A MIDDLEWARE FRAMEWORK PROVIDING ADAPTIVE QUALITY OF SERVICE FOR BLUETOOTH

By

THOR EGIL SKAUG

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY

School of Electrical Engineering and Computer Science

May 2004

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of THOR EGIL SKAUG find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGMENT

I would like to thank my advisor, Dr. David E. Bakken, for talking me into coming to WSU, for all the assistance and help he has given me, for devoting so many hours of his own time, and finally, for his incredible kindness towards his students.

I would also like to thank Dr. Sirisha R. Medidi and Dr. John C. Shovic, for their help, and for taking the time to be on my committee, and Ms. Ruby Young for her kindness and helpfulness towards the graduate students at EECS.

I have to thank all of the current and previous MicroQoSCORBA project members; you have all done some outstanding work: Dr. A. David McKinnon, Olav Haugan, Tarana Damania, Dr. Wesley Lawrence, Kevin Dorow, Eivind Næss, Kim Christian Swenson and Ryan Johnston.

A special mention to all my friends back in Norway – I miss you and I wish I could see you more often, and the friends I have made during my time here at WSU; you have definitely made this worthwhile.

Barb and Alan – thank you for your incredible hospitality and good company, you have made me feel at home in the US.

I give the most sincere thanks to my parents and my brother for making it possible for me to come to the WSU, for supporting me throughout my studies, and for believing in me. Without you I would never have been able to do this.

This research was supported in part by two Cisco University Research Program donations and Grant NSF-CISE EHS-0209211 from the National Science Foundation's Embedded and Hybrid Systems program.

Finally: Thank you Emily, for always being there for me, for your inspiration and support, and your patience. This thesis is dedicated to you.

PUBLICATIONS

Thor Egil Skaug, David E. Bakken, and John C. Shovic, "A Middleware Framework Providing Adaptive Quality of Service for Bluetooth", submitted to *The 5th International Middleware Conference*, Toronto, Ontario, Canada, October 18th – 22nd, 2004.

A MIDDLEWARE FRAMEWORK PROVIDING

ADAPTIVE QUALITY OF SERVICE FOR BLUETOOTH

Abstract

by Thor Egil Skaug, M.S.
Washington State University
May 2004

Chair: David E. Bakken

Bluetooth was originally conceived as a replacement for wires on human interface devices such as keyboards and headsets. More recently, its range has been extended such that it has the potential to become a viable and inexpensive alterative to other wireless technologies. However, its suitability for more general-purpose applications and traffic is an open question, especially with regards to application-level quality of service (QoS) control. This thesis analyzes the QoS mechanisms and hooks Bluetooth provides in terms of their potential as a building block for middleware-level mechanisms. In particular, the ability to add higher-level mechanisms useful for adapting to changing conditions on a wireless link is assessed. The thesis describes the design and implementation of a configurable set of middleware-level mechanisms that provides such adaptation, and provides an experimental evaluation of this framework.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Audere est Facere

Chapter 1

Introduction

Embedded applications have become increasingly commonplace in recent years. In the last decade embedded CPUs and supporting chipsets have been made much cheaper and smaller. As a result, over 90% of all CPUs produced in recent years are used in embedded systems [TUR03]. In recent years, embedded processors are commonly being networked, and increasingly with wireless network technologies.

Bluetooth is a popular standard for wireless networking. It was originally conceived as a replacement for wires and infrared communication on human interface devices ranging from cellular phones, keyboards, GPS devices, and headsets. Its initial range was limited to 10 meters (Class 2 Devices). In recent years, the range has been extended to 100 meters (Class 1 Devices). Since Bluetooth is generally considered to be less expensive to implement than other wireless standards such as IEEE 802.11b, this opens up the possibility for Bluetooth to be a less expensive and smaller replacement in some circumstances. This could in turn open up even more application domains for control or monitoring by embedded processors.

Bluetooth offers a variety of quality of service mechanisms in its API. However, all of its mechanisms are reactive, meaning they detect problems and correct them after the fact, very much in the style of stop-and-wait protocols. This opens up an opportunity for proactive mechanisms on top of Bluetooth, ones that use different kinds of redundancy in anticipation of failures in order to lower latencies while maintaining reliable delivery. While there have been a few middleware systems implemented over Bluetooth, none employ proactive mechanisms. Additionally, on these Bluetooth middleware frameworks, the usage of Bluetooth QoS mechanisms is set no later than connection setup time.

1.1 Thesis Contributions and Organization

This thesis investigates the use of Bluetooth as a transport for distributed applications, and how middleware can improve the QoS of Bluetooth in peer to peer wireless networking. The contributions of this thesis are:

- An analysis of the quality of service mechanisms and hooks available in Bluetooth, in terms of which can be overridden by middleware or not. The thesis discusses whether middleware or Bluetooth likely does better under all realistic conditions if granted control of the mechanism and the potential tradeoffs of migrating specific mechanisms from Bluetooth to middleware.
- An overview of the design and implementation of MicroQoSCORBA-Bluetooth (MQC-BT), a Bluetooth transport layer for MicroQoSCORBA

(MQC), including a custom, minimalistic Bluetooth stack designed to facilitate middleware QoS mechanism integration. MicroQoSCORBA-Bluetooth extends Bluetooth QoS mechanisms by providing both proactive redundancy and the hooks to change the parameterization of these Bluetooth QoS mechanisms at runtime. MicroQoSCORBA-Bluetooth is the only middleware framework with a Bluetooth transport we are aware of that offers a QoS API to the application.

- A preliminary experimental evaluation of MicroQoSCORBA-Bluetooth, demonstrating MW QoS mechanisms that are better than Bluetooth in some failure-free operational conditions. The thesis also presents baseline (flat configuration; no QoS options) results indicating that this Bluetooth stack is faster than BlueZ, the open source Bluetooth protocol stack included in the most recent Linux distributions.

The remainder of this thesis is organized as follows: Chapter 2 describes MicroQoSCORBA; Chapter 3 gives background information on Bluetooth; Chapter 4 gives overviews the possibilities of QoS control in Bluetooth enabled middleware; Chapter 5 overviews the design and implementation of the MicroQoSCORBA-Bluetooth transport and QoS subsystem; followed by an experimental evaluation of this implementation in Chapter 6; related work is discussed in Chapter 7; and conclusions and future work is presented in Chapter 8.

Chapter 2

MicroQoSCORBA

The MicroQoSCORBA framework is a rethinking from the bottom up what can and should be configurable in middleware for embedded systems [McK03a, McK04, MQC04]. MicroQoSCORBA has organized the space of the features and flexibilities that can possibly be stripped out to support a small footprint. It makes these choices available to the developer at a fine granularity, and then tailors the middleware to both the embedded device's resource constraints and the flexibilities and features required by the embedded application software to run on it. This is crucial, because the large number of embedded systems have a very wide range in terms of both device resource constraints and application requirements.

MicroQoSCORBA also supports multiple quality of service (QoS) mechanisms. Its fault tolerance subsystem provides mechanisms for redundancy (temporal, spatial, value); reliability (best effort, reliable, atomic, uniform atomic); and ordering (FIFO, causal, and total) [DOR03]. Its security subsystem provides a variety of mechanisms for encryption, message digests, message authentication codes, error correction codes, and shared secretes [McK03b].

Ongoing work has added a configurable intrusion detection subsystem [NAE04]. Multiple mechanisms from one or more subsystem may be configured.

Our initial version of MicroQoSCORBA was in Java. We have a beta version of a C++ version. As of April 2004, it was the fastest CORBA implementation tested by one independent benchmarking expert for some key benchmarks, especially for message sizes below 2K bytes [GAU04]. Roundtrip times start at less than 60 microseconds. MicroQoSCORBA was not specifically designed to be fast, just highly configurable; its speed is a pleasant side effect of removing everything that is not absolutely necessary for a given application.

For the remainder of this chapter, we first present the base architecture of MicroQoSCORBA, then the choices it exhibits at each stage during the lifecycle of an application, and finally we show how quality of service constraints can be set static (design time) or dynamic (run time).

2.1 MicroQoSCORBA Lifecycle Epochs

During the design of any distributed application, the designer must provide information on how the application is to be configured, essentially a set of tradeoffs. MicroQoSCORBA supports this through an underlying architecture and toolkit that span the complete development cycle from first concept in the design stages to application runtime. A MicroQoSCORBA project's lifetime is divided into five epochs: Design, IDL Compilation, Application Compilation,

System/Application Startup, and Run Time. During each of these epochs, various constraints are bound. During the application's lifecycle, as each constraint is bound, opportunities exist for reducing and/or refining many key facets of the application. It is beyond the scope of this thesis to provide a complete list of constraints that can be bound, but a few key constraints are shown in Table 3.1.

Lifecycle Epoch	Constraint Bound	Examples
Design	HW Heterogeneity	Symmetric, Asymmetric
	HW Choice	X86, Tini, ColdFire
	Communications HW	Ethernet, Serial, Infrared, Bluetooth, IEEE802.11b
	Processing Capability	50MHz, 1GHz, 8bit, 32bit
	System Size	Small, Medium, Large
	Power Usage	Line, Battery, Parasitic Power
IDL Compilation	Communication Style	Passive, Proactive, Push, Pull
	Stub/Proxy Generation	Inline vs Library Usage
	Message Lengths	Fixed or Variable
	Parameter Marshalling	Fixed Formats
Application Compilation	Space/Time Optimizations	Loop Unrolling, Code Migration, Function and Proxy Inlining
	Library Usage	Static vs Dynamic
System/Application Startup	Device Initialization	
	Network Startup	Bootp, DHCP
	Major QoS Adaptation	Select Between QoS Modules
Run Time	Minor QoS Adaptation	Adjust QoS Parameters

Table 2.1: Partial listing of Constraint Bounds

Table 2.1 shows how MicroQoSCORBA attempts to constrain choices as early as possible in the lifecycle of an application. This can be done because of

the nature of embedded applications where it is possible to determine a large portion of the constraints early in the design phase. It is also not recommended for embedded applications to leave constraint decisions open for the startup and run-time stages, because it may be result in costly, additional resource consumption such as footprint and context switches. MicroQoSCORBA's approach is in contrast to many other reflective middleware frameworks such as QuO, see related work.

2.1.1 Design

The choices made in the design stage affect all subsequent stages. This is the stage where key decisions are made in terms of homogeneity vs heterogeneity, processor type and capability, symmetry in terms of processing power and power consumption, and means of communication (wired, wireless, Ethernet, Bluetooth, etc.).

2.1.2 IDL Compilation

During IDL compilation, MicroQoSCORBA exploits the constraints made during the design stage. An example is if an 8bit processor is used, all larger data types could potentially be dropped, at least in a homogenous system. The communication style and role of the devices will be set during this stage.

The IDL compiler will generate or leave out code based on the design constraints. Is there enough memory available to use inline proxy/skeleton

marshalling in the client and server implementation? Can messages be constrained to a fixed size? The result is smaller, but less flexible code.

2.1.3 Application Compilation

During the application compilation phase, MicroQoSCORBA plays a rather subdued role. Existing tools and compilers are used for optimal compilation, and a specialized compiler is beyond the scope of MicroQoSCORBA.

However, directing the performance of these compilers and tools is quite beneficial. Thus, if the developer knows that memory will be at a premium, the MicroQoSCORBA configuration tools can direct the compiler to optimize the compiled code so that space is conserved. Another constraint that is bound during this epoch is the choice of static versus dynamic linking of library code, highly dependant on the type of system to which the application is to be deployed.

2.1.4 System/Application Startup

When power is first applied to an embedded device, both the system and application will start running. The binding of a few run-time MicroQoSCORBA constraints may be delayed until this time. The embedded device may have some hardware configuration options that are set with buttons, switches, etc. and these settings could control the startup state of the embedded hardware. At startup, the device's networking parameters might be automatically configured (e.g., DHCP). Another key hardware factor is that ROM is often more plentiful than RAM.

Thus, multiple implementations could be written and burned into the device's ROM, an option very common in embedded systems. At startup the appropriate implementation could be loaded into RAM. This coarse grained adaptation allows a device to adapt to its environment.

2.1.5 Run Time

Until the main contribution of this thesis, an adaptive quality of service add-on to MicroQoSCORBA, runtime flexibility was limited. More and more embedded systems have sufficient computing resources that can support flexibility at run time, as was not the case when MicroQoSCORBA was initially designed.

2.2 MicroQoSCORBA Architecture

One of the key benefits of MicroQoSCORBA is its ability to target a range of embedded devices. This is accomplished by exploiting the various constraints that can be bound in each lifecycle epoch, as well as by using some novel adaptations in the standard CORBA architecture. The architecture of MicroQoSCORBA is shown in 3.2.

Note in Figure 3.2 that the IDL compiler has an increased role, and that the interaction between the ORBs and the underlying communications technology has changed. The remainder of this section discusses each of the key components of MicroQoSCORBA.

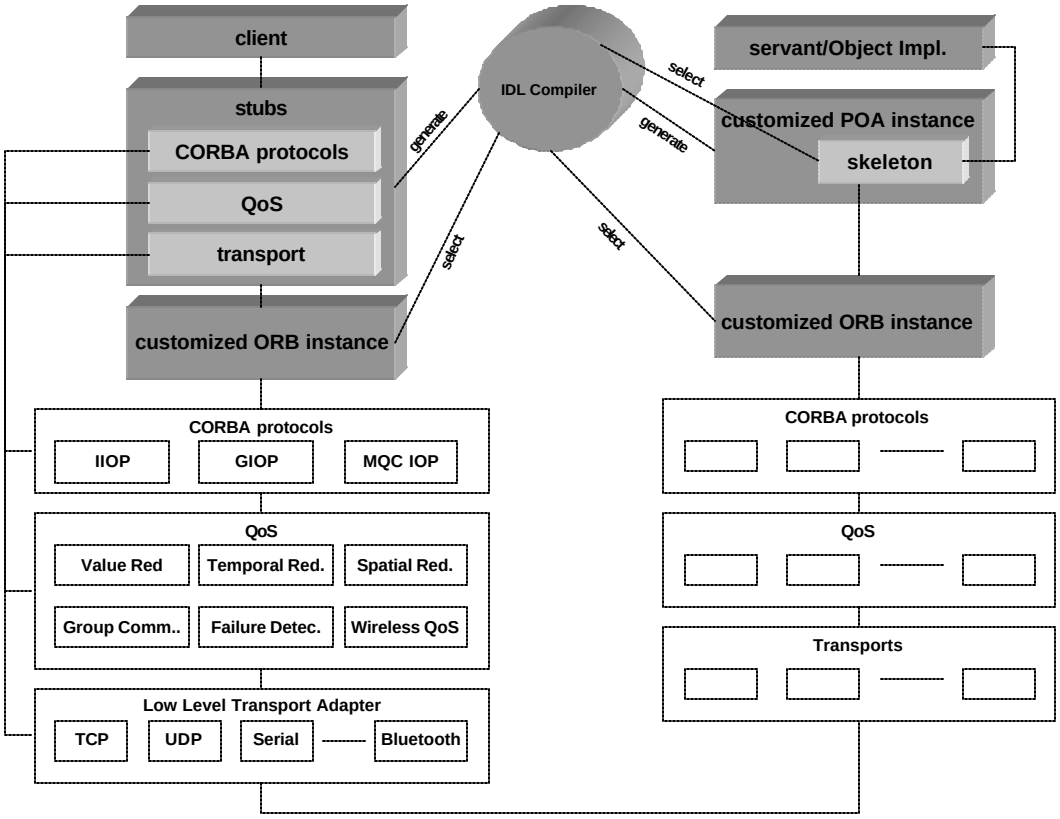

Figure 2.1: MicroQoSCORBA Architecture

2.2.1 IDL Compiler

Every CORBA development environment has an IDL compiler. This compiler is responsible for parsing an application's IDL files and producing the appropriate stub and skeleton code. Often, these IDL compilers assume that one canonical ORB implementation exists. For the standard desktop/workstation environment this is a reasonable assumption, since sufficient resources usually exist at the desktop to bundle in “everything” that is needed into one ORB implementation. But, this assumption is inappropriate for MicroQoSCORBA because “one size fits

all" simply does not scale down to small embedded devices. So in the MicroQoSCORBA development environment, the IDL compiler generates stubs and skeleton code that has been optimized for a customized ORB.

2.2.2 Customized ORBs and POAs

Only so much can be done in the stub and skeleton code to reduce (or improve) resource usage for a given application. Thus, MicroQoSCORBA supports the ability to use customized ORB instances, several of which could coexist in the MicroQoSCORBA development environment. Depending upon the choices previously discussed, multiple ORBs and POAs might exist. Some of these could be automatically generated via the CASE tools or they could be custom, "hand coded" ORBs that are finely tuned to a given application.

The IDL compiler is made aware of the existence of the customized ORBs via various configuration settings. During the IDL compilation, the compiler inserts appropriate statements into the generated stub and skeleton code so that the desired ORB/POA implementation is used.

2.2.3 Communications Layer

Increased functionality generally comes with an associated increase in cost. Thus many small, embedded devices have very limited communications abilities. For some applications, the support for IIOP interconnectivity may actually entail more code than is required for the application logic. In these cases, support for a lighter-

weight communication layer is needed. On the client side, the IDL generated stubs have a reference to the protocol and transport layer to be used. These references are given to the ORB so messages may be sent or received as needed. We note that the ORB could have used an abstract factory pattern [GHJV95], but that would have required linking in functionality for all of the MicroQoSCORBA's communication layers into a given application, something that was neither needed nor desired.

2.2.4 Quality of Service

With the recent port of MicroQoSCORBA for C++, we redesigned the architecture slightly to include a quality of service layer. With the increasing amount of QoS subsystems developed for MicroQoSCORBA, we identified the need for a common platform anchoring these. The QoS layer uses a messaging protocol for prepending GIOP messages with QoS sensitive information. We designed this layer so that MicroQoSCORBA can easily be extended with new QoS subsystems, and for all of them to work together in harmony.

We define two types of QoS in MicroQoSCORBA: Static and Dynamic. Static QoS occurs when the designer decides on the constraints for a given QoS subsystem at design time, while dynamic QoS occurs when the designer allows the middleware to find the best fit properties at runtime, usually within a set of design-time constraints. MicroQoSCORBA's profiling and CASE tools will aid the designer in picking and choosing a set of compatible quality of service

subsystem implementations and their settings, ruling out ones that do not make any sense to combine (something the developer cannot, in general, know).

GIOP messages are prepended with QoS information, see figure 2.2, according to the protocol shown in figure 2.3.

Figure 2.2: MQC Messaging Protocol

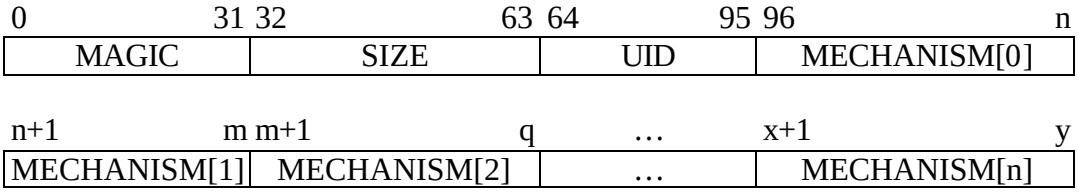

Figure 2.3: MQC QoS Fragment Messaging Protocol

The *magic* field is used for the QoS layer to identify a prepended QoS message from a GIOP message (this field always contains the byte sequence 'MQOS'), the size field indicates the length of the *mechanisms* payload field, and the *uid* field helps the QoS layer identify which client-server session. The *mechanisms* payload field is variable length, and includes 1..n QoS-mechanism messages. Each of these sub-messages includes information about a specific mechanism. An example is if proactive temporal redundancy is used. Then the field would contain information about the message sequence number for ordering purposes so that the receiving end can discard any redundant copies.

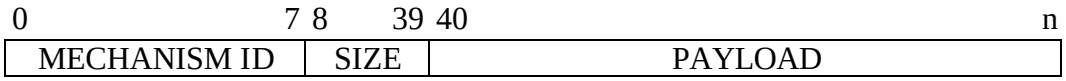

Figure 2.4: MQC QoS Mechanism Payload Protocol

Figure 2.4 shows the QoS mechanism payload field protocol. The first octet contains a unique identifier for the mechanism. Then there are 4 octets containing the size of the payload field and finally the variable length payload field. The payload field includes the information to be exchanged about the specific QoS mechanism. The different mechanisms and their payload contents will be discussed in more detail later in this thesis.

Chapter 3

Bluetooth Background

Bluetooth [BSI01, BSI03] is a short-range, low-cost, low-power wireless standard operating in the 2.4 GHz band, developed by the Bluetooth Special Interest Group, a trade association comprised of leaders in the telecommunications, computing, automotive, industrial automation and network industries [BT04].

One of the most important functionalities of a two-way wireless communication system is to determine how and when radio units at each node in the network can communicate. A solution common to many wireless technologies is to equip all nodes with a both a transmitter and a receiver, operating simultaneously on different frequencies [MOR02]. This is called frequency division duplexing, and is a form of full-duplex transmission, common to most cell-phone systems.

Bluetooth uses Time Division Duplexing (TDD), which utilizes the same frequency synthesizer, giving half duplex transmission. With TDD, one node transmits while the other receives, and the other way around [MOR02]. When two nodes communicate using Bluetooth, time is divided into *time slots* of 625 microseconds each. Time slots are either downlink slots or uplink slots. The

master transmits to slaves in downlink slots, and the slave transmits to the master in uplink slots. In a connection relationship, one node is the master, and the other is slave, the master controls the traffic in the network. Downlink slots are typically even numbered, while uplink slots are odd. *Polling* is when a node spends one or more time-slots listening for incoming data. The scheme gets slightly more complicated when the master has connections to several slaves. How this is achieved is, however, not important for understanding of the remainder of this thesis.

There are several types of Bluetooth baseband packets, with varying time slot occupancies. Bluetooth offers two types of physical links, namely asynchronous connectionless (ACL) and synchronous connection-oriented (SCO) links. ACL links are used for data transmissions, while SCO links are used for real-time two-way voice.

Transmission of Bluetooth baseband packets is allotted 366 microseconds per time slot, while the remaining 259 microseconds is used for frequency hopping. There are seven types of ACL baseband packets: DM*n*, DH*n*, and AUX1. M denotes medium speed, H high speed, and *n* the number of time slots occupied by the packet. Medium type packets use forward error correction (FEC), while both medium and high speed packets have error detection through CRC. Uncorrectable errors in medium type packets and detected errors in high speed packets result in the receiving device issuing a retransmit request. This stop-and-wait scheme is

called automatic retransmit request (ARQ) [BSI01, BSI03, VAL02a], and packets are retransmitted until completion.

Packets using more time-slots have less overhead per percent of payload, since the packet headers, error correction and detection codes occupy less of the total packet. However, if the transmission fails, and the packet has to be retransmitted, the latency increases with the number of time-slots used. AUX1 packets have no error detection or correction overhead, and thus no ARQ scheme. These are intended for calculating bit error rates (BER) by frequently transmitting a known bit pattern [BSI01, BSI03, MOR02].

Since Bluetooth has a raw data rate of 1 Mb/s, there is room for 366 bits of data in each time-slot. An access code and a header occupy 126 bits of each packet-fragment, and a further payload header occupies 8 bits for one-slot packets, and 16 bits for three and five-slot packets. The error detection and (for M-type packets) error correction overhead further decreases the space for application data in a packet.

A symmetric channel in Bluetooth means that the master and slave both use the same packet type for exchanging data, while the opposite is called an asymmetric channel. Assuming a BER of 0 and a symmetric channel, throughput varies between 108.8 kb/s for DM1 packets to 433.9 kb/s for DH5 packets. For a forward asymmetric channel, DH5 packets should have a throughput of up to 723.2 kb/s if we assume no bit errors.

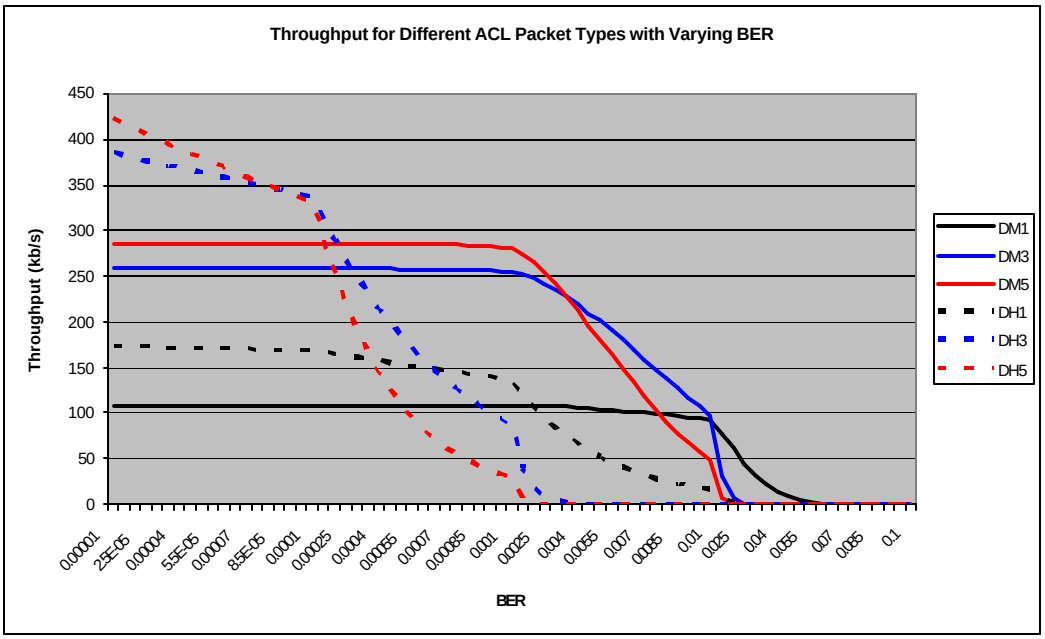

Figure 3.1: Bluetooth ACL Packet Throughput, Symmetric Channel

Even though DH5 packets have the highest throughput rate, this is not the case when one has to account for BER, and bit errors will happen when transmitting over radio. Figure 3.1 shows the result of computing throughput for the different packet types on a symmetric channel with different BER. From this we can derive the following coarse grained packet selection scheme:

- Use DH5 packets for BER less than 10-4.
- Use DM5 packets for BER between 10-4 and 10-2.
- Use DM1 packets when BER exceeds 10-2.

There is of course room for a finer grained granularity to this selection scheme, and additionally, all packet types are not necessarily supported by both devices.

Finally, BER can be computed in two ways; either by using the before mentioned AUX1 packets, or derived from the link quality (LQ) and received signal strength indicator (RSSI) of Bluetooth.

Chapter 4

Quality of Service: Bluetooth vs. Middleware

Bluetooth devices were initially not meant to be used in any kind of sizeable distributed application. With the introduction of class A Bluetooth devices, the technology is becoming more interesting for such purposes. In this case, we must assume that the range of hosts includes anything from very small embedded systems to powerful desktop computers; Bluetooth is rapidly becoming a cheap alternative to other wireless standards. In many cases, it is meaningful to move QoS control from the Bluetooth device that has limited computational power, into middleware. This chapter presents the tradeoffs, limitations and improvements of such a migration.

4.1 Kinds of High-Level Bluetooth Traffic

The Bluetooth specification provides the mechanisms for transmitting data or audio between two or more devices. There are three kinds of high-level Bluetooth traffic:

- Realtime audio/voice streaming
- Streaming data

- Intermittent data

4.1.1 RT Audio/Voice

Periodic analog data with hard real-time requirements transmitted over an SCO link. Impractical for use by MW, because of air coding/compression and the fact that once a connection is established, a consistent 64 kb/s data steam is opened between the two devices. Sending application data (not voice) over such a link would mean that the receiver would have to continually parse every part of the stream to receive even a few messages. Even more troublesome, the message could be corrupted by the compression, giving no guarantees of being able to recover every bit. Note that newer Bluetooth specifications allow transparent data, (meaning no air coding/compression).

4.1.2 Streaming Data

Periodic digital data interactions transmitted over an ACL link. Typical examples include, but are not limited to, streaming data files (MP3, MPEG4, etc.) and sensors pushing data at a fixed rate per second.

4.1.3 Intermittent Data

Aperiodic digital data interactions transmitted over an ACL link. Typical usage includes, but is not limited to, client-server request/reply messages and pushing alerts and sensor data updates based on a threshold trigger.

The specification has clearly not been designed with an aim to allow full control to middleware; most of the choices are done in the hardware implementation. Table 1 is an overview of QoS mechanisms, their current place according to specification, whether or not middleware can override any hardware implementations, and what QoS property that each mechanism fulfills.

4.2 QoS Mechanisms and Their Place

We enumerate QoS mechanisms in Table 1, in terms of what is offered by Bluetooth, what can be added with middleware. We discuss the different mechanisms, where their best place is, and whether or not it is possible to move certain mechanisms from Bluetooth to middleware given the current Bluetooth specification. In this section, when we use the term middleware, we refer to the host software protocol stack.

4.2.1 Error Control

Error detection (CRC) is used by Bluetooth hardware on all DM*n* and DH*n* packets, while forward error correction (FEC) is applied only to DM*n* packets. The Bluetooth specification does not allow this to be disabled or migrated to middleware; the only solution is to use AUX1 packets, which is explicitly disallowed by the Bluetooth specification. AUX1 packets are likely to outperform DM*n* and DH*n* packets within a certain BER range [VAL02a], if the middleware is located on a host with decent computational strength and a fast error control

algorithm is employed. The forward error correction scheme used for DM*n* packets is (15, 10) shortened Hamming code [MOR02].

Mechanism	Supported by Bluetooth HW Spec?	D	Does the mechanism have a better place in Middleware?	A	L
Error control FEC/ Detection	YES: FEC w/ DM*n*, detection with DHn	NO	Likely better if the use of AUX1 packets was allowed	Y	N
Automatic Repeat Requests	YES: automatic if DH*n* detects error, or DM*n*'s FEC cannot correct all errors	NO	Potentially, if both 1) and 2) can be disabled in BT HW	Y	N
Runtime Baseband Packet Type Selection	YES: SW provides set of allowed packet types, HW reserves the right to use DM1 at any time	NO	Likely, if complete control of packet type is available to MW, i.e. DM1 can be disallowed, and AUX is allowed	Y	Y
Isochronous Data	YES: timeout in message delivery. Software notified by event/interrupt if message cannot be completely transmitted in the given time	Not forced	Likely, can adaptively and proactively adjust timeouts based on runtime instrumentation	N[1]	Y
BT QoS Setup	Allows allocation of resources & config. parameter settings to "reserve" BW	Not forced	No, but can provide better params via IDL analysis and/or instrumentation	N[2]	Y
Proactive Temporal Redundancy	Not supported for unicast, only for broadcast. MW likely to be more effective	N/A	YES, BT HW & SW does not have enough app-level knowledge	Y	Y
Proactive Spatial Redundancy	Hardwire frequency hopping provides some crude spatial redundancy within BT frequency band.	NO	YES: replicated devices with active replication	Y	Y

D = Possible to disable hardware implementation, A = Availability, L = latency
[1] Determines availability in terms of data streams, but not guarantees.
[2] YES, if availability concerns means latency requirements

Table 4.1: Bluetooth QoS Mechanisms and Potential Middleware Improvements

4.2.2 Automatic Repeat Requests

Reactive temporal redundancy is used on all DH*n* and DM*n* packets, and cannot be disabled. As with error control, if AUX1 packets were allowed middleware would have the choice between proactive, reactive and no redundancy. Note that this ARQ scheme is not used on broadcast messages, in which case the middleware has full control of redundancy scheme. For DM*n* packets, a retransmit is requested only if the packet still has an error after the forward error correction scheme has been applied at the receiving device. DH*n* packets are requested retransmitted if an error is detected through the redundancy checksum. It is easy to derive from this that using DH*n* packets in general is a gamble for most BER intervals.

4.2.3 Runtime Baseband Packet Type Selection

Most Bluetooth hardware chooses baseband packet types in real-time using Channel Quality Driven Data Rate (CQDDR). CQDDR selects packet types based on the current channel BER, which is determined through communication between the local and remote link managers (LMs). During connection setup, the application specifies a set of allowed packet types (i.e. DM1, DM3 and DH5). However, DM1 packets are by specification always available for use by the hardware implementation. CQDDR makes the correct choice in most situations,

although for very small messages (10 bytes or less), this choice seems inexplicable; because initial experiments suggest that it can result in a higher RTT than with 100 byte messages. Note that for broadcast messages, the packet type used is undefined in the Bluetooth specification, although most vendors simply use DM1 packets. This is of course due to the fact that broadcast messages are intended for a number of recipients, and selecting packet types based on a number of different connection BERs is hard; DM1 is the safe choice.

4.2.4 Isochronous data

Middleware can set flush timeouts that the Bluetooth hardware can use to create isochronous data transmissions. Isochronous data transmissions are used when the application has a certain deadline for when the data has to be successfully transmitted. This deadline is called a flush timeout. There are no timeliness guarantees, other than that if the entire higher layer message is not successfully transmitted within the specified time window, the remainder of the message is discarded, and the application is notified. Middleware can possibly improve the use of isochronous data, because it has the potential to exploit various properties of the current state of the wireless network and the application. Note that this mechanism is intended to be controlled by software; the Bluetooth hardware simply acts on the timeout value if isochronous data is desired by the software protocol stack.

4.2.5 Bluetooth QoS Setup

Bluetooth has its own QoS mechanisms targeting link utilization (resource management), latency and timeliness requirements. The connection master decides this, usually after a negotiation between the higher stack layers on both sides. From a middleware standpoint, this gives a server the ability to assign priorities to different connections (fair or priority based). An example is when a Bluetooth device that has more than one active link, it has to decide how often to poll each one. By default, each link is polled 50% of the time.

Resource management is only useful when a server can handle multiple clients over Bluetooth. If there is only a single client, the connection should consume all the resources available for the link. If there is more than one client, the server must be able to direct the Bluetooth device's polling interval. There are several opportunities that middleware can take advantage of, including, but not limited to, load balancing for servers and connection based priority scheduling. For some distributed applications, with multiple clients and one server, different clients could have different soft real-time requirements, and the server can then schedule polling of the links accordingly. As we know, a Bluetooth device can have at most 3 active ACL connections, thus load balancing is probably not required, unless the application is hosted on devices with very strict and small processing and memory capabilities.

4.2.6 Proactive Temporal Redundancy

Temporal redundancy is to do the same thing more than once, in the same or in different ways, until the desired effect is achieved [DORTHESIS]. An example of temporal redundancy is the retransmission of a message in order to tolerate omissions due to electromagnetic noise or temporary receiver overflow [VER01].

In Bluetooth, temporal redundancy is reactive, that is, if a message is found to have errors, a new copy is requested. Bluetooth's reactive temporal redundancy is achieved through a stop-and-wait automatic repeat request (ARQ) scheme. Since this scheme is implemented in hardware, moving it middleware would probably not bear many fruits. Therefore, it is more interesting to explore the option of being proactive.

There are, however, two ways of implementing proactive temporal redundancy in middleware, both relying on ARQ to be (effectively) disabled:

- AUX1 baseband packets
- Peer broadcasting

AUX1 packets have as earlier stated, no retransmit scheme, and this allows middleware to be proactive, and also reactive, in its choice of redundancy. The use of AUX1 packets is mathematically proven to be more efficient (higher throughput, same reliability) than DMn and DHn packets for certain BER [VAL02a], but is hard to implement due to the fact that the Bluetooth

specification [BSI01, BSI03] does not allow ACL connections to use this type of packet. Some devices allow use of AUX1 through vendor specific HCI commands [ERI01], but these are mostly restricted (and intended) to calculating BER, not allowing user defined application data to be transmitted.

Use of ARQ packets also requires the software side to implement error detection and correction, but this could be viewed as a possibility more than a restriction for a Quality of Service mechanism.

Broadcasting is the second way of implementing proactive temporal redundancy. Bluetooth's active broadcast messages have as we have earlier seen no ARQ protocol, and because of this, no guaranteed reliability. In a simple CORBA application, it can be assumed that the piconet only contains nodes that are part of a client-server relationship. When a Bluetooth device issues a broadcast, it is only received by the nodes with which the device has an active connection. Additionally the node issuing the broadcast is required to be the master for the connection. This way of communicating is almost the same as using AUX1 packets, leaving everything but error control to middleware.

Both the abovementioned schemes will effectively disable Bluetooth ARQs, allowing middleware to be proactive, semi-proactive[1], and also reactive. It is obvious that due to failure probabilities and the overhead associated with issuing a

[1] Semi-proactive is a hybrid between proactive and reactive where the message is first transmitted a number of times without ARQ, then once with ARQ.

retransmit request, proactive schemes are more successful as the size of the message increases.

The drawback of using broadcasts as a means of unicasting in Bluetooth is the huge overhead associated with switching roles, recalling that only the link master can broadcast. Therefore, full duplex broadcasting is expected to have a significant latency increase compared to AUX1 and normal unicasting.

4.2.7 Proactive Spatial Redundancy

Spatial redundancy consists of having multiple copies of the same component [VER01]. In our case, there are two types of redundant components in the system:

- Replicated links: A client and a server sharing more than one wireless link to mask device and link failures. The receiving end picks the first available message, according to an ordering policy.
- Replicated servers: A client having links to several server replicas to mask link problems and server failures. The client sends requests to both servers and picks the first reply according to an ordering policy.

The overhead associated with spatial redundancy depends on the size of the request message, the middleware's ability to pick the best link for the first transmission attempt, and the computational overhead in middleware associated with maintaining two or more links, compared to one.

All the previous mechanisms can of course be combined with spatial redundancy, and other opportunities include proactive failure detection and handoff between servers.

4.2.8 Value Redundancy

By definition, value redundancy is adding extra information to a message [VER01], usually in order to increase some property of fault tolerance or security. Bluetooth exhibits a lot of value redundancy in terms of CRC and FEC. By the nature of MicroQoSCORBA's QoS messaging protocol, it is also clear that MicroQoSCORBA exhibits some crude value redundancy when any QoS mechanism that needs communication between the client and the server QoS layers is used. Note that MicroQoSCORBA also offers a set of fault tolerance mechanisms, including security that is part of the value redundancy domain.

4.3 Adaptive Quality of Service

The mechanisms discussed in section 4.2 are all usually chosen at design time, hence they are static mechanisms. In wireless networks, such as Bluetooth, the choice of mechanism that seems valid at design time might turn out to be a poor choice due to unforeseen changes in the wireless network at runtime. Therefore, it is important that a middleware offering wireless transports is able to adapt to such changing conditions.

For Bluetooth, the before mentioned mechanisms that are suited for runtime adaptation are all related to the various types of redundancy (temporal, spatial and value). Additionally, as earlier discussed, Bluetooth hardware adapts at runtime by selecting the best-fit baseband packet type through the use of CQDDR.

Chapter 5

Design and Implementation of MicroQoSCORBA-Bluetooth

Figure 5.1 shows the software part of the Bluetooth protocol stack as defined in the Bluetooth Core Specification. The Host Controller Interface (HCI) protocol layer is used to format messages (commands, events and data) that are interchanged between the host (software stack) and the host controller (hardware stack). The Logical Link Control and Adaptation Protocol (L2CAP) layer multiplexes messages that are pushed upwards in the stack to the appropriate destination layer. L2CAP also segments and reassembles messages, and together with the HCI layer it delivers messages in order. Each layer above L2CAP also has its own messaging protocol.

Most general purpose Bluetooth software stacks adhere to the guidelines of the Bluetooth specification, giving a lot of flexibility for supporting various Bluetooth profiles: Headset, Printer, Bluetooth Network Encapsulation Protocol (BNEP), and so forth [GRA03]. When using Bluetooth as a CORBA transport, the underlying software stack can be optimized to include only the protocols needed for the middleware to function properly. This means that there is no need

to support profiles, service discovery and so forth, only connection setup, link management and data transport is of concern. For additional QoS control in middleware it is almost a requirement that the Bluetooth software stack used by the CORBA transport offers more than a general purpose stack. Hooks must be available for hands-on control of the Bluetooth device.

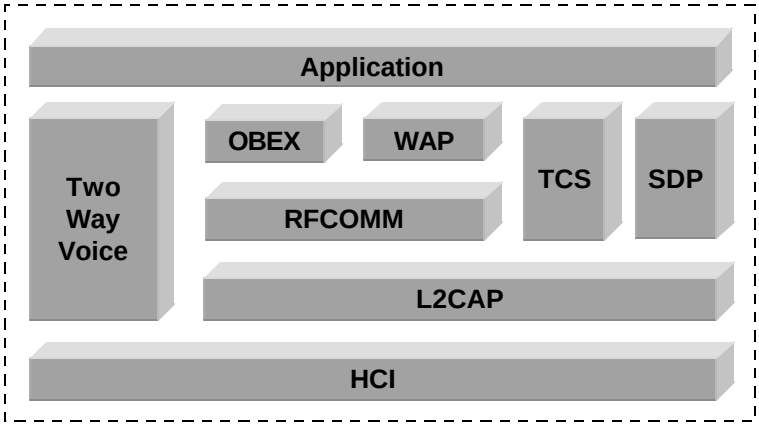

Figure 5.1: The Bluetooth Software Protocol Stack

We have developed a Bluetooth transport for MicroQoSCORBA. This includes a stripped-down object oriented Bluetooth software protocol stack[2], see figure 5.2, to communicate with Bluetooth devices. This stack includes only HCI and L2CAP protocol layers optimized for CORBA. There is no need for our L2CAP layer to multiplex data, as the only layers above it belong to the ORB. Its only function is to segment and reassemble messages, encapsulating them in the

[2] Not certified by the Bluetooth SIG and is used for MQC experiments only.

L2CAP messaging protocol. Any other tasks related to link QoS negotiation are performed by MicroQoSCORBA's QoS layer, encapsulated in the ORB. These two layers act as a socket between the Bluetooth hardware and the MicroQoSCORBA Bluetooth transport.

We tunnel GIOP messages over the MQC-L2CAP layer, compliant with the OMG specification for GIOP tunneling over Bluetooth [OMG03a]. Inter-ORB-operability is not currently supported due to the manner in which we accomplish middleware QoS control. Our own Bluetooth stack was needed in order to analyze the effects of moving QoS mechanisms into middleware. Most third party Bluetooth software stacks are developed according to the guidelines of the Bluetooth specification, not permitting the user full freedom of sending and receiving raw HCI commands and events. A nice side effect of such a small Bluetooth stack is that the memory footprint and the total stack time[3] of a message is reduced.

In order for Bluetooth QoS control and management to have a meaningful place in software, the host must have an advantage in computational power over the host controller. If not, the computational overhead introduced by moving some mechanisms from hardware to software is too high. Placing QoS control in middleware gives the application a lot more flexibility, and combined with

[3] We define total stack time as the time it takes from a message is submitted at one end of the stack (stub, proxy / OS device driver) until it reaches the other end (OS device driver / proxy, stub).

sufficient computational power, it is clearly an improvement over the existing Bluetooth QoS control and mechanisms. For each type of control action that is issued from the host to the host controller, there is significant overhead due to the interchange of commands and events.

MicroQoSCORBA is a configurable middleware, and its Bluetooth extension is also highly configurable. Table 5.1 shows the mechanisms and configuration options that are available. The various mechanisms that are needed in the application designer's configuration can be selected, and those that are not needed are not compiled into the binaries. We support a number of configuration options, including inquiry and page scan intervals, whether or not inquiries and page scans are enabled when a connection already exists, etc. We also support preset static and dynamic Bluetooth QoS setup parameters, proactive temporal redundancy with static or dynamically adaptive number of retransmits, and spatial redundancy.

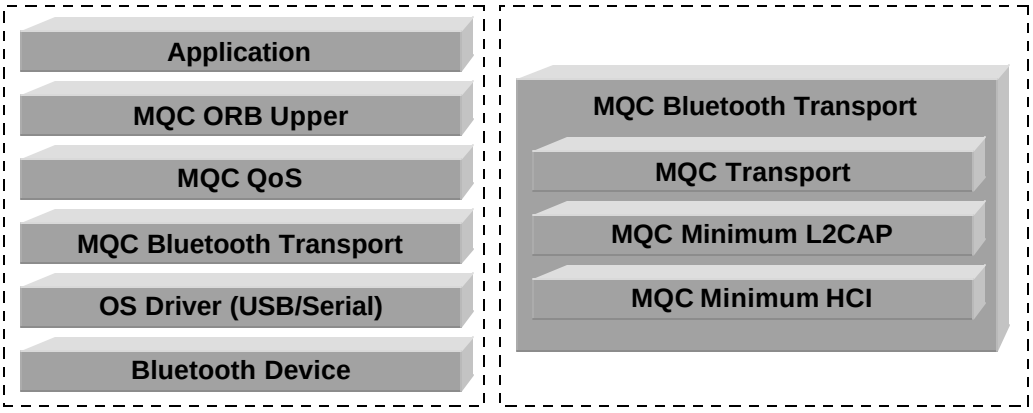

Figure 5.2: MQC Stack and MQC Bluetooth Transport

MicroQoSCORBA-Bluetooth masks heterogeneity in operating system, host hardware, and host controller hardware. All that is needed for it to work with any given hardware configuration is an implementation of the device driver for the HCI transport. We currently support serial and USB interfacing in Linux.

Mechanism	Design Time Choice		Can Be Excluded At Compile Time?	Run Time Behavior	
	Static	Dynamic		Static	Dynamic
QoS Setup	Preset Values	Value Range	YES	Preset Resource Allocation	Dynamic Resource Allocation
Isochronous Data	Preset Values	Value Range	YES	Preset Transmission Timeout	Dynamic Transmission Timeout
Proactive Temporal Redundancy	Preset Redundancy Level	Dynamic QoS Mechanism	YES	Preset Redundancy Level	Reflective Dynamic Adaptation based on Link Quality
Spatial Redundancy	Preset Number of replicated servers	N/A	YES	Client uses replicated servers	N/A
Packet Types	Preset Specified Set	N/A	NO	Preset packet types	N/A
Page and Inquiry Scan Intervals	Preset Values	N/A	NO	Preset Intervals	N/A
Page Scan with active connection	Preset Boolean Switch	N/A	YES	Enabled or Disabled	N/A
Inquiry Scan with active connection	Preset Boolean Switch	N/A	YES	Enabled or Disabled	N/A

Table 5.1: MicroQoSCORBA Bluetooth QoS configurations

5.1 Implementation Details

5.1.1 MicroQoSCORBA-Bluetooth Protocol Stack

The MicroQoSCORBA Bluetooth protocol stack is object oriented and implemented in C++. We implement a subset of the two lowest protocol layers of the Bluetooth stack, namely the L2CAP and HCI layers. In the implementation these two layers essentially expand into three core objects: lower HCI layer, upper HCI layer and a socket that is used by the transport wrapper.

The socket implementation complies with the GIOP Tunneling over Bluetooth Specification, and also offers a number of hooks for the controlling software to implement QoS mechanisms. The upper HCI layer is used for all commands and events that are interchanged between the stack and the device; complying with a subset of the HCI layer in the Bluetooth specification (we do not implement functionality which is not necessary). The lower HCI layer is dual threaded; the extra thread is used to receive events and data from the device. Received events and data are multiplexed here and delivered to their appropriate destination: events to the upper HCI layer, and data to the appropriate socket. There are two concerns here; first of all, access to the received data (events and data) must be efficiently synchronized between the two threads. Second, the data must be subject to ordering.

This is accomplished through a message pump, more specifically an asynchronous-synchronous message passing kernel with selective receives. This

solves the problem of fine grained synchronization. Ordering is important, and the following scenario is typical: The software issues a data send followed immediately by a command. Both actions will result in the device issuing one or more events for each action. Assuming the actions belong to different threads, there is a problem, since both threads will be expecting one or more events. Our message passing kernel will store these messages, and the expecting parties will be waiting for specific properties contained in the events. When such an event is inserted into the data structure, the waiting process/thread will be woken up and receive the correct event.

In our Linux implementation, we use the POSIX [LEW91] thread library to accomplish fine grained synchronization. Our code should be easily portable to Java with the recent introduction of the Java Synchronization Library [JCP04], while most embedded platforms have their own synchronization primitives, also making a port feasible.

For simplicity in porting, we have used a number of wrapper classes to implement a small MicroQoSCORBA synchronization library. Any port of the MicroQoSCORBA Bluetooth stack would only need to switch the synchronization anchor calls such as wait, signal, lock and unlock.

Note that it seems that most JSR-82 compliant implementations solve the problem of synchronization and ordering by reversing the flow control, in which case the host controller is to be considered inferior to the host in terms of

computational ability [MOR02]. This is of course not a desired solution, but is expected to change with the introduction of the before mentioned Java Synchronization Library. Our implementation neatly bypasses this problem by using the compact message passing kernel.

5.1.2 Implementation of QoS Mechanisms

The MicroQoSCORBA QoS layer is an ideal location for Bluetooth QoS mechanisms. This layer provides a platform for runtime QoS adaptations, a new addition to MicroQoSCORBA. The transport wrapper layer provides the QoS layer with the socket it uses for communications, in our case a MicroQoSCORBA-Bluetooth socket. The QoS layer implements a number of mechanisms targeting Bluetooth, and the socket is the interface providing the hooks that are the building blocks for these.

For our Bluetooth QoS implementation, the client and server side QoS layer implementations exchange information about link quality, received signal strength (both for BER calculation), temporal redundancy, spatial redundancy and packet types. Table 5.2 shows the various mechanisms utilized by the Bluetooth implementation.

Temporal Redundancy. Here the client and the server need to communicate sequence numbers in order to drop redundant messages.

Spatial Redundancy. This message is only sent from the server to the client, and includes the address of the server that sent it. Recall that spatial redundancy can either be over replicated links between one client and one server, or between one client and multiple server replicas. The server address identifies both the server and the link used.

Bluetooth Packet Type. This message is used for the client and server(s) to agree on a set of packet types. This is useful both to create symmetric and asymmetric links as well constraining unreliable packet types, such as DH*n* packets.

Bluetooth Communication Type. This message is used in connection setup stage for the client and server to agree on a style of communication. The octet is split into two 4 bit parts, the low order bits contain the client transmission style, and the high order bits contain the server messaging style. Styles of communication include, but are not limited to:

- Client unicast, server unicast
- Client broadcast, server unicast
- Client broadcast, server broadcast
- Hybrid broadcast, unicast schemes

In the case where both the client and the server broadcast messages, they need to be aware that they need to switch master/slave relationship between each transmission, and the exchange of this message ensures that.

Bluetooth BER. This message is only used in tandem with proactive temporal redundancy when application is configured to adapt to changes to link quality at runtime. The purpose is to exchange BER information, so that each node can be proactive in terms of heuristically calculating the number of redundant retransmits that are needed in order to achieve message completion.

Mechanism	**Mechanism ID**	**Parameters**	
		Description	**Size**
Temporal Redundancy	0x01	Sequence Number	4 octets
Spatial Redundancy	0x02	Server Address	10 octets
Bluetooth Packet Type	0x03	Packet Types	2 octets
Bluetooth Communication Type	0x04	Comm Type	1 octet
		Role Switch Required	1 octet
Bluetooth BER	0x05	Link Quality	1 octet
		RSSI	1 octet

Table 5.2: QoS Messages

5.1.3 Interoperabilty and Service Discovery

Bluetooth has its own device and service inquiry protocol. This is used by most Bluetooth devices to discover other devices in range, and which services are offered by each discovered device. An example is a roaming laptop user that needs to print a document. Using his Bluetooth device, the user discovers a number of Bluetooth devices, and some offering a printing service.

This style of service discovery is similar to the naming service approach of CORBA, but differs in the fact that Bluetooth directly addresses all devices in range, while a CORBA application will only be looking for a naming service, a centralized repository that keeps track of the available services. Both approaches use a broadcast-like discovery.

The OMG has developed a specification for wireless CORBA IORs: the Mobile Interoperable Object Reference standard [OMG03b]. This type of IOR is generalized to support all types of mobile communication technology, and MicroQoSCORBA does not need to fully comply in order to achieve object referencing. Currently, we embed only the Bluetooth hardware address as the location of a service in the object reference, much like the way a server's ip-address is used for any IP-based transport.

Chapter 6

Experimental Evaluation

6.1 Experimental Setup

Experiments were conducted on two machines with the following specifications: 1 Intel Pentium 4 2.4 GHz processor with hyper threading, 1 GB DDR RAM, 800 MHz FSB, Slackware 9.0 operating system with Linux kernel version 2.4.25. The Bluetooth devices used were Belkin F8T003 USB dongles with Cambridge Silicon Radio (CSR) chipsets.

6.2 Experiments and results

The basis for the three first experiments is to measure the roundtrip time (RTT) for a remote method invocation. The last experiment measures the elapsed time for a server push. For the purpose of evaluation we use a simple application based on the IDL in Figure 6.1. After each invocation, the elapsed time is computed and stored in a histogram data structure. By using a histogram we are able to filter events that are not part of our application's execution, for example execution time devoted to other tasks.

We measure RTTs for 1000 iterations of foo.bar(...) calls, using timing.idl. We define RTT to be the time that passes from when a client calls a method located on the server until the call completes. For the three first experiments we use a symmetric channel, while the last experiment makes use of an asymmetric channel, since this makes most sense in a push scenario.

```
module timing {
   interface foo {
      long bar (in long arg1);
   };
};
```

Figure 6.1: timing.idl

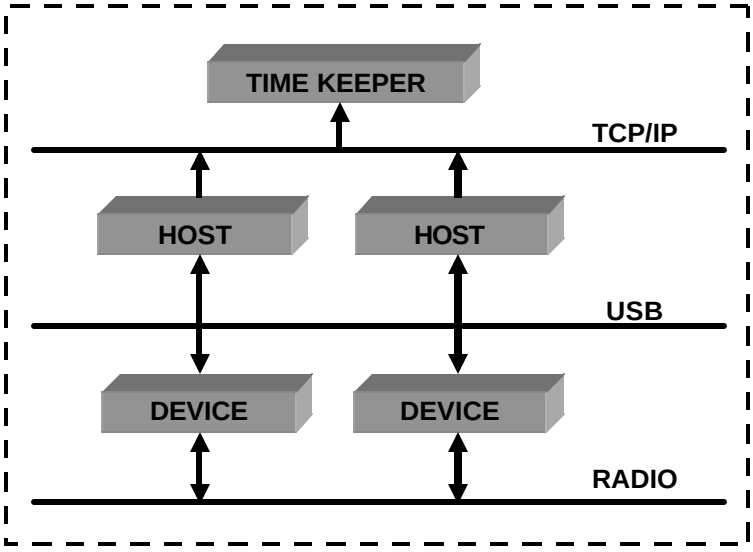

Figure 6.2: Physical Setup for Experiments

Experiments 1 and 2 were conducted to measure the performance of our Bluetooth stack (MicroQoSCORBA-Bluetooth), while experiments 3-10 were

conducted to measure the performance when the QoS control is assigned to the host controller. Experiments 11-16 were conducted to measure the performance with QoS control in middleware, to prove that Bluetooth are missing some important mechanisms Experiments 3-12 measure the RTT in ms in a client-server scenario, while we for experiments 13-16 measure the elapsed time from the client calls foo->bar(…) until the server receives this call in its skeleton. This is a typical server push scenario. Figure 6.2 shows the physical setup for the experiments.

1. Baseline MQC-BT versus baseline MQC with BlueZ transport wrapper
2. Middleware Time
3-9. Two way unicast, reactive redundancy, different packet types
10. Two way broadcast with a static number of retransmits, DM1 packets
11. Client Broadcast, Server Unicast, static proactive temporal redundancy, DM1 packets
12. Client hybrid broadcast unicast, server unicast, DM1 packets
13. Unicast
14. Broadcast with static temporal redundancy
15. Hybrid broadcast unicast with no redundancy (other than the extra unicast)
16. Proactive broadcast

#	Transport	RTT	MW Time
1	**MQC w/ MQC-BT Transport Wrapper**	10.42	0.011
2	**MQC w/ BLUEZ Transport Wrapper**	11.14	N/A

Table 6.1: Measurements for experiment 1 and 2 in ms; comparison between MQC-BT and BlueZ

#	Packet Type	Best	Worst	Avg	Freq	Rank
3	**DM1**	19.00	60.00	37.18	39.00	7
4	**DM3 | DM1**	9.00	43.00	10.40	10.00	1
5	**DM5 | DM1**	10.00	45.00	14.33	13.00	3
6	**DH1 | DM1**	15.00	45.00	32.21	34.00	6
7	**DH3 | DM1**	8.00	40.00	15.32	10.00	4
8	**DH5 | DM1**	10.00	42.00	17.13	12.00	5
9	**All packets**	9.00	45.00	10.42	10.00	2

Table 6.2: Measurements for experiments 3-9 in ms

#	Best	Worst	Avg	Freq	Dropped
10	398.00	961.00	434.07	437.000	10.0%
11	17.00	48.00	20.41	19.000	0.3%
12	18.00	47.00	19.24	19.000	0%
13	16.00	41.00	18.63	18.50	0%
14	8.00	28.00	8.52	8.00	4.8%
15	8.00	28.00	9.37	8.00	0%
16	8.00	39.00	12.22	9.00	0%

Table 6.3: Measurements for experiments 10-16 in ms

6.2.1 Baseline and Middleware Time (1-2)

We compare MicroQoSCORBA with our own Bluetooth transport against MicroQoSCORBA with a BlueZ transport wrapper. The BlueZ transport wrapper uses an L2CAP socket. Table 3 shows that our implementation has slightly lower roundtrip latency as the BlueZ stack, on average 720 microseconds, an improvement of 6%. Note that the BlueZ stack was compiled without SCO code

for optimization. We observe that the middleware time (computational time) for our implementation is negligible; most of the round trip time is spent during transmission between the two devices. We define middleware time to be the execution time from foo.bar(...) is called until the message is delivered to the low-level transport driver (in our case the USB driver).

6.2.2 Two Way Unicast (3-9)

We measure roundtrip times for a two way unicast of a call to foo.bar(...), with varying packet-type combinations, giving the Bluetooth hardware CQDDR algorithm different sets of packet types to use.

Table 4 shows the roundtrip times in ms for various packet type combinations, using MicroQoSCORBA over Bluetooth. Since there is no way of forcing the Bluetooth device not to use DM1 packets, these are always available for CQDDR to use, making it impossible to achieve a perfect symmetric link for testing purposes. Not surprisingly, Use of DHn packets alone perform badly over time, since the LM will often use DM1 to be sure of packet completion. For our GIOP messages, 3-5 time slots seem to be the best fit, and therefore DM1 and DH1 packets have the highest latencies.

6.2.3 Two Way Broadcast (10)

The second experiment measures roundtrip times for foo.bar(...) where the client and the server broadcast the messages. This scheme requires the sending node to

change its role to master before it can send a broadcast. We log the number of dropped messages. See Table 5 for results. Since we know that broadcast messages are sent using DM1 packets on devices using CSR chipsets, we can compare the results to the unicast measurements. We observe that the latency overhead resulting from the bottleneck of constantly switching roles is huge; this scheme is not suitable for client-server communication.

6.2.4 Client Broadcast, Server Unicast, static proactive temporal redundancy (11)

The third experiment is a combination of the first two, where the client sends requests using broadcast and the server replies using unicast. The removes the bottleneck of having the client and the server negotiate link roles each time a message is transmitted. Table 5 shows promising results, as the roundtrip times are ~ 20 ms less than with normal unicast of DM1 packets.

6.2.5 Client Hybrid Broadcast Unicast, Server Unicast (12)

A hybrid broadcast unicast scheme first broadcasts the message (with no broadcast retransmits), then immediately unicasts a redundant copy. For comparability, this scheme uses DM1 packets for the unicast. In most cases, bar high BER, the broadcast message will complete, and we know this is faster than a unicast based on the previous experiments. If it doesn't, it is backed up by the reliable unicast message.

We chose only to have the client use this scheme, while the server replies using unicast, to avoid the previously discussed overhead introduced by role-switching. As we see from Table 5, the RTT is further decreased with ~ 1 ms, and this scheme does not drop packets. There is, as expected, no additional overhead associated with sending the extra unicast message, because the transmitting node would otherwise be waiting for a reply message, thus this is performed when the previous experiment would be in an idle state. Likewise, the reception of a redundant message does not introduce any latency overhead.

6.2.6 Unicast (13)

We observe that the measured time is approximately half the roundtrip latency for DM1 packets, as expected. This experiment was run with on a symmetric channel to be able to compare apples to apples, and we would expect slightly lower latencies if an asymmetric channel was used.

6.2.7 Broadcast, Static Proactive Temporal Redundancy (14)

Here we observe the expected performance burst that we get by using proactive redundancy. We experience on average a 54% performance improvement over unicast, but this scheme drops almost 5% of the messages. Since we are only pushing data, the server does not have to switch role before it can send anything, and this scheme reaps the benefits. We note that 4.8% of the messages were lost,

but that must be expected in any scenario when a radio link is used with no temporal, spatial or value redundancy.

6.2.8 Hybrid Broadcast Unicast with No Redundancy (15)

This scheme uses the same means of transmission as the previously described hybrid scheme. We see that it is almost as efficient as using a proactive broadcast scheme and we do not have to worry about selecting the degree of redundancy at design time, since the second redundant message is 100% reliable. Most of the time the first (broadcasted) message is going to complete, and in the few cases it doesn't, the second 100% reliable message will. In those cases, when the link is not good enough for the first message to complete, we will experience a decrease in roundtrip time, and this is why the latency performance is slightly inferior to the one in the previous experiment.

6.2.9 Dynamic Reflective Proactive Broadcast (16)

Our proactive broadcast scheme is based on exchange of link quality information between the client and the server. It is possible to derive the bit error rate (BER) from link quality [HOL04] using the formula depicted in Figure 6.3. With knowledge of the link quality, a binomial distribution function is used to calculate the probability of a message being dropped due to unrecoverable bit-errors. Using this probability the server uses varying levels of proactive temporal redundancy when pushing data to the client.

$$BER(lq) = \begin{cases} (255 - lq) * 0.000025 & if \quad lq \geq 215 \\ 0.001 + (215 - lq) * 0.0008 & if \quad 214 \geq lq \geq 90 \\ 0.1 + (90 - lq) * 0.0064 & if \quad 89 \geq lq \geq 0 \end{cases}$$

Figure 6.3: Formula for calculating BER from link quality

The measured results in terms of latency are not as good as the above two schemes while we observe the same the reliability as the second scheme. We believe the reason this scheme has inferior latency measurements is the fact that the network conditions were not optimal for these mechanisms. In very good network conditions, the single broadcast of the hybrid scheme will complete most of the time, and therefore the reflective scheme should have better performance.

Chapter 7

Related Work

Most of the research on Quality of Service related to Bluetooth is either based on mathematical results, or simulation results from adding Bluetooth protocols to ns2 [NS04], related to routing, handoff/handover and management of pico- and scatternets. However, there are some contributions directly related to our research.

7.1 Bluetooth and Middleware

The Object Management Group (OMG) has developed a specification for wireless CORBA transports [OMG03b]. This specification includes mobile IOR, GIOP tunneling, and handoff solutions for wireless networks. Our Bluetooth transport does not comply with this specification, but it can easily be extended to do so. OMG also has a specification in progress for GIOP tunneling over Bluetooth [OMG03a], which matches our implementation of tunneling GIOP messages over L2CAP.

MIWCO is a research project at the Department of Computer Science, University of Helsinki [MIW03]. MIWCO is an open source implementation of the Wireless Access and Terminal Mobility in CORBA specification [OMG03b],

a wireless extension to the MICO ORB [MICO03]. VIVIAN is an extension to MIWCO that implements GIOP tunneling over L2CAP [VIV04]. Mobiware is an adaptive QoS API that abstracts wireless hardware devices as CORBA objects [ANG04], but is not generalized to Bluetooth technology.

Mascolo et al. provide a detailed characterization of various middleware systems designed to support mobility in distributed systems [MCW02], while Capra et al. present CARISMA, a mobile computing middleware that exploits reflective techniques to enable mobile application designers to address requirements such as context-awareness and adaptation [CEM03].

7.2 Bluetooth Latency Improvements

Valenti and Robert discuss the effects of using AUX1 packets in Bluetooth ACL links, and moving error control into software by using turbo coding, [VAL02b], later realizing that due to the small sizes of Bluetooth baseband packets, the full potential of turbo coding is not achieved [VAL02a]. Their results show that AUX1 packets have higher throughput than DM*n* and DH*n* packets for certain signal to noise ratios (SNR). All their results are based on the assumption that Bluetooth can allow AUX1 packets and packet selection in software. They do not discuss the presence of CQDDR, which is used by most Bluetooth devices today. Valenti et al. also provide a detailed analysis of the throughput of DM*n* and DH*n*

packets, showing that DH*n* packets never achieve maximum throughput [VAL02c].

Das et al. discuss enhancing the performance of ACL traffic by optimizing the L2CAP segmentation process for maximum slot utilization [DAS01]. This is interesting, and could be incorporated into a further optimized Bluetooth subsystem for MicroQoSCORBA, but was found unsuitable for this thesis, superseded by the need to compare experimental results to existing Bluetooth frameworks.

7.3 Quality of Service

Quality of Service for CORBA Objects (QuO) is a reflective middleware framework for adaptive, application level QoS [ZIN97]. QuO, in contrast to MicroQoSCORBA, leaves most constraints to be bound later in the cycle, in order to best facilitate runtime adaptivity. An embedded systems middleware framework cannot afford the late binding flexibility of QuO. For QuO it is necessary to support the runtime adaptivity necessary to deal with the dynamic characteristics inherent in the wide-area network environments it supports.

Scheiter et al. propose a system for applying QoS to MPEG-4 transmissions over Bluetooth, and mention the important role of middleware controlling and applying QoS to the underlying Bluetooth hardware [SCH03].

van Der Zee and Heijenk enumerate and categorize the different QoS options available in Bluetooth hardware [vdZ01], but do not discuss improvements or problems with the current Bluetooth specification, while Yaiz and Heijenk propose a guaranteed service approach for Bluetooth based on different polling schemes [YAI03].

7.4 Summary

We know of no middleware implementations that offer QoS over Bluetooth transports. Nor do we know of any other middleware framework (CORBA or non-CORBA) that allows both hardware and application constraints to be used to tailor the middleware. Middleware for small, embedded devices are sparse or non-existent, especially if be consider the fact that MicroQoSCORBA is designed to support multiple design time and runtime QoS property constraints. Adding a regular Bluetooth transport to any middleware is trivial, as most Bluetooth software stack implementations offer a socket interface, or in the case of JSR-82 [JCP02], a Java API for Bluetooth. However, custom Bluetooth stacks designed to control QoS are sparse, or non-existent. We reiterate that our research concerns middleware QoS control between two Bluetooth nodes, and not problems related to handoff and/or routing.

Chapter 8

Concluding Remarks and Future Research

8.1 Concluding Remarks

In order for middleware to successfully implement strong and flexible QoS mechanisms, there is a need for the Bluetooth specification to be extended in such a way that there are hooks that the overlying software can use to take control of QoS management. Further, in order for middleware to use Bluetooth while maintaining QoS, a custom Bluetooth software stack must be used. This is because, if the recommendations of the Bluetooth specification are followed, the software stack does implement the hooks necessary to control the most important QoS mechanisms needed by a Bluetooth peer to peer network.

The experiments show that middleware can improve the latency in Bluetooth networks by using broadcasts to tunnel GIOP messages. Even though this is not a desired way of transporting CORBA messages, it shows that there are lots of improvements that still can be made to the performance of peer to peer Bluetooth communication. This can easily be helped if the Bluetooth specification is extended to allow vendors to implement use of AUX1 packets for data transmission.

This thesis shows that error control, selection of temporal redundancy scheme and packet selection in many cases can be done more correctly and efficiently in middleware, and that our proactive schemes have lower latencies than the normal unicast transmission method commonly used by connection-oriented L2CAP data transmissions.

8.2 Future Research

Immediate future work should include performing experiments with hardware that supports AUX1 packets in environments where it is possible to deterministically choose the link quality. Further tests can then be performed with the reflective QoS mechanisms in environments where they are better suited.

Bluetooth broadcasting could be utilized for optimizing group communication and with that specialized MicroQoSCORBA fault tolerance subsystems for Bluetooth could be implemented. The effects of moving security from Bluetooth into middleware should be analyzed as part of a possible extension to the MicroQoSCORBA security subsystem. This would relieve Bluetooth hardware of the task of guaranteeing confidentiality, integrity and availability, as we suspect the greater computational power of most hosts will perform these mechanisms more efficiently.

Bibliography

[TUR03] J. Turley. The Essential Guide to Semiconductors. 1st edition, Prentice Hall, 2003.

[BSI01] Bluetooth Special Interest Group. Specification of the Bluetooth System, version 1.1. http://www.bluetooth.com, 2001.

[BSI03] Bluetooth Special Interest Group. Specification of the Bluetooth System, version 1.2. http://www.bluetooth.com, 2003.

[MOR02] R. Morrow. Bluetooth Operation and Use. 1st edition, McGraw-Hill, 2002.

[VAL02a] M.C. Valenti and M. Robert. Custom coding, Adaptive Rate Control, and Distributed Detection for Bluetooth. In *Proceedings of IEEE Vehicular Technical Conference (VTC)*, (Vancouver, BC), Sept. 2002, pp. 918-922.

[McK03a] A. D. McKinnon, K. E. Dorow, T. R. Damania, O. Haugan, W. E. Lawrence, D. E. Bakken, and J. C. Shovic. A Configurable Middleware Framework with Multiple Quality of Service Properties for Small Embedded Systems. In Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications, Cambridge, MA, April 16-18, 2003, 197–204.

[McK04] A. D. McKinnon. Supporting Fine-grained Configurability with Multiple Quality of Service Properties in Middleware for Embedded Systems. Ph.D. Dissertation, Washington State University, 2004.

[MQC04] MicroQoSCORBA. http://microqoscorba.net.

[DOR03] K. Dorow and D. E. Bakken. Flexible Fault Tolerance In Configurable Middleware For Embedded Systems. In Proceedings of the Workshop on Architectures for Complex Application Integration (WACAI2003), part of the 27th Annual International Computer Science Software and Applications Conference (COMPSAC 2003), IEEE, Dallas, Texas, November 3–6, 2003.

[McK03b] A.D. McKinnon, D.E. Bakken, and J. C. Shovic. A Configurable Security Subsystem in a Middleware Framework for Embedded Systems. Submitted for publication.

[NAE04] E. Næss, D. A. Frincke, and D. E. Bakken. Configurable Middleware-Level Intrusion Detection for Embedded Systems. Submitted for publication.

[GAU04] G. H. Thaker. Middleware Comparator. http://www.atl.external.lmco.com/projects/QoS/.

[OMG03a] Object Management Group. GIOP Tunneling over Bluetooth Specification. Adopted specification, OMG document dtc/03-05-06. May 2003.

[HOL04] Marcel Holtmann. CSR BlueCore Specific Information. http://www.holtmann.org

[NS04] The Network Simulator. http://www.isi.edu/nsnam/ns.

[OMG03b] Object Management Group. Wireless Access and Terminal Mobility in CORBA version 1.0. OMG document formal/03-03-64, March 2003.

[MIW03] An Open Source Implementation of Wireless CORBA. http://www.cs.helsinki.fi /u/kraatika/wCORBA.html.

[VIV04] VIVIAN Consortium. GIOP Tunneling over Bluetooth L2CAP. http://www-nrc.nokia.com/ Vivian/Public/Html/ltp.html.

[ANG04] O. Angin, A. T. Campbell, M. E. Kounavis, and R. R. F. Liao. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking.

[VAL02b] M.C. Valenti and M. Robert. Improving the QoS of Bluetooth Through Turbo Coding. In *Proceedings of IEEE Military Communications Conference (MILCOM)*, (Los Angeles, CA), Oct. 2002, pp. 1057-1061.

[VAL02c] M.C. Valenti, M. Robert, and J.H. Reed. On the throughput of Bluetooth data transmissions. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC)*, (Orlando, FL), March 2002, pp. 119-123.

[ZIN97] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, vol. 3, num. 1, April, 1997.

[SCH03] C. Scheiter, S. Rainer, M. Zeller, R. Knorr, B. Stabernack, and K. Wels. A System for QoS Enabled MPEG-4 Video Transmission Over Bluetooth for Mobile Applications. In *Proceedings of IEEE International Conference of Multimedia & Expo (ICME)*, 2003.

[vdZ01] M. van der Zee and G. Heijenk. Quality of Service in Bluetooth Networking - Part I. Technical Report University of Twente, TR-CTIT-01-01, January 2001, 61 pp.

[JCP02] Java Community Process. Java APIs for Bluetooth. Java Specification Request 82, http://www.jcp.org/en/jsr/detail?id=82

[GRA03] Dean A. Gratton. Bluetooth Profiles: The Definitive Guide. 1st edition, Prentice Hall, 2003.

[ERI01] Ericsson. Ericsson ASIC Specific HCI Commands and Events for Baseband C. 2003.

[VER01] P. Verissimo and L. Rodrigues. Distributed Systems for System Architects. Kluwer Academic Publishers, Boston, Massachusetts. 2001.

[BT04] The Official Bluetooth Website. www.bluetooth.com

[YAI03] R. Ait Yaiz and G. Heijenk. Providing Delay Guarantees in Bluetooth. In *Proceedings of Workshop on Mobile and Wireless Networks, MWN 2003*, Providence, Rhode Island, USA, May 19 – 22, 2003.

[DAS01] A. Das, A. Ghose, A. Razdan, H. Saran, and R. Shorey. Enhancing performance of asynchronous data traffic over the bluetooth wireless ad-hoc network. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 1, pp. 591-600, 2001.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

[WAP02] Open Mobile Alliance, Wireless Application Protocol Architecture Specification, 2002
http://www.wapforum.org/what/technical.htm

[LEW91] D. Lewine. Posix Programmers Guide. 1st edition, O'Reilly & Associates, April 1991.

[JCP04] Java Community Process. Concurrency Utilities. Java Specification Request 166, http://www.jcp.org/en/jsr/detail?id=166

[MCW02] C. Mascolo, L. Capra, W. Emmerich. Mobile Computing Middleware. In *Advanced Lectures on Networking*, Springer Verlag New York, Inc., pp. 20-58, 2002.

[CEM03] L. Capra, W. Emmerich and C. Mascolo. "CARISMA: Context-Aware Reflective Middleware System for Mobile Applications", *IEEE Transactions on Software Engineering*, 29(10):929-945, 2003.

[MICO03] MICO Object Request Broker. www.mico.orb

Appendix A

Implementing a driver for the Bluetooth subsystem

A.1 Serial Bus Drivers

MicroQoSCORBA comes with a complete serial bus implementation for Linux and Windows. For other platforms an implementation is required. To do so, the following steps are required:

- Add a preprocessor constant to the mqc_config file in the mqc directory in the section “OS Type”. This constant should be named `MQC_XXX`, where `XXX` is the name of the platform.

  Example:

  ```
  PREPROC_PARAMS+=-DOS_LINUX
  ```

- Add a code section to the `HCIDeviceSerialSocket` class in the `mqc/transports/bluetooth` directory. This section should be inside a preprocessor macro for the platform specified in the mqc_config file. See Figure A.1 for an example. The implementation must complete all the functionality defined in the class. Refer to the Windows and Linux implementations for examples.

```
#ifdef OS_LINUX

...implementation...

#endif
```

Figure A.1: Example of Platform Specific Device Driver Code Insertion

A.2 USB Drivers

A.2.1 Linux

USB drivers are a little trickier to implement. The current implementation of the Bluetooth subsystem includes USB functionality on Linux distributions for Bluetooth devices with Cambridge Silicon Valley chipsets. However, this implementation does not work "out of the box", a few "simple" steps are required as the implementation makes use of the BlueZ USB driver:

- The Linux Kernel must be rebuilt with the BlueZ stack implementation, but *NOT* with any SCO code.
- All the BlueZ components should be built as loadable modules, not linked into the kernel binary.
- Apply the Linux kernel patch, mqc`/transports/bluetooth/bzusbpatch`, to the kernel source code using the standard patch command included in any Linux distribution.
- Recompile the kernel and install the kernel modules

- After rebooting, start the USB device(s) by using the following command <**`hciconfig devno up`**>, where devno is the device number found by executing <**`hciconfig`**> with no arguments
- If **hciconfig** is not existent on the system, download this tool from www.bluez.org, and install it.

A.2.2 Other Platforms

For other platforms than Linux, an implementation is required. Refer to the steps of section A.1, but the implementation must be made to the `HCIDeviceUSBSocket` class. For details on how to implement the driver, refer to platform specific manuals, and [MOR02] pages 380-381.

Appendix B

Configuration of the Bluetooth Subsystem

There are several configuration switches for the MicroQoSCORBA Bluetooth subsystem. Since the C++ version of MicroQoSCORBA does not yet have a backend GUI configuration toolkit, these have to be set manually in the mqc/mqc_config file. The remainder of this appendix enumerates the available switches, and describes their functionality.

B.1 Serial Driver Macros

Refer to the specification for the Bluetooth device being used to determine the correct settings for the following macros.

B.1.1 MQC_BT_SERIAL_DEVICE_SOCKET

This macro switch enables the serial port Bluetooth driver. Code is compiled for Windows or Linux depending on what type of OS_ switch is set. Note that this switch is mutually exclusive with any other device driver socket macro switches.

B.1.2 MQC_BT_SERIAL_PORT

This macro should contain a file name for the serial port. For Windows platforms, an example could be: MQC_BT_SERIAL_PORT="COM1", whereas for Linux platforms an example could be: MQC_BT_SERIAL_PORT=\"/dev/ttyS0\".

B.1.3 MQC_BT_SERIAL_BAUDRATE

This macro contains the baud rate at which the serial connection will run. An example is: MQC_BT_SERIAL_BAUDRATE=CBR_57600.

B.1.4 MQC_BT_SERIAL_DATABITS

This macro contains the data bits for the serial connection. An example is: MQC_BT_SERIAL_DATABITS=8.

B.1.5 MQC_BT_SERIAL_PARITY

This macro contains the parity settings for the serial connection. An example is: MQC_BT_SERIAL_PARITY=NOPARITY.

B.1.6 MQC_BT_SERIAL_STOPBITS

This macro contains the stop bits setting for the serial connection. An example is: MQC_BT_SERIAL_STOPBITS=ONESTOPBIT.

B.1.7 MQC_BT_SERIAL_RTSCTS

This macro switches between RTS and CTS for the serial connection To enable: MQC_BT_SERIAL_RTSCTS=1, to disable: MQC_BT_SERIAL_RTSCTS=0.

B.2 USB Driver Macros

B.2.1 MQC_BT_USB_DEVICE_SOCKET

This macro is a switch that enables the USB Bluetooth driver. Code is compiled for Windows or Linux depending on what type of OS_ switch is set. Note that this switch is mutually exclusive with any other device driver socket macro switches.

B.3 QoS Configuration

B.3.1 MQC_BT_QOS_UNICAST

This macro will force the Bluetooth subsystem to use unicast to transport GIOP messages. Note that if not QoS configuration switches are set, this is the default.

B.3.2 MQC_BT_QOS_BROADCAST

This macro will force the Bluetooth subsystem to use broadcast to transport GIOP messages.

B.3.3 MQC_BT_QOS_HYBRIDCAST

This macro will force the Bluetooth subsystem to use hybridcast to transport GIOP messages.

B.3.4 MQC_BT_QOS_PROACTIVECAST

This macro will force the Bluetooth subsystem to use proactivecast to transport GIOP messages.

B.3.5 MQC_BT_QOS_ROLE_SWITCH

This macro *must* be set if unicast is not used on *both* the client and the server.

Appendix C

Additional Notes for the Bluetooth Subsystem

C.1 Debugging

The MicroQoSCORBA Bluetooth subsystem implementation is complete with detailed trace information. This is an important debugging tool when implementing new low level drivers (serial, USB, etc.), and also when porting the implementation to other platforms. To enable debugging, set the MQC_DEBUG macro flag in the `mqc_config` file.

To include additional debug statements, use the MQC_TRACE macro as defined in the `types.h` file.

C.2 Problems and Solutions for the Linux BlueZ USB Driver

The implementation is heavily tested on various Linux distributions using USB as the interface between MicroQoSCORBA and the Bluetooth device. Should the subsystem still malfunction due to some unforeseen event, it is recommended to take the following steps:

- Run **hciconfig down** for all the devices
- Unload all the BlueZ modules from the system

- Unplug the devices from the USB host controller
- Load the BlueZ modules
- Plug the devices back into the USB host controller
- Run **hciconfig up** for all the devices