Middleware-level QoS Differentiation in the Wireless Internet: the ubiQoS solution for Audio Streaming over Bluetooth

Paolo Bellavista
Dip. Elettronica, Informatica e Sistemistica
Università di Bologna
pbellavista@deis.unibo.it

Cesare Stefanelli, Mauro Tortonesi
Dipartimento di Ingegneria
Università di Ferrara
{cstefanelli, mtortonesi}@ing.unife.it

Abstract

The ultimate goal of mobile and ubiquitous Internet accessibility is not only the seamless integration of wireless devices with traditional fixed networks but also the dynamic differentiation of the provided levels of Quality of Service (QoS) depending on client characteristics. In this context, the paper presents the provisioning of audio streaming with different QoS levels in the application-level ubiQoS middleware. In particular, it focuses on how ubiQoS manages the QoS over the last segment of the audio distribution path towards Bluetooth clients by allocating different types of Bluetooth communication channels (unicast connection-oriented or broadcast connectionless) depending on the differentiated QoS requirements of different user classes. To this purpose, we have designed and implemented a library that extends the JSR82 standard specification with the support of Active Slave Broadcast, thus simplifying the Java-based management of Bluetooth communications. The reported experimental results show the feasibility of our application-level middleware approach in the challenging case of audio streaming with differentiated QoS to resource-limited Bluetooth devices.

1 Introduction

The expanding market of wireless-enabled portable appliances pushes towards the realization of spontaneous networks of devices located within the range of a single user. These spontaneous networks are often identified as Personal Area Networks (PANs) [1]. There is a growing and growing commercial interest in enabling PANs to seamlessly integrate with the fixed Internet. In the following, we will use the term *wireless Internet* to refer to the above deployment scenario where PANs work as the "last-meter" connectivity solution that extends the traditional Internet infrastructure [2].

A primary issue of the wireless Internet scenario is the wide heterogeneity of the hardware/software capabilities of access devices, e.g., screen size and resolution, operating systems, and supported multimedia formats [3, 4]. This heterogeneity makes almost impossible to provide statically tailored versions of Internet services to all the possible categories of access terminals. In addition, portable terminals usually have limited resources in terms of processing, memory, storage, and network connectivity. Resource-consuming services designed for the fixed network, such as multimedia streaming, requires being downscaled to suit the specific characteristics of the served limited clients.

In the wireless Internet, Bluetooth is emerging as the de-facto standard communication technology for

the realization of PANs [1]. Bluetooth-enabled portable devices can interconnect to form a particular incarnation of PAN called piconet, which consists of one master and up to 7 slaves. The master device has direct visibility of all slaves in its piconet and can handle two different types of packet-oriented traffic, with differentiated Quality of Service (QoS) levels: unicast Connection-Oriented (CO) traffic over Asynchronous Connection-Less (ACL) links with best-effort QoS support and broadcast connectionless traffic over Active Slave Broadcast (ASB) links with no QoS support. In addition, Bluetooth provides circuit-oriented Synchronous Connection Oriented (SCO) links, designed to support time-bounded transmissions, e.g., voice communication. Depending on service/user QoS requirements, streaming services to Bluetooth devices should be capable of choosing the most suitable type of available channels, even by considering the already accepted service requests.

The paper presents a middleware solution, called ubiQoS, for the QoS management of mobile multimedia services to portable devices in the wireless Internet. The application-level ubiQoS approach facilitates the application-specific tailoring and adaptation of multimedia flows, the differentiation of QoS levels depending on profiles of user requirements and of device characteristics, and the dynamic un/installation of middleware/service components, when and where needed [5, 6]. The paper specifically focuses on how ubiQoS supports audio streaming with differentiated QoS over the last-meter Bluetooth channels towards portable client devices. The main guideline is to exploit support proxies located at the edges between the fixed Internet and the Bluetooth PANs. Proxies work as masters in the Bluetooth piconets of access devices: they choose the most suitable Bluetooth channels for communicating to the target clients depending on the already accepted service requests and on client profiles, possibly by downscaling the served audio streams at provision time. ubiQoS proxies are implemented in terms of mobile agents, i.e., active entities that can migrate from node to node during their execution by carrying their code and by preserving their reached execution state. Agent mobility facilitates the deployment of ubiQoS components in the proximity of wireless access localities only when required, depending on the client positions and movements at runtime.

To achieve portability of the ubiQoS middleware over the open wireless Internet, we have designed and implemented a Java-based interface, called JSR82ext in the following, to have full control of Bluetooth channels from within the standard Java Virtual Machine (JVM). JSR82ext has been originally developed within the ubiQoS project and significantly extends the standard Java APIs for Bluetooth (JSR82): i) JSR82ext adds the support for ASB links, which are fundamental for efficient broadcast distribution of the same audio flow within a piconet; ii) it enriches the open source JavaBluetooth stack (with its current limitations to the support of only serial adapters [7]) to fully support JSR82-based ACL management on a wide range of Bluetooth adapters; iii) it additionally permits to exploit SCO-based communications, also by integrating with platform-dependent native SCO libraries via the Java Native Interface [8].

The paper finally reports the experimental evaluation of ubiQoS audio streaming with differentiated

QoS levels to Bluetooth devices. The results show that it is crucial to differentiate the usage of ACL-based channels and ASB-based ones depending on the already accepted service requests and on client profiles. The differentiated allocation of channels to different classes of users permits to achieve a simple and effective form of QoS differentiation in a single piconet. Most important, the experimental results show that our Java-based middleware approach at the application level introduces an acceptable and very limited degradation if compared to the theoretically maximum performance of the raw Bluetooth channels.

The rest of the paper is organized as follows. Section 2 gives an overview of the Java-based ubiQoS middleware to support streaming services with differentiated QoS levels. Section 3 describes the QoS-differentiated communication channels available in Bluetooth and presents the state-of-the-art of the integration of Bluetooth and Java programming. Then, Section 4 specifically focuses on how ubiQoS proxies allocate the different channels available in the Bluetooth-based last-meter connections, while Section 5 presents the design and implementation of our JSR82ext API for Bluetooth channel management. Section 6 reports the experimental results about the performance of the implemented ubiQoS prototype when exploiting the JSR82ext library. Concluding remarks and future work end the paper.

2 The ubiQoS Middleware for Audio Streaming with Differentiated QoS over Bluetooth

The provisioning of Internet services to wireless devices requires dynamically managing (and possibly downsizing) the provided QoS levels to suit the specific limits of served access terminals and the runtime resource availability in the wireless access locality, e.g., the currently unused piconet bandwidth. In particular, QoS tailoring and adaptation is crucial for very resource-consuming services such as multimedia streaming, especially when working over traditional best-effort networks. In addition, device mobility requires several other support operations that limited devices cannot perform on their own, e.g., local/global resource finding, binding, and adjustment. On the one hand, local discovery operations may consume a large amount of device resources in the environment exploration and in negotiations with available entities and services. On the other hand, even global identification and retrieval of user-related properties (such as profiles of user preferences, profiles of access device capabilities, and security certificates) from directory-based name services can be too long to be directly controlled and managed by terminals with limited resources [9].

Therefore, audio streaming in the wireless Internet can significantly benefit from distributed and active support infrastructures for QoS management, hosted in the fixed network. In addition, there is a growing research interest in the design choice of adopting mobile middleware proxies that work over the fixed network, on behalf of and close to associated portable devices [4]. Proxies can decide the best QoS management operations to perform and can act as distributed cache repositories of previously requested flows in successive service requests. In addition, they can support additional management operations, such as providing connectivity and discovering the needed resources/service components, either in the current

device locality or in the whole Internet environment. Mobile proxies could also follow device movements during service provisioning and install automatically, only when needed, locally to the fixed network localities visited by portable devices. Let us notice that the importance of proxies that move close to clients at runtime is recognized in several application domains and distributed computing technologies: for instance, in the Jini discovery solution, proxy objects are dynamically installed at the JVM on the client side to handle the interaction with the discovered remote services.

2.1 The ubiQoS Middleware Architecture

Given the above motivations and claims, we have developed the ubiQoS middleware with a primary design choice: to provide any wireless device with a mobile companion entity, called shadow proxy, which runs in a wired node in the same network locality that currently provides wireless connectivity to the device. ubiQoS shadow proxies are implemented as mobile agents, built on top of the SOMA programming platform[*], to enable the dynamic proxy migration where and when needed, i.e., to the network localities that currently host wireless portable devices requesting ubiQoS-based audio services. As depicted in Figure 1, ubiQoS proxies are hosted in execution environments, called places, that offer the basic services for mobile agent communication and migration. Places typically model fixed Internet nodes and can be grouped into domains that correspond to network localities, e.g., Local Area Networks with IEEE 802.11/Bluetooth access points providing wireless connectivity to portable devices (see Figure 1).

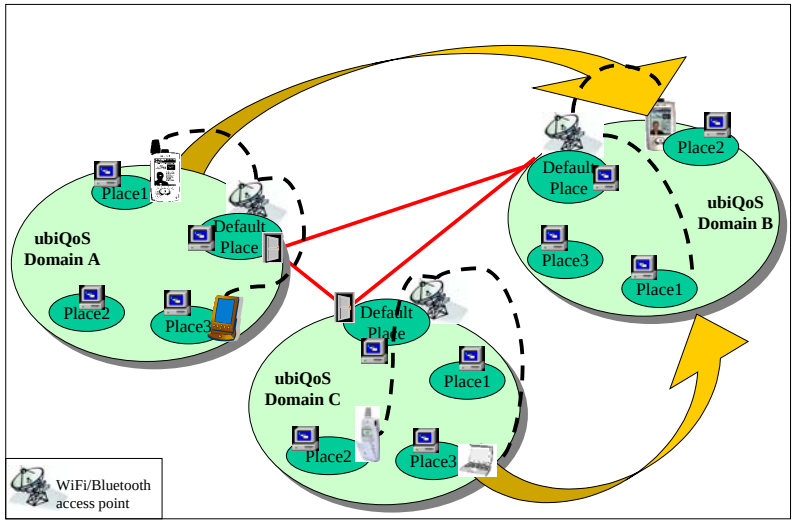

Figure 1. Wireless devices roaming among ubiQoS domains.

[*] Additional information about the SOMA mobile agent system and its full source code are available from our CVS repository at: `http://lia.deis.unibo.it/Research/SOMA/`

Shadow proxies represent portable devices over the fixed network and receive, cache, and coordinate the tailoring/adaptation of audio flows on behalf of their client devices. Any shadow proxy retrieves the profiles of its served companion device and of its currently connected user. Profiles play a central role in service discovery in the ubiQoS Portable Device Lookup Service (PDLS), as sketched in the following. In addition, the shadow proxy can maintain small amounts of audio flow information in a buffer ahead of playback (as in anti-shock mechanisms for portable CD players), in order to smooth temporary congestion situations in the network connections between the proxy and the server.

Shadow proxies interact with other ubiQoS components hosted in the same domain, as depicted in Figure 2: QoS adapters are in charge of on-the-fly tailoring of audio flows depending on client characteristics and requirements; client/server stubs enable the easy integration of the ubiQoS infrastructure with off-the-shelf PDA audio players and streaming servers; PDLS supports the intra-domain personalized discovery of needed middleware components; and the Profile Manager Service (PMS) stores profiling information for supported devices and registered users.

QoS adapters are in charge of compression and format transcoding of audio flows, e.g., reduction of bit/sample rate and conversion from WMA/OGG/WAV to MP3, to tune the provided QoS level to the profile of device characteristics and user preferences. They receive audio data, operate transformations on them, and forward processed flows to device-specific audio players. The current implementation of QoS adapters is based on the SUN Java Media Framework (JMF) for multimedia reception, transmission and processing [10]. For the transport and control of packet flows from audio streaming servers over the wired network, QoS adapters exploit the JMF APIs to integrate with the Real Time Protocol (RTP) and its corresponding RTCP control protocol [11]. Any ubiQoS middleware component is generally portable on any platform that hosts a JVM. For performance sake, QoS adapters sometimes exploit local plug-ins available as native components, by integrating them via the portable Java Native Interface [8]. To improve their portability, QoS adapters retrieve dynamically the list of plug-ins installed on their current place to bind only to the locally available native components.

Device/player-specific client stubs run on portable devices and are in charge of transferring audio flow requests from locally installed device-specific players to associated shadow proxies, of receiving flows, and of forwarding them to the local players. We have currently implemented two different stubs. The first one runs on top of the Java standard distribution and interworks with the standard JMF audio player; the second one is based on the Java 2 Micro Edition and acts as a stub for the Mobile Multimedia API audio player [12]. Similarly to client stubs, server stubs simplify the integration of the ubiQoS infrastructure with legacy components. They run on the places hosting audio servers; they are in charge of receiving flow requests and passing them to their local servers. In addition, they transparently encapsulate the server-provided flows into RTP ones, and forward the RTP flows to the requesting ubiQoS components. At the moment, we have implemented server stubs that can wrap JMF-compliant audio sources.

ubiQoS also includes two non-moving middleware components, PDLS and PMS. They are the only

infrastructure components that any ubiQoS domain should have installed to participate in audio distribution. PDLS is responsible for managing tailored lookup requests. When an intra-domain shadow proxy performs a discovery request for a specified audio flow, the local PDLS component does not provide a direct reference to an audio server that can answer the service request (the traditional behavior of lookup services) but a reference to a suitable QoS adapter acting as the intermediate between the shadow proxy and the actual server component. PDLS is based on Jini and extends the SUN Reggie reference implementation of the lookup server [13]. Differently from Reggie, it also considers the user/device profiles carried by the shadow proxy to identify the needed QoS adapter. Then, it binds the identified adapter to the server component that can provide the requested audio flow. When the shadow proxy place does not already host a suitable QoS adapter component, PDLS triggers both the migration of the adapter code to that place and the instantiation of the adapter.

PMS maintains profiles of supported devices and registered users. It implements a partitioned and partially replicated directory service specialized for profiles. PMS maintains local copies of profile information and is able to coordinate with PMSs in other domains, via either LDAP or HTTP, to provide global profile visibility to shadow proxies. Device and user profiles are expressed according to the W3C Composite Capabilities/Preference Profile standard specification [14].

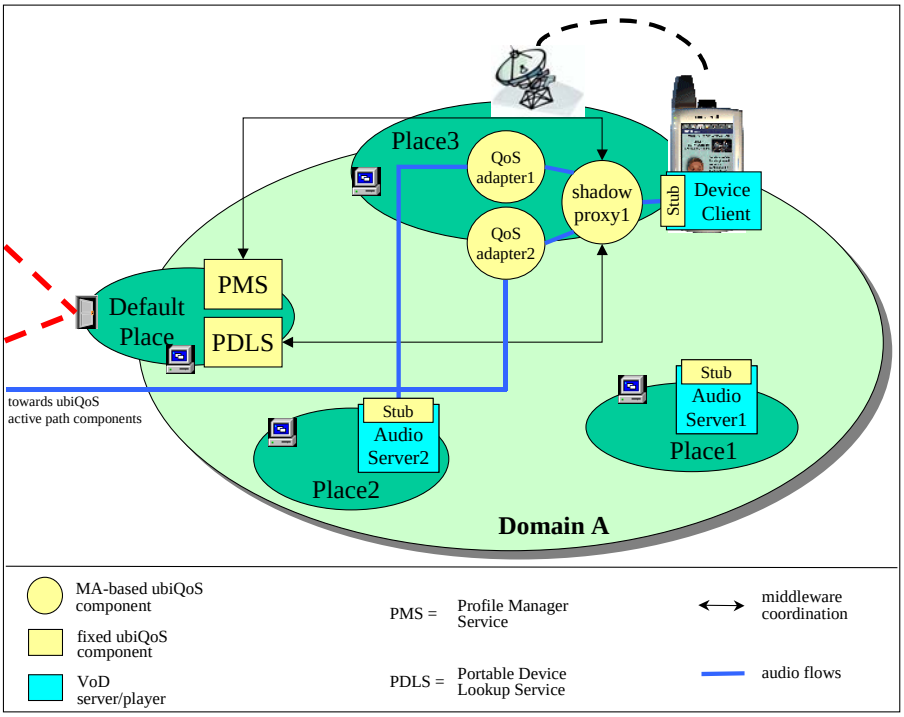

Figure 2. The ubiQoS middleware components in one domain.

The detailed description and implementation insights of the different ubiQoS middleware components are out of the scope of the paper and can be found in [9]. In the following, instead, we specifically focus on the last hop of the audio distribution path because it represents the most crucial point of resource

discontinuity when serving portable devices in the wireless Internet. In particular, the paper specifically focuses on how ubiQoS shadow proxies can support different QoS levels for audio provisioning by assigning different types of Bluetooth channels to different classes of ubiQoS users, also by considering the current state of the piconet, e.g., the already accepted local service requests. In addition, the paper shows how it is possible to perform Java-based Bluetooth channel management, which is crucial when dealing with the open wireless Internet.

3 Bluetooth and Java Programming

ubiQoS manages differentiated QoS levels for audio streaming to wireless Internet clients by choosing the most suitable communication solution for the last-meter wireless connection. In particular, when considering Bluetooth-based last meter connectivity, the main idea is to allocate different types of channels to different classes of users, also depending on the already accepted audio streaming requests in the considered piconet. Before describing the ubiQoS channel allocation strategies, this section briefly overviews the different kinds of communication solutions available in Bluetooth and sketches the state-of-the-art of the software supports to exploit these communication possibilities when working with the Java language.

The Bluetooth specification defines two main classes of traffic. On the one hand, framed data traffic exploits Logical Link Control and Adaptation (L2CAP) channels for both connection-oriented and broadcast data transfers: unicast CO traffic is transferred over ACL links with best-effort QoS; broadcast transmissions work over ASB links with no QoS support. On the other hand, unframed data traffic are provided for isochronous communication with guaranteed QoS requirements [15]. It is carried over Synchronous Connection-Oriented (SCO) links or the recently introduced extended Synchronous Connection-Oriented (eSCO) ones (only available in Bluetooth v1.2 specifications).

L2CAP channels built on top of ACL links support the specification of QoS settings to indicate the desiderata for the delivery of data frames. QoS settings are typically used to instruct the Bluetooth core system to discard undelivered packets after a given lifetime or to specify the reliability characteristics of the data transmission. In fact, ACL links exploit an error detection algorithm that can trigger a simple acknowledgement/repeat request (ARQ) protocol. This permits to achieve an enhanced reliability by re-transmitting packets that do not pass the error checking algorithm at the receivers. When transferring latency-sensitive packets, it is possible to modify the above re-transmission choice by indicating to discard unsuccessfully transmitted packets if their useful life has expired. In addition, the Bluetooth v1.1 specification provides the QoS Setup command of the Host Controller Interface (HCI) to specify QoS settings on ACL links symmetrically. The QoS support in Bluetooth v1.2 is still more advanced. For instance, it introduces the HCI Flow Specification command to specify QoS flow parameters in an asymmetrical way for (even already established) ACL connections. Unfortunately, most firmware of the Bluetooth chips in commerce still provides only a partial implementation of the Bluetooth QoS support,

and very first chips compliant with the Bluetooth v1.2 specification are being commercialized in these days [16].

L2CAP channels built on top of ASB links are suitable for the broadcasting of data flows from the master to a group of slaves located in the same piconet and do not provide any form of QoS management. However, ASB links are unreliable: they have no feedback route and this makes impossible to exploit the ARQ scheme adopted for reliable communications over ACL links.

Bluetooth also supports SCO and eSCO links, which are point-to-point, bi-directional, isochronous, and with constant bit-rate (fixed to 64kb/s for SCO and user-defined for eSCO). One master can support up to three SCO connections to different slaves in the piconet. However, SCO traffic is isochronous and requires strict reservation of transmission slots, thus leaving very few of the piconet bandwidth for other applications. Since signaling between Bluetooth devices is primarily done via L2CAP, it is practically impossible to have more than two contemporary SCO links working, as they would completely exhaust the available piconet bandwidth by leaving no space for any other signaling exchange. Let us observe that, at least at the moment, SCO links are a practicable solution only for analog voice traffic, e.g., from headsets to associated mobile phones. In fact, most commercial Bluetooth chips transfer application data over SCO as audio pulse code modulation samples, by exploiting lossy codecs based on continuously variable slope delta modulation before transmitting the data over the air. In addition, the SCO isochronicity requirements hardly match with software scheduling and the SCO support in most Bluetooth stacks for Linux and Microsoft Windows is demonstrating to be still immature, thus leading to very poor performance [17].

3.1 Java-based Management of Bluetooth Communications

Java is the primary programming language to achieve easy portability not only in the case of mobile agent-based infrastructures such as ubiQoS, but also in any portable and dynamically deployable middleware support for the open wireless Internet. For this reason, here the paper describes which is the state-of-the-art of the Java-based programming support available for the management of Bluetooth connections.

Nowadays, Java-based applications can interwork with Bluetooth via the recently approved JSR82 standard interface [18]. The JSR82 API allows the creation and the management of Bluetooth L2CAP-based CO channels on top of ACL, but does not provide any support for ASB and SCO links. In addition, JSR82 supports device discovery, service registration/discovery via the Service Discovery Protocol, Bluetooth security facilities (encryption, authorization, authentication), and object transfer via the Object Exchange protocol [15]. Moreover, the specification defines optional profiles with additional functionality on top of the JSR82 core: the Generic Access Profile, the Service Discovery Application Profile, the Serial Port Profile, and the Generic Object Exchange Profile. Most relevant, JSR82 has been designed by taking into consideration the characteristics of resource-limited portable devices: the JSR82 API can work on top

of any compliant implementation of the limited Java 2 Micro Edition with the Connected Limited Device Configuration [19].

Even if JSR82 represents a fundamental first step in opening the Java support towards Bluetooth-based connectivity, its specification lacks some important functionality, especially when considering the provisioning of services with differentiated QoS levels. First of all, the JSR82 specification does not include the support for ASB, SCO, and eSCO links, thus making impossible to adopt the standard API as it is in portable Java-based middleware solutions such as ubiQoS. In addition, JSR82 does not consider at all some advanced Bluetooth features, such as the possibility to set QoS parameters for L2CAP-based CO channels. Finally, JSR82 introduces the Bluetooth Control Center as one of its main architectural components, with the intent to enable users and OEMs to change Bluetooth settings (basic security settings, security policies for connection authorizations, lists of known/trusted devices) in a portable way. However, the specification does not standardize the API to access the Control Center facilities, with the result of making that part of the JSR82 API unusable in really open environments.

Also in response to the above JSR82 API limitations, several Java libraries, alternative to JSR82, continue to be proposed, such as JBlueZ [20]. These libraries typically provide Java-based applications with a more extensive and complete access to the native functions of existing Bluetooth software suites, by defining non-standard proprietary APIs. They often exploit the lower-level interface exported by the underlying Bluetooth stack implementation they work in conjunction with, and integrate with those software stacks via JNI, the standard Java API for interfacing native software modules and the JVM [8]. However, these libraries tend to provide only a partial support to the QoS management of the different types of Bluetooth channels and to propose non-portable solutions that are tightly associated with a specific implementation of the Bluetooth software suite (Bluetooth stack + operating system).

For the above reasons, to permit the full exploitation of the communication possibilities offered by the standard Bluetooth specifications, which are needed when distributing audio streaming with differentiated QoS levels within a piconet, we have decided to define and implement an original extension of the standard JSR82 API within the ubiQoS project. Section 5 describes how JSR82ext extends the standard JSR82 and gives some insights about its implementation.

4 Last-Meter Bluetooth Channels in ubiQoS

ubiQoS proxies run in Bluetooth-enabled fixed network places and act as piconet masters for their PANs. They interwork with local QoS adapters and with the other ubiQoS middleware components over the fixed network to send to their clients the audio flows with the properly downscaled QoS level. As already stated, in the following, the paper focuses on the PAN side and on how the proxies manage the different Bluetooth channels available in their PANs.

The ubiQoS proxy can support different classes of users in its piconet, i.e., platinum, gold, silver, and bronze, with differentiated QoS levels. It retrieves the profiles of both involved users and requested audio

flows, e.g., user QoS requirements and the bit-rates of requested flows. Based on these metadata, the proxy chooses the ranking of the currently served clients and how to allocate the Bluetooth channels over the different types of link.

The ubiQoS proxy exploits both unicast ACL and broadcast ASB links. In particular, since ACL links are expected to provide a higher quality transmission than ASB ones, the main guideline in ubiQoS is to allocate ACL links to privileged users if there is enough piconet bandwidth; when the number of clients in a single piconet is too high for exploiting only ACL links, ubiQoS will use ASB links (less bandwidth-consuming and of minor quality than ACL) to transfer audio streaming to less privileged clients. In addition, the proxy supports the preemption of less privileged clients in the case of more privileged new clients joining the service in the piconet.

More in detail, ubiQoS decides to accommodate only one platinum client in a piconet at a time. The platinum client exploits an asymmetric ACL link with DH5 packets (maximum bandwidth achievable in theory = 721 Kbps), by leaving to other local slaves a small amount of bandwidth that is allocated to remaining non-platinum clients via ASB links. When the piconet does not host a platinum client, the ubiQoS proxy can decide more articulated strategies for link allocation. Its default choice is to serve gold clients via ACL links and bronze clients via ASB. Silver clients, instead, are assigned with either ACL or ASB links depending on the piconet state, i.e., the number and the type of the already serving audio streaming requests in the locality. It is the ubiQoS proxy that dynamically decides to assign either ACL or ASB links to silver clients with the goal of maximizing the bandwidth utilization in its piconet, as shown quantitatively in the experimental result section.

Let us finally observe that the ubiQoS proxy can also modify the link allocation to clients at provision time. For instance, after the arrival of a new gold user in a piconet hosting 3 gold and one silver slaves, all currently served with ACL links, the proxy can de-class the silver client by switching it to an ASB-based connection. This switching operation frees one ACL-based connection, which is assigned to the new gold slave. The next section show how ubiQoS proxies can decide which types of link to exploit for their master-slave connections by exploiting our JSR82ext API.

5 The JSR82ext API Implementation

Within the framework of the ubiQoS project we have designed and implemented the JSR82ext library by extending the open source JavaBluetooth JSR82 implementation [7]. The main goal is to provide Java-based applications with functions for the application-level QoS management of different types of Bluetooth channels: ACL-based unicast CO, ASB-based broadcast connectionless, and SCO channels (the code of the JSR82ext library is available at `http://lia.deis.unibo.it/Research/ubiQoS/audioStreaming/`).

Figure 3 depicts the layered architecture of the JSR82ext stack. The primary components originally developed within the ubiQoS project are L2CAP CO, L2CAP broadcast, L2CAPLink, SCO CO, and

SCOTransport. The abstractions provided by the three upper-layer components allow the QoS-enabled transmission and retrieval of application-level data between Bluetooth-enabled applications.

The L2CAP CO module allows the establishment of unicast L2CAP CO channels on top of ACL links, by exporting a completely JSR82-compliant API to the ubiQoS middleware. In addition to the standard JSR82 functions, it permits to specify the desired QoS characteristics for the best-effort CO channels. For instance, the module allows ubiQoS proxies to control the priority of the Java threads responsible for managing the buffers associated with the transmission channels and also to set link-layer QoS requests (token rate, peak bandwidth, and latency) by interworking with the HCI QoS Setup command. The L2CAP ASB module supports the creation and exploitation of broadcast communications on top of ASB links, by proposing an API similar to the JSR82 one. It also permits some forms of software-based QoS management (differentiated prioritization of application-level buffer scheduling), given the lack of the QoS functionality provided in this case at the link layer.

The L2CAPLink module significantly extends the JavaBluetooth L2CAPLink by implementing the L2CAP protocol logic, e.g., channel creation, packet segmentation, and reassembly, both for ACL and ASB links. This component is built on top of the HCIDevice layer, whose task is to provide a simple abstraction of Bluetooth devices, to interact with available adapters at the HCI level. The HCIDevice module allows the transmission and retrieval of data, command, and signaling packets between the upper layers of the stack and the Bluetooth controller installed on the adapters.

The BlueZTransport component is a low-level interface between the HCIDevice layer of our protocol stack and BlueZ, the native Bluetoothv1.1 protocol stack included in the standard distributions of the Linux kernel. This module enables the HCIDevice layer to communicate with Bluetooth devices at the HCI level via JNI [8] and the low-level BlueZ native API, thus allowing the usage of JSR82ext with the wide range of Bluetooth devices supported by BlueZ [21]. We have developed the BlueZTransport module to overcome the limitations of the JavaBluetooth support which, as a 100% pure Java system, cannot interface with native Bluetooth device drivers and can only support serial Bluetooth adapters by means of the javax.comm API.

We have also developed from scratch the Java-based SCO module, by integrating with basic support mechanisms provided by native BlueZ SCO libraries. The platform-dependent native support for SCO sockets is integrated in a portable way via JNI in the SCOTransport module that provides an interface for data transmission/receiving over Bluetooth SCO links. Additional details about the SCO support in JSR82ext are out of the scope of the paper and can be found at the ubiQoS audio streaming Web site.

JSR82ext is not the only JSR82-compliant stack working on Linux. The recently released Avetana protocol stack is tightly integrated with BlueZ and can significantly contribute to leverage the deployment of Bluetooth-enabled Java applications on the Linux platform [22]. However, since the implementation of the lower layer of our stack is based on the exploitation of native API to access the HCI interface of Bluetooth adapters (commonly available in most native Bluetooth protocol stacks nowadays), our

approach goes beyond Aventana in providing a good separation between the platform-independent middleware components that implement the upper layers of the protocol stack and the native components that access Bluetooth adapters via native platform-dependent APIs. As a result, the JSR82ext solution is more easily portable among different operating systems and Bluetooth software stacks; this characteristic is crucial in the open wireless Internet and even stressed by the current availability of a large set of heterogeneous Bluetooth support implementations competing on the market.

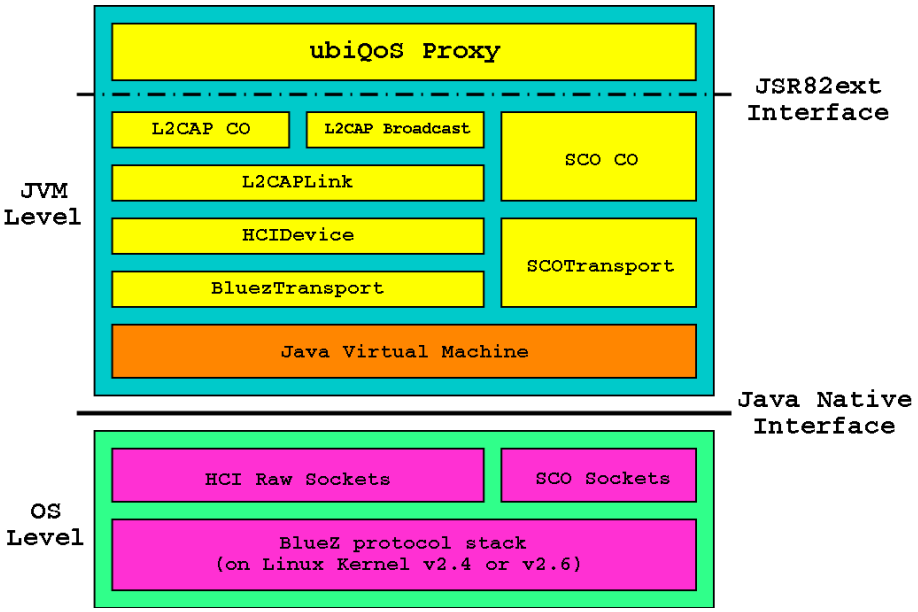

Figure 3. Architecture of the JSR82ext protocol stack.

6 Experimental Results

We have measured the performance of the ubiQoS middleware when providing audio streaming over Bluetooth, to test the feasibility of the Java-based ubiQoS management of the different types of Bluetooth channels. In particular, our experiments have measured throughput, packet delay, and standard deviation in packet delays when the ubiQoS proxy exploits CO and broadcast channels via the JSR82ext interface.

In the testbed, the master is a PC Pentium III 1.13 GHz, while the piconet slaves are laptops, which move during service provisioning but stay all located within the range of the Bluetooth master visibility. The master and the slaves host Linux (kernel version 2.4.26-mh1) with the latest version of the BlueZ userspace tools (bluez-utils 2.7, bluez-libs 2.7).

The master can distribute different audio flows, with different QoS levels, to the different slaves. In the experiments, we have used audio flows requiring a pulse code modulation stream with high-quality constant bitrate of 512Kbps (note that phone quality audio requires 64Kbps, while CD-quality flows need 1378Kbps), in order to stress the bandwidth available in the piconet.

Figures 4a, 4b, and 4c report, respectively, throughput, packet delay, and standard deviation (their average values on a set of one hundred tests) for ACL-based and ASB-based audio distribution, with a

varying number of slaves. In all the tests we have used the DH5 packet type, which gives the best performance results among all the types of packets without payload protection available for ACL and ASB links (DH1, DH3, DH5). We have not considered DM packets, whose payload is Forward Error Correction-coded, because less suitable for audio streaming [23].

As expected, in the ASB case the performance is mostly independent from the number of slaves. In the ACL case, instead, the performance has shown to significantly deteriorate with the increasing of the number of slaves. The low throughput of ASB links is motivated by the small size used by the controller of currently available Bluetooth USB adapters for ASB packets (22 bytes, with a 17 bytes payload and a 22.7% overhead posed by packet headers) in comparison with a significantly greater size for ACL packets (681 bytes with 672 bytes of payload).

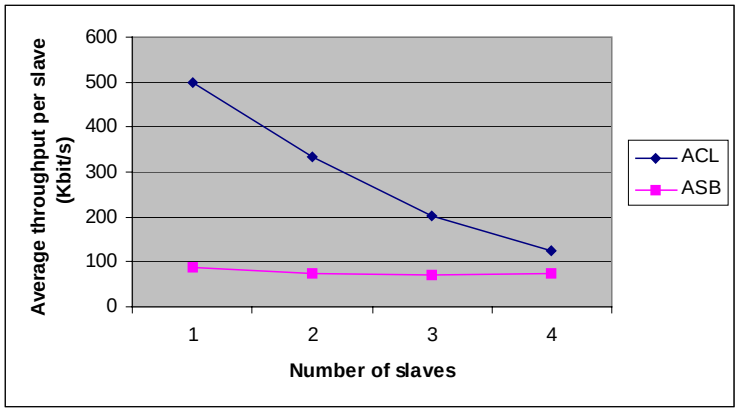

Figure 4a. Average throughput per slave for ACL and ASB links.

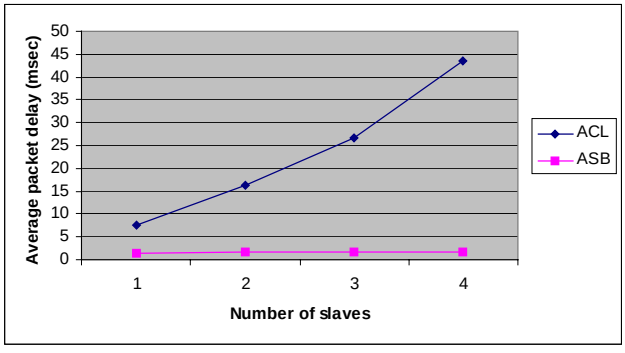

Figure 4b. Average packet delay for ACL and ASB links.

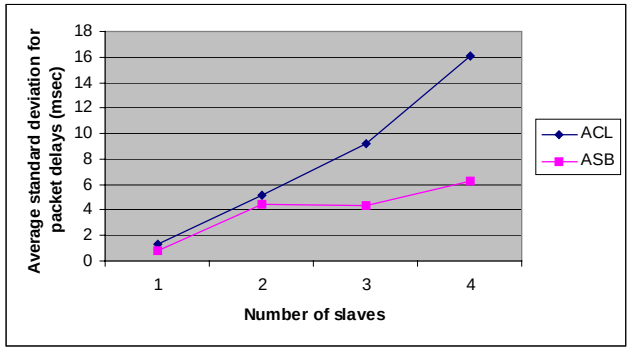

Figure 4c. Average standard deviation in packet delay for ACL and ASB links.

In addition, we have also measured the packet loss for both ACL and ASB links. In the ACL case there is a negligible percentage of lost data. In the ASB case, we have experienced a higher packet loss (around 7% on average), mostly independent of the number of slaves in the piconet. This packet loss affects the overall quality of the transmission over ASB links but, however, is acceptable for audio streaming applications. This corroborates the idea that ASB links can be a valid low bandwidth alternative to ACL links for audio streaming, especially in the case of time-bound traffic and of a high number of concurrent

slaves. For this reason, to exploit the piconet bandwidth at its best, the ubiQoS proxy decides to provide silver users with ACL links if and only if there are no more than 4 ACL links already in use within the locality. On the contrary, if 4 ACL links are already in use and the new client is not privileged with regards to the already accepted ones, the proxy serves it by exploiting an ASB link not to deteriorate the performance of the working ACL-based connections.

Finally, it is worth noticing that the experimental results also point out that the Java-based application-level approach achieves performance results not far from the raw Bluetooth hardware performance, in terms of both throughput and packet delay [23]. This confirms the viability of flexible application-level overlays to support QoS-enabled audio transmission in the Bluetooth-enabled wireless Internet.

7 Conclusions and Current Work

Bluetooth is emerging as a market-successful connectivity technology for the last-meter access to the wireless Internet. The integration of the traditional best-effort Internet with Bluetooth PANs calls for middleware overlay networks capable of supporting QoS-enabled services, in particular by operating QoS management operations at the wired/wireless edges. Recent standardization efforts are making possible to design and implement first Java-based portable middleware solutions for these scenarios. The development and deployment of the ubiQoS prototype have produced first experimental results that demonstrate the feasibility of the application-level middleware approach, at least for supporting audio streaming at the usual Internet transmission rates.

The encouraging experimental results obtained are encouraging further research activities within the framework of the ubiQoS project. We are extending the ubiQoS middleware to include also the Bluetooth guaranteed QoS management functions supported by the novel eSCO specifications. In addition, we are working on the porting of the JSR82ext native part implementation over both the Affix stack for Linux and the FreeBT Bluetooth stack for Microsoft Windows XP [24, 25]; the next JSR82ext release will be able to dynamically discover the locally installed Bluetooth software stack and the underlying operating system, so to bind to the needed platform-specific JSR82ext modules only at provision time.

Acknowledgments

Work supported by the Italian Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR) in the framework of the FIRB WEB-MINDS Project "Wide-scale Broadband Middleware for Network Distributed Services" and by the Italian Consiglio Nazionale delle Ricerche (CNR) in the framework of the Strategic IS-MANET Project "Middleware Support for Mobile Ad-hoc Networks and their Application".

References

[1] K. Sairam, N. Gunasekaran, S.R. Redd, “Bluetooth in Wireless Communication”, *IEEE Communications*, Vol. 40, No. 6, June 2002.

[2] P. Johansson, M. Kazantzidis, R. Kapoor, M. Gerla, “Bluetooth: an Enabler for Personal Area Networking”, *IEEE Network*, Vol. 15, No. 5, Sept.-Oct. 2001.

[3] H. Xu, J. Diamand, A. Luthra, “Client Architecture for MPEG-4 Streaming”, *IEEE Multimedia*, Vol. 11, No. 2, Apr.-June 2004.

[4] P. Bellavista, A. Corradi, R. Montanari, C. Stefanelli, “Context-aware Middleware for Resource Management in the Wireless Internet”, *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, Dec. 2003.

[5] R. Oppliger, “Security at the Internet Layer”, *IEEE Computer*, Vol. 31, No. 9, Sep. 1998.

[6] P. Bellavista, A. Corradi, C. Stefanelli, “Mobile Agent Middleware for Mobile Computing”, *IEEE Computer*, Vol. 34, No. 3, March 2001.

[7] Sourceforge.Net – *The JavaBluetooth Stack*, http://sourceforge.net/projects/javabluetooth

[8] Sun Microsystems, Inc. – *The Java Native Interface 1.1 Specification*, http://java.sun.com/j2se/1.4.2/ docs/guide/jni/spec/jniTOC.html

[9] P. Bellavista, A. Corradi, C. Stefanelli, “Application-level QoS Control for Video-on-Demand”, *IEEE Internet Computing*, Vol. 7, No. 6, Nov.-Dec. 2003, pp. 16-24.

[10] Sun Microsystems, Inc. - *The Java Media Framework (JMF) API*, http://java.sun.com/products/java-media/jmf/

[11] T. Braun, "Internet Protocols for Multimedia Communications - Resource Reservation, Transport, and Application Protocols", *IEEE Multimedia*, Vol. 4, No. 4, 1997.

[12] Sun Microsystems, Inc. – *The Mobile Media API (MMAPI)*, http://java.sun.com/products/mmapi/

[13] Sun Microsystems, Inc. – *The Jini Reggie Discovery Implementation*, http://java.sun.com/developer/ products/jini/index.jsp

[14] W3 Consortium - Composite Capability/Preference Profiles (CC/PP) – http://www.w3.org/Mobile/

[15] Bluetooth SIG – *Bluetooth Core Specification v1.2*, https://www.bluetooth.org/foundry/adopters/ document/Bluetooth_Core_Specification_v1.2

[16] M. Holtmann, *Bluetooth Hardware Support for BlueZ*, http://www.holtmann.org/linux/bluetooth/ devices.html

[17] R. Kapoor, Ling-Jyh Chen, Yeng-Zhong Lee, M. Gerla, “Bluetooth: Carrying Voice over ACL Links”, *4th IEEE Int. Workshop on Mobile and Wireless Communications Network*, 2002.

[18] Java Community Process – *Java APIs for Bluetooth (JSR82)*, http://jcp.org/en/jsr/detail?id=82

[19] Sun Microsystems, Inc. - *Java 2 Platform: Micro Edition (J2ME) and Connected Limited Device Configuration (CLDC)*, http://java.sun.com/j2me/

[20] Sourceforge.Net – *JBlueZ: the Java Extension for the BlueZ Bluetooth Protocol Stack*, http://jbluez. sourceforge.net

[21] BlueZ Project – *BlueZ, the Official Linux Protocol Stack*, http://www.bluez.org

[22] Avetana – *The Avetana JSR82 API Implementation*, http://www.avetana-gmbh.de/avetana-gmbh/jsr82.xml

[23] S. Zurbes, "Considerations on Link and System Throughput of Bluetooth Networks", *11th IEEE Int. Symp. Personal, Indoor and Mobile Radio Communications (PIMRC)*, Sep. 2000.

[23] Sourceforge.Net – *The Affix Bluetooth Protocol Stack for Linux*, http://affix.sourceforge.net

[24] FreeBT – *The FreeBT Bluetooth Protocol Stack for Windows*, http://www.freebt.net