available, although PCI ATM network cards do exist for the Mac.

Whilst the lack of suitable implementation for the available() method limits the use of the software written for more general purpose applications, this project has demonstrated that the concept is viable. Future implementations, perhaps using Winsock2, should make possible a more general-purpose version of the software, open up wider possibilities and provide the benefits of platform independence offered by Java together with those of high bandwidth and QoS offered by ATM networking.

One such possibility, currently being investigated in Glasgow as part of the MiniMS project, is the development of a distributed or shared dsp unit for audio applications. The front end might be developed in Java providing a consistent control interface across platforms; and the audio data could be routed, via ATM, from the host machine to the dsp unit where it is processed and sent back potentially in real-time [12] [13].

References

[5] All references to Java in this paper refer to JDK version 1.0.2, unless otherwise specified.
[7] Only Fore Systems’ hardware is used at Glasgow University.
[11] This is because TCP uses a streams based model which, for ATM Adaption Layer 5, is not appropriate.

Acknowledgements

Prof. D. McAuley, Dept of Computing Science, University of Glasgow.
The Scottish Higher Education Funding Council.