CONFIGURATION AND CODE GENERATION TOOLS FOR MIDDLEWARE TARGETING SMALL, EMBEDDED DEVICES

By

OLAV HAUGAN

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
December 2001

To the faculty of Washington State University:

The members of the Committee appointed to examine the thesis of OLAV HAUGAN find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGMENT

I wish to thank my advisor, Dr. Dave Bakken, for all the help and advice I have received during my years at Washington State University, and for all the help, support, and ideas he has given me during my research work.

I would also like to thank my committee members, Dr. Curtis Dyreson, and Dr. John Shovic, for their comments and support.

I wish to thank Ruby Young, the graduate secretary, for her tremendous kindness in helping students.

I also want to thank my parents for being the nicest parents there is, and Mandy for her support and for keeping up with me. I also wish to thank all my friends for being supportive of me.

CONFIGURATION AND CODE GENERATION TOOLS FOR

MIDDLEWARE TARGETING SMALL,

EMBEDDED DEVICES

Abstract

by Olav Haugan, M.S.
Washington State University
December 2001

Chair: David E. Bakken

Distributed systems are playing an increasing role in computer applications, and many systems are becoming smaller and smaller. Middleware, such as CORBA helps the developer meet the increasing focus on distributed applications by providing development tools. However, current middleware tools are not suited for very small, embedded applications.

MicroQoSCORBA provides an architecture and a development toolkit targeting small, embedded devices. This toolkit is the topic of this thesis. The toolkit features several tools to configure and constrain the application, enabling the system developer to design and create small, embedded system applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

As networking technology has become faster and cheaper in recent years, traditional stand-alone applications are evolving into distributed ones. Wireless network technologies such as Bluetooth [BLU01] and IEEE 802.11 [IEE99] accelerate this advancement in distributed computing. The importance of distributed systems is increasing. This trend is occurring not only in large corporate computing centers but also, increasingly, is reaching down into the embedded systems market [TEN00]. The number of devices manufactured across this spectrum is staggering, at eleven billion parts per year. The embedded systems marketplace includes a wide range of devices from small kitchen appliances to jumbo-jets and ocean-going vessels. Devices that previously were thought of as stand-alone devices are now more and more interconnected and forming distributed, networked systems.

Middleware, such as Common Object Request Broker Architecture (CORBA), has proven to be a valuable tool in dealing with the many facets of distributed systems [BAK01]. CORBA provides excellent object-oriented and platform-neutral middleware architecture for developing distributed applications [OMG01]. It provides elegant separation of the interface and implementation, along with its encapsulation of communication protocols and processing environments, and a rich set of services and industry-specific APIs. CORBA ensures access, location, and persistence transparencies. Access transparency makes the local and remote

data objects accessible in the same way. Location transparency gives us the ability to access objects without the knowledge of their location. Persistence transparency hides the processing, storage, and communication functions for a distributed object. The object does not need to be concerned with loading and unloading of the object from persistent storage. These features of CORBA allow not only for language independence and a high-level programming building block, but also mask heterogeneity and allow for many different implementation configurations and optimizations that can be employed transparently to the client.

Current middleware tools are being applied to small, embedded systems with varying degrees of success. Many high-end workstations have more onboard CPU cache than some embedded systems have available memory in both RAM and ROM. Needless to say, tools developed in these memory rich environments often fail to scale down to memory starved environments. Other solutions have been developed for small, embedded devices, but typically they are point solutions that do not have the flexibility to widely cover the embedded systems market. They do not provide the particular embedded application designer with the right set of constraints appropriate for its specific application functionality and target hardware.

We have designed and implemented most of MicroQoSCORBA, a middleware framework that can scale down to memory-starved environments. In doing so, we allow the middleware to be tailored both to the precise properties of the hardware

as well as to the application's configuration requirements. Additionally, limited memory is only one of the many facets of small, embedded devices. We have thus built an architecture that addresses other facets of MicroQoSCORBA such as fault tolerance, security, power usage, and system performance, and we are presently extending the framework in these areas. Some of these are typical resource constraints and others may be perceived as Quality of Service (QoS) issues.

As MicroQoSCORBA's architecture was being defined to encompass each of these facets, it was quickly determined that traditional distributed systems concepts such as "client", "server", "push", "pull" and so forth were defined at too coarse a level and were not orthogonal enough for use in the fine grained configuration control required for very small environments. To that end, we have further refined these terms and broken them down into their respective orthogonal components. This improved taxonomy of terms proved to be a key component in the successful design of MicroQoSCORBA's architecture.

As the embedded systems market expands, many developers will transfer into this area with a limited amount of prior experience. Hence, developing an application would require a lot of research into distributed system facets such as details about protocols and communication. Developers with little or no experience with embedded or distributed systems will be incapable of directly implementing the great flexibility that MicroQoSCORBA affords. To solve this problem,

MicroQoSCORBA provides a set of CASE tools to aid the developer. The developer specifies the overall system constraints, and then the tools work behind the scenes to meet these requirements. For example, a developer might know that a special project requires certain kinds of interactions with other computers and only a few data types. The CASE tools aid the developer in choosing between the various interaction styles and data type choices that best meet his or her design constraints (e.g., low power, minimal memory). Thus, these CASE tools aid developers who are not experts in embedded systems, middleware, or quality of service to leverage the experience of the MicroQoSCORBA development team as instantiated in the toolkit.

The contributions of this thesis are a toolkit targeting embedded systems developers or application domain experts and the MQC-IOP protocol. The toolkit helps the developer in the design and development of a distributed system by offering a wide spectrum of configuration options in an easy to use tool. The toolkit reduces the time it takes to develop a distributed system application targeting an embedded system platform. The wide range of configuration options helps the programmer reduce the application size to ease the deployment of the system to an embedded system with sparse resources. The MQC-IOP protocol is based on the standard GIOP protocol [HER99]. The MQC-IOP protocol takes advantage of the MicroQoSCORBA architecture to decrease the size and increase the performance of an application.

The thesis is organized as follows: Section 2 describes the refined taxonomy of distributed system terms. Section 3 overviews the base architecture of MicroQoSCORBA. Section 4 gives a detailed discussion of the design and implementation of the configuration and code generation tools. Section 5 contains a description and evaluation of our implementation prototype of MicroQoSCORBA. Section 6 discusses related work, and section 7 contains a short conclusion and future work.

2 ARCHITECTURAL TAXONOMY

The space of embedded applications is diverse in terms of interaction with other applications, the type and size of data transferred, and whether or not the application is active or passive in a number of ways, etc. Moreover, the space of embedded devices also varies widely in terms of RAM, ROM, power consumption, and many other ways. While the space of application and hardware characteristics is quite wide and must be supported, a given embedded system is typically designed with one task in mind, often a relatively simple one, and it is almost always designed for a specific target hardware platform. To take advantage of these specific constraints, we were required to divide the facets of embedded systems and the middleware architecture into a more fine-grained definition.

The dedicated task assigned to an embedded system and target platform together dictate the tradeoffs and constraints a given embedded application allows its middleware to make in order to meet its (often constrained) requirements for RAM, ROM, power, and QoS. Since MicroQoSCORBA was designed for these dedicated applications, we found it necessary to best categorize some of the useful fundamental facets of embedded systems to support the wide range of tradeoffs outlined above. As shown in Table 2.1, four key categories of interest were identified: Hardware, Roles, Communication, and subsets of CORBA's Interface Definition Language (IDL) [MCK01].

Roles (Client/Server/Peer)		
Connection Setup	**Data Flow**	**Interaction Style**
Initiates Conn. Setup Receive Conn. Requests Bind • Name Service • Hardwired Logic • Other	Bits In Bits Out Bits In/Out Passive Pro-Active	Sync (Send/Receive) Async (One-Way Msgs) Msg Push Msg Pull Exceptions Event & Notif. Services
Embedded H/W	**IDL constraints**	**Communication Support**
Homogenous Asymmetric Communications Support Processing Capabilities System size • Small • Medium • Large	Message Types Data Types Exceptions Message Payload # of interfaces # of methods	Wired Wireless Transports • Ethernet • Serial • 1-wire 1-Tier (i.e., IIOP) 2-Tier Approach (i.e., G.W. to IIOP)

Table 2.1: Refined Distributed Systems Architectural Taxonomy

2.1 Embedded Systems Hardware

The choice of what hardware to support is a critical factor for an embedded application. A constraining hardware choice allows the MicroQoSCORBA designer to constrain code generation and other optimizations performed by MicroQoSCORBA. One key choice is the decision regarding the heterogeneity of

the embedded systems hardware as well as the hardware to which these devices will be connecting. Typically, middleware is built to support a large degree of heterogeneity, but this does not have to be the case with embedded systems. Many embedded systems designers have substantial control over their deployment environment, so they can (and often do) reasonably dictate a common platform for all devices within the system. These fine-grained choices regarding hardware configurations means that we lose some of the transparency that CORBA provides. However, an embedded systems developer needs to know what type of hardware the application is deployed to, and using this knowledge gives strength to MicroQoSCORBA ability to deploy applications to different system platforms. It can also be less expensive, in a global sense, to deploy asymmetric hardware, especially if only a few nodes need to be resource rich and the rest can be resource poor and thus less expensive.

The hardware support for a communications layer is consequential. Extremely 'small' devices might not need a full Ethernet solution or perhaps Ethernet will be too expensive for the target application. Support for other communication layers should then be available for those devices. The capability of the processor holds an important role. The power of the processor specifies how fast an application can process information and communicate with other devices. This must be addressed and the application must be design thereafter.

The size of the system is also included in Table 2.1 to indicate that the size of the individual systems and the networks that they compose will vary widely. An increase in system size usually yields a more complex and resource hungry system.

2.2 Role Definitions

One can easily see that the role a device plays has an impact upon both the software and hardware that needs to be deployed at a node. We started our initial design based upon a simple client/server/peer stratification of our devices. However, it became apparent that more constrained role taxonomy would allow MicroQoSCORBA to package only the most limited functionality with a given application. For example, a parasitically powered device cannot initiate connection to a remote system, so in the extreme case, one could remove code that initiated connections with other devices. Some devices may only have the capability (or need) to either transmit or receive data. If this is the case, then one may remove software from these devices that either receives or transmits data.

Different types of object location management may also be applied based upon the different needs in each individual application. Some applications may not need a naming service to locate the peer, but a hard coded reference or an IOR [OMG01], found in a shared memory location, could be sufficient.

The data flow can be restricted to only bits out or bits in, passive, or pro-active. These restrictions can again reduce the complexity and memory footprint of a distributed application by limiting the features of the marshaling library. Restrictions can be applied to the interaction style such as one-way messages and message push or message pull only.

2.3 Communications Support

It is unreasonable to assume that all embedded devices will be able to support a full IIOP solution. In some applications, the cost of implementing such a high-level communications protocol will be too costly. MicroQoSCORBA provides support for additional environment specific protocols. Some devices may also want to support several different protocols simultaneously. A server would want to receive requests from different clients on different communication channels. This allows for great flexibility when developing a distributed system on different platforms.

When extremely resource constrained devices need to communicate it might be appropriate for them to do so in a two-tiered approach. Namely, these devices would use a 'resource poor' protocol to connect to a gateway machine that would then bridge the data and commands from these devices onto the Internet at large, with a standardized IIOP protocol. An example of such a device is the iButton [DAL01]. The iButton speaks only 1-Wire, and needs a gateway/bridge to the

Internet to be able to communicate to other devices connected to the Internet. Such a bridge can be build using the TINI [LOO01] platform running MicroQoSCORBA with support for multiple protocols.

2.4 IDL Constraining

CORBA has a rich and powerful Interface Definition Language that is used to define an application's functionality. Not all embedded systems require the full power of CORBA IDL, thus the potential exists for MicroQoSCORBA to support a small subset of the standard IDL mappings. An application ideally suited for an 8-bit processor could forgo support for 32-bit values. By itself this is not a large improvement, but combined with the fact that exception support may be removed or message formats might be constrained based on this IDL constraining, the potential exists for significant gains in both reduced resource usage of the hardware and simpler software. Moreover, restriction on the number of interfaces and the number of methods allowed can reduce the complexity of protocols and protocol overhead.

3 MICROQOSCORBA BASE ARCHITECTURE

The flexibility designed into MicroQoSCORBA increased the role that the IDL compiler has, as seen in Figure 3.1. Our architecture can target a wide range of systems of different sizes and properties. This meant that various novel approaches had to be taken to be able to support these requirements. We will discuss these choices below.

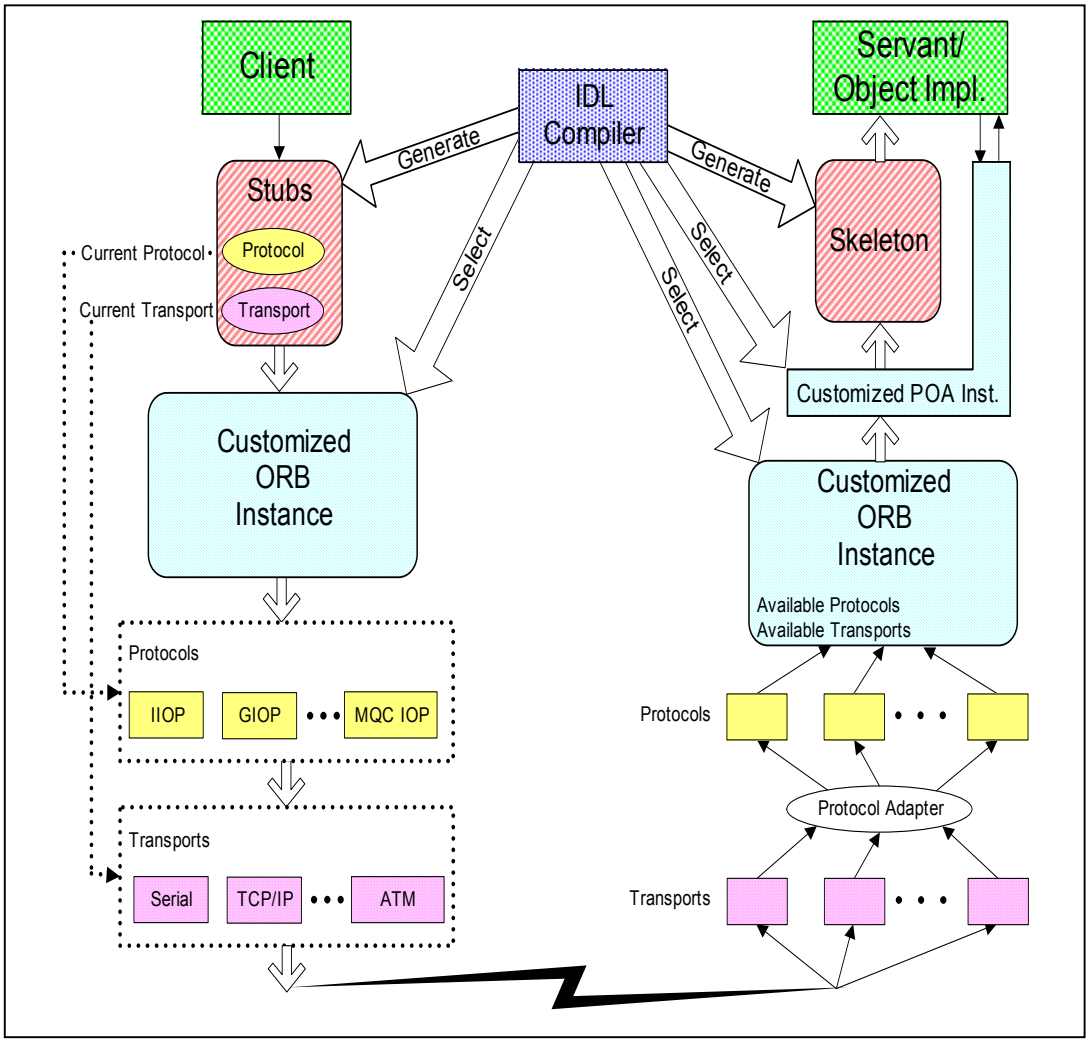

Figure 3.1: MicroQoSCORBA Architecture

3.1 IDL Compiler

CORBA makes use of an IDL compiler to help the developer to implement the application by producing code for the client-side and server-side communication. This is mainly stub and skeleton code to marshal and un-marshal method arguments and return values. Since a standard CORBA implementation does not have the flexibility and configuration options that we have seen designed into MicroQoSCORBA, the IDL compiler plays a relatively small role. However, because of MicroQoSCORBA flexibility we needed to expand the role that the IDL compiler plays in our architecture. A standard CORBA implementation provides one ORB with all the functionality needed. The functionality found in such an ORB does not always fit the functionality that is needed in an embedded application. We addressed this problem by extending the IDL compiler to generate more "smart" stubs and skeletons that interact with a customized ORB that is selected by the IDL compiler based on the requirements given to the application.

The application layer, which consists of the implementation of the client application and the server object implementation, keeps the transparency seen in the standard CORBA architecture. We will not lose any transparency to the application by adding more functionality to the stubs. The API between the client application and stubs is the same for every configuration. The same client application can make use of a stub generated by MicroQoSCORBA using the

TCP/IP transport and the GIOP protocol and a stub using the UDP transport and the MQC-IOP protocol. However, we give information to the IDL compiler about the target hardware, which is not the case for a standard IDL compiler. We can specify whether the hardware is homogeneous or heterogeneous. By doing this, we lose the hardware transparency that is provided by the CORBA standard. We also lose the portability of the application to other environments. However, the application developer can choose to not know anything about the hardware and configure the system as heterogeneous. This will work for both homogeneous systems and heterogeneous systems.

3.2 Customized ORBs and POAs

To adhere to the increased flexibility and functionality that our architecture provides we had to make changes to the current CORBA architecture. MicroQoSCORBA makes use of customized ORB instances, several which can coexists in the MicroQoSCORBA development environment. Using CASE tools, the developer will be able to choose which ORB instances to use for his or her system implementation. A customized ORB could be generated by the CASE tools, or provided by the development environment.

Moreover, the same functionality that is needed from the ORB on the server side is not necessarily the same functionality needed by the client side. We have

chosen to divide the ORB into one ORB for the client and one for the server to reduce the size of the client and server.

The IDL compiler is made aware of the existence of the customized ORBs via various configuration settings. During the IDL compilation, the compiler chooses the selected ORB combination both for the client and server, and generates appropriate code that makes use of the desired ORB.

3.3 Protocols

We intend for MicroQoSCORBA to support many different protocols and at the same time keep the sizes of the client and the server as small as possible. We want the developer to be able to choose which protocols to use and which protocols that are not needed. MicroQoSCORBA must be able to include only the protocol modules that are chosen to limit the size of the application. We solved this on the client side by adding a reference to the protocol wanted in the IDL generated stub so that the application knows which protocol to use, and so that we can only include that protocol in the ORB library. On the server side we had to solve this a bit different since we want the server to support several protocols simultaneously and call comes from the bottom up. The IDL generated proxy cannot know what protocol is used before the actual proxy is called. The ORB includes a list of protocols that are active. At compile time, we generate a configuration class that is used by the server side ORB to choose which protocols to support.

MicroQoSCORBA will have support for several different protocols, including IIOP and IIOPlite. We have developed a protocol called MQC-IOP that is based on IIOP and take advantage of the constraints that can be applied to MicroQoSCORBA to increase the performance of an application.

3.3.1 MQC-IOP

The MQC-IOP protocol is based on the GIOP 1.1 protocol [HER99]. We have implemented certain constraints to achieve better performance. We removed certain fields that are not needed for our specialized protocol. This yields a shorter message header, which in turn simplifies the header construction. We have removed the version field, the service context field, and the requesting principal parameter. This saves at least ten bytes in a request message and at least six bytes in a reply. However, the main improvement from the GIOP standard protocol is the use of a fixed length object key. This means that the (inline) marshaling code becomes simpler and more efficient, since at code generation time you know how long the MQC-IOP header is. The IDL compiler can generate code that uses a fixed size buffer. Marshaling of method arguments and return values becomes much simpler. Normally, at compile time it cannot be determined how long the message request or reply header is, since the service context and object key fields are of variable size. The IDL compiler must generate marshaling code that checks if the storage space is large enough and then re-size it if it is too small. This tends to be inefficient. Using a fixed size header found in MQC-IOP helps the IDL

compiler to generate simpler and more efficient code. A fixed size buffer is used and no re-sizing of storage space is needed.

The MQC message header contains a magic number, byte order, message size, message type, and 3 bytes are reserved for future use.

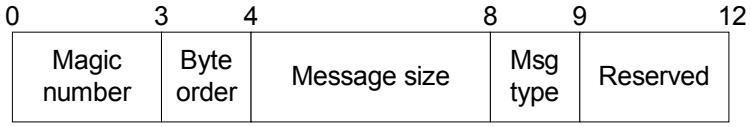

Figure 3.2: MQC-IOP message header

The magic number is a three-byte sequence holding the value “MQC”. This is used to distinguish the MQC-IOP protocol from other protocols. The receiving end reads the three first bytes of the message and determines what protocol it is.
The byte order is one byte which specifies whether the message is encoded in big endian or little endian. A value of 0 indicates a big endian byte order, and a value of 1 indicates a little-endian byte order. The rest of the message is encoded using the specified byte order.

The message size is a four-byte entity holding the total size of the message excluding the 12 bytes of the message header.

A one-byte message type member holds the type of message that follows the message header. The message types currently used are request message (0) and reply message (1).

A request message holds the following fields: Request ID, response expected, object key, the operation length, and the name of the operation requested.

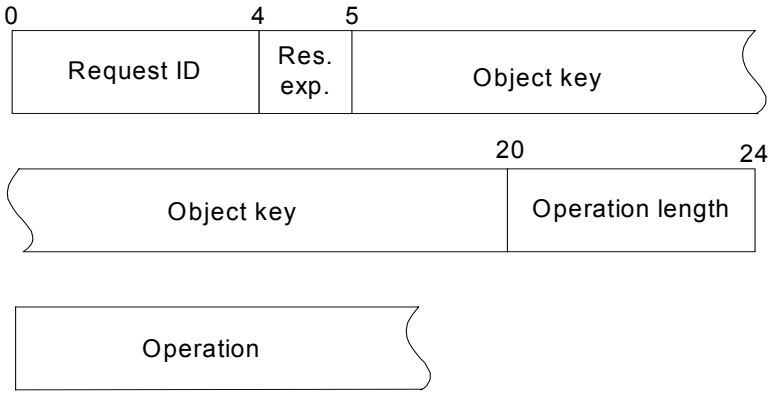

Figure 3.3: MQC-IOP request message header

The request identifier is a 4-byte field that uniquely identifies the message sent. A client might expect several replies from different requests, and the client might receive those replies out of order. The request ID helps the client to associate request sent with replies received.

Response expected is a one-byte field that holds a value of 0 if no response is expected. This value is used when the client neither expects a reply nor wants a reply. A response expected field with value 1 means that the client wants to receive a reply back.

The object key holds a 15-byte key used by the server to identify the target object residing on the server. A server may contain several objects and the object key uniquely identifies the requested object.

The next field specifies the 4-byte length of the following operation name.

The operation field is a variable length field specifying which operation or method to invoke on the server.

A reply message is the answer sent from the server back to the client. It contains a request id and a reply status.

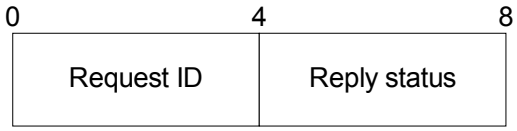

Figure 3.4: MQC-IOP reply message header

The request ID is the same request ID used in the request message header. When a server sends a reply back to the client, the server uses the request ID of the request to identify the reply.

Reply status holds a 4-byte value specifying the status of the operation performed. The values applied to this field are no exception (0), user exception (1), and system exception (2). A no exception value is used when the operation was performed and no exception was thrown. If the implementation throws a user

exception then the user exception value is used. When the server cannot find the implementation, the operation does not exist, or some other failure happens, the server sends a system exception.

3.4 Communications Layer

In the same way as protocols are selected, the communication protocols are also selected using references in the stubs on the client side. On the server side it is obvious that we cannot leave the choice of which communication protocols to use to the IDL generated proxy. The ORB needs to know what communication ‘ports’ to listen to when the ORB starts up. A list of available transports is therefore included in the ORB in the same way as protocols.

We note that the ORB could have used an abstract factory pattern [GAM95], but that would have required linking in functionality for all of the MicroQoSCORBA's communication layers into a given application, something that was not needed or desired.

3.5 Quality of Service

A key component of MicroQoSCORBA's architecture is the use of customized ORB and POA components. One size does not fit all, and this is even more true with quality of service issues. Thus, our design allows for the use of customized ORB instances. We are presently designing quality of service subsystems to

support real-time performance, security, and fault tolerance. In each of these subsystems, there are multiple implementations of the subsystem, offering different tradeoffs of quality of service versus resource consumption (e.g., encryption strength versus memory and power usage). MicroQoSCORBA's profiling and CASE tools will allow the user to pick and choose a set of compatible quality of service subsystem implementations, ruling out ones that do not make any sense to combine (something of course the developer cannot, in general, know).

4 TOOLKIT

To aid the embedded systems developer in design and configuration of a wide spectrum of architectural configuration choices, we have developed several CASE tools. The MicroQoSCORBA CASE tools currently include a GUI-based configuration tool and an IDL code generator. These tools provide the developer with an easy way to develop a distributed system for embedded devices. Figure 4.1 shows the interaction between the tools in the toolkit.

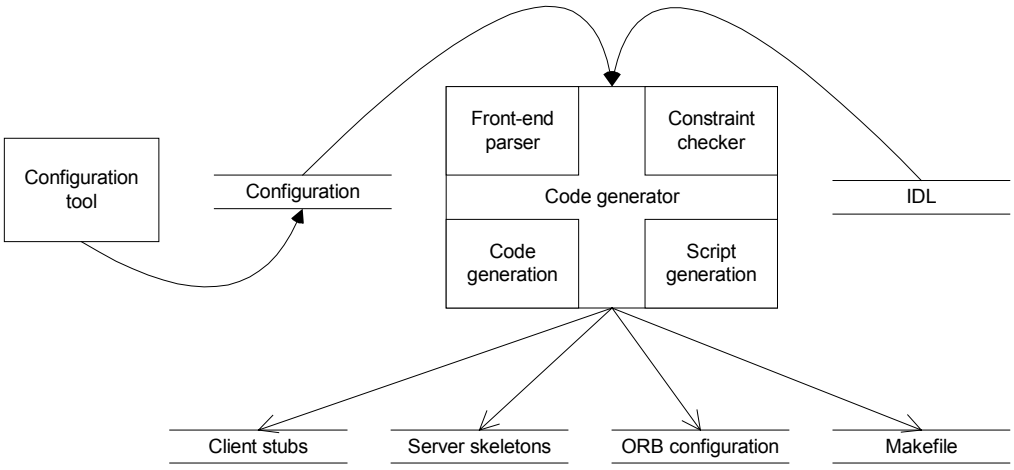

Figure 4.1: Tool interaction

The application developer runs the configuration tool to configure and constrain the application that he or she wants to build. The configuration is saved to an application specific configuration file. The code generator reads the configuration file together with the IDL specification. Based on the information gathered from these files, the code generator generates the client-side stubs, server-side skeletons, ORB configuration file, and a makefile to ease the compilation of the

application. We will outline the configuration tool and the code generator in the following sections.

4.1 MicroQoSCOBRA Configuration Tool

The configuration tool's main purpose is to let the developer determine what type of architecture on which to build the application by choosing from the constraints outlined in Section 2 and 3. The configuration tool consists of a graphical user interface (GUI) where the application developer can set the properties and constraints on the application. Once gathered, these constraints are stored in an application specific configuration file. The IDL compiler tool uses these values to customize and specialize each application. We have divided our discussion of these constraints into four different sections: Data types (IDL constraining), Quality of Service, Communication and Protocols, and Miscellaneous options.

4.1.1 Data Types

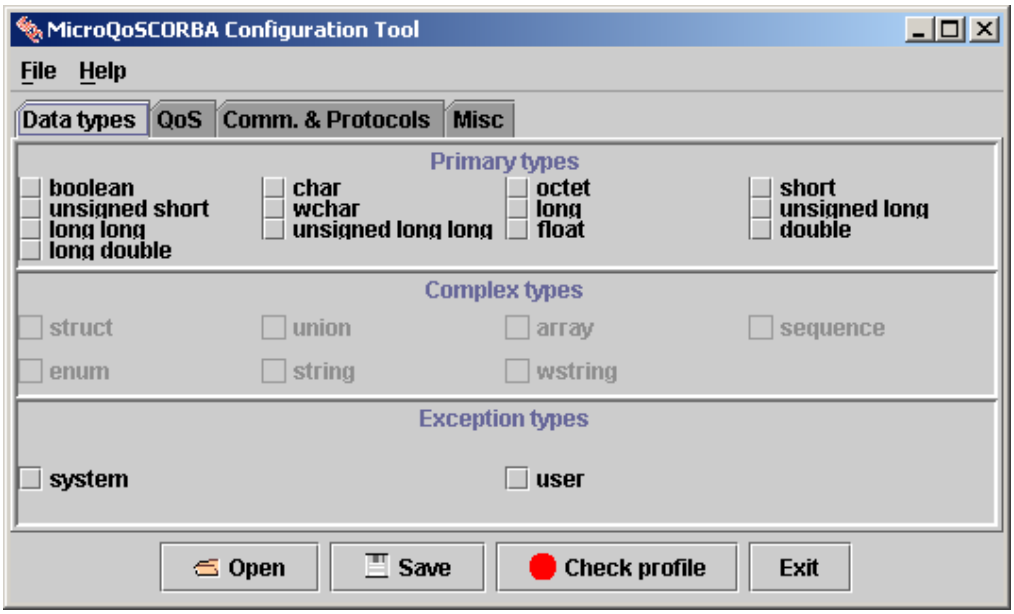

Figure 4.2: Data Type Constraints

Many small, dedicated systems do not need support for all the different data types that a standard IDL mapping encompasses. We thus support only a subset of the data types by selecting which specific types to use in our application (see Figure 4.2). We have divided the data types into 3 sections. These are primary types, complex types, and exceptions. Primary types are the 13 basic types that are supported by the IDL standard. They range from one byte boolean to 16 byte long double. Each data type can be individually selected. The complex types are mainly types composed of one or more primary types. Complex types include struct, union, array, sequence, enum, string, and wstring. All of the complex types are similar to their C++ counterparts, except for sequence. The sequence type is

an unbound array of one (complex) data type. The exception types consist of system exceptions and user exceptions. System exceptions are exceptions defined by the system and thrown when there is a marshalling error, the object is not found, the operation does not exist, or other system errors. A user exception is an exception defined in the IDL file, specified by the developer, and thrown from the object implementation.

Reducing the number of data types yields a smaller and simpler marshalling and un-marshalling library, and thus reduced code size and memory footprint. By removing support for user and/or system exceptions, we can remove the code that handles exceptions and thus generate smaller and simpler code for stubs and skeletons.

4.1.2 Quality of Service

Quality of service is important in many distributed systems. We want to be able to select what type of QoS subsystems to employ, what implementation is needed from each subsystem chosen, and how to parameterize each. The QoS part of the tool is still the subject of ongoing research, but is presented here in the interests of completeness.

4.1.3 Communication and Protocols

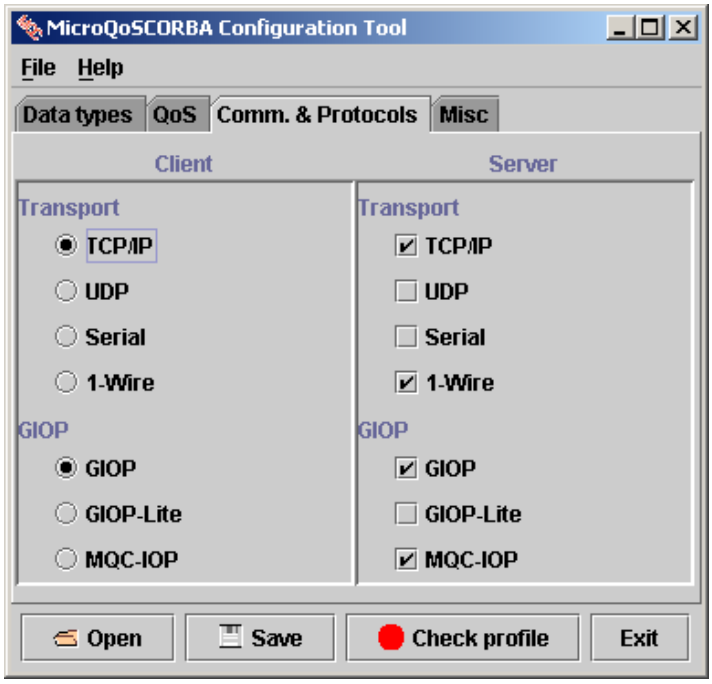

Figure 4.3: Communication and Protocol Constraints

To allow for flexibility in the client/server application we need to be able to specify what communication layer and protocols to support. Embedded systems are developed on many different platforms, and so each application needs to be able to adapt to these different environments. The different configuration options, with regard to transports and protocols, are shown in Figure 4.3. We have chosen to consider the client and server separately. The relationship between a client and a server is often many-to-one. A minimal client only needs to be able to use one specific transport and one specific protocol; this is reflected on the left side of Figure 4.3. This means that the client may be simple and small. The server

however, often must have the ability to communicate with several different clients on different systems. The server side can, therefore, be configured to support more than one transport layer and protocol type.

The client can chose to use one of four different transport protocols. A client with an Ethernet card available would have the choice between the connection-oriented TCP/IP protocol and a connection-less variant in UDP. A device like the iButton [DAL01] would use the 1-wire transport, since this is its only mean of communicating to the outside world. Devices have also the possibility to use a serial channel for communication.

The client can also chose one out of three different protocols to use. If interoperability is needed with other ORB's the client can use the standard GIOP protocol. Accepting reduced interoperability, the client can also use GIOPlite [COR00]. The client can then only communicate with ORB's that implement GIOPlite, e.g. TAO [LEV98]. If no interoperability is needed, then MQC-IOP can be used, which yields a smaller and more efficient client.

The server has the same options as the client in the choice of transports and protocols. However, as mentioned earlier, the server might want to support more than one protocol or transport at the same time. An application could make use of this as a bridge between two different transport layers. A server with support for

1-wire and TCP/IP could operate as a bridge between a 1-wire device and the Internet.

4.1.4 Miscellaneous Other Configuration Constraints

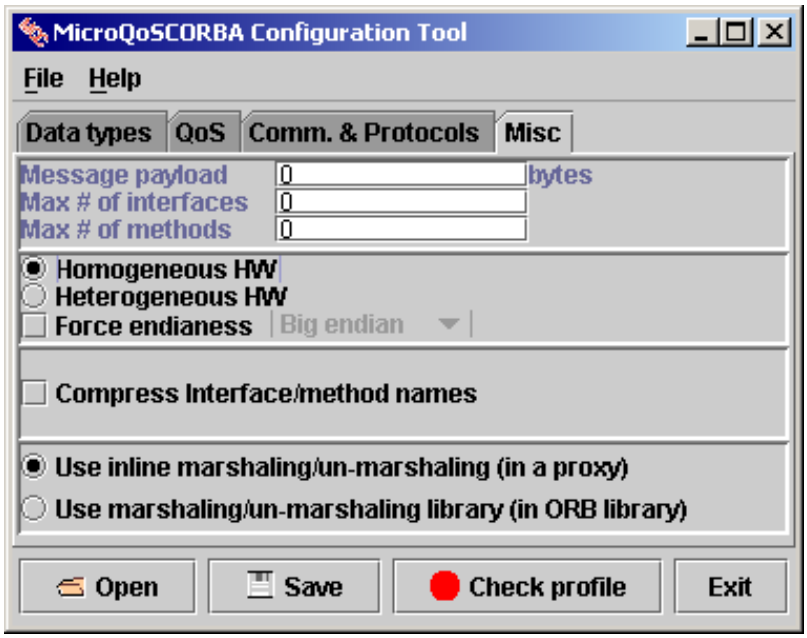

Figure 4.4: Miscellaneous Configuration Choices

An embedded systems developer generally knows the hardware on which a given system is going to be deployed. MicroQoSCORBA can utilize this information by using these hardware restrictions to bind and constrain various software implementation choices. This will reduce the code size and simplify the generated stub and skeleton code. As shown in Figure 4.4, the developer may set a number of constraints in the GUI in order to accurately reflect the hardware bindings. One

of these choices is the use of homogeneous hardware. (The meaning of homogeneous hardware in this context is the endianess of the system.) Once this GUI switch has been set, the GUI instructs the IDL compiler to generate simplified marshalling and un-marshalling code. On a heterogeneous system, the compiler needs to generate code that supports both big-endian and little-endian marshalling and un-marshalling. Taking advantage of the knowledge of developing for a homogeneous system, the code generator generates code that only supports big endian or little endian, and not both, which reduces the code size and complexity. Endian-ness can also be forced if needed. This is useful when you know what type of endian-ness the remote system uses, and you want to simulate a homogeneous system.

Other optimizations that are available are restriction of message payload, the number of interfaces, and the number of methods. These restrictions can help to reduce the complexity of the client and server communication protocols and reduce the application's data flow. The application developer also has the option to compress interface names and method names into a fixed, n-bit integer. This will reduce the size and the overhead of a message, since method names are normally coded into the request message by its full name. The final configuration option available is the choice of using inline marshalling and un-marshalling or to use a library. Depending on the number of interfaces, methods, and parameters to

the methods, choosing to use inline code instead of a library might increase the speed and reduce the code size of the application.

The developer can also check the configuration for errors or impossible combinations. For example, GIOPLite and heterogeneous hardware cannot be used together, since GIOPLite depends on being run on homogeneous hardware. The check is run when the user pushes the “Check profile” button. If no error is detected, the red light on the button changes to green. However, if errors are found, a message box will pop up and explain the errors as seen in Figure 4.5.

Figure 4.5: Configuration error

4.2 MicroQoSCOBRA IDL Code Generator

The task of an IDL compiler is to generate client-side code and server-side code that works with the ORB library. The compiler reads the interface definitions found in an IDL file and produces the appropriate stubs for the client and skeletons for the server. These stubs and skeletons are generated so that the

system developer can concentrate on the top-level functionality of the application instead of designing and implementing the low-level communication details.

An overview of the MicroQoSCORBA IDL code generator is shown in Figure 4.6. The MicroQoSCORBA IDL compiler is based on the ZEN [DOC01] IDL compiler. We have taken the ZEN IDL compiler and extended it to generate code for MicroQoSCORBA.

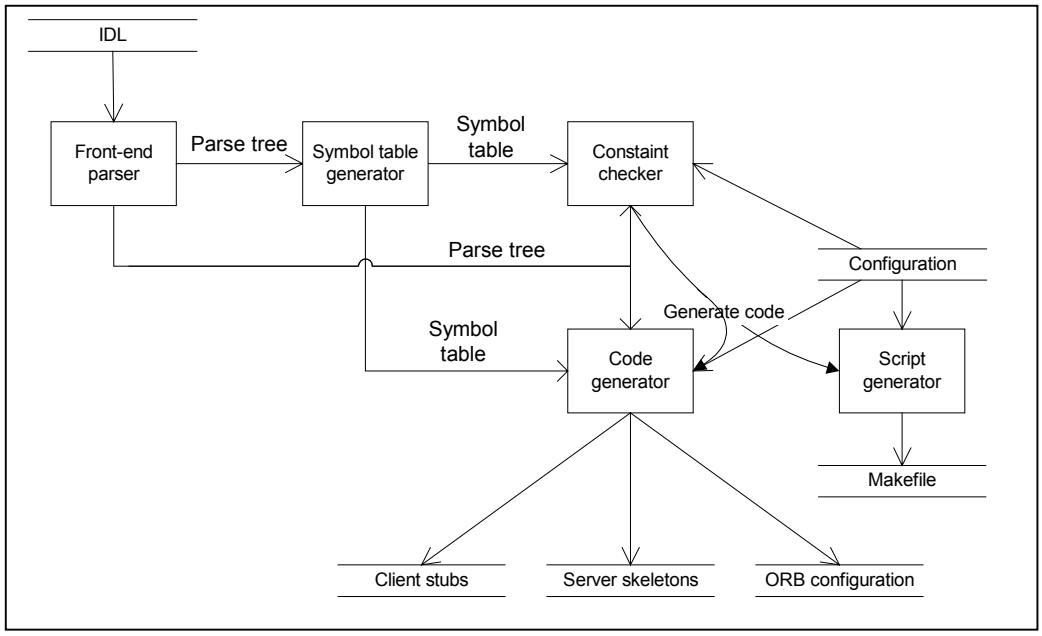

Figure 4.6: Overview of MicroQoSCORBA code generator implementation

The IDL compilation can be divided into three phases: Front-end parsing and symbol table generation, constraint checking, and code generation. In the first phase the compiler reads the IDL specification file and checks for syntax errors in the definition. If no errors are found, the front-end generates a parse tree that is used by the two other phases. After a parse tree is produced, the symbol table

generator is run to generate a symbol table used by the code generator to ease the generation of code. The second phase consists of checking the IDL definitions for compliance with the IDL constraining constraints set by the developer in the configuration tool. If a violation is found, it is reported and the compiler does not produce any code. However, if no violations are found, the final phase is executed. The code generator uses the parse tree, symbol table, and configuration options to produce the desired code. Based upon the configuration selected earlier by the developer, the code generation tool will generate stubs and skeletons that reflect the choices already made by the developer. During code generation, the tool makes decisions regarding what transports and protocols to use, what type of marshalling and un-marshalling code to produce, and what libraries to include. The script generator produces a makefile that eases the compilation of the client and the server.

5 IMPLEMENTATION EVALUATION

An initial working MicroQoSCORBA prototype has been developed. Our initial prototype has been developed in Java, though we have done interoperability experiments with C++ ORBs as discussed below. Several small devices have been deployed with small JVMs (e.g., Dallas Semiconductor's TINI board [LOO01]), so we felt that even though Java would not allow us to initially target the extremely small, embedded devices, Java's flexibility would allow us to quickly develop a cross-platform, working prototype.

Our working MicroQoSCORBA implementation has successfully interoperated with a number of other ORB implementations (e.g., JacORB [BRO97], TAO [LEV98], VisiBroker [BOR01], and ZEN). Our implementation supports multiple IDL modules and interfaces. It also supports the standard simple IDL data types (e.g., boolean, char, octet, short, long, float, double). We were also able to deploy our implementation on both an x86 Linux box and a TINI board.

For the purpose of evaluation, we developed several simple applications in MicroQoSCORBA, JacORB (release 1.3.30), and ZEN (June 2001 build). The first application was based upon the IDL definition shown in Figure 5.1.

```
module sensor {
     interface volts {
          long read_smart_transducer(in long arg1);
     };
};
```

Figure 5.1: Experimental interface #1

The MicroQoSCORBA options used were: GIOP protocol, TCP/IP transport, inline data marshalling, and interface/method name compression. JacORB and ZEN were run with their respective default settings (no debugging output). The results of our evaluation are shown in Tables 5.1, 5.2, and 5.3. In these tables, Java RMI, ZEN, and JacORB results are not reported for the TINI platform. This is because the TINI's limited memory (512 Kbytes) and stripped down JVM (support for Java 1.1 only with no dynamic class loading and no support for Java RMI) was too constrained an environment for Java RMI, JacORB, and ZEN. All results were obtained using SUN's JDK 1.3.1 and the TINI class libraries.

5.1 Application Size

The application sizes reported for Linux, shown in Table 5.1, are the total sizes of all the class' files for the application, including libraries. MicroQoSCORBA provides two ORB libraries, one for the client and one for the server, with the client library being the smaller of the two. Both ZEN and JacORB use only one library for their clients and servers. The file size numbers reported under the TINI

column in Table 5.1 represent the sizes of the compiled executable generated by the TINIConverter tool. One can see from the results shown in Table 5.1, that MicroQoSCORBA was able to produce a very small application. But, we readily acknowledge that these results are unfair to a degree. Namely, MicroQoSCORBA does not provide support for any of the standard CORBA services (e.g., Naming, Event, Trading) whereas JacORB does. These services take up about 3.7 MB of the total size. JacORB and ZEN also implement the standard Java definitions that are specified and required by the IDL Java language mapping as specified in CORBA 2.3, which is not done in MicroQoSCORBA. The size of the abstract classes that define the standard java language mapping is about 1 MB for JacORB and about 200 Kb for ZEN. A better size comparison would factor out the size of ZEN and JacORB code that implement CORBA features that MicroQoSCORBA does not have. MicroQoSCORBA is not fully implemented with all the features designed in the architecture and will of course increase in size with the addition of these features. However, the point of this comparison is to show that we have a tool to reduce the size of an application, which the other two ORB's do not have.

	Class File Size on Linux (bytes)		**Executable File Size on TINI (bytes)**	
	Client	**Server**	**Client**	**Server**
Java RMI	4,385	4,597	N/A	N/A
MicroQoSCORBA	35,182	43,412	14,598	22,117
ZEN	356,080	351,704	N/A	N/A
JacORB	6,596,196	6,592,640	N/A	N/A

Table 5.1: Application Size Comparison

The first column of the table shows the size of the same interface implemented using Java RMI. We see that the client and server sizes are very small. This is because most of the code for the client and server are hidden inside the Java class libraries, which is not accounted for here. Although using Java RMI provides us with portability across platforms, we loose the interoperability with other systems that are implemented using other programming languages. However, you can achieve interoperability using RMI-IIOP, but only if all the remote interfaces are originally defined as Java RMI interfaces. We also loose the flexibility that is designed into MicroQoSCORBA.

Adding interfaces and methods to the application will only increase the total size by the size of the added implementation, and an increase in the generated stubs and skeletons, since the libraries will be the same.

Using the IDL definition found in Figure 5.1 we produced an application with the 'maximum' size by including all the features of MicroQoSCORBA. These figures are shown in Table 5.2. The client side configuration used where the following: Support for system and user exceptions, TCP/IP transport, GIOP protocol, heterogeneous hardware, no method compression, and marshalling and un-marshalling library. The size of the client does not increase substantially because the client can only be configured with one transport and one protocol. The increase in size comes from the use of marshalling library instead of inline

marshalling. The server was configured the same way, except the server was configured with support for GIOP, GIOP-Lite, and MQC-IOP. The server was only configured with the TCP/IP transport since the other transports aren't fully implemented in the prototype and thus adding UDP, 1-Wire, and serial support would not yield any increase in the server image size.

	Class File Size on Linux (bytes)	
	Client	**Server**
MicroQoSCORBA	39,665	58,085

Table 5.2: Maximum size

5.2 Timing Results

The times measured and reported in Table 5.3 are the average times it takes for a method call, starting from when the call is first issued and ending when client receives the return reply. The average time on Linux is computed from 10,000 calls, while on TINI the same number is 1,000. The client and server ran on the same processor with no other network traffic. The Linux results were obtained on a Celeron 400 MHz PC with 128 Mbytes RAM running Red Hat 7.1, using the Ethernet loopback device. The TINI results were obtained on the TINI board, which is powered by a 36.864 MHz DS80C390 CPU, using its loopback adapter as well. The results show that MicroQoSCORBA performs well because of its small size and simplicity, and partly because of the method name compression. Using inline marshalling and un-marshalling does not show the performance gain

expected. By performing several experiments with more complex interfaces and methods, we saw that using a library instead of inline code performs better. A reason for this result could be the nature of the implementation of the library. Also, the Java compiler itself does some inlining and thus could improve the performance of the code using the library.

	Average call time (ms)	
	Linux	**TINI**
MicroQoSCORA	0.56	210.00
ZEN	2.03	N/A
JacORB	1.46	N/A

Table 5.3: Timing Results

5.3 Memory Usage

The amount of memory used on the TINI board was measured by taking the difference between the amount of memory available at the beginning of the application and the amount available at the end. The number for the server is the amount used after the server is ready, but before any calls are made to the server. For each call to the server, about 14k bytes of RAM is consumed. These numbers are initial and dependent on when the garbage collector is run. The numbers shown are the worst-case results that were observed.

	Memory usage on TINI (bytes)	
	Client	**Server**
MicroQoSCORBA	31,840	35,552
ZEN	N/A	N/A
JacORB	N/A	N/A

Table 5.4: Memory Usage

The second application was developed to show how the call time increases with the increased complexity of the application. The application was based upon the IDL definition shown in Figure 5.2.

```
module signal {
     interface alarm {
          long alarm_sensor_reading(
                         inout double arg1,
                         inout double arg2,
                         inout double arg3,
                         inout double arg4,
                         inout double arg5,
                         inout double arg6,
                         inout double arg7,
                         inout double arg8);
     };
};
```

Figure 5.2: Experimental interface #2

5.4 Timing Comparison Between GIOP and MQC-IOP

MicroQoSCORBA was first run with the same settings as described earlier. During the second run the same configuration was kept except for now the MQC-

IOP protocol was used, homogeneous hardware, and the endianess was forced to big-endian. The result is shown in Table 5.5.

	Average call time (ms) on Linux
GIOP	0.78
MQC-IOP	0.66

Table 5.5: Timing comparison between GIOP and MQC-IOP

The result shows an expected increase in the average call time. The increase in call time is due to heavier payload on each message transmitted and because of more marshalling and un-marshalling operations must be performed in the client stub and the server skeleton. The payload is now 64 bytes in the request and 68 bytes in the reply, compared to four bytes in the previous example. We can also see the advantage of using the MQC-IOP protocol in combination with homogeneous hardware. The average call time is less than for the GIOP protocol because of simpler marshalling and un-marshalling using fixed-sized buffer.

5.5 Library Marshalling versus Inline Marshalling

Figure 5.3 shows the number of two-way arguments of a function versus the size of the application. We used one interface with one method with the number of two-way arguments as shown on the x-axis. Each argument was a 4-byte long. MicroQoSCORBA was configured with TCP transport, GIOP protocol, heterogeneous hardware, and method name compression. The figure shows that

for the client we get a smaller application size using inline marshalling when we have less than 13 arguments. Using inline marshalling for the server shows that we get a smaller application size with less than 14 arguments. If we increased the number of methods and distributed the number of arguments evenly over the methods, we would get a smaller pivot number since adding more methods would add more code to the stubs and the skeletons. Reducing the size of the argument would yield a larger pivot number, while increasing the size of the argument would give us a smaller pivot number.

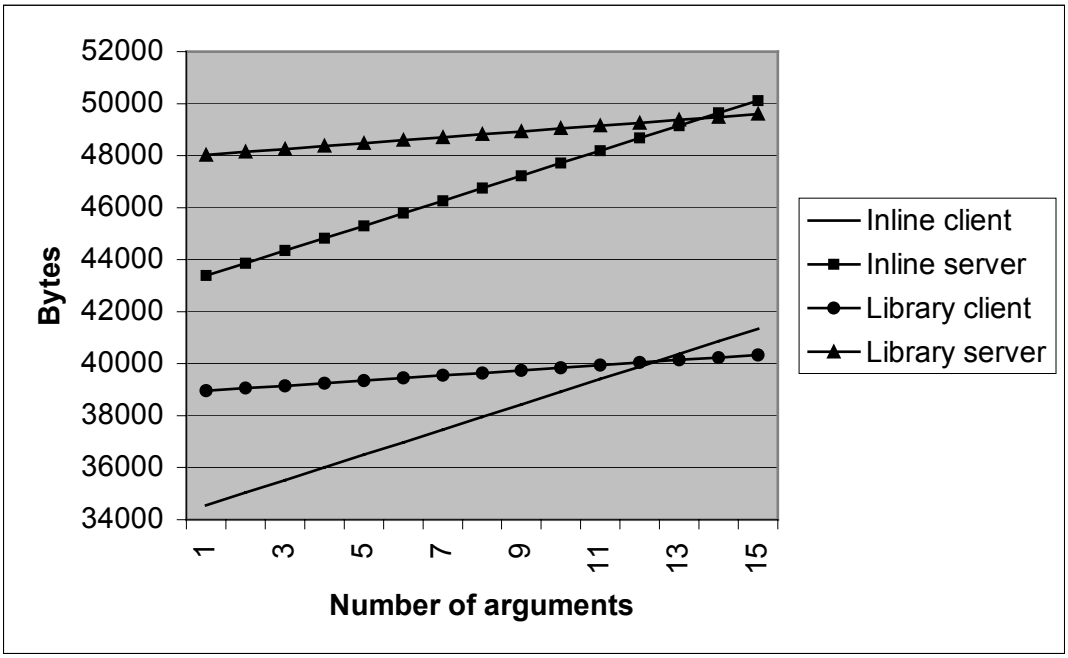

Figure 5.3: Library marshalling versus inline marshalling

6 RELATED WORK

We know of no CORBA or non-CORBA middleware framework that allows both the application and hardware constraints to be used to tailor the middleware, or of middleware for small, embedded devices that has been designed to support multiple QoS properties. MicroQoSCORBA is also allowing the constraints to be chosen at a much finer granularity than any other system of which we are aware of, because we are designing it from the device level up rather than top-down from a reflective architectural reference model.

The Software Architect's Assistant [KRA96] is a graphical CASE tool to automate the process of design and construction of a distributed system. The Software Architect's Assistant is component based, and generates Darwin [DUL93] code based on the design done in the CASE tool. However, this tool does not provide the same flexibility as we need, and it is specific to the Regis [DUL94] programming environment.

Another GUI CASE tool is found in [FOS96]. It is similar in to the Software Architect's Assistant in that it is component based and build upon the Darwin configuration language. A CORBA implementation is supported, but it does not target the low-end embedded systems market.

Emerging CORBA standards and products are only beginning to address the deeply embedded systems market. The minimumCORBA [OMG01] is a cut-

down version of the full CORBA specification. It is targeted embedded systems with limited resources. Support for Dynamic Invocation Interface (DII), Dynamic Skeleton Interface (DSI), Dynamic Any, the Interface Repository (IR), and parts of the Portable Object Adapter (POA) are removed because these are not needed in a static system after the application has been burned into the ROM on an embedded system. However, removing these features from a minimumCORBA implementation only reduces the memory footprint by about a half.

The e*ORB™ [VER00] framework by Vertel gets much smaller and closer to what is needed with respect to memory footprint. However, this is a point solution that does not allow application developers to tailor their constraints in ways appropriate to their applications to meet resource constraints. Further, memory footprint is only one of a host of resource and quality of service (QoS) issues that must be addressed for small, embedded devices.

LegORB [CAM00b, MIC00] is the "little brother" of dynamicTAO (described below) and has implementations at 6K (Palm Pilot) and 20K (Windows CE). It does allow some customization. However, while there are very few details available, there seems to be much less granularity in the choice of constraints than with MicroQoSCORBA, and there is no support for QoS.

Another related work that is beginning is the OMG Smart Transducers Interface Request For Proposal (RFP) [OMG00]. Smart transducers are small, single

purpose devices (e.g., sensors and actuators) with some level of built-in processing and communications support. This RFP is seeking to standardize a very lightweight communications API for these devices. Thus, this effort is focused on only a small part of the MicroQoSCORBA framework, namely the communication subsections.

Other CORBA-based frameworks have explicit support for Quality of Service or employ reflection, or both, as does MicroQoSCORBA, but are not intended to scale down to small devices. These systems also generally allow adaptively much later in the design lifecycle than does MicroQoSCORBA, which is appropriate given they are not targeting small footprints. MULTE is a multimedia middleware platform that handles a range of latency and bandwidth requirements [ELI00]. MULTE provides a framework that supports dynamically (re)configuration of QoS. A reflective architecture is implemented in the OpenORB Python Prototype [AND00]. The OpenORB architecture uses reflection to achieve a flexible and adaptable middleware solution. OpenORB provides openness and a configurable component based architecture. The dynamicTAO framework allows dynamic adaptation and allows for replacement of different strategy modules for concurrency, scheduling, and security; its footprint is never less than one MB [CAM00a].

MMLite [FOR01] is a modular system architecture that allows a system to be built from object-oriented components. Each application is built, either statically

or dynamically, from a set of components that determine its behavior. Depending upon the choice of components, a system with a memory footprint as low as 10 KB has been achieved. The QoS properties of an MMLite application can be changed by selecting different subsets of components. The primary QoS property that MMLite targets is real-time behavior and this is achieved in part via the use of multiple implementations of the scheduling component. MicroQoSCORBA has a much broader QoS breadth (e.g., security, fault tolerance, and real-time behavior). Another advantage of MicroQoSCORBA is its support for analysis tools that help quantify the impact of various component (both functional and non-function) costs and compositional properties. Another difference between MMLite and MicroQoSCORBA is that MMLite uses the COM object model and XML based SOAP as its communication framework, whereas MicroQoSCORBA is CORBA based and uses GIOP (and subsets thereof) to communicate.

The research on Application Specific Operating Systems [HUM01] and on VEST [STA00] [STA01], in particular, is focused on component based real-time embedded systems. Their focus is on building lightweight components that are suitable for either static or dynamic use within embedded systems. They also have a strong emphasis on providing adequate analysis tools that can provide the designers with information on the performance of both functional and non-functional aspects of the developed application. Aspects are used to weave in support for non-functional requirements, but their non-functional requirements of

concern are tightly focused on real-time performance. Our research is focused on a slightly higher development layer, namely application middleware, and we are also seeking to develop non-functional components for additional non-functional components such as security and fault tolerance.

7 CONCLUSION

7.1 Summary

MicroQoSCORBA is a middleware framework targeting small, embedded systems. MicroQoSCORBA is a highly configurable architecture with support for Quality of Service and has the ability to scale down to memory-starved environments. Together with a development toolkit, MicroQoSCORBA architecture offers a development environment for embedded systems developers.

Section 2 described a refined taxonomy of distributed system architecture concepts, while section 3 defined the base architecture of MicroQoSCORBA. Section 4 presented a toolkit available for the developer to easy design and implement a distributed system application. The toolkit consist of a graphical configuration tool and an IDL code generator producing client stubs, server skeletons, and scripts. Section 5 describes the implementation evaluation.

The result of this research is a toolkit for developing distributed system applications for embedded systems with sparse resources. The toolkit enables the developer to choose from a wide range of configuration options using a graphical user interface. The toolkit produces the necessary code for client-side and server-side communication, reducing the time it takes to develop an application.

7.2 Future Work

Future work for MicroQoSCORBA toolkit includes further development of the IDL code generator, design, implementation, and integration of profiling tools, and Quality of Service support.

The IDL code generator can be extended to support complex data types such as struct's, unions, and strings. Extending its flexible marshalling options by including an option to generate the marshalling and un-marshalling library (during code generation) for the supported data types can also further extend the code generator. Also, the code generator can be improved to generate more efficient code in regard to data marshalling and un-marshalling.

Profiling tools are useful for the application developer to make decisions regarding resource usage versus performance tradeoffs. Profiling tools can be integrated into the MicroQoSCORBA development environment so that the developer can get estimates of the resource usage and performance of a specific configuration for a distributed application.

The toolkit has been designed with support for Quality of Service properties such as security, real time, fault tolerance, and bandwidth properties. A concrete implementation of these configurations can be implemented into the toolkit.

8 BIBLIOGRAPHY

[AND00] A. Andersen, G. Blair, F. Eliassen. OOPP: A Reflective Component-Based Middleware, in *NIK, Bodø*, Norway, November 2000.

[BAK01] D. E. Bakken. Middleware, chapter in Encyclopedia of Distributed Computing, J. Urban and P. Dasgupta, eds, Kluwer Academic Publishers, 2001, to appear. <http://www.eecs.wsu.edu/~bakken/middleware.pdf>, November 2001.

[BLU01] Bluetooth SIG. Specification of the Bluetooth system, February 22, 2001, <http://www.bluetooth.org/>, November 2001.

[BOR01] Borland software corporation. Visibroker: CORBA technology from Borland, <http://www.borland.com/visibroker/>, October 2001.

[BRO97] G. Brose. JacORB: Implementation and Design of a Java ORB, in *Proceedings of DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems*, Cottbus, Germany, September 30 - October 2, Chapman & Hall 1997.

[CAM00a] R. H. Campbell, F. Kon, P. Liu, L. Magalhaes, J. Mao, M. Roman, T. Yamane. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB, in *Proceedings of the FIP/ACM*

International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), New York, April 3-7, 2000.

[CAM00b] R. H. Campbell, F. Kon, D. Mickunas, M. Roman. LegORB and biquitous CORBA, *Workshop on Reflective Middleware* at Middleware'2000, New York, April 2000.

[COR00] C. O'Ryan, O. Othman, D. C. Schmidt. Applying Patterns to Develop a Pluggable Protocols Framework for ORB Middleware, chapter in Design Patterns in Communications, Linda Rising, ed., Cambridge University Press, 2000.

[DAL01] Dallas Semiconductor Inc. iButton: Java-powered, thermocron, digtal jewelry, TINI, <http://www.ibutton.com/>, 2001.

[DOC01] DOC: Distributed Object Computing Laboratory, UCI. Real-time CORBA with ZEN, <http://www.zen.uci.edu/>, 2001.

[DUL93] N. Dulay, J. Kramer, J. Magee. Structuring Parallel and Distributed Programs, *IEEE Software Engineering Journal*, 8(2), March 1993.

[DUL94] N. Dulay, J. Kramer, J. Magee. Regis: A Constructive Development Environment for Distributed Programs, *In IEE/IOP/BCS Distributed Systems Engineering*, 1(5), September 1994.

[ELI00] F. Eliassen, B. Hafskjold, T. Kristensen, R. H. Macdonald, T. Plagemann, H. O. Rafaelsen. Flexible and Extensible QoS Management for Adaptable Middleware, in *Proceedings of International Workshop on Protocols for Multimedia Systems (PROMS 2000)*, Cracow, Poland, October 2000.

[FOR01] A. Forin, J. Helander, P. Pham. Component Based Invisible Computing, *IEEE Real-Time Embedded Systems Workshop, part of the 22nd IEEE Real-Time Systems Symposium*, December 3, 2001. See also http://rtds.cs.tamu.edu/helander.pdf.

[FOS96] H. Fossa, M. Sloman. Implementing Interactive Configuration Management for Distributed Systems, *Third IEEE Int. Conference on Configurable Distributed Systems*, Annapolis, May 1996.

[GAM95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995.

[HER99] T. Herron, P. Klinker, W. Ruth. *IIOP Complete. Understanding CORBA and middleware interoperability*. Addison Wesley Longman, Inc., 1999.

[HUM01] M Humphrey, C Lu, R Poornalingam, J Stankovic, R Zhu. Application Specific Operating Systems, <http://www.cs.virginia.edu/~stankovic/asos.html>, December 2001

[IEE99] IEEE, IEEE P802.11, The working group for wireless LANs, <http://grouper.ieee.org/groups/802/11/index.html>, November 2001.

[KRA96] J. Kramer, J Magee, K. Ng. A CASE Tool for Software Architecture *Design Journal of Automated Software Engineering*, 3(3-4), Kluwer Academic Publishers, August 1996.

[LEV98] D. Levine, S. Mungee, D. C. Schmidt. The Design of the TAO Real-Time Object Request Broker, *Computer Communications Special Issue*, 21(4), April 1998.

[LOO01] D. Loomis. *The TINI™ Specification and Developer's Guide*, Addison-Wesley, June 2001.

[MCK01] D. A. McKinnon. Interface Description Language, chapter in Encyclopedia of Distributed Computing, J. Urban and P. Dasgupta, eds, Kluwer Academic Publishers, 2001, to appear.

[MIC00] R. Campbell, F. Kon, D. Mickanus, M. Roman. LegORB and Ubiquitous CORBA, *Workshop on Reflective Middleware*, part of *IFIP/ACM Middleware 2000 Conference*, April 2000.

[OMG00] Object Management Group. *Smart Transducers Interface Request For Proposals*, Document orbos/2000-12-13, Object Management Group, Framingham, MA, 2000.

[OMG01] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.5,* Object Management Group, Framingham, MA, 2001.

[STA00] J. Stankovic. VEST: A toolset for constructing and analyzing component based operating systems for embedded and real-time systems, University at Virginia TR CS-2000-19, July 2000. Available at <http://www.cs.virginia.edu/~stankovic/psfiles/asos_paper.pdf>

[STA01] J. Stankovic. VEST: A toolset for constructing and analyzing component based embedded systems, to appear, Springer-Verlag lecture notes. Available at <http://www.cs.virginia.edu/~stankovic/psfiles/lake-tahoe.ps>

[TEN00] D. Tennenhouse. Embedding the Internet: Proactive Computing, *Communications of the ACM*, May 2000.

[VER00] Vertel Corporation, e*ORB™ Technical Description, Vertel Corporation, <http://www.vertel.com/>, November 2001.

APPENDIX A: INSTALLATION INSTRUCTIONS

Installation

The installation of MicroQoSCORBA development toolkit is done in one simple step. Unpack and un-tar the distribution to your preferred directory on a Unix system. Assuming the current directory is "WORK_DIR" and the name of the distribution is "MQC.tar.gz", the command for unpacking and un-tarring is:

```
$ tar zxvf MQC.tar.gz
```

This will unpack the necessary files to "WORK_DIR".

Test of installation

Running one of the example applications that comes with the distribution can test the installation. Several steps must be taken to ensure correct operation.

1. Assuming the current directory is WORK_DIR, change directory down to MQC/examples/simple.
2. Edit the Makefile found in the directory. Make sure the **JAVA** and **JAVAC** variables point to the java command and the javac command respectively. Change the **IDLPATH** variable to **IDLPATH=.:/WORK_DIR/MQC/CASE/**

Run the IDL compiler by executing

```
$ make idl
```

Next, edit the generated Makefile.mqc file. Make sure the **JAVAC** variable points to the javac command. Change the **CLIENT_TARGET_DIR** and **SERVER_TARGET_DIR** to point to the directories where you want the application to reside, e.g. **CLIENT_TARGET_DIR= /WORK_DIR/MQC/test/client/** and **SERVER_TARGET_DIR= /WORK_DIR/MQC/test/server/**. Also change the **PATH** variable that points to where the MQC libraries are stored. This would be **PATH= /WORK_DIR/MQC**/.

Next, build the application:

```
$ make –f Makefile.mqc
```

To start the server, change directory to /WORK_DIR/MQC/test/server and execute

```
$ java –classpath . Server &
```

To start the client, change directory to /WORK_DIR/MQC/test/client and execute

```
$ java –classpath . Client `cat ../server/simple.ior`
```

If the installation is working correctly, the client and server prints several “success” to the terminal, meaning the test was successful.

APPENDIX B: EXAMPLE CLIENT AND SERVER SOURCE

```
import MQC.*;
import MQC.holders.*;

public class Client
{
	public static void main(String[] args)
	{
		C_ORB orb = new C_ORB();
		System.out.println("ORB created!");

		Object object = orb.string_to_object(args[0]);
		System.out.println("String to object!");

		simple.test1 remote = simple.test1Helper.narrow(object);
		System.out.println("Narrow!");

		byte res = 0;
		res = remote.fun1((byte) 127);
		System.out.println("Reply: " + res);
	}
}
```

```
import MQC.*;
import MQC.holders.*;

import java.io.*;

public class Server
{
	public static void main(String[] args)
	{
		S_ORB orb = new S_ORB();
		System.out.println("ORB created!");

		POA rootPOA = new POA(orb, "RootPOA");
		test1Impl impl = new test1Impl();

		Object object = rootPOA.servant_to_reference( impl );
		String ior = orb.object_to_string(object);
		System.out.println("object to string!");

		try
		{
			FileWriter out = new FileWriter(new File("simple.ior"));
			out.write(ior);
			out.close();
		}
		catch(IOException e)
		{
			System.out.println("Error writing IOR");
			return;
		}
		System.out.println("Waiting for requests...");
		orb.run();
	}
}
```