The ubiQoS Middleware for Audio Streaming to Bluetooth Devices

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 full and seamless integration of wireless devices with traditional fixed networks is more and more important to foster the mobile and ubiquitous access to the Internet. In particular, the heterogeneity and resource limitations of wireless devices motivate novel support infrastructures that can facilitate the wired-wireless integration and can provide service tailoring depending on client characteristics. The paper presents an application-level portable middleware, called ubiQoS, for QoS-enabled audio streaming to Bluetooth clients. ubiQoS exploits support proxies for QoS tailoring and for managing the QoS over the last segment of the audio distribution path towards the clients, by using different types of Bluetooth links. Proxies execute at the wired-wireless network edges and can even migrate to follow the device movements, where and when needed. The reported experimental results show the feasibility of the application-level approach in the challenging case of QoS-enabled audio streaming to resource-limited Bluetooth devices.

1. Introduction

The growing market of Bluetooth portable appliances is enabling the realization of spontaneous networks of devices that are located within the range of a single user, and often referred to as Personal Area Networks (PANs). In addition, there is an increasing research and commercial interest in allowing PANs to 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 [1]. The ultimate goal is to ubiquitously provide both traditional and location-dependent services in the wireless Internet [2].

Service providers and wireless network operators have to address new technical challenges for the seamless integration of portable devices in the wireless Internet. A primary issue is the wide heterogeneity of the hardware/software capabilities of wireless access devices, e.g., screen size/resolution and supported multimedia formats/players. In addition, portable terminals usually have limited resources in terms of processing, memory, storage, and network connectivity. In this scenario, resource-consuming services designed for the fixed network, such as multimedia streaming, requires being downscaled to fit the limited clients. Moreover, the wide heterogeneity makes impractical to provide statically tailored service versions to all the possible categories of access terminals.

One of the most widespread wireless technology is Bluetooth, which is the primary solution today for enabling the realization of PANs [3]. Bluetooth-enabled portable devices, such as laptops, phones, and PDAs, can interconnect to form a piconet, which consists of one master and up to 7 slaves. The master device has direct visibility of all slaves in the piconet and can handle two types of connections with different Quality of Service (QoS) levels. Asynchronous Connection-Less (ACL) links provide a packet-oriented service. Synchronous Connection Oriented (SCO) links, instead, are circuit-oriented connections, designed to support time-bounded transmissions such as voice streaming. The usage of SCO links may leave very little of the piconet bandwidth available to ACL links [4]. Depending on service and user requirements, streaming services to Bluetooth devices should be capable of choosing the most suitable link type.

The paper proposes a middleware-level solution, called ubiQoS, for the QoS management of mobile multimedia services to Bluetooth devices in the best-effort wireless Internet. ubiQoS adopts an application-level approach that facilitates dynamic un/installation of infrastructure/service components, application-specific service tailoring and adaptation, security, and

Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)
0-7695-2208-4/04 $20.00 © 2004 **IEEE**

interoperability [5, 6]. ubiQoS supports multimedia streaming with differentiated QoS levels and can dynamically tailor service provisioning to the specific requirements of the served users, the device characteristics, and the available wireless connections [7].

In particular, the paper focuses on how to support the last-meter Bluetooth-based connection for QoS-enabled audio streaming services. The main design guideline is to exploit support proxies that are located at the edges between the fixed Internet and the Bluetooth PANs. Proxies work as masters in the Bluetooth piconets of access devices and dynamically tailor the QoS level of audio streams to the characteristics of both the Bluetooth links used and the target clients. 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 and where needed. In addition, mobile proxies can continuously serve their associated clients by following the device movements between different wireless points of attachment to the Internet.

To achieve portability in the open wireless Internet, we have designed a Java-based interface, called JSR82+SCO interface in the following, that extends the standard Java APIs for Bluetooth (JSR82) [8] with the SCO link support. On the one hand, our implementation extends JavaBluetooth (with its current limitations to the serial profile [9]) to fully support JSR82-based ACL management. On the other hand, we have developed from scratch the Java-based SCO support, also by integrating with platform-dependent native SCO libraries via the Java Native Interface [10]. ubiQoS proxies exploit the implemented JSR82+SCO interface to control Bluetooth links from within the standard Java Virtual Machine (JVM).

The paper also reports the first experimental results of the performance of the ubiQoS prototype in the provisioning of audio streaming with differentiated QoS levels to Bluetooth devices. The results show that the Java-based application-level approach introduces an acceptable and very limited degradation of the Bluetooth link performance.

The rest of the paper is structured as follows. Section 2 introduces Bluetooth, describes the QoS characteristics of ACL/SCO links, and briefly overviews the state-of-the-art of the Java-based technologies to interwork with Bluetooth. Section 3 motivates the adoption of proxy-based service tailoring. Section 4 gives an overview of ubiQoS, while Section 5 describes how ubiQoS proxies manage the Bluetooth-based last-meter links. Section 6 reports some performance results of the implemented prototype. Concluding remarks and directions for future work follow.

2. Java-based Bluetooth Support for the Provisioning of QoS-enabled Services

Bluetooth is an emerging technology in the wireless world and is the standard solution for PAN applications. The Bluetooth specification defines two main classes of traffic: unframed data traffic, with guaranteed QoS requirements, and framed data traffic, with both best-effort and guaranteed QoS [11]. Unframed data traffic is carried over SCO and enhancedSCO (eSCO) baseband links; framed data traffic is carried over ACL and Active Slave Broadcast (ASB) links.

SCO and eSCO baseband links are point-to-point bi-directional, symmetrical (eSCO links can also be asymmetrical), isochronous, and have a constant bit-rate. The bit-rate is fixed to 64Kb/s for SCO and user-defined for eSCO. SCO and eSCO logical transports do not support the multiplexing of data streams; when needed, multiplexing operations should be performed at the application level.

ACL links are bi-directional, connection-oriented, asynchronous or isochronous, and support the specification of QoS settings to indicate the desiderata for the delivery of the 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. 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, e.g., the new specification introduces the HCI Flow Specification command that can be used to specify QoS flow parameters in an asymmetrical way for (even already established) ACL connections. Unfortunately, the largest part of the (firmware of) Bluetooth chips in commerce still provides only a partial implementation of the QoS support, and very first chips compliant with Bluetooth v1.2 are being commercialized in these days.

In addition, Bluetooth also defines connection-less ASB links for broadcasting traffic from the master to all the slaves in the piconet. The ASB-based transmission is best-effort, with no possibility of QoS management.

Java-based applications can interwork with Bluetooth via the recently approved JSR82 standard interface [8]. JSR82 allows the creation of different types of connections, e.g., L2CAP and RFCOMM, and supports the Service Discovery Protocol and the Object Exchange

protocol [11]. In addition, JSR82 provides the support for several profiles defined in the Bluetooth specification: 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; as a result, the JSR82 API can be also offered on top of any compliant implementation of the limited Java 2 Micro Edition with the Connected Limited Device Configuration [12].

Unfortunately, JSR82 lacks some important functionality. First of all, the JSR82 specification does not include the support for SCO, eSCO, and ASB links, thus complicating the portable implementation of Java-based audio streaming applications. In addition, 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 services, thus making it unusable in an open environment.

Also in response to the above JSR82 limitations, several Java libraries, alternative to JSR82, continue to be adopted and proposed, such as JBlueZ [13]. These libraries provide Java-based applications with full-featured access to existing Bluetooth stacks via a non-standard proprietary API. The libraries often exploit the lower-level interface exported by the underlying Bluetooth stack implementation they work in conjunction with. They usually integrate with their Bluetooth stack by using the mechanisms of the Java Native Interface (JNI), the standard API for interfacing native modules and the JVM [10]. The adoption of JNI ensures the code portability over any standard JVM implementation.

3. Middleware Approaches to the QoS Tailoring of Audio Streams

Service provisioning to wireless portable devices requires dynamically downsizing service content to the characteristics of both access terminal and wireless connections. In particular, the tailoring of the dynamic content is crucial for resource-consuming services such as audio streaming. For instance, when streaming an audio guide to a small group of museum visitors with Bluetooth-enabled earphones, DVD-quality 153.6 Kb/s audio flows in the .WAV format should be dynamically transcoded to minor quality 30 Kb/s MP3 versions, in order not to overload, uselessly and excessively, the Bluetooth ACL links in the visitors PAN.

An important design choice is where to operate the tailoring of the transmitted audio flows, typically by reducing the audio quality and transcoding the representation format. In server-based solutions, the host that provides the service is in charge of either selecting the version of the requested flow with the most suitable QoS level among a pool of statically pre-determined ones (off-line tailoring) or dynamically operating downscale transformations on the unique high-quality stored version (on-the-fly tailoring). These solutions concentrate service and tailoring functions on the server side. This approach has several drawbacks, from overloading the server computing capabilities to making distributed caches ineffective.

On the opposite, client-based tailoring requires a significant amount of computational resources on the client side and has higher bandwidth requirements. However, it usually improves the effectiveness of distributed caching solutions. Let us observe that most Bluetooth-based devices have very limited resources and cannot directly perform downsizing operations on-board. When the downscaling is needed, e.g., because the client device cannot play an audio file with the provided format/QoS level, the client necessarily has to delegate the service tailoring to middleware components hosted in the fixed infrastructure.

This motivates the design of a distributed and decentralized infrastructure consisting of middleware proxy components working on behalf of the device. Proxies can help in smoothing the discontinuities between the fixed Internet and the Bluetooth PAN. They are mainly in charge of tailoring the QoS level of audio flows depending on the device characteristics, and of properly supporting the device connectivity by adapting service provisioning to the exploited Bluetooth link.

4. The ubiQoS Middleware

ubiQoS is a middleware support for the QoS-enabled provisioning of dynamically tailored multimedia flows to heterogeneous clients. ubiQoS has been first developed for nomadic computing deployment scenarios over traditional wired networks [7]. Here, the paper focuses on the significant ubiQoS extension to the wireless Internet, and, in particular, on the peculiar aspects of the QoS management of audio streaming in the case that Bluetooth is the connectivity technology enabling the last-meter client connection.

ubiQoS provides any Bluetooth client device with one companion entity, called *shadow proxy*, and with

several application-specific processors, called *QoS adapters*.

The shadow proxy works on behalf of the associated Bluetooth device, usually at the edge between the device PAN and the Internet. Depending on user requirements, device characteristics, and available audio servers, the proxy decides the tailoring operations to perform. QoS adapters, instead, are the middleware components responsible for the actual QoS management operations (reduction of bitrate, format transcoding) on audio flows.

ubiQoS proxies and QoS adapters are hosted in execution environments, called places, that offer the basic services for mobile agent communication and migration. Places typically model nodes and can be grouped into domains that correspond to network localities, e.g., Local Area Networks with IEEE 802.11b/Bluetooth access points providing wireless connectivity to WiFi/Bluetooth portable devices (see Figure 1).

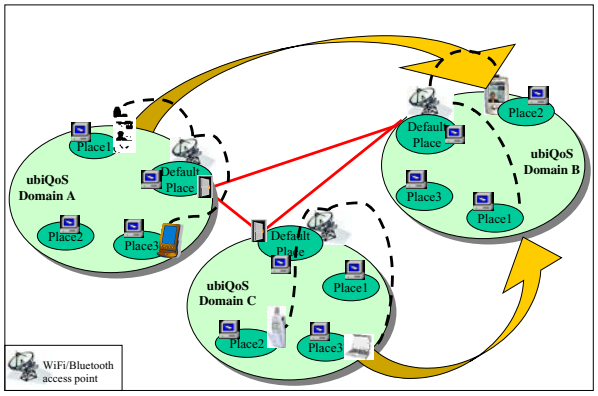

Figure 1. Wireless devices roaming among ubiQoS domains

Figure 2 concentrates on a single domain and depicts a typical ubiQoS deployment scenario. At the client request to look up for audio flows, the shadow proxy interrogates the ubiQoS naming service [7] to obtain the list of available audio servers, either in the locality or in other ubiQoS domains. Then, the proxy retrieves the applicable profiles, i.e., the metadata describing the characteristics of its companion device, the preferences of its currently logged user, and the characteristics of the requested audio flow. Depending on these profiles, the proxy decides which downscaling transformations to perform by instantiating and interworking with the suitable QoS adapters.

Shadow proxies and QoS adapters are implemented as mobile agents on top of the SOMA platform (available at `lia.deis.unibo.it/Research/SOMA`). They execute in the ubiQoS domain where the portable devices are currently attached and can follow them in the case of runtime movements between different points of attachment to the fixed Internet. We usually associate one shadow proxy for each portable device (with a 1-to-1 mapping). However, it is also possible to define group shadow proxies in charge of managing a set of portable devices with synchronization constraints, e.g., when distributing a synchronized audio guide to a tourist group visiting the rooms of a museum.

QoS adapters are in charge of audio compression, e.g., reduction of bit/sample rate, and format transcoding, e.g., from WMA/OGG/WAV to MP3, to tune the provided QoS level to suit the profiles of both device characteristics and user preferences. Their mobile agent implementation permits to dynamically migrate the needed adaptation code to the places where it is not already available, by following the movements of associated shadow proxies. Let us rapidly observe that SOMA mobile agent migration only moves the code not already present at the new place; if the code is already there, SOMA only migrates the state and re-instantiates the agent by exploiting the transmitted state and the local code [6]. QoS adapters receive audio flows, operate the flow transformations decided by their associated proxy, and forward processed flows to device-specific audio players. The current implementation of QoS adapters is based on SUN Java Media Framework (JMF) [14]. For the transport and control of packet flows towards the servers, QoS adapters exploit the JMF APIs to integrate with the Real-time Transport Protocol (RTP) and its corresponding RTCP control protocol [15].

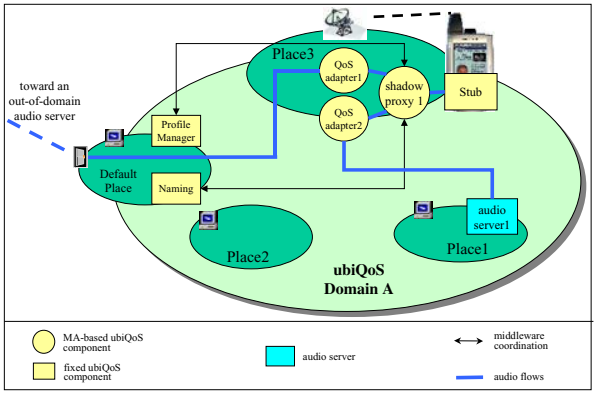

Figure 2. The ubiQoS middleware while distributing audio flows to wireless clients

Shadow proxies and QoS adapters are generally portable on any platform that hosts a standard JVM. For performance sake, QoS adapters sometimes exploit local

plug-ins available as native components. To achieve portability, they retrieve dynamically the list of plug-ins installed on their places to bind only to the locally available components. The current implementation of ubiQoS exploits a proprietary lightweight solution to discover in-place/domain available resources and service components; we are working on migrating towards open standard solutions to interface with local resources, such as the OSGi Alliance proposal [16].

In addition to these two primary middleware components, ubiQoS includes a profile manager service and device-specific stubs. The profile manager service stores profiles of supported devices, registered users, and available audio flows. It implements a partitioned and partially replicated directory service specialized for profiles. ubiQoS profiles are represented according to the W3C Composite Capabilities/Preference Profile (CC/PP) standard format [2, 17]. CC/PP profiles are processed via a proprietary Java library for profile parsing and management developed within the ubiQoS project, due to the lack of standard Java-based supports for CC/PP at the time of ubiQoS development. The next ubiQoS release will replace that library with the recent reference implementation of the JSR188 API for CC/PP profiles processing [18]. Device-specific stubs are the only middleware components that run in the Bluetooth-enabled access terminals. They handle the communication forwarding between the on-board audio player and the associated shadow proxy and are implemented on top of the J2ME/CLDC/MIDP/JSR82 software suite.

Implementation insights about the above ubiQoS middleware components and functionality are out of the scope of the paper and can be found in [19]. In the following, this paper specifically focuses on the peculiar aspects of distributing audio streaming over the Bluetooth last meter with differentiated QoS levels. In particular, it details how ubiQoS shadow proxies handle the different Bluetooth connections available, how they decide the allocation of Bluetooth connections to device-specific stubs depending on the associated user classes, and how ubiQoS achieves a portable Java-based implementation of such a support.

5. The Management of Last-Meter Bluetooth Links 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 receive audio flows with the properly downscaled QoS level. In the following, the paper focuses on the PAN side and on how the proxies manage the different Bluetooth links available in their PANs.

The ubiQoS proxy can support different classes of users, e.g., gold, silver, and bronze, with differentiated QoS levels. It retrieves the profiles of involved users and audio flows; depending on these metadata, it chooses how to manage the different Bluetooth communication links. As already stated, the ubiQoS proxy has been designed to exploit both SCO and ACL links. In particular, the proxy performs an application-level ranking of the current clients to decide the allocation of the different links. The usage of one SCO link is suggested when providing an audio flow of 64 kbps or less to a single gold client with guaranteed QoS requirements. However, even in the best case of SCO links using HV1 packets and requiring only 2 time slots every 6, the SCO exploitation leaves very little of the piconet bandwidth available to other links. In practice, no more than two SCO links can be concurrently active in the same piconet and, also in this case, the other connections may easily get starved [4].

For this reason, when concurrently serving several clients, the ubiQoS proxy mainly exploits ACL-based connections, and possibly allocates one single SCO link to the best-ranked gold client that requires guaranteed QoS and requests a bandwidth-compatible audio flow. The allocation of the single SCO link can be dynamically modified: for instance, after the arrival of a new gold user with better ranking, the proxy can choose to de-class the client that currently exploits SCO, by switching it to an ACL-based connection.

Let us observe that the current ubiQoS implementation does not take into consideration the usage of eSCO and ASB links. This choice is motivated by the fact that the two link types above are still not fully supported in most Bluetooth commercial solutions. eSCO has been introduced in the recent Bluetooth 1.2 specification and, to the best of our knowledge, the market still lacks eSCO-compliant devices. In addition, most of the widely adopted Bluetooth stack implementations, such as BlueZ [20], do not currently support the broadcast transmission over ASB links.

To achieve a portable implementation of ubiQoS proxies, we have implemented them by using the Java technology. The Java choice is almost compulsory when willing to provide middleware components that can change their location at runtime, as demonstrated by the implementation of the most recent mobile agent platforms [21]. To have full control of both ACL and SCO links, Java-based ubiQoS proxies exploit the JSR82+SCO interface, which we have designed and implemented within the ubiQoS project.

Figure 3 depicts the layered architecture of the JSR82+SCO interface (the light colored blocks in the

JVM level part represent the components originally developed in our research). Our implementation significantly extends the open source and JSR82-compliant JavaBluetooth protocol stack [9]. JavaBluetooth does only support serial Bluetooth adapters by exploiting the *javax.comm* API and cannot interface with native Bluetooth device drivers. To overcome these limitations, we have developed BlueZTransport, a low-level interface between the JavaBluetooth protocol stack and BlueZ, the native Bluetooth v1.1 protocol stack included in the Linux kernel, via the Java Native Interface (JNI) [20]. This enables our JSR82+SCO implementation to communicate with the HCI interface of Bluetooth devices via low-level BlueZ native API, thus allowing the usage of our JSR82+SCO stack with the whole range of Bluetooth adapters supported by BlueZ.

We have also developed from scratch the Java-based SCO support by integrating with basic support mechanisms provided by native SCO libraries. The platform-dependent native support for SCO sockets is integrated in a JVM-independent way via JNI. From the developer point of view, the creation of SCO connections in JSR82+SCO closely resembles the one defined by JSR82 for the creation of L2CAP connections. The Generic Connection Framework of the J2ME CLDC software suite provides the base connection for the implementation of the communication protocol. JSR82+SCO allows developers to exploit SCO connections almost transparently: developers should simply indicate SCO-specific URLs (starting with *btsco*) and should not specify a Protocol Service Multiplexer parameter for their SCO connections.

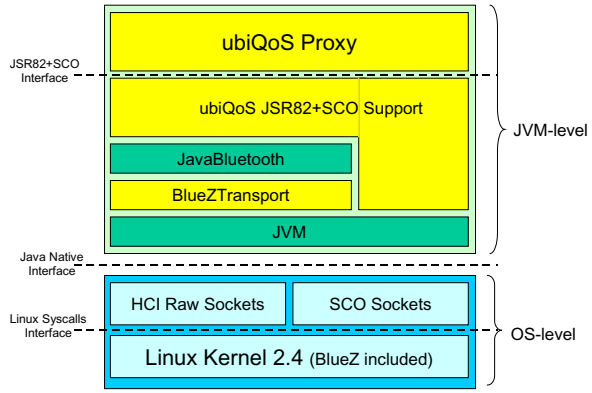

Figure 3. The software architecture of the ubiQoS proxy

Additional details about the implementation of the JSR82+SCO interface and its downloadable code are available at `http://lia.deis.unibo.it/Research/ubiQoS/audioStreaming/`

6. Experimental Results

We have organized an experimental testbed to measure the performance of the ubiQoS middleware when supporting audio streaming over Bluetooth, especially to verify the feasibility of the Java-based application-level approach. In particular, our experiments have focused on measuring the average throughput, the average packet delay, and the average standard deviation for packet delays when the ubiQoS proxy handles Bluetooth links via the JSR82+SCO interface.

The testbed consists of one workstation, which hosts the execution of the proxy (piconet master), and of some laptops (piconet slaves), where the ubiQoS device-specific stubs and the audio players run. The laptops are all located within the range of the Bluetooth master visibility. The master and the slaves host Linux (kernel version 2.4.25-pre7) with the latest version of the BlueZ userspace tools (bluez-utils 2.4, bluez-libs 2.5). They exploit Bluetooth USB adapters based on the Cambridge Silicon Radio Bluecore chip [22], e.g., D-Link DBT-120 and Belkin F8T003 adapters.

The master can distribute different audio flows, with different QoS levels, to the different slaves. To show the ubiQoS performance under heavy network load conditions, the reported experimental results are obtained by transmitting audio flows with a constant bitrate of 512kbps and with packets of 672 bytes. Such high bandwidth-requiring flows saturate the piconet bandwidth with a limited number of concurrent slaves. Note that, for instance, when adopting uncompressed pulse code modulation, a phone quality audio requires 64kbps and a CD-quality one 1378kbps, while by using MPEG-3 compression very high-quality audio flows typically require 320kbps.

First, we have measured the throughput, the packet delay, and the associated standard deviation when serving a growing number of concurrent slaves by exploiting only ACL. We have experimented all the different types of packets (DH1, DH3, DH5) available for ACL links without payload protection. We have not considered ACL DMx packets, whose payload is Forward Error Correction-coded with a simple 2/3 rate block code, because less suitable for audio streaming [23]. Figures 4, 5, and 6 report, respectively, the average results for throughput, delay, and standard deviation.

The results show that ACL links can well support the distribution of audio flows when exploiting DH3 and DH5 packets, also under different load conditions of the piconet. The total bandwidth is efficiently exploited and, when the number of slaves grows, is equally distributed between the slaves. This is a consequence of the centralized control in the access to the Bluetooth channels in a piconet, which successfully prevents the

slaves from starvation. DH5 packets achieve the best performance because of the lower overhead due to the DH5 header/payload ratio. It is worth observing that DH3 packets achieve performance results similar to DH5, while DH1 performs significantly worse and does not fit well the audio streaming requirements, especially in terms of packet delay when the number of concurrent slaves is greater than 3.

With regard to SCO links, we have experienced that the BlueZ SCO support in the Linux kernel is still immature (in both v2.4 and v2.6) and leads to very poor performance. In addition, our experiments have pointed out that the performance of ACL links significantly suffers from the concurrent usage of SCO. This result has also emerged in [4] that, with a different approach and different goals, exploits SCO links for voice transmission. For this reason, it seems reasonable to use a single SCO link only when an overloaded PAN hosts a single gold user with guaranteed bandwidth requirements.

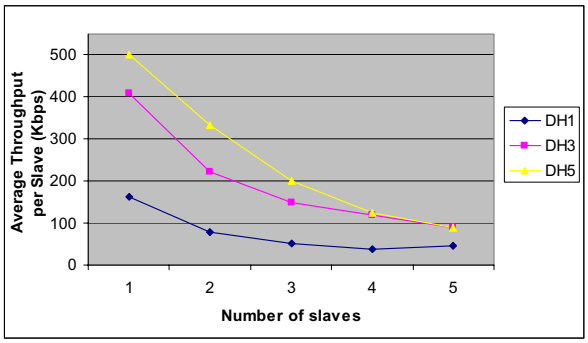

Figure 4. Average throughput per slave when exploiting ACL DH1/DH3/DH5 packets.

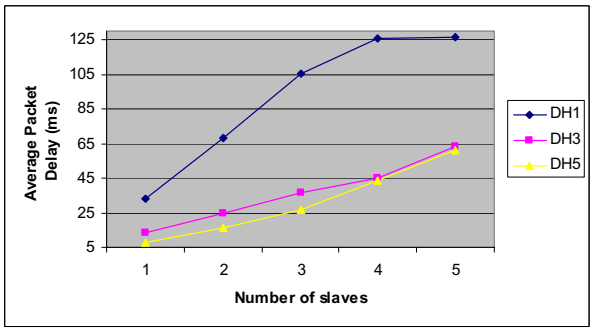

Figure 5. Average packet delay when exploiting ACL DH1/DH3/DH5 packets.

As a conclusive remark, the collected data point out that the performance of our Java-based middleware, in terms of both throughput and packet delay, is close to the Bluetooth raw hardware performance [4]. This confirms the viability of flexible application-level overlays to support QoS-enabled audio streaming, even when exploiting the Bluetooth links as the last-meter connectivity technology in the wireless Internet.

Figure 6. Average standard deviation for packet delays when exploiting ACL DH1/DH3/DH5 packets.

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 pointing out that the middleware approach can satisfy both the flexibility and the performance requirements of audio streaming services over Bluetooth. In fact, notwithstanding the Java-based implementation and the application-level approach, ubiQoS achieves throughput and packet delay performance close to the Bluetooth raw hardware one, thus confirming the possibility to exploit Bluetooth-based last-meter connectivity also for audio distribution, at least at the currently usual Internet transmission rates. At the same time, the prototype deployment has revealed that the BlueZ SCO support in Linux is still immature to fully enable the exploitation of SCO links.

Further research activities are going on within the framework of the ubiQoS project. We are currently working on the exploitation of ASB links to distribute the same audio flow to slaves in the same piconet, in order to enable services such as synchronous group visits to a museum exhibition. In addition, we are extending the middleware to include also the support for the guaranteed QoS HCI Flow Specification of Bluetooth v1.2. Finally, we are working on the porting of the implementation of the JSR82+SCO module over the BluePC Bluetooth stack for Windows XP, also to verify the effectiveness of the SCO support in this software suite [24].

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] P. Johansson, M. Kazantzidis, R. Kapoor, M. Gerla, "Bluetooth: an Enabler for Personal Area Networking", *IEEE Network*, Vol. 15, No. 5, Sept.-Oct. 2001, pp. 28-37.

[2] 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, pp. 1086-1099.

[3] K. Sairam, N. Gunasekaran, S.R. Redd, "Bluetooth in Wireless Communication", *IEEE Communications*, Vol. 40, No. 6, June 2002, pp. 90-96.

[4] 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.

[5] R. Oppliger, ″Security at the Internet Layer", *IEEE Computer*, Vol. 31, No. 9, Sep. 1998, pp. 43-47.

[6] P. Bellavista, A. Corradi, C. Stefanelli, "Mobile Agent Middleware for Mobile Computing", *IEEE Computer*, Vol. 34, No. 3, March 2001, pp. 73-81.

[7] 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.

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

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

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

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

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

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

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

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

[16] Open Services Gateway Initiative – *OSGi Service Platform Release 3 Spec.*, http://www.osgi.org

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

[18] Java Community Process – *Java APIs for CC/PP Processing (JSR188)*, http://jcp.org/en/jsr/detail?id=188

[19] P. Bellavista, A. Corradi, "How to Support Internet-based Distribution of Video on Demand to Portable Devices", *7th IEEE Int. Symp. on Computers and Communications (ISCC'02)*, pp. 126-132, July 2002.

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

[21] R.H. Glitho, E. Olougouna, S. Pierre, "Mobile Agents and their Use for Information Retrieval: a Brief Overview and an Elaborate Case Study", *IEEE Network*, Vol. 16, No. 1, pp. 34-41, Jan.-Feb. 2002.

[22] Cambridge Silicon Radio – *CSR's Bluecore Chips*, http://www.csr.com

[23] S. Zurbes, "Considerations on Link and System Throughput of Bluetooth Networks", *11th IEEE Int. Symp. Personal, Indoor and Mobile Radio Communications (PIMRC)*, pp. 1315-1319, Sep. 2000.

[24] Impulsesoft – *The BluePC Bluetooth Protocol Stack*, http://www.impulsesoft.com/HomePage/products/bluepc.html