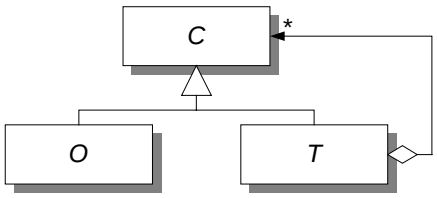
XOIP – XML Object Interface Protocol

COT/3-34



Centre for Object Technology

Centre for
Object Technology

Revision history: V0.1 16/2-2001 First COT release

Author(s): Morten Kvistgaard Nielsen, Teknologisk Institut
Allan Bo Jørgensen, Teknologisk Institut

Status: Draft

Publication: Public

Summary:

XOIP describes a way in which heterogeneous networked embedded systems can interface to a variety of distributed object architectures using XML. An implementation of XOIP is available for download. This document is a thesis for the Masters Degree in Computer Science at the University of Aarhus

The Centre for Object Technology (COT) is a three year project concerned with research, application and implementation of object technology in Danish companies. The project is financially supported by The Danish National Centre for IT-Research (CIT) and the Danish Ministry of Industry.

Participants are:
Maersk Line, Maersk Training Centre, Bang & Olufsen, WM-data, Rambøll, Danfoss, Systematic Software Engineering, Odense Steel Shipyard, A.P. Møller, University of Aarhus, Odense University, University of Copenhagen, Danish Technological Institute and Danish Maritime Institute

Abstract:

In this thesis we shall present our solution to the problem of achieving interoperability between heterogeneous distributed object architectures and paradigms. What makes our solution special is that it is specifically designed to address the problems faced by embedded systems, where lack of system resources have hitherto prevented their participation in distributed object systems. Since embedded systems are more likely to be placed in heterogeneous object systems than their desktop counterparts, the two issues are naturally linked.

This work is supported in part by the Center for Object Technology, a part of the Center for Information Technology, and by the Danish Technological Institute, Center for Information Technology.

Table of contents - overview

Table of contents

Introduction

In our thesis work, we have concentrated our efforts on two primary goals, namely

1) Making a system for allowing distributed object invocations to be performed across distributed object system architecture boundaries.
2) Designing the system in such a way that it will be possible to utilize it on embedded hardware.

This has led us to investigate a number of interesting aspects of distributed object systems and their architectures, and reporting on these findings make up the first two parts of our thesis. The first part consists of the preliminary studies we've conducted and the background information we've gathered, and part two consists of the more theoretical discussion of how to achieve interoperability between object architectures. A third part consists of a description of the design of our own solution to the problem, XOIP, and the considerations that led to our particular solution. A fourth part consists of information about the inner workings of XOIP, which has been implemented in a proof-of-concept state, available for download under the GNU Public License.

Our motivations for working with object system interoperability and accessibility to embedded systems are:

- The wired household - merging household appliances with smart technology (e.g. Ericsson and Electrolux)
- Increasing desire to integrate systems from the smallest embedded systems to huge mainframe installations (e.g. remote control and diagnostics)
- The lack of a "Grand Unified Theory" of distributed object models: Existing models fits well into specific (albeit broad) scenarios, but can not easily be extended to deal with the issues of integrating the variety heterogeneous systems
- Systems should enjoy freedom of choice when deciding on an object and programming model (e.g. JINI vs. DCOM and class-based vs. prototypical programming languages)

Solutions that will allow heterogeneous platforms and architectures to cooperate are needed, without imposing an architecture and programming model on every subsystem.

In this thesis we will explore issues related to interoperability between distributed object models and the requirements that must be met for embedded systems to participate in these. We describe and motivate a possible solution to interoperability between distributed object systems based on XML as the message format. The requirements to the underlying transport layer are modest. Based on our findings and ideas we provide a proof of concept implementation that allows us to access DCOM, CORBA and Java RMI objects from an embedded device in a general fashion – that is, it makes no difference to the embedded device whether it invokes methods on an object in any particular object architecture.

By using this approach we show that it is possible to interoperate between heterogeneous systems without changing the server and with few requirements to client-side programming. And we demonstrate that this is a viable approach on embedded systems, by making it possible for the system to operate on scant resources.

Part One: Preliminary findings

1. What characterizes embedded hardware?
2. Embedded communication systems
3. CANbus higher level protocols
4. COM and run-time type information
5. Distributed Object Systems
6. Distributed Garbage Collection

1. What characterizes embedded hardware?

This chapter on embedded systems is composed of two individual sections. The first describes the current range of embedded processing hardware (CPU's and related), the second describes the current state of real-time operating systems (RTOS) for embedded hardware.
The focus will be on best practice in industrial use, not a survey of state-of-the-art technology. Each section will contain a conclusion about its subject.

Embedded CPU's

A wide variety of CPU's for embedded systems are in use today. They range from single digit MHz 8-bit processors to 32 or more bit processors running at several hundred MHz. Each has their place in different embedded applications, as the need for processing power varies substantially more than is the case for desktop systems. In this thesis we will focus on 32 bit processors only, since most of the recently developed embedded CPU's are 32 bit, and not much effort is spent upgrading older (usually meaning 8 or 16 bit) designs to newer production technologies, meaning these processors are relegated to running at sub-par clock speeds, further reducing their processing power. Note that low-clock cycle 8 or 16 bit architectures are occasionally used deliberately, since higher clockspeeds and bandwidth imply an increase in power consumption, which for handheld devices is very problematic. There are more factors involved than clock speed, however. The number of Programmable Input/Output (PIO) ports, the number of Interrupt Request (IRQ) lines and the number and resolution of programmable timers are important in most embedded applications, since these are usually about reading values and computing new settings based on these values.

Most current embedded processors claim to use 32 bit, but some confusion exists as to what actually constitutes a 32 bit processor. Most 16 bit processors can combine internal registers to perform limited 32 bit arithmetic, and some have 16 bit data buses combined with 24 to 32 bit address buses. This does not necessarily make them full-blown 32 bit processors, and the consensus in the industry seems to be that a processor is 32 bit primarily if it has 32 bit general data registers. Most of these have from 24 to 32 bits of address space, so a de facto definition for 32 bit processors would be 32 bit general data registers and at least 24 bits of address space.

Although the list of such processors seem long and the variety great[1], it comes down to a few key factors, namely:

Clock speed
The range of clock speeds for embedded processors is big indeed. Even for 32 bit processors, it ranges from single digit to several hundreds of MHz. Most of the popular designs are currently in the range 25 to 100 MHz, but as technology advances and prices drop this looks certain to increase. Many processors come in several clock speeds.

[1] There are literally several hundreds of these processors – most can be found through [EG3]. They are not necessarily all that different – for instance, Motorola makes about 10 different variants of their popular ColdFire embedded RISC processor.

Integration, size and programmability of input/output system
Most current embedded processors have on-board PIOs, timers and other communication-related features. Some have TCP/IP stacks, some have Ethernet adapters and others have CAN controllers more or less built in. Some have these communication abilities embedded in co-processors tightly integrated with the main processor. The typical range of onboard PIOs is 3-6 serial and/or parallel ports, and most have 2-4 programmable timers. The number of IRQ-lines varies considerably more, with a range of 4 to 8 covering most current processors. Sometimes the interrupt system is placed in another, tightly coupled, dedicated processor. This area seems to be where there is the most divergence among processors, as embedded applications seem varied enough to warrant such diversity of products.
Some of the most powerful embedded processors are naturally found in embedded devices requiring lots of processing power, for instance PostScript printers. These are typically quite expensive, high performance processors that are partially tailored to their particular task. Processors for this purpose branches off from the line of general-purpose processors out of necessity, as their ancestor processors were simply not powerful enough. They have had a high enough volume or unit cost to warrant specialized development, making them markedly different in several respects (specialized instructions and/or registers, features included/omitted).

Type of instruction set
As with processors for desktop system, embedded processors can be placed somewhere on a CISC to RISC axis, with most being placed at either end. There are a growing number of hybrids and more recent processor designs are trying to marry the two, to combine the compactness of instruction representation inherent in most CISC designs, with the ease and speed of instruction decoding that most RISC designs enjoy. This is naturally done to further increase processing power, and is likely to follow the same development, as will desktop processors.

Programming model
The programming model of most current embedded processors is very similar to their desktop counterparts. This is due to several factors, primarily that most embedded processors are modified versions of desktop processors, typically one or more generations delayed. This is probably due to reuse of very expensive chip-design, but also to facilitate easy development and porting of code and tools to the embedded system. Some varieties exist that have, for instance, no concept similar to a system stack, which typically increase the complexity of programming these system as the expense for simpler hardware design with possible gains in other areas. However, these tend to be highly specialized chips aimed at particular industries or even particular companies. Some of these chips are found in several current cell phones[2], where design decisions reducing the chip size (translates to reduced power consumption) are rated higher than ones easing the life of the programmer.

Conclusion

The majority of embedded processors used are very similar to the desktop processors used a few years ago. They are, in fact, typically based on the same chips with on-board additions for interfacing/communicating with the surrounding world. This makes the process of developing software for such systems less arduous for programmers well versed in desktop programming.

[2] The AMD 29K chip, for instance

For this reason it would seem advantageous to choose embedded processors closely related to successful desktop processors. This idea is, to some extent, supported by the number of units sold, where processors based on the Motorola 68030 processor used in vintage Sun workstations and Macintoshes are very popular (see [MPR]). On the other hand, enough specialized embedded devices exist to warrant developing processors for their specific purposes. So the obvious conclusion would seem to be that for embedded devices whose processing and power requirement can be met by using a modified desktop processor, it would indeed be a good decision to use just such a processor.

Real-time operating system

In this section we will discuss the fundamentals of selecting a real-time operating system (RTOS) for embedded systems, and look at some of the existing products. A surprisingly high number of different products in this category are on the market today, and a lot more are in an experimental state. There are, in fact, lots of good RTOS implementations and while most have a large intersection of features, some have special features that make them different. These commercial implementations come at a cost, however, and most commercial RTOS'es are quite expensive. There seems to be two trenches into which most RTOS'es fall. Either they're commercial products, expensive to buy, covering lots of processors and with a load of tools, or they're give-away freeware, tailored to one or two platforms and has limited tools available. There does not seem to be many products in the middle area.
We've compiled a list of some of the more interesting and widespread products in Appendix A. They were chosen from a technical standpoint, meaning that only those products that offer a solid set of operating system features were admitted to the list. This set of features includes task scheduling, resource management (memory, ports etc), program loading and control and some level of communication support.

Factors involved in the selection process

First of all, no two projects are alike. This means, that while certain factors are very important in one project, they are next to irrelevant in another. We will therefore not attempt to quantify the factors involved in selecting an RTOS, but merely present the ones that occur most often. Most of these are taken from [SRTOS].

Hard vs. Soft real-time
Lots of products claim to be real-time, but not all support hard real-time. Hard real-time means that you can be assured that a specific portion of code will be allowed to run before some fixed amount of time has transpired, regardless of system load. In some systems, this can be very relevant. For instance, you'd like to be sure that the airbag in your car inflates within a few milliseconds of a crash detection. This can usually be achieved, but not guaranteed, by soft real-time, and it's the guarantee that separates the two variants of real-time.

Scheduling algorithm
There are a number of different scheduling algorithms for concurrent/real-time operating systems. We'll not go into detail as to how these work, as any good book on operating systems will do so. Instead we'll make the statement that the scheduling algorithm CAN have a major influence on how your system needs to be designed. Typical ones in today's RTOS'es

include ordinary round robin, priority-queues, dynamic priority queues and rate monotonic scheduling[3].

Price/licensing
As mentioned before, most RTOS'es are either expensive commercial products or open source. Since choosing between the two basic variants can have a major influence on overall project economy, this factor is not to be discarded lightly.

Development tools
To facilitate easy development for embedded systems, it is paramount that good tools for doing so exist. Of course it is possible to write it all in assembly language and just download the binary code directly, but it is not generally a feasible approach. At the very least, you tend to need a cross-compiler, a debugger and perhaps a simulator. Good integration with the RTOS requires extensive sets of well-documented modules, to allow you to choose the pertinent ones for your particular project. Whether these modules are supplied with the RTOS, have been purchased as third part products or have been written by yourself, they can have serious implications in terms of development time.

Processors supported
A critical factor is whether the RTOS can run on the hardware in question. Most currently support the Motorola 8xxx series and the Intel x86 series, but lots of other good processors exist and support for these is not necessarily a given thing.

Communication
Some RTOS'es have built-in support for communicating between embedded devices. Others have no support whatsoever. Most are somewhere between those two extremes, and support some parts of a communications standard, leaving the rest up to the developer. In certain ways this makes good sense, as the actual communication hardware can be very varied. However, most commercial RTOS'es have good support for standard communication protocols and APIs, for instance a TCP/IP stack and/or a sockets interface.

Other factors are naturally important in various projects, but we've listed the ones that are generally perceived as important in the embedded industry, as exemplified by most discussions on [EG3].

Conclusion

When concluding which, if any, of the listed products we prefer, we've chosen not to rate them against some theoretical notion of a 'good operating system'. This is due to the fact that embedded systems are usually heavily constrained by a number of real-world issues that make theoretical considerations less applicable, since these tend to work with ideal situations. Another factor is that most products surveyed have just about the same feature set in terms of core operating system services (task switching, memory handling, device handling etc.), with variations typically limited to whether a specialized scheduling algorithm has been implemented this or that way. Further, the technologically inept products were not admitted to the list in Appendix A, so the 'good operating system' considerations have already been taken into account.

[3] Rate monotonic analysis and scheduling is explained in more detail by [RMA].

We believe that RTEMS is one of the most interesting candidates, and we've summarized our reasons below:

Widely implemented
RTEMS is implemented on a wide range of processors, making it an excellent choice for a cross-platform embedded project.

Price/performance
Among the freeware implementations, RTEMS has the unique position of being ported to most of the currently popular embedded processors. Since it is a proven product used in military devices for years, it is expected to be as robust and error-free as the major commercial products. It has a community of developers supporting it, much the same way Linux does – albeit on a much smaller scale.

Existing CORBA implementation
It was recently announced that omniORB was ported to RTEMS, a major factor in our decision. It may or may not be a good implementation, but at least it is there. If nothing else it can function as a viable comparison tool.

Recommended by industry partner
Our industry partner recommended we use RTEMS, and they are currently using it themselves. This experience is not to be overlooked, as they have many years of experience with embedded systems.

2. Embedded communication hardware

This chapter will examine the typical hardware used for communication between embedded systems. Contrary to the situation in the last chapter concerning processors and operating systems, few types of hardware are used for communication among embedded systems. Focusing on the most widespread, we will look at three different systems of communication between embedded devices. First we will look at serial linking, the simplest communication system in use. Second, we will look at Ethernet connections and finally we will review the CAN communication standard.

What demands must communication in embedded devices meet, and how do they differ from those of ordinary desktop systems? Most demands are like those of desktop systems, and the all-important difference is reliability – many embedded devices must work in very noisy (electromagnetic noise) environments, and that introduces lots of errors on most current communication media, excluding fiber-optics, but currently price inhibits that technology. Usually, the necessary reliability is achieved at the expense of data throughput, as it is normally so that transmitting at lower rates reduces the risk of some transmission errors, and increases the chances of catching the actual errors (the error is spread over fewer bits). Several initiatives have been taken to design the communication system to fit the needs of embedded systems more precisely, and one of the most successful in that respect is CAN, the third subject of this chapter.
Note that not very many embedded systems are, in fact, communicating with other systems. This is probably the reason why the range of used communication systems is as narrow as indeed the case. This is likely to change in the coming years, as it becomes more and more likely that home appliances will have on-board computer systems. This view is shared by [VNU].

Serial linking

The simplest communication system possible, a serial link is basically a single wire running from embedded device to embedded device. This obviously means lower cost of cabling, and it allows the hardware at both ends of the cable to be extremely simple. It is still necessary, however, to implement some form of data arbitration, to bring order to the stream of bits. Although protocols for this purpose do exist, they are not favored in the embedded industry. This is probably caused by the historically low throughput of normal serial links, which have forced developers to make proprietary protocols to maximize throughput for their particular application. Being a dated technology we will discuss it in no further details, but simply conclude that it mainly exists because it was the only possibility in earlier system. This does not mean that current embedded devices do not use serial linking – it is, in fact, still widely used to monitor/control closely coupled devices, but not commonly used to communicate between embedded devices.

Ethernet

The most widespread standard for computer connectivity, the Ethernet is a most successful communication technology. It is a general-purpose specification for connecting a wide variety of computer networks via a multitude of cables. Some years back, it was deemed too

expensive for embedded devices due to its relatively complicated hardware compared to, e.g. serial interface, but as prices have continued to drop it is not uncommon to find Ethernet connectors on more powerful embedded devices. To further support Ethernet connectivity a growing number of embedded processors feature built-in TCP/IP stacks (probably the most popular protocol on Ethernet). This makes porting existing code to embedded devices simpler, and also allows several protocols built on top of TCP/IP to be easily implemented. A modern trend is that embedded devices are controlled by a web-server running on the embedded device, and using that to serve HTML pages that allows to remote user to change settings or read the state of the system. This is greatly simplified by using the standard HTTP transport protocol, TCP/IP. To what extent TCP/IP allowed HTTP or HTTP required TCP/IP is not clear, but it's a given fact that they are both present today. Rather than repeating what others have said on how Ethernet works we will simply refer to [ETH], a standard reference in this respect.

Controller Area Network

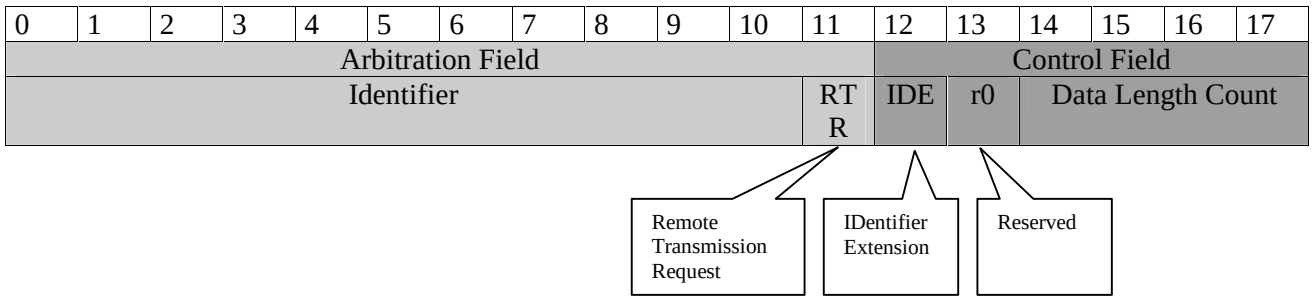
A widely accepted industrial standard, the Controller Area Network (CAN) is used in numerous embedded devices. Major applications include the aerospace and automotive industry (BMW, Rover, Mercedes et. al.). It has been standardized in ISO 11098 and was originally designed in the late 1980's for the automotive industry. It features high resistance to electromagnetic noise and can even operate on partially severed wires. It also features a high level of automatic error detection and correction which, coupled with an efficient scheme for real-time priorities for data transmission, has made is as popular today for embedded applications as is the case.

CAN conform loosely to the 7-layer ISO model, but implements only the Physical Layer, the Data Link Layer, small parts of the Network Layer and the Application Layer. This partial implementation is due to the reduced amount of resources on embedded systems and the fact that the extra facilities in the remaining layers have not been needed so far in embedded systems. The link between the top- and bottom layers is usually handled by dedicated software, designed to meet the needs of a particular embedded system or a particular industry.

CAN normally transmit on standard twisted pair cables (occasionally fiber optics), and typically operates at 500 KBaud (2 MBaud is the current limit, but is rarely used). It uses differential voltage signaling, where the two signal lines (CAN_H and CAN_L – high and low respectively) sits at 2.5 V when the line is idle. A '1'-bit is signaled by CAN_H being higher than CAN_L and a '0'-bit is signaled by CAN_L being higher that CAN_H. This signaling scheme allows CAN to continue operating when one of the signal lines is cut, giving it a high degree of fault tolerance (see [CAN] for details). It is possible to use CAN without this feature, if it is deemed unnecessary.

CAN nodes do not transmit while the bus is busy. If two nodes decide to transmit at the same time, the one with the lowest Identifier (see below for header description) is allowed to transmit. This is done in a bitwise fashion by looking at the Arbitration field (the first part of the header transmitted), and giving preference to the first node transmitting a '0' where the other node transmits a '1'. This makes it possible to make assumptions on the maximum delay a node will experience if Identifiers are used appropriately - typically by assigning lower numbers to more critical messages.

A CAN message packet is very short, consisting of a header, 0-8 bytes of data and a footer. The header is laid out as follows (the numbers indicate the transmission sequence):



The RTR-field specifies whether a message packet contains actual data. If it does, it contains a ‘1’. If not, it contains a ‘0’. This means that among packages with identical Identifiers, those without actual data (the shortest ones) are given preference.
The IDE field indicates whether the message packet conforms to Basic CAN (as described here) or Full CAN (which, in essence, allows 18 additional Identifier-bits to follow the Control Field and mandates that masking of messages should be possible on the individual nodes). A value of ‘1’ indicates a Full CAN message.
The data length Count field combines with the RTR and IDE fields to yield the actual length of the message package, in a non-trivial way described in [CAN].

The actual data is transmitted using 5-bit stuffing. This means that following 5 consecutive identical bits, a bit of the opposite value is inserted (stuffed). This always happens in hardware, and allows receivers to flag an error whenever 6 (or more) consecutive identical bits are received. This is actually a clever algorithm since most transmission errors span several bits in length, and usually just force a certain constant value onto the cable, as described by [AST]. The destuffing also takes place in hardware, so it is a transparent process.

The header consists of a 15 bit CRC-checksum, calculated from a formula described in [CAN], followed by a two bit acknowledge (ACK) field. The ACK field is special in that it consists of an ACK slot (1 bit) and the ACK delimiter. The ACK slot is sent as a ‘0’, and upon successfully receiving the message, receivers overwrite it with a ‘1’. In this way, the sender will know if the message was received at all. This also constitute the sole area where we’ve found the CAN specifications to suffer from problems – the ACK field is not included in the checksum (or verified in any other way). This means that an intermittent error changing the ACK slot to a ‘1’ (and doing nothing else) will erroneously indicate to the sender that the message was received, regardless of whether this was indeed the case. In safety-critical applications the designers of the system should address this problem.
CAN is extremely reliable – according to [CAN] it has a $3{,}7 \times 10^{-11}$ risk of not catching an error, making it a very reliable system indeed.

Conclusion

The right communication system for a system of embedded devices must have the right combination of features, and high among these are reliability, price and familiarity. Another feature necessary in many systems is real-time arbitration of data frames, allowing the ones with the highest priority to suppress others. These demands can be met in many ways, but it is our firm conviction that following standards is a desirable way to go. Several approaches to

this seem feasible, but the ones we believe to be best are CAN for very noisy industrial applications, and Ethernet (or any similar desktop technology) for relatively noise-free environments. One could imagine adapting Ethernet to noisy environments, but since that would involve either lots of checking/correcting errors at higher levels or reducing transmission speed (or both), it would more or less just reduce Ethernet to being an inferior version of CAN. Another problem is adapting Ethernet to handling real-time prioritization of data frames, a problem that we have yet to hear of any proper solution to. Since neither standard Ethernet nor CAN is costly, price is not expected to be the deciding factor between the two.
Since familiarity is also important, it would seem a good strategy to have, for instance, a TCP/IP stack (or similarly familiar technology) for use in communicating. Since this can readily be achieved when using Ethernet, that's an argument for choosing Ethernet over CAN, where implementing TCP/IP is expected to require more work (treated in a later chapter). Not because CAN is faulty, but because the underlying design decision in CAN focuses on other features than TCP/IP usually provides. How the real-time arbitration in CAN could be taken into account in the TCP/IP model, is a question we can not currently answer but would be interested in seeing answered. The later chapter will touch on this too.

3. CANbus higher level protocols

A widely accepted industrial standard, the Controller Area Network (CAN) is used in numerous embedded devices. Major applications include the aerospace and automotive industry (BMW, Rover, and Mercedes to name the most prominent). It has been standardized in ISO 11898 and was originally designed by Robert Bosch GmbH in the late 1980's for the automotive industry. It features high resistance to electromagnetic noise and can even operate on partially severed wires. It also features a high level of automatic error detection and correction which, coupled with an efficient scheme for real-time priorities for data transmission, has made is as popular today for embedded applications as is the case.
This chapter explores higher level protocols for the CAN, and assumes familiarity with the CAN-bus.

What must a CAN-HLP do?

A general consensus in industry seems to be that a CAN-HLP (high level protocol) is a communication protocol built on top of the CAN-bus, which features one or more of the following:

- Large data transfers
- Node identification
- Peer-to-peer communication (may be connection-oriented or not)

Other features may or may not be present in different solutions, but the above items are considered to be crucial ones. The selection of features is inspired by the features normally connected with TCP/IP, but does not extend to the full functionality of this protocol. It is furthermore inspired by the necessary transport for a CORBA GIOP implementation. For that reason the following requirements are, in our context, mandatory (taken from [IIOP]):

- No loss of data
- No reordering of data
- No duplication of data
- Connection oriented with drop detection
- Transport can be viewed as a stream of bytes

This set of features makes it possible to create a communication system as illustrated in the following figure. The CAN-HLP layer is supposed to provide the transport layer in the OSI 7-layer model, where the GIOP and the ORB provide functionality normally related to the application, presentation and session layers.

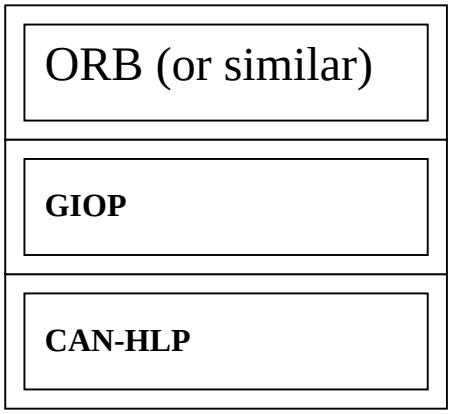


Figure 1: CAN-HLP in relation to a GIOP

Limitations and features of raw CAN

The CAN-bus is, in itself, rather low-level. It features an excellent native error-detection system, but no facility for error correction in any form, be it retransmits or otherwise. Furthermore, its packet numbering system is related to real-time arbitration and packet identification, not receiver or connection identification. Raw CAN features no notion of data streams from peer to peer, and to provide the quality of service normally expected from an OSI transport layer, something akin to TCP/IP socket communication needs to be in place. This implies that communicating nodes should be able to reliably connect to any desired node, knowing only its unique network address, transmit a sequence of bytes and disconnect in an orderly fashion. This includes an orderly disconnect in the instance of communications breakdown, such as a malfunctioning network, sender or receiver. If two-way communication is desired, two channels of communication in opposite direction can simply be established.

Different approaches to achieving this can be (and has been) explored. The focus in the remainder of this paper will be on analyzing the approach one could use in designing/implementing such a system, and a relatively brief breakdown of several existing solutions. It is still an open question to what degree the real-time features of the CAN-bus are, in fact, compatible with a GIOP implementation.

Approach 1: Implement TCP/IP

This approach comes naturally to mind, since IIOP (the most common implementation of the CORBA GIOP specification) runs on top of TCP/IP. Since IIOP implementations already exist in several flavors, this could make it easier to implement that particular element. Furthermore, TCP/IP is a thoroughly tested approach that has proved its worth. This leaves two essential questions, namely:

- Is it a good approach for a CAN-bus?
- Is it a good approach for an embedded system?

The latter question is easily answered with a “yes”. One reason for this is the observation that several, if not numerous, embedded hardware solutions have a TCP/IP stack in firmware (either soft- or hardware). This would indicate that TCP/IP is fairly well suited for an embedded system, but the TCP/IP stack is somewhat complicated and tends to require more resources to be available than more simple solutions do. The very fact that it is widespread makes it a good choice too, since chances are that developers have some previous knowledge of TCP/IP.
The former question, however, is where TCP/IP fails (in our opinion). This is caused primarily by TCP/IP more or less hiding the very useful features of CAN-bus such as real-time arbitration. One could argue that these features could be added to the TCP/IP protocol, but that would effectively make the implementation non-standard, and the argument above about being widespread would become moot. This could imply that a less complex non-standard implementation would be better.

Approach 2: Use a commercial system

This approach has the obvious advantage of freeing us from implementing the protocol ourselves. This also means that we are given much less control over the features that are included or omitted. There are currently three popular choices based on open standards, namely CANopen, DeviceNet™ and SDS™ (see [CHLP] for a more detailed break-down of the three protocols, including references to their definitions). In the following sections the main characteristics of each of these three systems will be listed and discussed.

CANopen

CANopen is an almost exclusively European approach to a higher level protocol for CAN-bus. Of the three protocols examined it is probably the least complicated, but also has a number of limitations. It provides a high degree of control over how message ID’s are used, requires very little overhead in form of network load or processing power, but also falls short on several key issues such as fragmentation and connection-orientation. CANopen is based exclusively on message passing, with messages unable to exceed the CAN-bus’s native 8-byte limit. For that reason alone, substantial extra development is necessary to make use of CANopen. With no facility for node identification beyond that of CAN itself, it is not suited for immediate use as a protocol on which to implement GIOP.

DeviceNet™

DeviceNet™ is a far more ambitious protocol that handles a broad spectrum of the facilities necessary, and as such is an appealing choice. Developed by Allen-Bradley, standardized and monitored by ODVA (Open DeviceNet Vendor Association), it provides node identification, somewhat intelligent use of message ID’s, fragmentation to transmit packages of unlimited length and a good notion of connections. It is, however, a very complicated protocol with, in our opinion, too many bells and whistles. The layout of a standard header is flexible yet complex, and surprisingly efficient for the many varied uses DeviceNet™ is intended for. But since many of the facilities of DeviceNet™ are attempts to make an, albeit somewhat limited, distributed object-oriented system, it provides features that are non-standard and parallel to GIOP, but not sufficiently general to provide the foundation for a good GIOP implementation. The connection system, fragmentation and header layout are very well designed, and are in itself major arguments in favor of choosing DeviceNet™. Since it has been implemented completely in silicon, it places little or no burden on the individual nodes, but a curious

limitation is that of 32 concurrent general-purpose connections per node. This may or may not prove troublesome, but since the implementation is in silicon, it is impossible to work around without substantial work. Given that a CORBA-server may have lots of concurrent request/response connections open at any given time, this limitation can potentially cause severe delays while waiting for the server to free up precious connections.

SDS™

Designed primarily for networks of sensors and actuators, SDS™ (Smart Distributed System) was originally designed by Honeywell Micro Switch. It features an object-oriented hierarchical device model in the OSI Application Layer that allows devices to operate in unison. The SDS™ system uses a header almost similar to that of a subset of DeviceNet™ (the so-called group 2 messages), and uses a very compact way to encode packet function. This is due to the fact that it was designed to control relatively simple sensors and actuators (an example could be a thermostatic controller, trying to maintain a constant temperature in some chemical reaction tank). This means that relatively few commands can be issued to the individual node on the network, and these typically contain little or no data. This meant that the designers of the fragmentation system limited the system to 256 bytes of data, spread over 64 packets of 4 bytes each, which is quite inefficient since half the packet size is “wasted” on header information (compare with a single byte lost per fragment by DeviceNet™). Some of this extra header information is used to provide more extensive acknowledgement than DeviceNet™, so rather than wasting bytes it is merely a question of priorities. The 256 byte fragmented packet length is a serious limitation though, especially considering the many packets necessary to transmit them. One could argue that SDS™ does not have a fragmentation system, since it is so limited that, for all practical purposes, it needs to be augmented with an additional layer. This would not have been a serious problem, if it weren’t for the fact that the fragmentation protocol use as much network bandwidth as is the case. For this reason, it is our opinion that SDS™ is not a good choice as a platform for implementing GIOP. The idea about a limited distributed object model is very good, but the implementation is geared specifically towards sensor/actuator networks, and as such provides little of use for our particular task.

Conclusion

The three reviewed products have quite different approaches to providing a higher level protocol for CAN. CANopen tries to use a minimalistic approach where only the badly needed features are included. This has, in our opinion, lead to the situation where is provides too little to be useful to us. The two commercial systems, DeviceNet™ and SDS™, have taken two different but wide-reaching approaches, with DeviceNet™ being the more general of the two in our view. However they both suffer from being too specialized, with limitations that are perfectly reasonable with the intended uses ([CHLP] elaborates on this) for both protocols in mind, but seems to limit their usability for our particular use. We were impressed with the efficient use of network resources found in DeviceNet™, and the very flexible use of the 11-bit addressing space of a basic CAN-bus. If we were to make a choice exclusively among the three systems, we’d probably decide to use DeviceNet™. We are simply not convinced that it is easier to make a good GIOP implementation on top of DeviceNet™ rather than on top of CAN-bus itself, since much of the fine thinking that went into DeviceNet™ and SDS™ can easily be reused.

Approach 3: Do it all yourself

This approach would enable us to tailor-make a protocol that enables a GIOP implementation to run on a CAN-bus with the minimal overhead possible. It would also allow us to optimize such a protocol to include the very useful real-time features of CAN into the protocol. Since minimal overhead includes both memory footprints and network load, resources that are usually scant in embedded devices, the inherent advantages of this approach looks appealing. The downside is, of course, a more complicated and lengthy process of developing the actual implementation, difficulty in maintaining such a proprietary standard and the risk of omitting relevant features simply because no-one thought of them in the design phase.

Relevant issues in this area include

Error detection

Since CAN-bus has a very high chance of detecting transmission errors in hardware[4], one could argue that this would be an adequate error detection facility. Positive acknowledgements are handled very efficiently in CAN, but it is signaled whenever ANY communicating node receives it correctly. This conflicts with the way peer-to-peer communication works, because it is not relevant that other nodes correctly received the message – only the intended receivers ability to receive has relevance. This problem must be solved.

Fragmentation

Packets in CAN are very short, from 0 to 8 bytes in length. This means that a fragmentation system must be constructed that allows for longer messages to be passed reliably from node to node. This problem must be solved, and it would seem prudent to use a fragmentation mechanism like that found in DeviceNet™.

Node identification

CAN has no notion of identifying nodes. This just wasn't relevant for the problems CAN-bus was originally designed to solve, and therefore this needs to be addressed. Some limited functionality exist in the so-called extended CAN specifications for allowing nodes to set up a mask for filtering messages, effectively giving nodes a sort of address (see [CAN] for details). This problem must be solved, and preferably without requiring extended CAN (considerably more costly per unit in terms of hardware).

Connection-orientation

CAN has no notion of a connection from node to node, since it has no notion of identifying nodes (see preceding paragraph). This must be solved.

Drop detection

A facility that allows either end of a connection to correctly determine that the connection has failed and handle the situation accordingly. CAN has rudimentary support for this, with its signaled error state system (see [CAN] for details). A more comprehensive system is likely to be needed.

[4] CAN-specifications specify that this MUST occur in hardware, and the risk of an undetected transmission error is in the order of $O(n^{-11})$, see [CAN] for details

Design issues

There are several issues that must be kept in mind when designing a system to solve the problems mentioned above. First of all, the short length of CAN packets means that longer messages must be encoded in lots of CAN packets. Since CAN-bus allows for real-time arbitration of packages, this could mean that high-priority packages of substantial length could monopolize the CAN-bus for extended periods of time. This may or may not be viewed as a problem, but it must be addressed somehow. One way could be to use the fragment number as part of the real-time ID of the CAN packet, thus ensuring that lower priority connections could get a fragment through occasionally. Another way could be to simply conclude that this monopolization is the desirable behavior.
The CAN-bus operates at a low speed (typically 500 Kbps). This means that in situations with comparably little simultaneous communication there could be considerable delays involved. This can not be solved without either lowering the need for communication or increasing the speed of the CAN-bus (speeds of 1-2 Mbps have shown themselves to be feasible, but are not yet widespread). Care should be taken in designing the protocol to waste as few bits as possible on overhead such as headers and flags. For that reason, it may be necessary to make slight alterations to the way GIOP packets are normally constructed, primarily in terms of alignment. Since this will take effect on the CAN-bus only, and the CAN-bus is not routable as stated in [CAN], it is, in our opinion, safe to do so – any communication with outside systems would need a gateway in any case, and such a gateway should have no problem dealing with any format changes.
CAN is based exclusively on broadcasts. This means that lowering the processing power needed on the nodes that are not part of a given connection is very important, and should preferably be handled in hardware. This could lead to a demand for extended CAN to be part of the solution, due to its filtering capabilities, or to a silicon-based commercial solution.
Since CAN-bus packets are given ID’s to facilitate real-time arbitration (and, of course, to identify packets), the scheme for handling network addressing could possibly utilize this to allow real-time arbitration of connections between nodes, thus enabling the distributed system to perform some level of system-wide prioritization. Care should be taken in designing any such feature, however – there are many issues and parameters whose effects are not easily discerned, and it remains to be seen if such a feature is, in fact, necessary. Since the real-time CORBA specifications are not currently mature, any such feature would be proprietary.

Conclusion

After reviewing the different approaches given in this chapter, we are left with no really satisfactory conclusion. We see our work as being about distributed object systems, not about transport protocols. So unless we can find compelling reasons for finding otherwise, it is our tentative conclusion that a CAN-bus approach is too much trouble in this respect.

4. COM and runtime type information

This chapter deals with COM (Microsoft's Component Object Model) and how it is possible to obtain information at run-time about the methods offered by the various interfaces supported by object instances. This system, sometimes called 'introspection', is of vital importance when compiling – at runtime – a list of available methods for any given object.
The parts about COM itself will not be exhaustive – an overview will be given to better understand how its particular version of introspection works.

COM

The Component Object Model [COM] grew out of Microsoft's experiments with a larger, more complex, related technology called OLE (Object Linking & Embedding). It was meant to provide it's office tools with the ability to embed "live" objects within one another. The original OLE implementation was terrible, and fortunately Microsoft decided to re-write it from scratch. The new design featured COM as its basic technology with a new version of OLE on top.
This approach has the advantage that the COM design was done from scratch[5], and didn't grow out of an older insufficient design. In this way newer design ideas could be (and were) incorporated. The downside is that it was partially designed to form the basics of OLE, a heritage that still shows up in the most unexpected ways. In this respect is has proven to be counter-productive that COM/OLE has always been a proprietary Microsoft standard – it was designed to solve a problem at the time.
Later Microsoft made the Automation system for "remote-controlling" applications, building this technology on COM and OLE. It gradually turned out that COM was actually a good technology on which to base a number of products and APIs. It supports

- Encapsulation
- Interface inheritance[6]
- Programming-language independence
- Polymorphism

COM has a distributed counterpart called DCOM [DCOM]. It is basically COM augmented with distribution. It is often compared to CORBA [CORBA231], and although the two are used for the many of the same tasks they are very different from a technical perspective (see [SBS] and [AVDO] for a comparison). DCOM is generally perceived to be difficult to administer and setup, but easy to use for programmers.

COM and Interfaces

The basic idea behind COM objects is that they offer a set of methods (no attributes) packaged in an **interface**. An interface is a collection of related methods that the designer/programmer decided were coupled logically. Objects may support more than one

[5] Although it was designed to provide the foundation for OLE, its implementation was done from scratch
[6] This differs somewhat from class inheritance, as we normally perceive it. The following section should shed some light on the differences.

interface, and while a single object reference can point to only one supported interface at any given time, a copy of the object reference can be made and requested to point to a different interface for the same object. Any useful object usually support at least a handful of interfaces, and it is rarely, if ever, the case that only one is supported. Implementationwise, an interface is little more than a vtable (a table of pointers to virtual methods) augmented with reference counting and the ability to change to another interface when requested to do so through a (small) fixed set of methods.
A COM object is often referred to as an "interface instance" – it is not necessarily an object at all (some are written in plain C or Visual Basic), and the only structure instantiated is the interface. Since interfaces do not facilitate data storage in the form of attributes – in order to provide total encapsulation – there is no such thing as an "object" that can be pointed to or instantiated. In this text we shall follow the norm and use "COM object" and "interface instance" interchangeably.

IUnknown

The IUnknown[7] interface is the most basic of all interfaces, and MUST be supported by any and all COM objects. It has three functions, namely

- AddRef
  Increments the reference-count by one

- Release
  Decreases the reference-count by one, and marks the object for removal if it reaches zero.

- QueryInterface
  Returns a pointer to an interface (on the same object) that supports the queried interface.

Note that the IUnknown interface is special in that it must also be the first three methods of any interface, since the reference counting and interface querying must be universally available. This means that the vtable for any interface must have, as its first three entries, AddRef, Release and QueryInterface. This is commonly achieved by having interfaces as derivatives of the IUnknown interface, but this is not mandatory.

IDispatch

The dispatch interface is probably the most widely used interface[8] (except, of course, IUnknown). It supports 'very late binding', where the method to call is given as its name (a string) along with an array of parameters, and the object implementation then decides which member function (if any) to actually call. In this respect COM handles member function calling somewhat like SmallTalk (see [SMT]) does. IDispatch also supplies the methods by which the runtime method information is available. Since most of the available tools for producing COM objects are able to make IDispatch objects with little or no effort (most actually default to this)[9], this makes IDispatch a good choice for providing runtime type information since it

[7] Interfaces are normally named in this fashion: A capital i followed by a (hopefully) descriptive name.
[8] All visual controls are IDispatch, as are practically all controls for scripting purposes, which make up the vast majority of available controls today.
[9] Visual Basic and Delphi are very common examples of this

- Is widespread
- Is a de facto standard
- Requires little – usually no – effort from the programmer

Type information through IDispatch

The process of getting access to type information is quite straightforward in COM. It involves little more than calling the QueryInterface method on the object and requesting the IDispatch interface. Since almost all IDispatch objects are used as such, this step can normally be omitted. The IDispatch interface has a method called GetTypeInfo, which returns a pointer to an ITypeInfo instance. This somewhat complicated object holds all the information necessary to analyze the runtime type information for the original object.
Since interfaces specify no data attributes, all members are functions. There are about a handful of different types of member functions, and while the differences among them are important from an implementation point of view, the information provided about them is essentially the same. For each interface, there is simply a list of its supported methods, each returned as its own structure. This structure primarily contains a way to obtain the member functions name, the type returned by the member and an array of parameter descriptions. Note that it is quite possible to obtain a dispatch object with no corresponding type information – in this case, the GetTypeInfo call simply returns NULL. Note also that it is quite possible to get a ITypeInfo instance by other means than through IDispatch[10].

Calling through IDispatch

The way to call methods through the dispatch interface seems straightforward – you identify the method to call, the parameters for it, and execute IDispatch::Invoke. In reality, it is almost that simple.

- It is not possible to simply give the name of the method to invoke – a so-called dispatch ID must be given instead. The translation can be done by way of IDispatch::GetIDsOfNames or ITypeInfo::GetIDsOfNames.
- All parameters must be given as Variants, which is the OLE way of storing all types of data in a uniform way. Any Variant has two fields – its type (called **vt**) and its value (named according to type).
- Arguments can be named too – that is, they can be specified in any order and their names given along with their value.
- It must be decided what locale[11] to use.
- The return value also comes as a Variant.

[10] Any COM interface can have type information attached, and this is strongly recommended by Microsoft to allow editors to provide the programmer with type information during coding, and to assist the compiler in generating more efficient code.
[11] A locale is a code used to specify what language settings the method should execute under. If the method handles locale-specific information such as dates, numbers and currency this locale must be set accordingly.

When these things have been handled, it is possible to call Invoke - sample code (taken from Microsoft Developer Network, see [MSDN]) could look like this in C (declarations omitted for brevity):

```
szMember = "ShowMe";
hresult =  pdisp->GetIDsOfNames(IID_NULL, &szMember, 1,
           LOCALE_USER_DEFAULT, &dispid);
dispparams.rgvarg[0].vt = VT_I2;
dispparams.rgvarg[0].ival = 1;
dispparams.cArgs = 1;
dispparams.cNamedArgs = 0;
hresult =  pdisp->invoke(dispid, IID_NULL, LOCALE_USER_DEFAULT,
           DISPATCH_METHOD, &dispparams, pVarResult, pExcepInfo,
           puArgErr);
```

This shows how to call the method named `ShowMe` with a single 16-bit integer parameter, on the interface instance pointed to be `pdisp`, assumed to be an IDispatch pointer. The return value is stored in the Variant pointed to by `pVarResult`, any exception arising from the execution of the method is stored in the Exception pointed to by `pExcepInfo` and the index to the first parameter (if any) causing the error is stored in the unsigned integer pointed to by `puArgErr`. Note that the Exception referenced has no connection to language-specific exceptions as seen in C++ or Java.

The work involved in getting this done is relatively small, but from experience we know that it is annoying to constantly convert between Variants and native datatypes. Unfortunately, this is usually necessary because the Variants are much harder to manipulate than native datatypes in most languages.

5. Distributed Object Systems

Distributed systems are inherently more difficult to design and build than their local counterparts. The reason is found in the requirements and characteristics particular to distributed applications [NDC].

Practical distribution issues

Compared with single process applications, multi process applications introduce a new class of problems that has to be addressed: inter-process communication, cross-process memory references, unavailable server processes etc. When distribution is introduced the number of problems increase; distributed systems has to deal with network communication failures, representation of data across different architectures etc. Important practical considerations include:

- Latency of remote invocation and communication
- Representation of remote object references
- Life cycle management
- Memory management

Invocation latency

Compared with inter-process and cross-process method invocations, remote object invocations incur substantially greater overhead. A method invocation within the same address space can often be reduced to a single lookup in a *virtual function table*, which is very efficient [STROUSTRUP, COM]. Invocations across local process spaces require simple *marshaling* of parameters and must treat standard types differently than memory pointers. Although some extra work must be done, this can still be achieved quite efficiently by using shared memory or local named pipes [STEVENS].

The picture changes dramatically when an invocation must travel across machine boundaries. Parameters must be marshaled, but since the client and server can reside on different architectures the marshaling must package the parameters in a common format. The marshaled invocation must now travel from the client to the server, which unmarshals the invocation, does the computation and then repeats the process in the opposite direction to the client.

Call latency combined with the marshaling rate gives an idea of the performance of a distributed architecture. The call latency is simply the minimum cost of sending any message whereas marshaling rate is the speed that data can be sent across the network by. According to [ADVCORB] the numbers for a CORBA ORB running on a contemporary machine[12] can achieve call dispatch times between 0.5 msec and 5 msec; put another way the call rate is between 200 and 2000 invocations per second. The Marshaling can be in the 200 KB/sec to 800 KB/sec range. Local method invocations are measured in the millions, and local memory

[12] A typical UNIX workstation anno 1999 equipped with a 10 Mbit ethernet card

bandwidth is measured in Mbps or even Gbps. Without regard to the actual numbers it should be evident that the overhead when invoking methods on remote objects is several orders of magnitude larger than local invocations.

Accessor methods (e.g. setName()/getName()) are often used extensively in object oriented applications, but as should be evident from the above discussion excessive use can lead to poor performance because each invocation of an accessor must travel across the network. It is obvious that this calls for careful thought when designing an application that will use remote method invocation. The design of the client as well as the interface and implementation of the remote object is affected by latency considerations and can have a profound impact on performance.

Life cycle management

Dealing with objects is part of a topic known as *object life cycle*. The central issues to life cycle management are

- Object creation and destruction
- Object activation and deactivation
- Object cloning
- Object migration

Since remote objects in general cannot be created remotely by calling their respective constructors, object systems often rely on the factories to create objects (i.e. based on the factory pattern [PATTERNS]). That is, invoking methods on a factory object creates the desired object.

While factories typically create objects in distributed object systems, destruction is commonly an operation on the object itself. The rationale behind this distinction is inherent in the distributed nature of such systems. In a distributed application where object references are passed between a number of clients, the clients will eventually want to inform the object that it is no longer needed. Since the client cannot be guaranteed to have any knowledge of how the object is obtained, the client must either obtain that information somewhere or ask the object reference itself to dispose of it self. The chapter on distributed garbage collection provides a more in-depth treatment of this subject.

Distributed object-based applications share a common characteristic - they need to support a large number of objects with a relatively long lifetime. A mechanism is needed that does not require every object to be running and thus consuming system resources. Some way of restoring object references in case client and server have been disconnected (e.g. system failures) is also required.

Passive objects, which do not use system resources, and active objects, which do, should be distinguished. An active object is associated with a process and passive objects are not associated. Changing an object from a passive to an active state is called activation. This may involve creating a process, loading the implementation and restoring state from a persistent store. Conversely changing the state of an object from active to passive is called deactivation or passivation.

There are several aspects to activation not treated here. A general treatment of the subject can be found in [WOLLRATH95]. Among those are;

- When are objects activated (e.g. eager or lazy activation)?
- How are objects activated (e.g. per-client, per-request or persistent activation)?

The need to copy or clone an object often arises as part of application development. Creating identical object copies corresponds to polymorphically creating stateful object without passing the initial state for the new object as parameters. Various programming language construct treat this special case of object creation in different ways, e.g. copy constructors in C++ [STROUSTRUP] and the virtual member function *clone* in Java. Even though this appears to be a straightforward problem, the issues involved have parallels to cloning in real life: should the object and its clone be considered identical, can they handle the same requests etc. A common convention is to treat the object and its clone as different objects leaving it to the application logic to resolve these questions.

Moving or migrating objects from one location to another faces the same kind of issues as do object cloning: what happens if communication failures occur while the object is in transit, what happens to existing object references etc. If location transparency is one of the design goals for an object model running on heterogeneous platforms, then allowing clients to move a remote object clearly violates this goal. Not only must the client know where the object should move to, but also some level of knowledge about the destination platform is required to ensure that the object can continue running at the destination server.
Secondly the problems involved in moving objects in a heterogeneous environment has no simple solution. Nevertheless object migration is often an important part of the infrastructure of distributed object architectures, where it is used to achieve load balancing and fault tolerance. Some distributed systems specifically support moving objects as an integral part of their programming model [ARA, EMERALD, JAVA]. A common trait among those is that the underlying platforms are homogenous, i.e. platform specific issues are hidden.

Memory management

Memory management or garbage collection in distributed systems poses a number of problems caused by distribution. This is the topic of the following chapter on distributed garbage collection.

Generic distribution issues

These practical problem leads to a more generic class of problems represented by:

- Partial failure
- Distribution transparency

As mentioned above distributed applications must deal with additional sources of failures, which can lead to situations where an application is only partially running, i.e. some remote object partly responsible for a systems functionality may not respond.

A common goal among distributed object systems is to provide an abstraction that will allow the programmer to treat local and remote objects in much the same way. Although complete distribution transparency is appealing in theory, practical considerations often require some level of knowledge about the nature of the objects involved computations. Not only must the programmer be able to handle errors relating to distribution; some level of knowledge about the location of objects is also needed to make a robust and scalable application. Latency considerations play a particular role when implementing an efficient distributed application.

All of these problems combined are among the challenges facing the design and implementation of distributed object systems

Distributed object models

Contemporary distributed object models used for building applications today have a lot in common [AVDO]. Among the more important common characteristics are:

- Object references are used to access the functionality of remote objects
- Interfaces are used to represent the logical functionality of remote objects
- Support for a synchronous programming model[13] is supported(request/response)
- Facilities for activation and deactivation

Beyond these commonalties, one will find the discrepancies illustrating the different backgrounds and philosophies behind their design.

Distributed Component Object Model (DCOM)

DCOM has its roots in Object Linking and Embedding (OLE) a technology developed by Microsoft to facilitate reuse of document centric applications, e.g. using a spreadsheet within a word processor. It soon became apparent the ideas behind reusing applications could be generalized to smaller units than applications, i.e. components. The Component Object Model (COM) is a binary specification that initially only dealt with local objects both inter-process and out-of-process. Distribution was added as a natural extension to COM. This heritage means that DCOM is often considered a "brick and mortar" technology useful for building application architecture and less often a suitable domain modeling abstraction. DCOM nevertheless possesses many of the same characteristics that other distributed architectures do.

COM both encourage the use of factory objects for object creation and provides an interface *IClassFactory* that provides a single method (*CreateInstance*) for creating object instances. *CreateInstance* allows the programmer to create an instance of a given class and get a pointer to a specific interface. Unlike the constructors typically found in todays object oriented programming languages *CreateInstance* does not accept initialization parameters, thus leaving the created object stateless or in a default initial state. Further initialization must be done using ordinary method invocations, which in a distributed system introduces additional network traffic compared with parameterized object creation.

[13] Not only is a synchronous programming model supported it is in fact often required

```
interface FactoryFinder {
     Factories find_factories(in Key
factory_key)
        raises(NoFactory);
};

interface GenericFactory {
     boolean supports(in Key k);
     Object create_object(
          in Key k,
          in Criteria the_criteria)
```

In non-trivial COM based systems other objects are acting as specialized factories, with the ability to encapsulate the creation and proper initialization of objects, manage object creation.

DCOM is an extension of the programming model used by COM to function across a network. Among the additions is support for location transparency, better threading models, security and administration facilities. There are no fundamental changes to the programming model (see the chapter on COM and runtime type information)

Distribution is achieved through DCOM Object RPC (ORPC) based on Distributed Computing Environment RPC (DCE RPC) [DCEDEV]. RPC is extended with an object reference datatype and the Service Control Manager (SCM) responsible for activating servers is able to communicate with remote SCM's to enable activation of remote objects. When a client request a factory for a remote object the local SCM contacts the remote machine's SCM, which in turn locates and activates the server and returns a RPC connection to the requested object factory. This allows the client to proceed and create object instances exactly as in a non-distributed scenario.

DCOM provides some support for activation and deactivation through the Running Object Table (ROT), without the ability to persist the state of the object.

The latest addition to the COM family is COM+. COM+ integrates DCOM with the Microsoft Transaction Server (MTS) providing better facilities for scalability and management along with new features such as asynchronous method invocations [UNDCOMP].

Common Object Request Broker Architecture (CORBA)

CORBA relies on factory objects to create objects but unlike DCOM does not require a factory object to support a particular interface for object creation. In fact there is no requirement in CORBA that factories should create instances. The OMG Life Cycle Service (*CosLifeCycle*) specification specifically recommends using the factory pattern to create objects. Unlike other CORBA services the Life Cycle Service is not implemented by an ORB supplier and used by clients. The Life Cycle Service provides design and implementation idioms in the form of design patterns and interfaces that optionally can be used to implement life cycle management of objects.

The most important interfaces defined by the Life Cycle Service are *FactoryFinder* and *GenericFactory*.

Interfaces FactoryFinder and GenericFactory

FactoryFinder provides a simple way to locate suitable factories for a given object key[14]. Since the inception of OMG Life Cycle Service new services has been added to the CORBAServices specification [CORBAServices]. Specifically the OMG Trading Service offers a strong and flexible mechanism for discovering objects, and is able to function as a more general factory locator.

The *GenericFactory* interface is the CORBA equivalent of the COM interface *IClassFactory*, with one important additional feature: The ability to provide initialization parameters (*Criteria*) to the instantiation. Initialization parameters are passed as a sequence of name-value-pairs, i.e. a string and a CORBA *any* value. Passing the initialisation parameters in a generic way using the CORBA data type *any* trades static compile-time safety for dynamic run-time safety. The OMG Trading Service provides interface that are more powerful and flexible than a simple generic factory, which does not impact type safety in the same way as *GenericFactory*.

CORBA provides extensive mechanisms for handling activation through the Portable Object Adapter (POA) policies and the Implementation Repository. The POA activates servants responsible for handling request to specific objects. It is beyond the scope of this overview to cover these, suffice to say that they cover a broad spectrum of activation policies. For more details refer to [CORBA231].

CORBA supports the notion of a location transparent object reference. Although information about physical locations, such as IP addresses, can be gleaned from object references, this not an intended use. In fact the CORBA object model has no concept of location.

CosLifeCycle LifeCycleObject

```
interface LifeCycleObject {
     LifeCycleObject copy(in FactoryFinder
there,
          in Criteria the_criteria)
        raises(NoFactory, NotCopyable,
InvalidCriteria,
          CannotMeetCriteria);
     void move(in FactoryFinder there,
          in Criteria the_criteria)
        raises(NoFactory, NotMovable,
```

The Life Cycle Service defines an additional interface that could be implemented by objects needing some form of life cycle management. This interface supports the basic lifecycle related operations copy, move and remove.

The interesting member function of this interface is move: what is the intention of a move operation in an object model that has no concept of location? It is a dichotomy that the design of a lifecycle related operation under CORBA requires knowledge about location; something that is unsupported by the object model. Even if we do not consider the move operation as migrating an object between locations, but instead as a move between "object repositories", the client would still need to know if the object to be moved can reside on the target platform.

[14] An object key in this context is a *Naming::Name*, as defined by the CORBA Naming Service (CosNaming)

Another contrast with the design of CORBA is the concept of protocol transparency: what happens if an object is instructed by a client to move to another server that does not support any protocols known to the client. This will render the object invisible to the client (see the chapter on Object Visibility for more details). To remedy this situation, some kind of protocol bridge must be introduced or else the concept of protocol transparency will be violated.

The relevance of the move operation is probably most suitable for administrative tasks by the ORB, such as load balancing, and not as a general part of the object model exposed to clients.

Other distributed object systems

Alternatives to distributed object models covered above exist in abundance. They basically fall into two categories:

1. Heterogeneous distributed object models that are platform independent
2. Homogeneous distributed object models that define a platform wherein objects must reside

CORBA and DCOM belong to the fist category along with distributed object models such as IBM's Distributed System Object Model (DSOM). In homogenous distributed object models the objects relies on facilities found in the run-time environment. Included in this category are Emerald [EMERALD], ARA [ARA], Distributed Beta [DISTBETA] and Java RMI[15] [JAVARMI]. Because of the homogeneity provided by the run-time environment, these object models often provide facilities to move objects between applications.

In Summary

To be able to support interoperability between different distributed object systems we must support a sensible set of features found across different object architectures. The industry standard distributed object models, CORBA, DCOM and Java, expose their remote objects and the functionality of related services (e.g. factories and naming services) by way of object references and interfaces defining the functionality.

It is evident that in order to be able to properly take advantage of interoperability between these kind of object architectures, interfaces and object references must therefore be supported. It is our contention that beside these features little else is required to provide a general interoperability mechanism.

[15] CORBA IIOP can be supported by Java RMI by sacrificing call by value semantics, i.e. moving objects between servers. This makes categorizing Java RMI harder

6. Distributed Garbage Collection

Garbage collection is a well-known memory management method, used extensively in most new programming languages. It removes the burden of handling object destruction manually at the expense of fine-grained control. Not all local garbage collection algorithms can be adapted to a distributed object model.

In theory the problem is quite simple: distributed objects should remain alive as long as they are referenced by clients or root objects and collected to reclaim resources when they become unreferenced [PLAINFOSS]. Several issues make this problem harder to solve in practice:

- Since distributed objects and references can be created, destroyed and even migrated dynamically across the network, deciding when an object is unreferenced is non-trivial.
- Servers and clients may crash or otherwise become unavailable during garbage collecting operations
- Network problems can cause loss or even duplication of messages between clients and servers
- Loss of communication can result from network partitioning
- Distributed systems are often decentralized to avoid bottlenecks and single points of failure

An excellent overview of various distributed garbage collection strategies can be found in [PLAINFOSS] including a taxonomy of popular distributed garbage collection techniques.

Summary of garbage collection techniques

Distributed Reference Counting

A straightforward adaptation of standard reference counting techniques, i.e. implementing a count on every remote object, is vulnerable to lost or duplicated messages. And the sequence in which reference counting messages arrives is vital: If a decrement message is sent after an increment message, but the decrement message arrives first (race condition), then this may cause the garbage collector to reclaim the remote object prematurely. Lost decrement messages can lead to rapid exhaustion of system resources, because remote objects, that did not receive a decrement message before becoming unreferenced, would continue to consume resources. Reasons for message problems are mainly client or network failures.

Distributed reference counting techniques are often augmented with techniques such as pinging [DCOM] and leases [RMISPEC] to remedy the message problems. Other improvements include optimisations in network roundtrips, since sending a message for every local duplication of the object reference is a considerable overhead.

Beside the problems specific to distribution, reference counting suffers from the inability to reclaim cycles of objects. One advantage of distributed reference counting is scalability, since no central coordinating mechanism is required for it to function.

Distributed Tracing

The common approach to extending the tracing based garbage collection to distributed environments consists of local tracing garbage collectors coordinated by a global tracing garbage collector. Local garbage collectors mark reachable objects and await marking messages from other collectors with remote references to its objects.

This makes the global collector sensitive to message race conditions regarding the sequence of coordinating messages the global collector. Another problem is the synchronization required between the global and local collectors, which creates a single point of failure and introduces scalability issues.

Other Techniques

Distributed garbage collection is the focus of extensive research with the aim of overcoming the limitations of well-known local garbage collection strategies. Among the different approaches to garbage collection in distributed object models are:

- Hybrid cycling techniques - a combination of reference counting and low frequency tracing to handle cycles of unreferenced objects
- Trial deletion - reference counting with heuristic guesses of references that form unreferenced cycles and thus would be candidates for deletion

Other techniques exist, but current distributed object models in general employ a variation of reference counting [ADVDGC, GCINTER].

Distributed Garbage Collection in Java RMI

The design of a distributed object model for Java was inspired by the Network Objects of MODULA-3 [MOD3NO] and included people from the original design of CORBA[16]. One of the foremost requirements to the design of distributed objects in Java [WOLLRATH96] was that it must fit well into the Java programming model; more specifically it should be natural (language integration), simple (ease of use) and support:

- Seamless remote invocation of distributed Java objects including activation of persisted objects
- Garbage collection of remote objects
- Differences between the local and distributed object models should not be hidden

These requirements are well in sync with [NDC].

Garbage collection strategy

The algorithm for distributed garbage collection used in Network Objects MODULA-3 [MOD3NOGC] has inspired the one used in Java RMI [RMISPEC]. It is based on a reference

[16] Jim Waldo

counting strategy integrated with the garbage collection mechanism used for local objects in the Java virtual machine.

```
package java.rmi.dgc;
import java.rmi.server.ObjID;

public interface DGC extends java.rmi.Remote {
   Lease dirty(ObjID[] ids, long sequenceNum,
           Lease lease)
     throws java.rmi.RemoteException;
   void clean(ObjID[] ids, long seqNum, VMID
vmid,
```

The Java distributed garbage collector interface, DGC

Once a reference to a remote object enters a virtual machine for the first time, a message is sent to the object telling it that it has been referenced by the virtual machine. What actually happens is that the client requests a time-limited lease on a remote object reference, by calling the dirty method on the remote objects DGC interface with a parameter specifying the duration of the lease. The client is not guaranteed to get a lease for the requested duration; the lease may be granted for a shorter duration.
References to the remote object, passed around within the same virtual machine, rely on the reference counting and thus garbage collection of the local virtual machine. When the last reference to the remote object becomes unreferenced, the virtual machine sends a message to the remote object server informing that the object is no longer referenced by this virtual machine (by calling the clean method on the DGC interface). This reduces garbage collection related network traffic to a message when an object is entering and leaving a virtual machine, unless the duration of the lease is exceeded in which case the lease must be renewed with a call to the dirty method.

When a client is crashed or has lost its network connection object references should be invalidated. Remote objects may as a consequence be garbage collected, even though a client holds a reference to it. As a result of this, the Java RMI specification does not guarantee that a remote reference points to a live object. Any remote invocation may throw a RemoteException. The distribution model of Java does therefore not enforce complete distribution transparency [NDC].

For a detailed treatment of garbage collection in Java RMI refer to [JAVARMI]. The lease based garbage collection strategy used in RMI is central to life-cycle management in JINI; a Java based infrastructure for making services available in distributed environments [JINI].

DCOM and Distributed Garbage Collection

COM provides support for control of an objects lifetime through the common interface IUnknown. IUnknown, which every COM object must implement, contains two methods AddRef and Release used to implement a reference counting scheme. According to the reference counting rules of COM [COM] reference counts are kept per interface pointer and not per object.

Application programmers will usually make frequent calls to the reference counting methods, either explicitly or through some abstraction (e.g. smart pointers), e.g. whenever an interface pointer is passed along its reference count should be incremented. The overhead of calling the reference counting methods must therefore be small compared to the overall execution time. In traditional COM this requirement is satisfied, since local method invocations are very efficient [INCOM].

Extending the COM model to distributed environments obviously influences the way resources allocated to objects is reclaimed. DCOM [INDCOM] is addressing two primary issues of generalizing the standard COM reference counting scheme to a distributed scenario: Reducing network communication and dealing with communication failures.

Reducing network traffic

A new equivalent to the basic IUnknown interface, IRemUnknown[17], is introduced. The purpose of IRemUnknown is to minimize network round-trips, resulting in a reduction of network communication. This is achieved by a set of interface methods capable of performing operations corresponding to a number of IUnknown operations in a single call. RemQueryInterface method can request several interface pointers in a single call. RemAddRef and RemRelease can increment and decrement the reference count of an object by an arbitrary number (DCOM for Windows typically request five references [DCOMARCH]).

The COM apartment where the server object resides automatically provides an IRemUnknown implementation. Clients will use IRemUnknown as a replacement of IUnknown, i.e. IUnknown is never remoted, instead calls on IRemUnknown results in local calls to QueryInterface, AddRef and Release. The IRemUnknown extension thus provides greater network efficiency compared with IUnknown.

Communication failures

As we previously discussed one of the primary obstacles to distributed garbage collection is dealing with communication failures. The DCOM solution to this class of problems is to introduce a “pinging” mechanism. Each remoted object has an associated pingPeriod time value and a numPingsToTimeOut, which multiplied gives the “ping period”[18]. If a client fails to ping a server object within the specified “ping period” interface references to from the client to this server are considered expired. This allows the server to garbage collect the object based on local knowledge (this scheme is similar to Java RMI). If garbage collection is deferred somehow and the server receives a ping from a lost client then DCOM is permitted to reactivate the remote references.
The traditional COM programming model is synchronous and consequently vulnerable to environments with fragile networks and high latencies such as the Internet. In the most recent version of DCOM an asynchronous programming model is introduced, which makes distributed garbage collection issues more difficult to handle.

[17] A derived interface, IRemUnknown2, allows the result of interface queries to be any marshaled data.
[18] DCOM for Windows uses fixed values pingPeriod = 2 minutes and numPingsToTimeOut = 3

The DCOM approach to distributed garbage collection is a pragmatic one. In essence DCOM builds on the ideas underlying COM, by identifying problematic distribution issues and augmenting COM with new facilities. In our opinion the result is just that: a version of COM with support for distribution. Compared with similar architectures, most notably CORBA and RMI, DCOM falls short in features and elegance found in systems designed from inception with distribution in mind.

An overview of the DCOM architecture is provided in [DCOMARCH] and a detailed treatment of DCOM and the underlying network protocol refer to [DCOM].

CORBA and Garbage Collection

In a CORBA context, garbage collection refers to reclaiming resources occupied by objects, not the object itself. To see why this distinction is made consider a case where an object representing a person is becoming unreferenced. Does this mean that all resources including the persistent state of an object should be collected? The implied meaning of garbage collection for transient objects is self-evident, whereas the meaning of garbage collecting long-lived persistent objects is not immediately apparent: Shall resources occupied by the objects persistent state be reclaimed along with resources occupied by the servant of the object?

For all its facilities and features CORBA newcomers are often surprised to learn that there are no provisions for automatic garbage collection in the specifications. Ensuring that resource utilization is kept reasonable is not a service provided by a CORBA ORB. It is up to the application developer to make sure that resources are reclaimed in an orderly fashion. The most common approaches are based on the evictor pattern: A servant manager is used to instantiate servants, making sure that the number of active servers is within some limit. If the limit is reached the servant manager will evict an existing servant[19] before instantiating a new servant to handle a request.

The designers of CORBA have been unwilling to make compromises that would allow a simple garbage collecting strategy into the specification that does not answer the fundamental issues what object destruction means and how to determine likely candidates. Until further research clarifies these questions for general-purpose object models like CORBA, garbage collection is unlikely to become a part of the CORBA specification.

Choosing a garbage collection strategy for general interoperability

In lieu of the later discussion on requirements to a general interoperability mechanism, most notably that it should be non-intrusive to existing systems, we believe that a lease-based strategy similar to Java RMI is most suitable. Compared with other strategies the characteristics in terms of message failure and overall distributed garbage collection capabilities are satisfactory [PLAINFOSS, BIRRELL116].

The role of the garbage collection scheme in a general interoperability mechanism is to bridge the various schemes found in common distributed object architectures of today, without

[19] Usually by employing a least recent used (LRU) of least frequently used (LFU) strategy

requiring any changes to the systems making objects remotely accessible via this interoperability mechanism. Lease-based garbage collection provides an attractive tradeoff between simplicity and flexibility. Pure reference counting schemes[20] trivially maps to a leasing scheme, since requesting a reference is equivalent to obtaining an infinite lease. Only granting leases corresponding to the ping period can approximate pinging mechanisms as used in e.g. DCOM.

One of the driving goals for the interoperability mechanism proposed in this thesis, is that embedded systems should be able to participate in a distributed environment of heterogeneous systems. Some situations occur more frequently in embedded systems and must be taken into account.

Error prone communication

Consider the case where a manufacturing company would like a production control application consisting of an existing production planning application and embedded applications controlling the machines running on the production line.

The production planning part of the system is located in a conventional client/server environment, where any kind of communication errors is considered fatal. In contrast to this, the machine control applications are located in a noisy embedded environment[21], where communications errors occur frequently without being fatal.

If the embedded applications have acquired a reference to an object from the production planning application (e.g. a production scheduler), what should happen if communication fails intermittently and reference counting messages (increment or decrement reference count) were out of band or lost? Simple reference counting would lead to uncollected objects (lost decrement message) or prematurely collected objects (lost increment or out of band messages).

Clearly this is not optimal. One could argue that the problem of prematurely collected objects only affects the client of a remote object and thus can be dealt with by that user (similar to non-existing objects in RMI). Uncollected objects are an altogether different issue, since there must be some assurance to systems that expose objects that they will not be rendered unusable by the overhead of uncollected objects. If no such assurance is available this will severely limit the types of systems that can be expected to expose objects. Adding leases to a reference counting scheme remedies this problem by ensuring that objects eventually are collected and at the same time lessens the impact by temporary communication failures on the client.

Disconnected objects

When a client application has obtained a remote object reference, the client may temporarily be unavailable, intentionally or by accident. Roaming wireless devices are good examples. When moving between different transceivers the devices may become unreachable for a period of time, like moving with a cellular phone while being connected. By using time based

[20] E.g. as used by CORBA to garbage collect client side stubs
[21] One of the scenarios CANBUS based systems are well suited for

leases systems can cater for this scenario without making itself vulnerable to clients becoming permanently unreachable.

Disadvantages of leases

In situations where only short-term leases can be granted (e.g. because of locking issues), clients will have to renew the leases frequently. This leads to an excessive amount of network traffic to ensure that the leases are renewed and valid. Other schemes exist that are more appropriate for this scenario than leases, some of which require bi-directional communication in the form of remote object to client notifications [OOSHVAR].

Compared with simple reference counting the client is not only responsible for requesting and releasing object references, but must do some additional bookkeeping to make sure references are renewed and therefore valid.

Part Two: Interoperability issues

7. Achieving cross-architecture object invocations – how is that possible?
8. Object Architecture independence and transparency
9. Integrating Object Architectures

7. Achieving cross-architecture object invocations – how is that possible?

When trying to perform invocations between two different object system architectures, a new set of problems arise. How can an object in one architecture hold a reference to an object in another architecture, and expect to be able to invoke methods across the architecture boundary?

Practically all the work on distributed object systems that we've encountered focus on a single architecture, and seems to take for granted that there is no need to go beyond that architecture. Why this limitation? A lot of the work that goes into distributed object system architectures is intended to make it function across heterogeneous hardware and operating systems. A good set of goals, for sure. We suggest that making systems work across heterogeneous object system architectures is an equally worthwhile undertaking. In an increasingly networked world, it is to be expected that larger and larger networks of distributed objects will come to be. As is the case for just about any networking product category (browsers, ftp clients, e-mail programs, web servers etc.) there will probably be an array of competing products and specifications. Arguing whether this is good or bad is beyond the scope of this text, but it is the view of the authors that something along those lines will happen eventually. This also means that no single object system architecture can be expected to dominate completely, but that some interfacing between different architectures will need to take place. Looking at the major contenders today (DCOM and CORBA), one could argue that interoperability with other object system architectures is handled in hindsight – if at all[22]. SOAP (see [SOAP]) has some interesting properties with regards to acting as "glue" between object system architectures, but it is not quite object-oriented in the classical sense of the word since it does not maintain state between calls and has no notion of object identifier or references – it is in our view essentially a procedural RPC protocol, albeit a clever one[23].

But what does it take for an object to have the ability to invoke an object in another object system architecture? Some way of transforming invocations to the foreign architecture must be available, or some way for the object servers to support several object system architectures simultaneously. The latter solution looks promising, allowing clients to call in any architecture they desire, and have the server translate to some internal representation which is then used for invoking the actual method. The major downside is that all servers must have a system installed to handle the translation, and having that would – in our opinion – make it much harder for new object system architectures to come into being, since potentially all reachable servers need to be able to handle invocations in that architectures, a maintenance nightmare on a global scale. Since it is unlikely that any object system architecture should be the perfect match for any and all current or future computing needs, this is a serious issue that disqualifies the solution.
The former solution, where invocations are translated either at the initial point of invocation or somewhere en route to the server, looks to be much more flexible. A realistic scenario would be "islands" of one object system architecture needing to communicate with "islands" of another object system architecture. These islands could internally use their own

[22] CORBA specifies a CORBA-DCOM bridge that tries to span the gap, but other than that we know of no initiatives that address this issue.
[23] The design goals (section 1.1 in [SOAP]) explicitly disregard objects-by-reference for the sake of simplicity. It is further argued that one can simply implement object references in SOAP, but we find this to mean that SOAP is as object oriented as ANSI C. In other words, it isn't.

architecture, and use some form of gateway or common representation to facilitate method invocations on objects in another island. Assuming this to be a realistic situation, the problem in this chapter boils down to this question: Where should the gateway(s) be located? In the following sub-sections, we will examine this question in more detail.
But first, we will list the criteria with which we will weigh the solutions. These are:

- No gateway necessary when addressing object within the same architecture
  This is to ensure that however the solution may look, the distributed invocations that worked before will still work with no modification of code.
- A reasonable level of scalability
  This one can be hard to quantify – and how much scalability is needed? In this scope, scalability means the ability to have more clients perform cross architecture invocations without incurring too heavy a price on the changes needed to make that possible.
- A reasonable level of flexibility
  Another hard-to-quantify property – and how much flexibility is enough? In this scope, flexibility means the ability to function under changing outer circumstances which are not under ones own control. Defective gateways, for instance – can another just be used? This would be indicative of high flexibility.

Solution 1: One gateway per client

A simple solution would be to leave the responsibility for translating to a foreign distributed object system architecture to the client needing to perform the invocation. In this way it is relatively simple to achieve cross-architecture invocations. In essence, this means that any client wishing to invoke a method on an object in another architecture, simply needs to be a client in that object architecture. This would mean that getting started with cross-architecture invocations is easy, but if more and more client machines need to perform such invocations, they need to be multi-architecture clients. While this solution would probably work, it provides absolutely no transparency in terms of architecture whatsoever. Since none of the major distributed object models have facilities to overcome this problem, neither of them would make a good choice for selection as a unifying architecture. One could then argue that a new distributed object model would be needed for this purpose.

In summary, the solution can be visualized as below in figure 1. The open boxes represent clients wanting to perform cross architecture invocations, and the filled boxes represents gateways. Note that in the figure, three architectures are shown, requiring each client to have two gateways each. If any client should need only to invoke methods in one of the other architectures, it would naturally have only a single gateway installed.

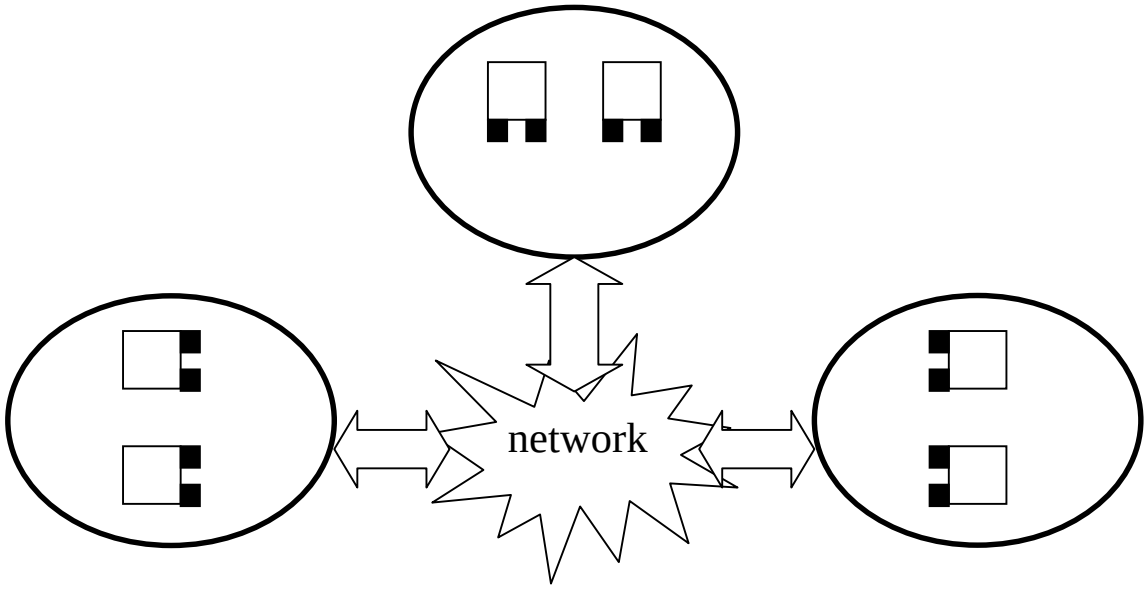


Figure 1: One proxy per client

As can be seen from figure 1 and the preceding discussion, the complexity of installing gateways – and the sheer number of them – can be considerable, especially if several architectures are to be used and numerous client machines participate. For this reason, we doubt that this would be a good solution, since scalability would be very poor. There are probably numerous ways that auto-updating or versioning could be applied to this problem, but this does not change the fact that the scalability problem does exist, and will therefore not be discussed here. Flexibility is average, since a malfunctioning gateway would often imply a malfunctioning client, making the entire invocation a questionable matter. When it does not imply this, there is no built-in way to fall back onto another gateway. One could come up with a scheme where any gateway within the island could be used, but this is essentially solution 2, described below.

Solution 2: One gateway per island

This solution tries to remedy the problems of solution 1 by making a few central servers (to have some level of fault tolerance) in each island responsible for gateway tasks, possibly a few per foreign object architecture. This reduces the number of gateways considerably, and makes for greater scalability since any client wishing to use a given object architecture only needs to know what machine to contact for translating its invocation to the target object architecture. Alternatively, gateways may be provided by the foreign object architecture islands, and one could dispatch the translation task to that location. The difference lies in whether the invocation has been translated to the target architecture before or after travelling over the connecting network. The idea of using gateways provided by foreign architectures is not very appealing, since it is less scalable – if an invocation needs to take place to some foreign architecture in another island, it is not within the powers of the organization in the local island to make it possible. Instead, some way of bartering with the foreign organization

is needed, severely lowering scalability. This means that having the gateway(s) belong to ones own island is the preferable approach (in our point of view).
Flexibility is high – if any gateway is unavailable, another can simply be chosen. Since the number of gateways is likely to be limited for cost reasons, this could in extreme cases reduce availability, but it is not a likely event.

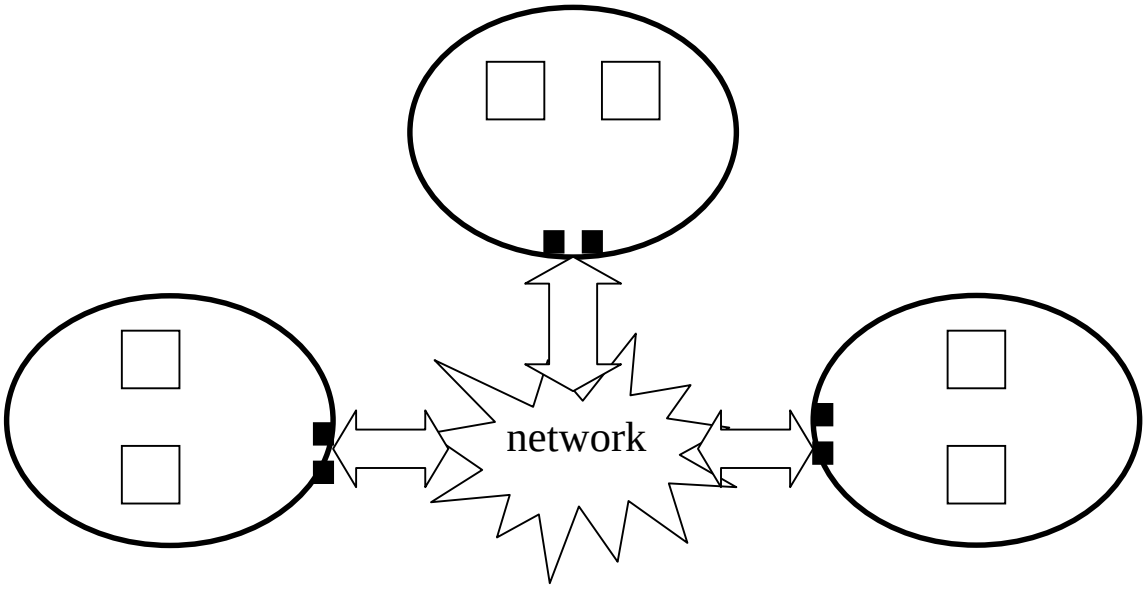
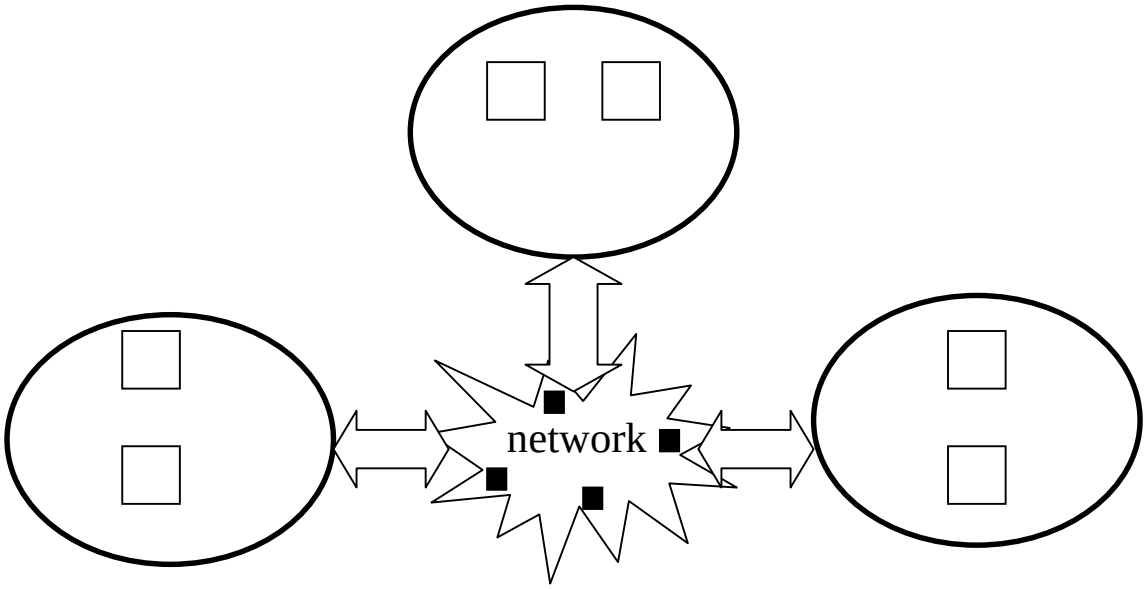


Figure 2: One gateway per island

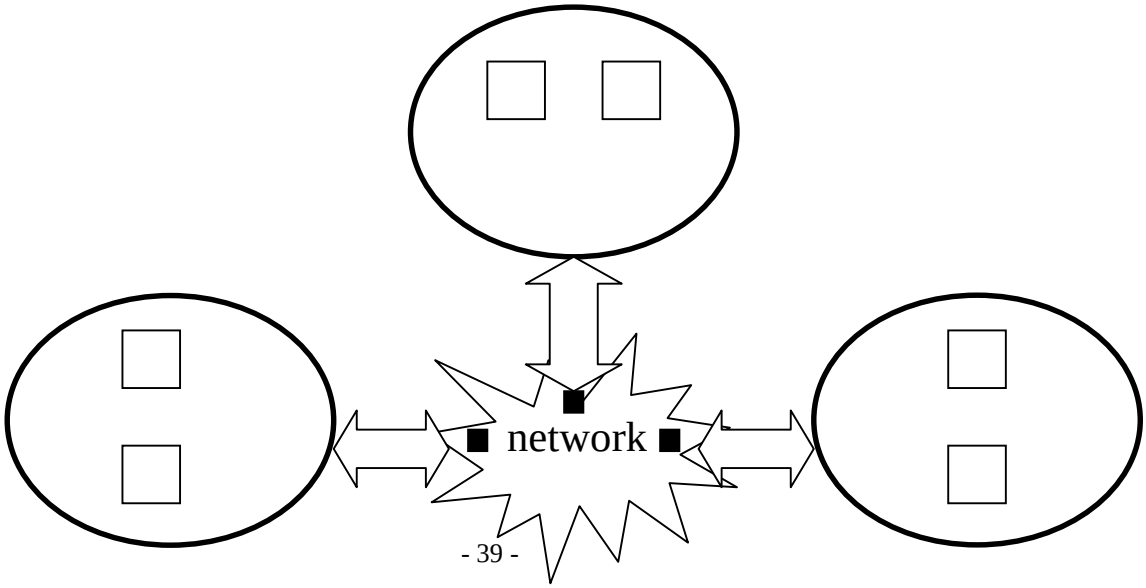
Solution 3: A network-provided gateway

Taking the idea of solution 2 to the extreme, one could imagine having just a number of gateways in the connecting network, which could take care of invocation translations as needed. In this way, as few gateways as possible are needed (while still maintaining a certain level of fault-tolerance), and the clients wishing to become multi-architecture clients still only needs to know the location of the necessary gateway. This is an appealing approach because it provides maximum flexibility and scalability, but it would require that “someone” provide the gateway service. Assuming that these “someones” can be found, this would be an effective approach. If not, each island would need to provide the gateways themselves, in effect reducing to solution 2. A downside is that these gateways become single points of failure, making the cross-architectural invocation system more fragile. A simple distribution system among gateways would solve this problem, in a manner not unlike the situation with top-level domain name controllers on the Internet.



Solution 4: An intermediate protocol

Assuming that some common protocol could be constructed, so that invocations could be carried across the connecting network in this single protocol, it would make sense to translate invocations to this common protocol when an invocation leaves an island, and translate it to the native architecture when it enters another. In this way, the gateway itself could be perceived as being distributed and split in two (less complicated) halves. This is classical divide-and-conquer and would make the task of actually providing gateways somewhat simpler, which in turn is likely to mean that more will be provided. This is a very appealing approach, but it is unfortunately quite unrealistic. Assuming that such a protocol could be constructed, much less agreed upon, is a far stretch. Object architectures ARE different, and the mechanisms by which they provide their functioning makes it very difficult to find a good common ground. [SBS] has some good points in this respect when looking solely at DCOM and CORBA, and in the next chapter we deal with this separately. Imagine the mayhem one would encounter if 5, 10 or 20 architectures were to be incorporated into one. Chances are, that no single intermediate protocol would ever come to be, and that trying to do so would result in a number of competing intermediate protocols, leaving us with essentially the same problem as before. Having said that, it would be far simpler to deal with, say, 4 intermediate protocols than 20 architectures, and therefore the solution is likely to be very useful.



Tentative conclusion

We are very fond of solutions 3 and 4 which both provide good scalability. A mixture of solutions 3 and 4 would be a good choice, in our opinion. Having access to some service on the connecting network that can translate between native object architectures and a few intermediate protocols would provide a very flexible system with a high degree of scalability and stability. As noted earlier, solution 3 may degrade to solution 2 in case no central servers are provided, but this does not change the properties of our mixed solution.
One could imagine augmenting solution 1 with the intermediate protocol idea, which would make it more interesting and simpler to handle. But this would essentially boil down to using an intermediate protocol as THE distributed object architecture, which is not likely to happen (as discussed before). This does not mean, however, that solution 1 should be totally dismissed. We think that it would be beneficial to not disallow solution 1 in the mixed solution we propose, since using the principles in solution 1 can be beneficial of very few objects and architectures are involved.

Our proposal

Each cross-architecture client communicates with one or more network-provided gateways that translate invocations to a common intermediate protocol. The translated invocation then travel to another network-provided gateway that translate it to the target architecture. From there, it is transmitted to the actual server holding the invoked object, and is finally executed. The return value(s) travels back through a similar mechanism, getting translated to an intermediate protocol during transit.
If one needs to further simplify this scheme, it is possible (but not necessary) to use the intermediate protocol directly at either end. This would be the case on systems that are not in a given island, and would not benefit from being so. A handheld device could be an example of this. Embedded devices too small for a "real" object architecture could be another[24]. Undoubtedly, other examples exist. This is, in out view, a good argument for the validity of our proposal, since very simple clients can thus be constructed. Furthermore, for these clients there would be a high level of architecture transparency, since they would not need to know the architecture of the actual objects – the gateway would take care of resolving this.
Since this added bonus for simple systems come at no cost to more advanced systems (regardless of chosen solution, some translation WILL take place), we think that our solution would be very useful across a wide range of platforms and machine types. It will provide a vehicle for actual cross-architecture invocations and also provide a common protocol for use when such a thing is required. It further requires only a limited number of gateways to exist, and provides good scalability, flexibility and robustness to failure[25]. We have not addressed privacy (protection against eavesdropping on parameter values) in this context, but no matter which of the solutions we've outlined were chosen it is common among them that invocation parameters are transmitted from the client to the server. Choosing a single (or a few) common protocol(s) would enable privacy to be built into that protocol at a native level, rather than

[24] Being a full-fledged object system architecture client incurs some cost. In some case, this cost can be quite high. For instance, even small implementations of CORBA are usually in the hundreds of kilobytes range (see [???]).
[25] We have chosen not to address partial failure due to network congestion or failure, since these issues have been dealt with in detail by many others, and for cross-architecture the problems posed by these issues are essentially identical to the single-architecture case.

trying to augment all existing distributed object system architectures with a good security model, an extra argument in favor of our solution.

8. Object architecture independence and transparency

In this chapter we shall examine some of the problems that one faces when two (or more) different object architectures are to be integrated. The focus will be on probing what level of architecture independence and transparency it is realistic to achieve. This will be done by gradually extending an existing solution to handle the problems we encounter. This leads to an interesting solution that presents a good deal of object architecture independence at minimal cost.
It is beyond doubt that other solutions could be found to the problems presented, but we have chosen to focus on generalizing a single solution to a high level to explore several different issues, that arise only after the more basic problems have been solved. In this way we hope to both explore the problem and present what looks to be a workable solution.

What is object architecture independence?

Object architecture independence means that it does not matter whether one or the other object architecture is used for a particular object. This does not imply that the choice does not affect the programmer, since some work may still be needed to make object architectures interoperate. Rather, it implies that it is possible to choose an object architecture without concern for whether or not it will be possible to make it interoperate with another architecture.

What is object architecture transparency?

Total object architecture transparency means that the difference between different object architectures have been abstracted away to such an extent, that it makes no difference at all whether one or the other object architecture is used, and that the programmer is never confronted with any issues related hereto. Having no object architecture transparency means that the programmer is entirely responsible for whatever happens across object architecture boundaries. Many levels of object architecture transparency exists between these two extremes, and the aim of this chapter is to shed some light on the factors involved in deciding what a reasonable level is.

Good and bad transparency

In our view, transparency is only to be used when the transparency can be fully obtained. If the transparency is presented to the programmer, but there are cases where the transparency prevents the programmer from realizing that some action is not possible or desirable, then the transparency is not good. In this respect, we are very much in line with the ideas presented by Waldo et al in [NDC]. They deal with distribution transparency, but the principles still apply. They argue that since distributed procedure calls can fail partially, it is an illusion to model it like this risk didn't exist. In other words, no transparency should be promised when it can not be carried through. In this respect, we like the distribution abstraction in Java RMI much better, since it explicitly does not guarantee that any remote invocation can be carried out[26].

[26] Partly because a distributed invocation might fail due to network problems, partly because of the use of leases for garbage collection

Likewise, we find that architecture transparency must not guarantee what can not be completely delivered.

Having stated this, we must admit that some features of architecture transparency can seem so rewarding, that a few constraints to the actions possible to the programmer, in exchange for a high degree of architecture abstraction, can be a very good tradeoff.

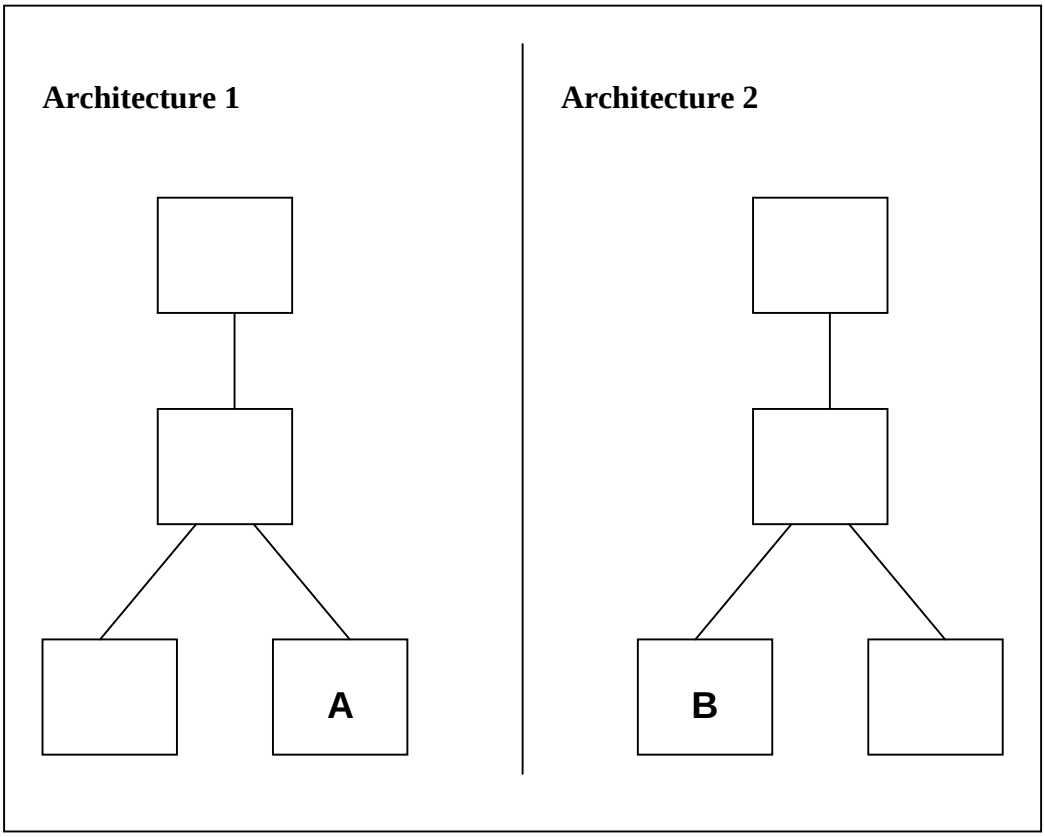
In [GD], some amount of cross-architecture is achieved. This is primarily done by providing a "wrapper" which is sufficiently loose to encompass a sleuth of object architectures and programming paradigms. While very interesting in itself, this does not constitute architecture transparency in our view, since the architectures supported does not become available to one another. Rather, they are accessible only through their DOM system, which attempts to be a "Grand Unified Theory"[27] of programming paradigms and architectures. Nevertheless, this is one of the approaches we've seen that might actually work, if programmers can be persuaded to use their DOM system.

In the CORBA-DCOM bridge specified by CORBA a different approach is taken, which attempts to make it possible for the two object architectures to interoperate. This does work, but the transparency is limited. Furthermore, it is not a general approach – it is tied directly to DCOM and CORBA.

Among many other things, SOAP (see appendix B) attempts to act as "glue" between disparate object architecture, but in our view it is still too much in its infancy for us to make conclusive statements about its merits. We will merely state that we it looks promising, but that the distribution model it supports is too weak in our opinion, primarily lacking object references. It should, however, provide a good protocol design vehicle because of is basis on XML (see appendix G).

By limiting ourselves to two architectures we hope to simplify the issue somewhat. This is not a serious limitation, since integrating more than two object architectures basically consists of a matrix of two-architecture integrations. The following inheritance diagram will serve as the ongoing setting:

[27] An attempt by physicists to make a single theory encompassing all existing theories of physics, which has so far failed because they seem too different



In the scope of the above inheritance diagram, we will investigate the following issues:

1. Can an instance of A be transferred to an instance of B? This was discussed in terms of proxying and network issues in chapter 7, but here the focus will be on investigating how object architectures can be integrated on a modeling level.
2. Can A be subclassed in architecture 2? This would be a step towards unifying object architectures.
3. Does it make sense to transfer a reference to an instance of A to an instance of B? Does it make sense to subclass A in architecture 2? What does this achieve?

Can a reference to an instance of A be transferred to an instance of B?

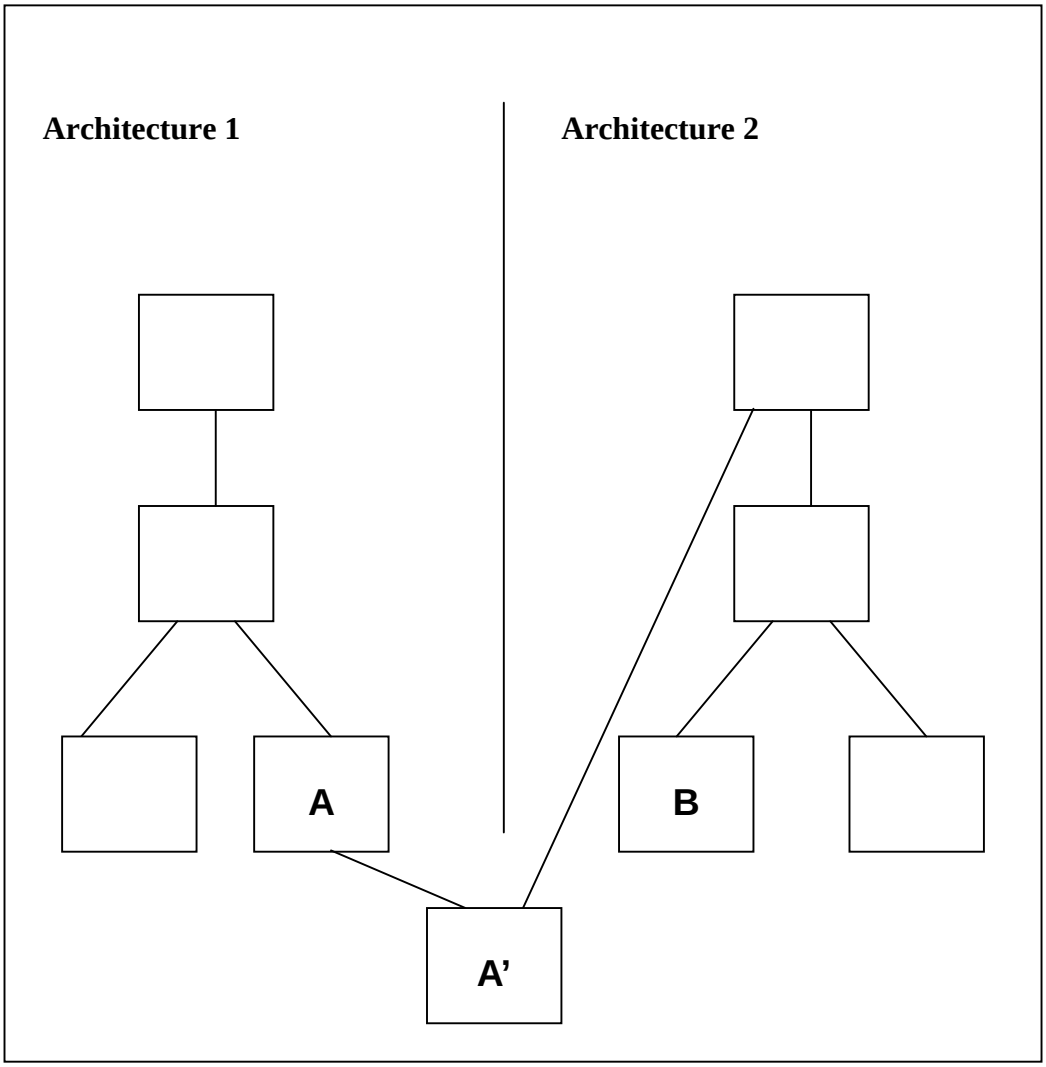
Assume that B is an objectList of some kind, and that it has a method named Add. This method takes a single parameter, which must be of the basic object type of architecture 2, thereby allowing any derivative in architecture 2 to be given. If the use of two architectures is to be transparent at a modeling level, it must be made possible to give an instance from architecture 1 as a valid parameter to the Add method.
First of all, we have found no references demonstrating this to be possible when this text is written. But what would it take to make it possible? In other words, how can A be made type-compatible with the basic object type of architecture 1?

Tie objects

In CORBA, it is usually recommended that non-CORBA objects be wrapped in “tie objects”. This is a simple delegation solution, which provides a wrapper around the foreign object. Any call to the CORBA interface is simply translated to the corresponding call on the “tied” object, and the call is thus delegated. A generalized version of this approach would solve most problems, since it would allow a “tie-A object” to exist in architecture 2. This object would then delegate all calls to the actual A instance for execution. At least one problem prevents this from being a good general solution, namely the problem we refer to as namespace pollution. This is the problem of having pre-defined (and mandatory) methods within the individual objects’ namespace. Examples are AddRef, Release and QueryInterface in DCOM, but more certainly exist. It is not difficult to imagine a situation where a pre-defined method in architecture 2, which would be implemented on “tie-A” to maintain type compatibility with architecture 2’s basic object type, would mask out a method with the same name on the delegated object, effectively preventing any calls to that particular method on the actual implementation of A. One could then argue that the designer chose the method names in A poorly, and a point could probably be made for this. But it is, in our view, not the main issue here. In terms of architecture transparency, it is bad that you need to have prior knowledge of the object architectures your object might need to interoperate with. And if names are to be chosen defensively to avoid causing problems because of namespace pollution, the level of architecture transparency falls markedly.
Assuming that tie objects are a viable approach in general, the namespace pollution problems still hold. Using just simple tie objects, we fail to see how the problem can be overcome. Besides, tie objects corresponds closely to solution 1 in chapter 7, which we argued would not be a very good general solution because of the excessive amount of proxies needed and the poor scalability.

Cross architectural multiple inheritance

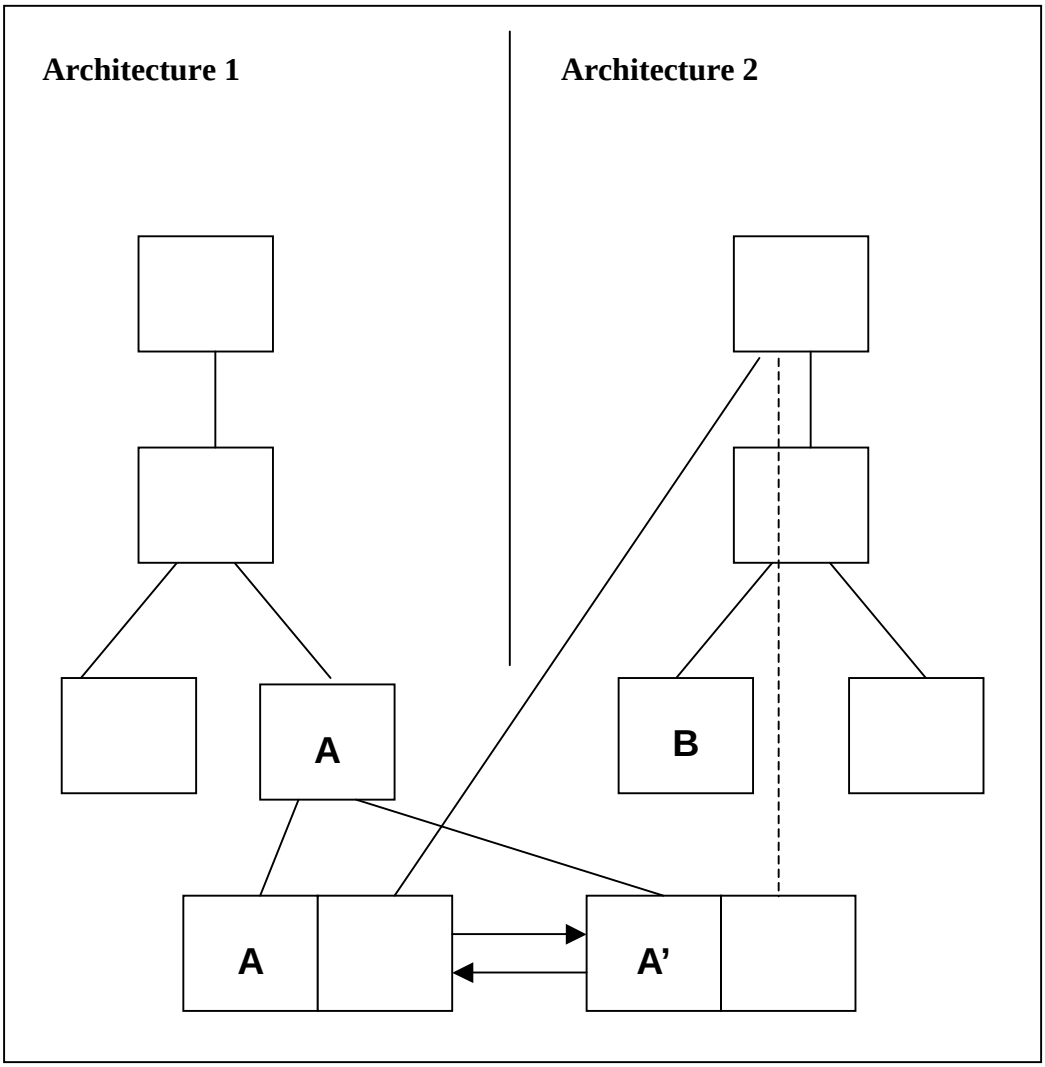
If we were somehow able to derive a common class from A and the basic object type of architecture 2, any instance thereof (or of a class derived from the common class) would be type compatible with the basic object type of architecture 2. The situation is shown below:



This would imply that A' is type-compatible with both A and the basic object type of architecture 2. One way of looking at this is that A' is a double tie object with two interfaces. One would be the interface defined by A and the other would be the interface defined by the basic object type of architecture 2.
But how would such a class work? It can not be placed in any of the two architectures exclusively – this would move us no further to a solution. Placing it in a third architecture would only serve to worsen the problem. What is needed is a system for expressing the same class in both architectures simultaneously, in such a way that instances of it also exists in both architectures simultaneously. One possible solution to this would be what be call "mirrored ties", which we will explain in the following sub-section.

Mirrored ties

The idea is to have an object in both architectures, with both having two interfaces and using delegation. Both objects support the same interfaces, namely A and the basic object type of architecture 2. The object in architecture 1 would rely on delegation to implement the interface for the basic object type from architecture 2, and just use the inherited A-interface to handle A-specific invocations. The object in architecture 2 would rely on delegation to implement the interface for A, and just use its native object methods to handle the interface to the basic object type from architecture 2.



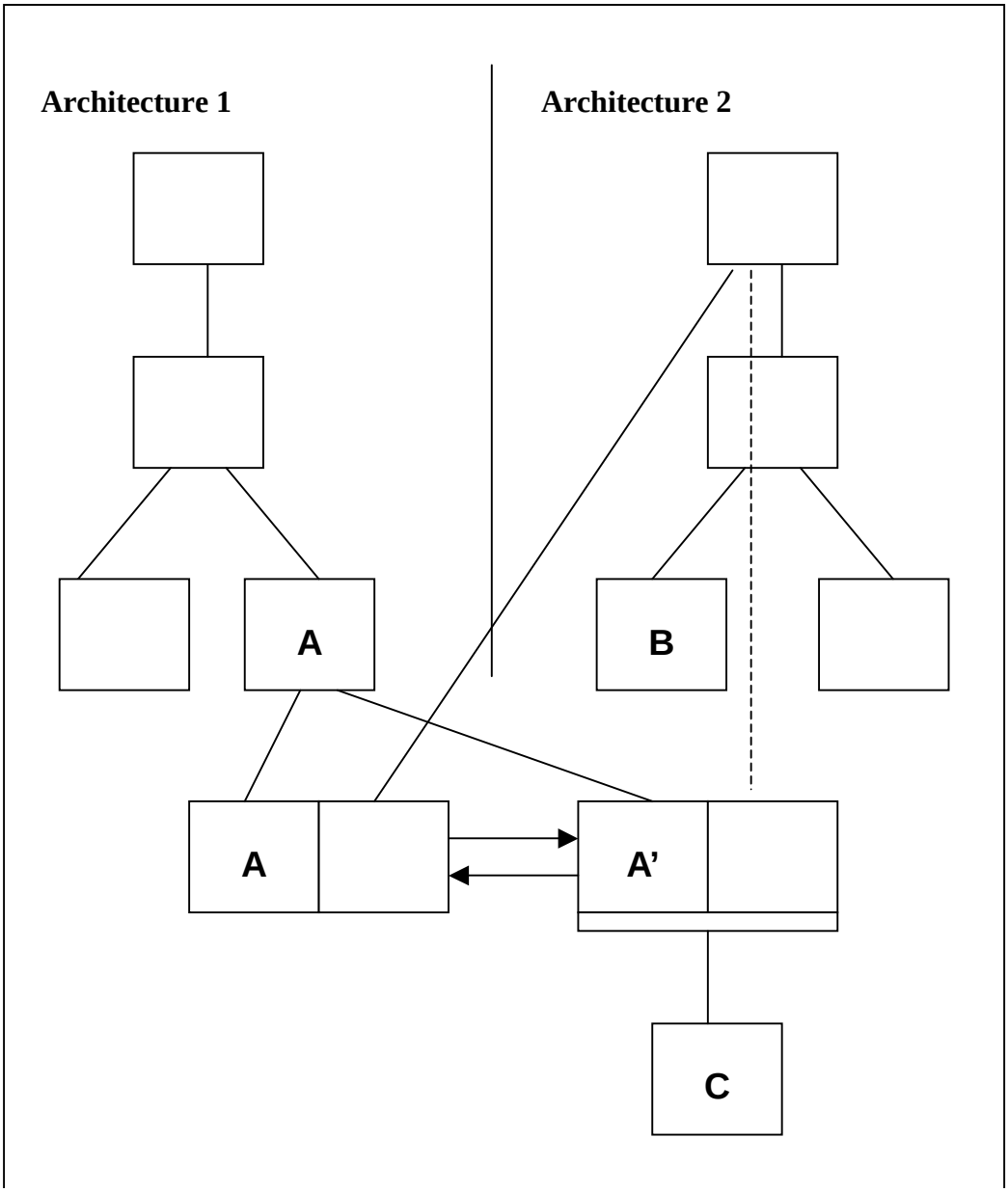
The dotted line indicates that this inheritance is mandatory anyway, since all classes in architecture 2 must be derived from the basic object class in that architecture.

This puts the object in both architectures simultaneously, as they use each other to implement the parts they don't supply themselves. Note that this is crucial – the state for the two objects must be the same. Not have the same value, but be the same state in such a way that changes to either is reflected immediately in the other. This is achieved by simply using delegation (as explained above).

In the light of this suggestion, a solution to the namespace problem looks possible. By giving the object in architecture 1 affinity towards using the implementation provided by its own architecture, but also the opportunity to cast itself to the supplied interface matching the basic object type of architecture 2, we should get the ability to call those methods, if needed. Due to the symmetry in the mirrored ties, the same also holds for the object in architecture 2 – it has affinity towards calling the methods from its own architecture unless it explicitly casts itself to the A interface. Since casting to this other interface is no different from casting to any other interface in the object's own architecture, the namespace pollution problem is reduced to the same problem that is always present when doing multiple inheritance, and while these problems can be troublesome, methods exist to overcome this limitation. We will not address these in this context, since they are specific to the object architecture in question, but merely conclude that such a system could probably be modeled successfully.

Can A be subclassed in architecture 2?

Looking at the above sub-section, this should be possible. By deriving a class from the part of the mirrored ties placed in architecture 2, using just the features for such an action found in architecture 2, we achieve both interface and implementation inheritance. Note that for the derived class, there is no counterpart in architecture 1. The situation is illustrated below



Since the class containing the A' above is an object in architecture 2, it can be sub-classed to C. Any method invocation on an instance of C, where the invoked method is actually implemented by A, finds its way to A by way of A', just as one would expect from the way inheritance normally works.

Does it make sense?

We are of the firm belief that architecture independence is beneficial, since it gives designers/programmers the choice to use it or not. It is not hidden from the programmer's view that the architectures are, in fact, different. Contrast this with architecture transparency, where it makes no difference at all what architecture is chosen and the programmer is not aware that several architectures may (or may not) be in use. This does not imply that transparency is bad, merely that because it represents a higher level of abstraction, some details will invariably be lost for the programmer. If the higher level of abstraction offered by transparency is sufficiently beneficial, this loss of detail could be seen as a good trade-off for the benefits.
We will discuss whether or not this makes sense in terms of transparency – that the independence is beneficial is a given.

While the described solution would seem to work (it has – to our knowledge – not been implemented, and consequently has a number of unresolved issues), it is an open question why anyone would want to do so. This is a big issue, and in the context of this text we will only be able to scratch at its surface. As stated earlier, such a construct makes sense if it is easier to solve some set of problems with it than without it. Since the main problem it addresses is achieving architectural independence and limited transparency, the question boils down to whether or not such transparency is actually desirable.

Transparency is everywhere

A sizeable part of computer science and related fields of work is working on transparency. Operating system transparency (most programming languages), distribution transparency (DCOM, CORBA and many others), machine architecture transparency (Java and Linux, for instance), resolution and color depth transparency (PostScript), paradigm transparency ([GD], for instance). Numerous other transparency related projects exist and were omitted only for brevity. While being widespread certainly does not in itself label all attempts at achieving transparency worthwhile, it does give a strong indication that transparency is desirable. As previously stated, we agree completely with Waldo, Wyant, Wollrath and Kendal with their argument in [NDC] that transparency is not to be overdone, as it can "gloss over" some issues that should not have been glossed over, since they are issues that can – in essence – not be solved. Distribution transparency is the key example here – how can anyone guarantee that a remote method invocation is possible at all times? Of course this is not possible – maybe the network cables were severed, thereby making the host unreachable. Or the host may simply have disappeared from the net. This makes issues like distributed garbage collection infuriatingly difficult, as we discussed in chapter 6. But by realizing that the guarantee can not be enforced, and designing the distribution system with that in mind, much more realistic (and therefore useful) systems arise.

In the view of the authors, this means that the ways in which architecture transparency can fail should be probed to give a fair assessment of the areas where the transparency is based on false guarantees. This is difficult without having access to a running system with features somewhat like the solution we described in the previous chapters to experiment on. Of course one can probe the issues in theory, and many good things can come from that approach. But to really get a handle on the problem, an implementation should be available. Since no such implementation exists at the time of writing this text, we have chosen to forego such an analysis. We would very much like to implement exactly such a system, but have simply chosen to not do so, for no other particular reason than to limit the scope of our thesis work.

9. The Object Visibility Problem

In distributed object systems, problems may arise when objects are given as parameter values in invocations of other object's methods. Are the objects actually able to call one another's methods? Several classes of problems can occur, depending on the object system in question. In this paper we will explore two of these problems and their possible solutions.

Problem background

When an object reference (referring to what could be called the parameter object) is sent as a parameter to another object (the called object), it is not automatically so that the called object is actually able to call methods on the parameter object. While this problem may not be obvious, looking at the situation in CORBA serves as a good illustration. While the GIOP specification makes sure that ORBs can communicate with one another, this is only the case for ORBs using the same GIOP implementation. Since GIOP implementations are network-dependent, problems may arise if an object reference for an object served on TCP/IP is handed to an object served on DECnet[28]. When the DECnet object tries to resolve the object reference for the TCP/IP served object, the address for the server is one of the items to resolve. Since the TCP/IP served object uses IP addresses and DECnet does not, the address of the server will have no meaning to the DECnet served object, and the TCP/IP served object is therefore not visible. This situation can be handled by manually adding code to the system to handle this issue, but no general mechanism exists to handle this. Nor is one likely to come to be, since the situation is particular to the two network protocols in question.

Other issues than network addressing discrepancies exist, and we will deal with the above issue and that of having sufficient type information available.

This may, or may not, be an issue on stand-alone systems, but it is very much an issue when dealing with distributed object systems, which will be our sole focus in this paper. Most distributed object systems have faced the problem, and have tried various approaches with different levels of success. Where relevant, we will discuss how a certain object system tries to solve a given problem or use a given suggestion. This chapter is not intended to "rate" the object systems according to some scale, merely to point out that several of the approaches we mention have, in fact, been tried.

[APP] deals with a subset of this problem, namely that of analyzing whether object parameters should be transmitted by reference or by value in a distributed system, and how the "by-value" choice could be implemented with some degree of efficiency through graph optimization algorithms taking its input primarily from "clues" given by programmers. While this touches on some of the same issues as we do in this paper, our focus is not primarily efficiency of calling – we focus more on the factors involved in deciding whether the invocation can be carried out in the first place.

[28] TCP/IP and DECnet are not the only combinations to yield problems. In general, ALL combinations can be quite troublesome.

For the two issues listed below, we will attempt to explore whether it is expected to play a role, and if so, how one could possibly avoid the problems. We will not limit ourselves to any particular distributed object architecture, but rather explore the field a bit more generally.

1. Network configuration issues
2. Type information availability issues

We do not expect to provide clear-cut answers to the issues at hand, merely point out directions in which we think solutions could very well be found[29]. There is no doubt in our minds that more issues exist, but we have chosen to limit ourselves to the two mentioned ones since we are of the opinion that they are relevant ones. Another very relevant issue is that of cross-object architecture references, but this is dealt with separately in chapters 5 and 6.

We will not vigorously pursue complete location transparency in our quest for solutions, since we agree with Waldo, Wyant, Wollrath and Kendal (see [NDC]) that such transparency is not necessarily a good thing.

Scenario

Imagine the following scenario: On machine I, some instance of object X lives (we'll call it iX). On machine II, some instance of object Y (called iY) holds a reference to iX. On machine III, some instance of object Z (called iZ) lives, and object Y also holds a reference to iZ. Figure 1 details the situation.

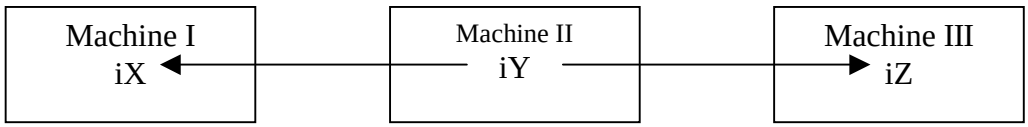


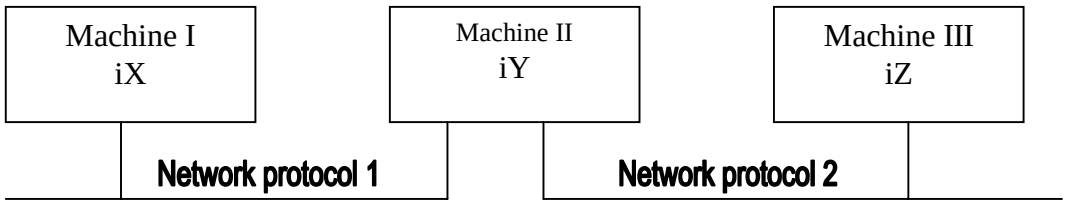
Figure 1: The test scenario

Suppose now, that object iY wishes to invoke a method on iZ, giving it iX as a parameter. Can iZ then proceed to actually invoke methods on iX? In the following subsections, we'll discuss the two issues stated in the problem background in the context of this scenario.

[29] These solutions may well be of use when straight calling of objects is involved, ie not in invocation through an object reference obtained through a parameter. However, simpler solutions for this simpler problem probably exists and may be considered more pertinent.

Problem 1: Network protocol discrepancies

Imagine that the three machines sit on the following network:



Can iZ invoke a method on iX, if given the iX reference by iY? The potential problem here is that the invocation may not be able to cross the boundary between the two network protocols. Why is that? In some popular distributed object architectures (most notably CORBA, see [CORBA]), object identifiers often contain the address of the server holding the actual object (this means that through **local** interpretation of a reference to iX, we should be able to get the value "Machine 1"). If iZ attempts this interpretation in its local network scope (network protocol 2), there is no guarantee whatsoever that it will work, or even report an error if it fails. The reference to iX may use a different encoding specific to network protocol 1 (since that's the native network scope for iX), and an address in one network protocol may have no meaning in another, as described earlier in this chapter.

If the distributed object architecture does not support automatic use of gateways between the two protocols, the call will not be carried out. For instance, in CORBA you normally have GIOP[30] implementations that are specific for network protocols. The main one is IIOP (Internet InterOrb Protocol) for TCP/IP. But others exist to cope with CORBA calls across other types of networks. How can the ORB on machine III know that the IOR referring to iX is local to network protocol 1, and treats it as such? Attempts to interpret it as an IOR local to network protocol 2 would almost assuredly NOT work. No such cross-protocol communication is possible unless either:

1) The underlying network is powerful enough to make the discrepancies in protocol totally invisible.
   Naturally, this would solve the problem. However, we have yet to hear of such a thing to exist and function perfectly.
2) Machine II somehow acts as an intermediate (or proxy) server through advanced marshalling of the IOR of iX.
   This approach is somewhat feasible, since Machine II could insert itself as server for iX in the reference passed to iZ, and then silently pass on invocations to iX from iZ to Machine I, in effect acting as a proxy server for object iX. However, this will require considerable efforts in terms of administration and implementation, since the number of possible permutations between network protocols can be somewhat large. For this approach to be efficient, no such proxy behavior should be implemented when it is not needed, so Machine II must be able to deduce when an invocation is between network protocols and take steps in that case only.

[30] General InterOrb Protocol, a specification of how ORBs in CORBA can communicate with each other

3) Machine III is able to deduce that its reference to iX is not valid in its own scope, and has enough information to be able to ask for it to be either changed to a valid one or routed through some intermediate server (Machine II in our example).
   This might work, and it would be rather obvious to trace the reference to iX back to its originator (Machine II) through determining where the call came from (a kind of distributed stack-trace). Then ask the originator to pass along the request, continuing the backward trace until the network protocol that the reference originated in is encountered. In our example, that would be the case once the call got passed to Machine II. This would require that machines be able to perform this backward tracing with little or no performance penalty.
4) A number of object address stores are maintained, where the individual objects can look up the actual location of servers for distributed object references. Assuming that such a store is visible in every network protocol segment, and that the network is able to route network packets between protocols, this could be a solution. Keeping such an object store in sync would be a major undertaking, and it would – in our point of view – not be possible to keep it completely so if it is distributed. But a single central server could not possibly sit in all network protocol segments unless it sits at the center of a star network topology, severely impeding the design of the overall physical network.

Conclusion

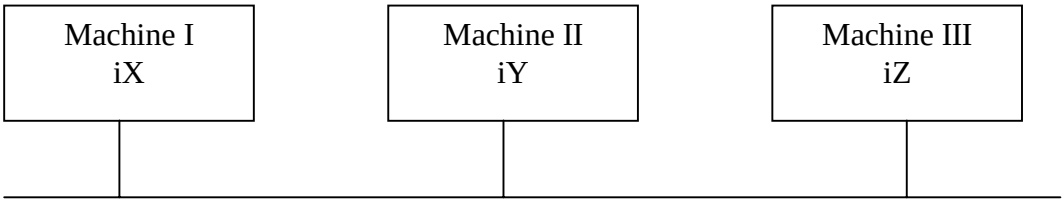
It is clear that calling across network protocol boundaries can present considerable trouble, or at the very least require that non-trivial steps be taken. Solutions 2 and 3 are – in our opinion – the only viable ones, and are actually rather similar. The primary difference lies in WHEN steps are taken to correct the situation. Solution 2 attempts to make sure that the problem will not arise through marshalling and proxy serving, whereas solution 3 attempts to solve the problem as late as possible (immediately before invocation is to take place). Solution 2 has the advantage that no cross-platform distributed object reference scheme needs to be devised – the existing ones are used and conversions take place as needed. Solution 3, on the other hand, looks to require less implementation and will correct only those distributed object references that cause problems.
To our knowledge, no existing distributed object system addresses this problem. It would be interesting to solve it, and we intend to attempt this in our solution.

Problem 2: Type system discrepancies

In most of the strongly typed object languages (C++, Java and similar), some amount of information about type compatibility is needed at compile time for object reference passing to be possible. At the very least, the actual parameter object usually needs to be an instance of a class descending from the formal parameter's class. Since distributed object systems are often (but not always) coupled with one of these languages, the same requirement holds for most distributed object systems. For languages with less strict typechecking this requirement may not exist, but that does not necessarily mean that no typechecking is done – it may just be delayed until the invocation arrives at the server, and it is decided that type safety is not met. It is also possible to imagine distributed object systems where it would make sense to have no type checking whatsoever, and naturally these would not suffer from this particular problem. But otherwise it is clear that, either at compile or run-time, type information MUST be

presented to the caller in order to make sure that the desired call is indeed typesafe. We will therefore concentrate on the situations where type information is indeed needed.

Now, imagine that the three systems from the scenario sit on the following network:



Would iZ be able to invoke a method on iX through a reference obtained as a parameter from iY? The answer would be "yes", if iZ already possesses the type information needed to perform such a call in a typesafe manner, or the information could somehow be obtained when needed.
While this problem may seem to be trivial, it is not necessarily so. Making sure that the type information is universally up-to-date is an intractable problem (see [???] det der med generalerne og adjudanterne). Having said that, it is our experience that type information is usually not very dynamic, so a less than perfect distribution system will probably suffice.

To summarize: In a typed object system, it would be required that either

1) Type information is available at compile-time, to allow the compiler to check type safety.
2) Type information is available at run-time, to allow the run-time system to check type safety
3) No type information available at any time before the call is made, but calls are nonetheless possible to make by foregoing type safety, possibly failing type check at the object servers location and the failure sent back to the caller.

Some distributed object systems allow for a mixture of these. For instance, DCOM (see [DCOM]) allows – in most languages – the use of any one of the three through

1) vtable binding.
2) distributed IDispatch invocation (a so-called type library may be available for constructing calls in a typesafe manner[31]).
3) distributed IDispatch without checking type safety, and quite possibly getting a "parameter incorrect" exception as a result.

Note that the choice of which method to use is entirely up to the programmer, and that no programming or runtime environment – to our knowledge – makes it possible to combine several approaches in an automatic manner. This means that if the programmer chooses approach 1), and no MIDL[32] file or equivalent is available at compile-time, the program will

[31] This is similar to the approach chosen for XOIP's COM handler – see chapter ??? for details
[32] Microsoft Interface Definition Language, a proprietary way to define COM/DCOM interfaces. Not to be confused with IDL, the Interface Definition Language defined by the open OMG (Object Management Group, the driving force behind CORBA)

simply not compile. If the programmer chooses 2), and no type library is available, the program will fail. If the programmer chooses 3), it is very likely that sooner or later some call will fail, since type safety can not be enforced very well, and could possibly leave the application in an indeterminate state. One could argue that iZ has no business calling iX in that case, and that could be a valid point (depending on the distributed object system). Our goal here is to examine under what circumstances calling is possible, not whether a point could be made about such calls being bad practice – practices have a habit of changing over time. Furthermore, a point could also be made that a good distributed object system should be able to handle what may be deemed bad practice in a reasonable manner. For instance, in SMALLTALK (see [SMT]) the notion of calling what may, or may not, be supported on some object is engrained in the language's object hierarchy, and doing so does not necessarily mean that some application will malfunction. A distributed version of this approach might be useful, and indeed distributed SmallTalk has been made, but has not received widespread use.

A very different solution would be to migrate objects when calling. For instance, when iY calls iZ, iY will migrate to Machine III (where iZ lives) and perform a local call instead. At Machine III the type information on iZ is available since the object is hosted there. When iZ then calls iX, iZ is migrated to Machine I, with similar benefits. Naturally, this means that distributed object identifiers can not contain an identification of the server hosting the object (since it can change without notice), or that all object servers must be able to respond to a call with information about where the object went, thereby allowing an object to choose a more correct destination. The latter solution has the distinct disadvantage that migration livelock may occur. Since objects may live on different machines, method execution is inherently parallel. Suppose that two objects (not on the same machine) decide that it is time to call one another. They both migrate to where they thought the other was living, and discover that the object has left. They then migrate to where the other objects went, only to find out it's no longer there and so on – essentially a livelock situation. This can probably be solved eventually through such a simple measures as random waits (like the way collisions are handled on Ethernet, see [ETH]).
As a final thought, an object locator service could be queried when looking for an object. Such a service could only guarantee that it's location is correct by denying objects the right to migrate if they are to be called, thereby acting as a migration referee. This could have serious implications on performance, and could effectively prevent an object from migrating any further, if one or more other objects repeatedly call it. Through some ingenious protocol and serialization of object access this can probably be avoided (we shall not delve further into this in this paper), but the main argument – as we see it – for the object migration scheme, its conceptual simplicity and elegance, is somewhat lost. For an example of a functioning distributed object system with migrating objects, look at Ara [ARA] or Emerald [EMERALD].

Conclusion

It is clear that some way of distributing type information is needed, especially when using distributed objects in conjunction with strongly typed languages. Whether it is through compile-time bindings or run-time bindings can be viewed as less relevant. What is not clear, however, is how easy it is to implement run-time type binding in a distributed object system. If the database of type information is centralized, it becomes a single point-of-failure that tends to make such a system more fragile. If it is too distributed it may encounter update problems.

In CORBA, the interface repository is responsible for requests concerning type safety at run-time. It is essentially distributed, and to our knowledge and experience works just fine.
In DCOM, the IDispatch interface[33] is coupled with a type library to provide the information to perform type safety checks. An inherent weakness in this system is that you need either an instance of the object to query its methods, possibly wasting resources in the process, or you need remote access to the server's registry[34] which is NOT something one should grant promiscuously (if at all). Another possibility is to have the DCOM type-library installed locally but that is, unfortunately, a manual task.

Through the use of object migration instead, the problem can be avoided. Migration is not a silver bullet though – as we've pointed out in the previous sub-section, migration has its share of problems.

[33] In COM and DCOM, an interface represents the public methods available on an object. A special interface, the IDispatch interface, was devised to allow calls to objects for which no type information was available at compile-time.
[34] A registry is a Win32-specific centralized storage for settings on a given machine, including settings about supported DCOM objects and their class properties – among those, type information.

Part Three: XOIP

10. Design criteria
11. XOIP method calls

In this part of our thesis we will explain the rationale behind our design of the XML Object Interface Protocol (XOIP). We will make heavy use of the points made in parts one and two, as these provide most of the background for choosing the set of criteria and the overall functionality of XOIP.

10. Design criteria

The primary purpose of XML Object Interface Protocol (XOIP) is to provide a set of facilities allowing clients to access distributed objects running in a number of given object system architectures, without being a real client in any of the distributed object system architectures. For instance, we'd like to make CORBA calls without having an ORB, or COM/DCOM calls without running on a Win32 platform or having a similar library giving wire compatibility[35]. Furthermore, very low demands must be placed on participating clients, both in terms of processing power and the amount of storage available. This is mandated by the focus we have chosen to put on embedded systems.

In chapter 7 we described ways to achieve cross-architecture method invocations, and came up with a proposal for how to do this. This approach will form the basis for our idea about how XOIP should function. In chapter 5 we outlined a possible "common pool" of features that a distributed object system must have, and this common pool will form the basis of the features that XOIP must offer to designers and programmers.

In many ways, we aim for a system similar to that described in [GD] by Nayeri, Hurwitz and Manola. Indeed, their method of achieving a generalized dispatch mechanism is rather similar to what we have in mind. We have a different focus, however. They seem to focus primarily on local object models and incorporating different paradigms into one all-embracing description system, and then attempt to broaden that system to incorporate distributed object systems. This is not to imply that they have failed in their endeavor, but it is our conclusion that they attempt to "gloss over" the difference between local and remote objects, giving themselves most, if not all, the problems described in [NDS]. Their focus is on object models and abstractions, while ours is more on designing a lower level system capable of enabling embedded systems to participate. We are impressed with their ability to incorporate class-based object systems, generic methods and rule-based systems into an all-encompassing system, but we actually think they have gone too far. Since the difference in semantics between invoking a CORBA method and matching rules in, say, Prolog is considerable, we are not sure whether it is beneficial to try to abstract the two paradigms into one system. The programmer must still know if one or the other paradigm is to be used, and must be familiar with them use them properly. For that reason, it might be confusing that very similar abstraction mechanisms can have large differences in semantics.
Actually, the system described in [GD] is somewhat similar to the IDispatch interface in COM (see chapter 3) – a dispatch mechanism for allowing code written in diverse languages to call another pieces of code through a generalized dispatch mechanism, without imposing this or that object model on any client or server. And just like COM is extended to a distributed object system through DCOM, the system described in [GD] has facilities added to make it distributed.
The proxying mechanism described in [GD] is similar to Solution 1 in chapter 4, where each machine (in fact, each object itself, if we understand their reasoning correctly) is responsible for making sure that the cross-architecture call is processed correctly. We are not sure if this is a very efficient system, and they point out themselves that they have not been concerned with

[35] Several wire-compatible products for using DCOM on non-Win32 platforms do exist, most notably from WindRiver and Intrinsyc. These are, however, tied to other operating systems – essentially the same problem.

efficiency. Since we wish to cater for embedded systems, we can not ignore efficiency related issues.
We agree that the system chosen to represent the cross-architecture related issues should be general enough to be useful with a wide variety of distributed object architectures and paradigms. Unlike Nayeri, Hurwitz and Manola we must design a package format instead of an abstraction mechanism, since very small embedded systems must be taken into account, and these are not expected to have the resources to provide rather high-level abstractions (their Scheme-inspired DOM scripting language, for instance), as was discussed in chapters 1 and 2.

Design criteria used

The following will describe the criteria that we considered important. Most of these follow from the desired end-goal. Some of these come into play in the design of the central features of XOIP, and some come into play when designing the actual implementation of XOIP.
We have chosen not to deal with issues pertaining to concurrency and partial failure. Not because these are not important issues, but simply to limit the size of the job at hand.

1. Use simple package format
   This means that the network packages travelling back and forth between the client and the server should be simple enough to construct that this can be readily done manually. The reason we consider this to be important is that by ensuring this we can lower the demands placed on embedded systems to be XOIP clients to just being able to transmit (and receive) streams of data on the network.
   If the package format is simple enough that calls can be created manually by the programmer with little or no effort, calls can simply be stored as a sequence of bytes in memory. This allows calls to take place without any marshalling, even if parameters are sent. If these parameters are to take variable values, they can be put into the sequence by very simple means. If a call is to take a number between 0 and 5000, enough room for such a number can be left intentionally blank at the position of the parameter, and it can be written into the area immediately prior to transmitting the buffer. While this approach can certainly also be used with not so simple package formats like GIOP, the entire GIOP package is not easily constructed manually – padding, alignment, and endianness needs to be addressed.
   Using a simple package format further allows very simple debugging of messages, since the programmer can decode the message with little effort.
   A simple package format can, however, easily be somewhat verbatim. It is furthermore usually so that a format easily readable to humans is not easily readable for computers and vice versa. We therefore realize that this criterion potentially clashes with criterion 5 about low bandwidth overhead.

2. XOIP must be able to run on a small embedded system
   This means that the actual implementation of XOIP must be sufficiently lightweight to run on a small embedded system itself. Since it can then obviously also run on more powerful systems, this criterion gives us flexibility in terms of where XOIP can actually be implemented and executed.
   If XOIP is sufficiently lightweight, it can run on an embedded system itself and possibly be hosted by one of the embedded systems already in the network, doing away with the need for further hardware. This would ease the adoption of XOIP.

We have decided not to aim at a specific memory footprint for XOIP, only that it should be as small as possible. Informally, we have aimed at less than 64 Kb for code and as little data as possible.

3. Type-checking must be performed as early as possible, but be flexible
   By this we mean that it must be possible for the client to obtain the type information needed to construct a type-safe invocation if necessary. It must also be possible to have XOIP do the actual type checking, and finally it must be possible to delay the type checking until it arrives at the object server. This will allow as much flexibility in terms of type checking as possible, and corresponds neatly with the three ways we described in chapter 7. This allows us to use XOIP to interface with object architectures that use any of the three methods for type checking.

4. Low architectural overhead
   We wish to make it possible to construct object oriented libraries in real object oriented languages with very little overhead in terms of speed, size, and complexity. This is desirable because embedded systems have scarce resources. These object-oriented libraries must not use any methods on the objects to achieve any functionality, as this would go against our conclusion that doing so is very detrimental to cross-architecture semantics because of namespace pollution.

5. Low bandwidth overhead
   As described in chapter 2, embedded systems usually have low bandwidth communication systems. For this reason, it is important that the wire-protocol does not require inordinate amounts of network traffic. A small increase is deemed acceptable, since this criterion clashes somewhat with criterion 1, which has higher precedence.
   We have chosen to measure ourselves against the packages constructed by GIOP, because this is the only other package format for heterogeneous currently in widespread use.

6. Low memory footprint on clients
   This is important for embedded clients, since memory is usually scarce (as we discussed in chapter 1). We have decided not to quantify this criterion more exactly, but for the minimal client functionality only a few Kb should be required. This is a tough criterion in our view, and one that will probably dictate many choices in design.

7. Some amount of cross architectural properties
   This is one of the central criterions for XOIP, as this is directly related to the raison d’être for XOIP. We would like to give XOIP as many cross-architectural properties as reasonable, and part two of our thesis provide much of the reasoning that will guide us.

8. Existing object servers should not have to be changed
   If existing object servers need to be re-written to take XOIP into account, it is highly unlikely that very many object servers will exist or come to exist. By not requiring this, we get immediate access to a huge code-base of object servers. The downside is, of course, that we are stuck with whatever object designs these servers mandated.

9. Object Visibility must be addressed
   The Object Visibility Problem described in chapter 9 poses a challenging question for heterogeneous object architectures. Some form of progress towards handling this problem must be made.

Chosen features of XOIP

Object references

The concept of object references is central to most object orientation. Even though these references give considerable problems in distributed object architectures, e.g. in conjunction with garbage collection as discussed in chapter 6, we have decided that this feature is desired. This decision comes from the fact that the major object architectures we'll interface with are DCOM and CORBA, which both have distributed references as a central feature.
Furthermore, this will enable us to make an object orientation abstraction closer to that of "classical" object orientation, which will enable us to better meet design criterion 4 about low architectural overhead.

Interface as the central mechanism

We have chosen to adopt an **interface** as the vehicle used for descriptions of objects. The reasoning for this is based on the discussions in chapter 5, where we argue that interfaces and class inheritance provide roughly the same level of abstraction. A mechanism for casting these interfaces must be provided, to allow the properties of up- and downcasting that C++ and Java offer. Note that this choice is also heavily influenced by the fact that DCOM and CORBA both have interfaces as central mechanisms. For object architectures such as Self or JavaScript (see [SELF] and [JSCR] respectively), where objects have whatever methods they choose to implement and a static list of these are not necessarily available, this can just be taken to be a dynamic interface, where it may or may not make sense to cast it to another interface. If it does not make sense in a particular object architecture, then the object just can not be casted, which does not mean that the general casting mechanism is badly chosen.

Avoid namespace pollution

In order to avoid namespace pollution, any and all features regarding object management will be placed outside the individual object's namespace. Contrast this with the way *narrow* is used in CORBA and *QueryInterface* in COM/DCOM. This is one of the major hindrances of proper cross-architecture object references, since it is quite possible that a DCOM object could decide to implement a method named narrow, that could then possibly conflict with the CORBA-specific use of narrow – that of performing typecasting. If the DCOM object does not imitate this behavior exactly, the call will simply fail to behave as expected. It is not to be expected that DCOM objects actually implement this, since the DCOM object is not required to be designed with CORBA-interoperability in mind.
Therefore, it is in accordance with design criterion 4 about low architectural overhead that avoiding namespace pollution is chosen.

Interface inheritance

Inheritance is one of the features usually described as defining object orientation. This means that inheritance must naturally be a part of XOIP, and since we have chosen to use interfaces

for object descriptions, it follows that we allow interface inheritance. Since we have argued in chapter 5 that interfaces and class descriptions are roughly similar, the major distributed object architectures should map easily to this mechanism. DCOM and CORBA uses interface inheritance already.
Design criterion 4 about low architectural overhead comes into play here too. Since part of the work done using XOIP is expected to be through a class library abstracting the actual packages away, designing the interface features to match closely with the abstractions of common object oriented programming languages looks to be a wise decision.

No transparency between local and distributed objects

XOIP is not to be concerned with local objects – it shall deal with remote objects only. Therefore, some of the problems described in [NDC] by Waldo et al. are not pertinent. Note that there is distribution transparency in terms of where the individual remote object is served, but no effort will be made to hide the distinction between local and remote objects. Rather, we wish to maintain that distinction, as we agree with Waldo et al. that this is beneficial to overall design.

Complete encapsulation of data

Data members are not accessible directly from an interface pointer. This means that all data access must occur though accessor functions (like attributes in CORBA and properties in DCOM), that the programmer is responsible for providing. We have chosen this method to avoid giving the impression that data is readily available, when this is, in fact, NOT the case because of distribution. Furthermore, it is a common belief (that we share) in object orientation that the provided encapsulation is beneficial to overall program design and maintenance. The reason for this is that implementation of actually storing/retrieving the data can be changed with no influence on the interface definition, and hence no alteration of code is necessary for programs that access the object in question, making maintenance much easier.

XML as protocol

We decided to use XML as the protocol layer for a number of reasons. First of all, XML allows us to fulfill design criteria 1 and 6 about simple package format and low memory footprint on clients. XML data can easily be constructed manually, and it is therefore not essential that a library of code exist for package construction (as is usually the case with another popular package format, GIOP from CORBA). Even though the data for the XML packages might end up being slightly larger than comparable GIOP packages, this is likely to be more than offset by the absence of a GIOP engine. Even when very optimistic about the ability to make a GIOP engine small, a library of 15 Kb seems like the lower limit (see [ICE]). These 15 Kb can hold lots of XML data instead, leading to what is probably a lower overall memory footprint. This is somewhat bad in regards to criterion 5 about bandwidth overhead, but we are optimistic that we can design an XML-based protocol that will reduce the overhead to a bearable amount.
The use of XML as protocol gives tremendous flexibility in terms of the data that can be represented. This is one of the big strengths of XML – data structures can be modeled with a very high level of flexibility, leaving us with ample leeway to design a very flexible and extendable system, that promises to adapt to practically any data structure imaginable[36].

[36] Even circular data structures can be handled through the use of the ID and IDREF attributes, or the more advanced mechanisms found in XLink [XLINK] or XPointer [XPTR].

Other systems somewhat comparable to our efforts use XML – SOAP (see appendix E), XML-RPC (see appendix F) and XIOP (XML InterOrb Protocol, see [XIOP]) are the prime examples here. Since these are some of the systems that XOIP could be expected to provide access to, choosing an XML based solution will allow interoperability to be as simple as a transformation of the XML, for which XML StyleSheet Transformations (XSLT) would work nicely (see [XSLT]).

Architecture independent call mechanism

Since XOIP is intended to provide a cross-architecture distributed object system, the essential mechanism in any distributed object architecture, the actual invocation of methods, must be performed in a manner that provides transparency in terms of object architecture. While object lifecycle issues might make it problematic to provide complete activation/deactivation transparency, making the call mechanism independent of object architecture is a simpler issue. Most object architectures provide quite similar types of invocation and parameter passing systems, and provided that we stay within a good-sized cross-section of these, we should have little or no difficulty achieving architecture independence. This is in accordance with design criterion 7, which mandate that some amount of cross-architecture properties should be present.
The actual choice of supported parameter types is given in a later chapter, but since we've chosen XML as our package format, any structure that can be represented in XML and validated using an XML Schema, is a valid structure. We briefly considered restricting the valid XML to that representing a chosen subset of datatypes, but decided against it. By choosing to make any valid XML (valid means that it complies with an XML Schema) a valid structure, we get a high degree of freedom to model datatypes and structures. This freedom could probably lead to slightly different representations of otherwise similar structures, since the schema to be used are supplied by the object systems themselves, and could therefore be used to model object architecture specific data structures. While this may be viewed as a problem, since it will somewhat blur the architecture independence, this is actually a strength of our system. Since we can not possibly design XOIP to take care of all kinds of (current and future) data structures, an open specification based on enforceability (in terms of type safety) seems prudent. By choosing XML as package format, we achieve in effect to make "type safety enforceability" equivalent to "valid XML". On the other hand, frequently used datatypes should have a uniform representation, in compliance with design criterion 1 about a simple package format – in this way, it will be easier for programmers to construct the most frequently used remote method invocations. For now, the choice of datatypes and structures to be defined in the XOIP standard follows a cross-section of OMG IDL and Microsoft IDL, in essence a cross-section between the datatypes and datatype constructors in CORBA and DCOM.
Problems may still arise in terms of passing object parameters between architectures, and as discussed in chapter 9, there are serious issues regarding this. For now, we have decided to simply not allow objects to be passed as parameters to foreign object architectures. We realize that this implies that the programmer using the object must have information about the native architecture of the object. This is in contrast with our desire to provide architecture independence, and weakens our effort to achieve architecture transparency. However, we are of the opinion that it is better to avoid transparency where it is not entirely clear that it is called for, as also stated in design criterion 7.

11. XOIP method calls

This paper describes the way to construct method calls for XOIP. A specification of how to translate from ordinary method calls to the XML needed for XOIP to process the request will be given, along with a specification of how the related XML Schemas for validating the calls looks.

Remote/distributed method invocation

XOIP is about performing remote method invocation on a wide variety of object systems uniformly. For this reason, it is necessary that the way to specify a method call is general enough to handle all reasonable variation on how to perform these invocations. In other words it must be designed to allow for as many commonly used constructs as possible, preferably in such a way that the less commonly used ones can somehow be adapted. The one true constraint on our solution is that it must be possible to make an XML Schema (see Appendix E) specification of how to validate that the method invocation is well-formed and correctly typed. Since anything that can be converted to a string can be tested that way (the current XML Schema proposal has regular expression support), the constraint is not likely to be a serious problem.

The anatomy of an XOIP call

On an overall basis, it looks like this:

1. Specification of object instance
2. Method identification
3. List of actual parameters

Since we have chosen to use XML as the data format, the three items will naturally be in the form of XML data. The following sub-sections will describe how each of the three will look. Note that the approach we have chosen corresponds nicely with how a method invocation looks in C++ and Java (and many other languages), namely *object.method(parameters)*. In this way we intend to make it easier to use XOIP directly, since it maps directly with the way most current object-oriented languages work, in accordance with our design principles.

Specification of object instance

To invoke any method through XOIP, it is necessary to have the object identifier for the object. We have chosen to denote a method invocation with the tag `CALL`[37]. It will look like this in XML:

```
<CALL OID="<the actual object ID>">
```

[37] Other suggestions included INVOKE and METHOD. CALL was chosen for no other reason than being shortest.

Naturally this will be accompanied by a `</CALL>` after the method name and parameters.

Name of method

The name of the method is an XML entity itself, structurally a child of the CALL entity, so to call a method named *foo*, one would merely use the following XML:

```
<foo>
```

Naturally, this will be accompanied by a `</foo>` after the parameters.

List of actual parameters

The list of parameters is where the native object architectures show the clearest. The general rule is – as stated before – that anything that can be checked for validity through an XML Schema is valid. For the simpler cases, like numbers and strings, we suggest that the following is standard: The list of parameters is either a list of named parameters with their values, or a sequence of (unnamed) parameters with their values, as discussed earlier. In the former case, each named parameter is its own XML entity, with the value of the parameter as the entity value. So the named parameters *hot* (an integer of value 42) and *dog* (a string of value "Linux") would look like this:

```
<hot>42</hot>
<dog>Linux</dog>
```

In the case of an unnamed sequence of parameters, each of the parameters are enclosed in an entity called XOIP:PARAM[38], making the above example look like this:

```
<XOIP:PARAM>42</XOIP:PARAM>
<XOIP:PARAM>Linux</XOIP:PARAM>
```

Notice the use of a namespace – this is to avoid namespace pollution, as was one of our design criteria.
If any parameter has a default value and can therefore be left out, its value can simply be left out. This must naturally be reflected in the XML Schema for the call. So to use that the hot parameter above had a default value, one would use

```
<hot/>
<dog>Linux</dog>
```

and

```
<XOIP:PARAM/>
<XOIP:PARAM>Linux</XOIP:PARAM>
```

[38] The XOIP: part indicates that the entity belongs to a namespace called XOIP – se appendix G for details about namespaces.

Examples

Call the method named **Progress** on the object **AAAAAAAAAB**, and gives the **CurrentPercentage** parameter as the integer 70.

```
<CALL OID="AAAAAAAAAB">
<Progress>
<CurrentPercentage>70</CurrentPercentage>
</Progress>
</CALL>
```

Call the method named **Progress** on the object **AAAAAAAABA**, and gives the first parameter as the integer 70.

```
<CALL OID="AAAAAAAABA">
<Progress>
<XOIP:PARAM>70</XOIP:PARAM>
</Progress>
</CALL>
```

Call the method named **setCoordinate** on the object **AAAAAAAATG**, with the named parameters X, Y and Z at values 50, 80 and 100 respectively.

```
<CALL OID="AAAAAAAATG">
<setCoordinate>
<X>50</X>
<Y>80</Y>
<Z>100</Z>
</setCoordinate>
</CALL>
```

Call the method named **setCoordinate** on the object **AAAAAAAATG**, with three unnamed parameters, leaving the value out for the middle one, and setting the first and last to 50 and 100 respectively.

```
<CALL OID="AAAAAAAATG">
<setCoordinate>
<XOIP:PARAM>50</XOIP:PARAM>
<XOIP:PARAM/>
<XOIP:PARAM>100</XOIP:PARAM>
</setCoordinate>
</CALL>
```

Pointers

XOIP can not handle memory pointers. Due to its entirely remote nature, no memory pointer has meaning on any other machine without making some kind of distributed memory system – hardly an effort central to XOIP. If the equivalent of a pointer MUST be implemented, we'd

suggest that an object could be created holding whatever the pointer should point to, and the object reference be sent as a parameter instead. At some level, an object identifier IS a pointer anyway.

Object references

XOIP does, however, support the use of distributed object references as parameters. These are always returned from the XOIP server to the client as a textual (stringified) representation, and can simply be copied and inserted as string values. To invoke the method registerObject on the object AAAAAAAAAA, taking a single parameter named whatObject, whose value is the object reference AAAAAAAAFF, the following could be used

```
<CALL OID="AAAAAAAAAA">
<registerObject>
     <whatObject>AAAAAAAAFF</whatObject>
</registerObject>
</CALL>
```

It is up to the XOIP server to handle any object reference visibility problems (discussed in chapter 9), and the call will fail if the AAAAAAAAAA and AAAAAAAAFF objects are not able to see one another. This is an attempt to make a reasonable solution to the question of architecture independence we discussed in chapters 7 and 8. Our solution requires that the programmer has prior knowledge about the visibility problems the two objects might have, and whether this is good or bad is touched upon in chapter 8.

Data structures

Most object systems allow some form of data structure to be passed as parameters and be returned from method calls. This also means that XOIP must allow for it too. Fortunately, XML is particularly well suited for this. As stated earlier, the only real limitation is the ability to perform validity checks by using an XML Schema. But as we also concluded earlier, it would be beneficial to give the most commonly used datastructures a consistent look.

The datastructures defined by XOIP are dealt with in the following sub-sections.

Records

Very similar to **struct** in C (see [ANSIC]).
The record is translated to XML by simply making an item for each data member. The following example (written in C++) should illustrate this concept:

```
Struct Person {
     int age;
     string name;
     bool isMale;
};
```

Assuming that we'd want to encode the data for one of the authors at the time of writing, we'd end up with the following XML data:

```
<Person>
     <age>31</age>
     <name>Allan Bo Jørgensen</name>
     <isMale>true</isMale>
</Person>
```

To call a method named **DoubleSalary** on the object **AAAAAAABBC** giving the above XML as the sole unnamed parameter would look like this:

```
<CALL OID="AAAAAAABBC">
<DoubleSalary>
<XOIP:PARAM>
       <age>31</age>
       <name>Allan Bo Jørgensen</name>
       <isMale>true</isMale>
</XOIP:PARAM>
</DoubleSalary>
</CALL>
```

There is no need to surround the **age**, **name** and **isMale** tags with a **Person** tag, since this is implied by the declaration of the **DoubleSalary** method. One could probably make a case for allowing it anyway, but we've decided against it to simplify the check for validity.

Please note that records are NOT slated for implementation in the first version of XOIP.

Arrays

An array is denoted using an ARRAY entity, which optionally takes the array length as an attribute. The ARRAY entity has the array elements as sub-entities, and these can be named as desired. We considered demanding that they should be called ELEMENT, but this would introduce a lot of overhead in terms of package size. Since no single (short) name appealed to us, we simply decided to leave the name open, and leave it up to XML Schema. Our recommendation is quite simply to allow whatever name the programmer decides, but to demand that the same name be used for all elements. An example:

```
<ARRAY length="5">
<ELM>5</ELM>
<ELM>10</ELM>
<ELM>15</ELM>
<ELM>20</ELM>
<ELM>25</ELM>
</ARRAY>
```

Please note that arrays are NOT slated for implementation in the first version of XOIP.

Return values

The final part of an XOIP method invocation is getting a return value. The return value consists of both the actual return value and the value of any output parameters.

A return value is wrapped in a RETURN tag. If the invocation provides no output parameters, the value is simply stated between the opening and closing RETURN tags. If output parameters are present, the return value and output parameters are listed as sub-entities of the RETURN entity, with the return value wrapped in a XOIP:RETURN entity to avoid namespace pollution, as our design criteria mandates. The following examples should shed some more light on this:

An invocation returns the single result of 42:

```
<RETURN>42</RETURN>
```

An invocation returns the string “Linux” and an output parameter named OSquality returns the string “Very high”:

```
<RETURN>
      <XOIP:RETURN>Linux</XOIP:RETURN>
      <OSquality>Very high</OSquality>
</RETURN>
```

It may be argued that the former case could also read

```
<XOIP:RETURN>42</XOIP:RETURN>
```

or even

```
<RETURN>
      <XOIP:RETURN>42</XOIP:RETURN>
</RETURN>
```

Good cases can be made for these points – they seem to provide a more uniform model. We have nevertheless decided to use the chosen method to minimize the overhead-to-data ratio in a return value, in order to reduce the bandwidth requirements for the return value – this is one of our design criteria. It is our experience that most methods return a single value, and that the protocol should be designed with that in mind.

Exceptions

Errors are signaled through an exception system. Whenever an error is reported to XOIP, or arises during XOIP server processing, an exception is returned instead of a return value. The exception is to be considered a specialization of a record structure and provide the necessary information for the client to be able to handle the situation in a structured manner. An exception is meant to look as exemplified below:

```
<EXCEPTION>
     <XOIP:noSuchObjectException>
          <OID>AAAAAAAAFF</OID>
     </XOIP:noSuchObjectException>
</EXCEPTION>
```

Notice that the use of an XML namespace in XOIP:noSuchObjectException is an indication of the type of exception. In this case, it means that XOIP discovered during method invocation parsing that the supplied object reference AAAAAAAAFF does not reference an existing object. This use of namespaces is meant to provide enough information to decide what part of the system first realized the error situation. For instance, an exception indicating that the following invocation

```
<CALL OID="AAAAAAAAAA">
<registerObject>
     <whatObject>AAAAAAAAFF</whatObject>
</registerObject>
</CALL>
```

failed due to a lack of memory, could look like this (assuming that the object referenced by AAAAAAAAAA is of XOIP class ObjectList)

```
<EXCEPTION>
     <ObjectList:outOfMemoryException
     xmlns:ObjectList="XOIP://ObjectList"/>
</EXCEPTION>
```

The xmlns attribute is the standard way to handle namespaces and is detailed in appendix G. Notice that the actual URI for the namespace is fictional, but is likely to be useable.

In this way the exception structure can be parsed for the information needed to transform the XOIP exception to whatever means of reporting errors used in the client's programming language and architecture, in accordance with our design criteria about low architectural overhead. If no abstraction is present on the client the XML representation of the exception can simply be parsed, in accordance with our design criteria about low memory footprint on clients.

XML schemas for XOIP calls

The main vehicle for providing type safety and checking is through the use of XML Schemas. These must be defined in such a way that they fulfill two major roles, namely parameter correctness check and providing type information to clients. Furthermore, they must abide by the guidelines given in the previous sub-section about representation of common datatypes. Apart from that, there are no restrictions.

The following two sub-sections will elaborate somewhat on the roles of the XML schemas.

Parameter correctness check

The XML schema must allow a standard schema-compliant XML parser to correctly parse only those invocations that are typesafe for the object. This means that the schema MUST provide enough information for the parser to decide if the given method exists for the object in question, and whether the list of parameters provided matches with the formal parameter-list for the method. If any parameter is given without a value, the XML schema must provide a default value. Note that there are no real restrictions on **how** to make such a schema, other than it being compliant with the XML schema standard (see [XMLSCHEMA0], [XMLSCHEMA1] and [XMLSCHEMA2]).

Provide type information to clients

Any client needing to invoke a method on a given object must be able to deduce, by looking only at the schema, how to provide correctly typed parameters, their direction and the type of return value. Furthermore, clients must be able to get a complete list of methods available for an object solely by looking at the XML schema.

Part Four – XOIP implementation

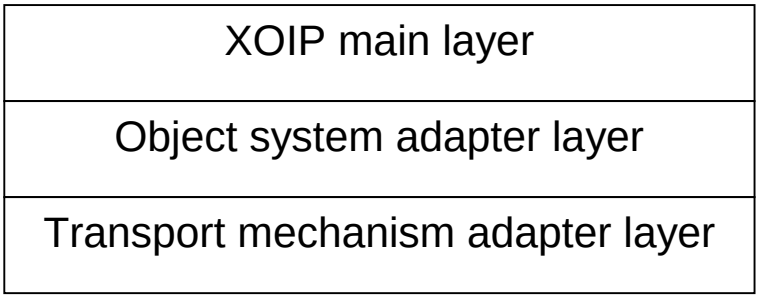
12. Modules and interfaces
13. Details from the proof-of-concept implementation

12. Modules and interfaces

This chapter describes the modules present in the XOIP system. It presents the layering model of XOIP along with the interfaces the layers expose. It further describes the responsibilities that the individual modules have in the XOIP system. Emphasis has been on achieving orthogonality and a very simple interface.

Layers

The XOIP system consists of three layers, as shown below.



The following sub-sections will describe each layer separately. Note that functionality concerning module maintenance (modules starting/stopping, retrieving modules based on name, etc.) are not detailed here, but naturally needs to be in place.

XOIP main layer

The XOIP main layer is responsible for providing the following services:

1) **Server for the XOIP boot object**
   The boot object provides elementary remote method services such as connecting to a specified object, requesting an XML Schema for an object or class of objects and closing a connection to an object. The boot object is special in that it is served directly on the XOIP server. To make the boot object operable a name server is required for mapping between object class names and actual object servers. The boot object is always connected to any client, and can not be disconnected. Note that the actual task of connecting to an object (creating one, if needed) is entirely up to the object system adapter (OSA) for the given object – the XOIP boot object simply passes the request to the relevant OSA. The boot object implements the following methods (declared in an OMG IDL-like fashion):

```
Connect(in String className, out OID objectID)
Disconnect(in OID objectID)
Schema(in String className, out String XMLSchema)
Cast(in OID objectID, in String className)
Copy(in OID objectID, out OID copyObjectID)
```

For now, all invocations are synchronous[39]. An example XML snippet that connects to a **logging** class object could look like this:

```
<CALL OID="BOOT">
<CONNECT>
   <CLASSNAME>logging</CLASSNAME>
</CONNECT>
</CALL>
```

The return value would look like this (assuming the call was successful):

```
<RETURN>AAAAAAAAAT</RETURN>
```

Note that the returned object ID is fictional and is likely to be different.
The Schema call is for requesting the XML Schema related to the class given, and disconnect is for reporting that a given object is no longer needed. The Cast and Copy operations are for type-casting an object and for making a new reference to the same object. These have not been implemented in our proof-of-concept implementation, and will not be discussed further. No attempt will be made to cater for these functionalities in the sub-sections dealing with the OSA and TMA layers.

2) Dispatch method calls

When remote method invocation requests arrive at a TMA, they are passed to the XOIP main layer. At this layer it is decided what OSA the request is to be passed on to, validation of type safety is performed, parameters are marshaled (if needed), the object identifier is reverted to its original state and the call dispatched to the chosen OSA. Note that due to the modularity of XOIP there is no inherent problem with receiving, for instance, native CORBA calls and dispatching them, in effect turning the XOIP server into a proxy CORBA server. Naturally this applies to all object systems and can optionally be implemented (we have chosen not to do so). The interim language used to communicate between the XOIP main layer and the OSA is chosen to be identical to the XML used for method invocations and return values, except for the object identifier which is reverted to its original state (as mentioned above). This means that the XOIP main layer must provide a single point of entry preliminarily named **Invoke**, which looks like this:

```
Invoke(in String XMLrequest, out String XMLreply)
```

The choice of format may change as the implementation proceeds – for performance issues, we may favor some XML interpreter dependant way even if it means sacrificing some of the module (and layer) orthogonality.

Object System Adapter layer

The OSA layer is where the bulk of the processing in the XOIP system takes place. This means that the OSA's in the system must have functions available for doing the basic operations on XOIP objects, namely

- Connection (creation)

[39] The reason is simplicity in terms of implementation.

- Disconnection (deletion)
- Invocation
- Schema request

Furthermore, it needs to decide whether any two objects are, in fact, able to address one another (see chapter 9). So any OSA must support 5 functions, and these will be elaborated in the following subsections.

Connection

To be able to invoke methods on an object, you need to somehow be able to contact the object. We have chosen to call this “connecting” to the object. The connection-function takes the native classname as input, and produces a native OID (Object IDentifier), pointing to an instance of the requested class, as output. For now, the connection request is always the result of calling the Connect method on the boot object, with the classname translated to be Object System Adapter specific.

Disconnection

When an object is no longer to be used, you need to disconnect from it. Note that this is currently necessary even if the native object system implementing the object does not need an explicit disconnection to get deactivated or garbage collected – the XOIP server needs the explicit disconnect to clean up internally (the object store). For now, all disconnect calls are the results of calling the Disconnect method on the boot object.

Invocation

Method invocations arrive at an OSA as XML snippets with the object identifier for the object on which to invoke the method reverted to (a stringified version of) its native version. The OSA is responsible for converting the XML invocation to an actual remote method invocation, invoking the method and parsing up the result as XML. In the CORBA OSA this means constructing a call through DII[40] and parse up the result to an XML snippet. In the COM OSA this means construction a call through the IDispatch interface and parse up the result to an XML snippet. The XML is then used as the return value. For this reason, the OSA implements an **Invoke** method similar to that of the XOIP main layer, which gets called exclusively by the XOIP main layer.

Schema request

There are two ways in which a schema can be requested from an OSA. It can either happen as a direct result of a call to the boot object’s Schema method, or as a result of the XOIP main layers Invoke function, which needs a schema to decide whether the parameters it received are correctly typed. In both cases, the call takes as its single parameter the native classname for which the schema is requested.

Visibility

[40] Dynamic Interface Invocation

When passing an object as a parameter, it needs to be decided whether or not the called object and the parameter object are actually able to call the other's methods. For this reason, the Visibility function can be called with the native classname of the called and parameter objects respectively. See chapter 9 for a more thorough discussion of the visibility problem.

Transport Mechanism Adapter layer

A Transport Mechanism Adapter (TMA) is responsible for sending/receiving network packages using a given transport mechanism on a given network hardware. All transport mechanisms, regardless of their actual physical incarnations, must provide a connection-oriented reliable transport mechanism for XOIP to use. To this end, all TMAs must provide the following functions:

- Connect
  Establishes a client connection to a remote machine. Takes a single parameter containing the address of the remote machine, and returns a connection identifier upon successful execution.

- Close
  Closes a connection. Takes a single parameter containing an identifier of the connection to close.

- Send
  Sends a buffer to a remote machine. Takes the connection to transmit over as one parameter, and the data to send as another parameter.

- Receive
  Receives a package of data from a connection. Takes the connection to receive from as one parameter and the buffer to receive into as another parameter. Blocks until data arrives.

- Server
  Establishes a server connection, that is, prepares the server to accept connections.

- Wait
  Lets the server to wait for a connection, and accept a request for connection. Returns a connection identifier for communicating with the connected client, and fills up a buffer with information about the client's identity.

While looking rather innocent, the actual implementation of these functions can be very complicated. For instance, to have a CANbus (see chapter 2) provide such a level of transport abstraction, substantial effort is needed in terms of programming, as we discussed in chapter 3. For this reason, we have chosen to only implement a TCP/IP TMA during our project. It is furthermore complicated by the fact that the client can be very thin, thereby reducing even further the resources available on the client to facilitate the protocol. This means that the protocol implementation will be "lopsided" in that the majority of the protocol logic must be handled by the TMA. While this is not necessarily a problem in terms of making the protocol work, it can make it somewhat difficult to reuse implementations of those existing protocols that focus on peer-to-peer networking.

13. Details from the proof-of-concept implementation

We have had to select what parts of the XOIP feature set to implement for our proof-of-concept implementation. We would have liked to implement XOIP fully, but not enough time was available to us. Naturally, we have chosen to implement what we think is the core of XOIP, and to not us excessive amounts of time polishing our implementation. We were, however, forced to optimize parts of the code considerably, since part of what we want to prove is that embedded hardware with limited resources is quite capable of participating in a distributed object system – this can be quite hard with bloated non-optimized code that takes up a lot of room. Instead, we have worked on reducing the memory footprint of our implementation.
The following sections will each address an issue we faced, and explain our rationale for the decisions we made.

Operating system

We decided NOT to use a Real Time Operating System (RTOS), even though we've discussed doing so in chapter 1. We decided to focus on some Linux variant instead, since we'd like to focus our attention on making a better XOIP. We know Linux already, and have no experience with an RTOS. So it is simply a matter of reducing the number of unknowns. Since Linux can expose the POSIX.4 (see [PSX]) interface like most RTOS implementations also do, we can simplify porting to an RTOS by staying within the POSIX.4 boundary. The POSIX.4 standard is primarily concerned with concurrency and hard real-time issues, which are the issues that arise when a task MUST get the chance to run within a specified amount of time, regardless of the other tasks running.

Actual embedded hardware

When we first started working on what would become XOIP, we were quite certain that we needed (and wanted) to implement parts of what we were to suggest. Since we were dealing with embedded systems, it seemed inevitable to us that actual embedded hardware should be used. After doing the survey work documented in chapters 1 through 3 we decided to use a uCSimm (see [UCSIMM]) module for the following reasons:

- Real embedded hardware
  As we concluded in chapter 1, it would be prudent to choose hardware with a strong resemblance to earlier desktop models. This uCSimm uses a Motorola 68328 processor, one of the embedded processor most widely used, and it is essentially a modified desktop processor used in old Apple Macintoshes. Since the very popular PalmPilot™ PDA uses almost identical hardware, it could be argued that this is indeed embedded hardware. Furthermore, we were recommended to look in that direction by an industrial partner in the COT project since they use these processors themselves.

- Low on computing power, but not intolerably so
  The uCSimm modules we use run at 16 MHz. This is quite comparable with many of the processors currently used for embedded devices. Some would argue that it is somewhat high, and others that it is somewhat low – it depends greatly on the task the device is to perform. On an overall scale, it seems a reasonable choice, partly because its low speed

(compared to the systems we use on our desktops) will prevent us from making decisions that lead to code that executes too slowly. This was one of the key reasons we forewent faster versions of essentially the same hardware. Note that the uCSimm is intended to host XOIP itself, not serve as a prototypical client.

- Sufficient storage
  The uCSimm has room enough to hold the operating system and our server program entirely in ROM. It also has 8 Mb of RAM, which is quite a lot for an embedded system. This should not prevent us from making a small program – it is quite easy to measure the footprint of a program and decide from that if a program is too big. We realize that we could be tempted to use too much memory with 8 Mb, but as we'll discuss later this didn't turn out to be the case.

- Runs uClinux (micro-controller Linux, see [UCLINUX]) with a good cross compiler
  This means that our Linux desktop machines could serve as development machines, and we could the trust the cross-compiler and uClinux combination to handle the move to the embedded platform when major milestones mandated testing - this works absolutely seamlessly and makes development really simple. While this is true for some RTOS'es too, it is usually in the form of simulation of the embedded hardware and RTOS combination, which usually requires a substantial investment in turns of software purchases. In the case of uClinux, all the software is open source and consequently free of charge.

Only core features were implemented

Certain features were omitted to allow the core features to be implemented. The core features include

- Connect, Disconnect and Schema on the boot object
  Connect is the way to get hold of an object reference and Disconnect releases an object reference and therefore both are essential. Schema is not entirely necessary, but included because all the logic to implement it was present due to the following point

- XML Schemas used for type verification
  Using schemas to validate actual parameter values is essential to providing the flexible type checking mechanism we desire. For that reason we included this in the core features.

- TCP/IP transport mechanism adapter
  We needed at least one transport mechanism adapter, and we chose TCP/IP. This was a simple choice since using TCP/IP provides us automatically with the level of transport abstraction we've described as necessary in chapters 2 and 4. We would have liked to include CAN support, but as we explained in chapter 2, this would require a lot of programming that is non-consequential to the task of writing an XOIP implementation for proof-of-concept.

- COM and CORBA object system adapter
  To demonstrate any level of object system architecture transparency, at least two object system architectures are needed. Since CORBA and COM are the ones most familiar to us, they were chosen over all other candidates. We briefly considered including Java RMI

into our collection of object system adapters, but decided against it to simplify development. Furthermore, two object systems should be sufficient to demonstrate the object system architecture transparency we have designed.

- Only basic types supported (integer, float, string, object) in the object system adapters
  This was chosen to simplify the task of writing the object system adapters. Since the groundwork of the system to support more advanced type constructions is essentially in place when basic types are supported, this omission does not yield serious problems.

- Rudimentary schema support
  Full schema support is very processor-intensive, and requires a large program package to handle. Since none of the existing packages we looked at ported easily to anything but desktop systems with fast processors, we decided to implement a different schema system, which is much simpler. It's organized around the same principles as genuine XML schemas, but uses far simpler syntax rules and does away with the more advanced data verification system. This mandates a lot of changes to the schema verification system before the system is ready for production, but since we've limited ourselves to basic types only (previous item), the chosen system is adequate. Since the schema handling on the XOIP server is abstracted into a few functions, changing it will not be too problematic. Note that the schema support functions are likely to need replacement in any case, since the XML Schema standard was only a recommendation at the time this thesis was written.

- No server concurrency
  The XOIP server does not currently handle concurrent requests. This is a simple decision to reduce the number of unknowns in our implementation, and should naturally be fixed in production code.

- Primitive error handling
  The errors encountered during processing are simply returned as an XML error entity, consisting of a single sub-entity that holds a textual description of the error. This should be replaced by an exception-like system in production code.

- Object Visibility problems are avoided by disallowing calls that could potentially cause the problem
  While not exactly the flexible solution we would have liked to make, it does make some sense. First of all, if the parameter and called objects are not supported by the same object architecture, the parameter is dismissed and an error occurs. If the parameter object is within the same architecture as the called object, it is left up to the object system adapter in question to decide whether they are visible to one another. They could be on different protocol segments, for instance (as described in chapter 9). This approach could conceivably be extended to allow a more flexible approach, where the amount of object architecture transparency that can be made to function, can be allowed. For instance, DCOM and CORBA have bridges between them already, and these could be put to use by leaving the decision entirely up to the object system adapters, and allowing these to take whatever measures needed.

- Parameters must be named and their values must be given
  Default values for parameters and the ability to give actual parameters anonymously were not included in the feature set. These features are not essential to process a method

invocation, and as such were simply omitted. Parameters must be given in sequence and by name.

- Nameserver data is kept in a text file
  The list of classes that XOIP allows clients to connect to is kept in a text file, which is simply parsed upon server startup. This means that the list is very easy to maintain, but that the server needs to be stopped and started to accommodate changes. Hardly a problem for a proof-of-concept implementation, but a more sophisticated system might be a good idea in production code, especially in terms of dynamically adding classes.

- All parameters are treated as “in” parameters
  No feature to handle directional attributes of parameters has been implemented.

The remaining features were left out. Note especially that casting and reference copying were left out. To some extent, this limits the modeling capabilities of our system. As stated in chapter 8 there are issues left unresolved with regards to this, which left us in doubt of how to design this system. Since we didn’t want to include a feature we were not sure had been properly designed, we decided against it.
This does not render our proof-of-concept implementation useless. Our aim is primarily to construct a protocol that allows embedded systems to participate in distributed object systems across object architecture boundaries, and this protocol is primarily concerned with obtaining object references and method invocations. Since this does not mandate casting, our primary goal can be met without including a casting feature.

Memory usage

We have heavily optimized the XOIP server implementation with regards to memory usage and speed – in that sequence of importance. While we were able to keep the strict interfaces described in chapter 12, this optimization has rendered the code modules that in unison makes up the full server somewhat obfuscated. Central pieces of code are very heavily optimized, and the XML parser is the extreme example in this respect. It uses the XML itself as its only buffer, and literally cuts it to pieces during parsing, in that way avoiding copying any of the data it parses.

By applying this approach liberally, we have been able to reduce the actual memory footprint of the XOIP server to just 37 Kb for program data, 50-150 bytes of data per active object and whatever room the schemas for parameter verification takes up (typically 10-20 bytes per method and parameter). It is parsed into a kind of parse tree, with a structure that makes it easy to use the parse tree as instrumentation data for the XML parser. These parse trees are cached from call to call to speed up processing, but this can easily be modified to including the option of disallowing caching, forcing the server to retrieve the schema before it verifies the parameters. This would be necessary for object architectures such as Self and JavaScript, where the set of supported methods can vary from invocation to invocation. Another approach would be to simply not check anything on the XOIP server, but leave the checking to the object architecture to be called.
The XML data for a single call is copied during the marshalling of object parameters, but this is done in a temporary buffer, which is released immediately after use.

Note that having the XOIP server use the XML as its only buffer automatically makes it easy to make the XOIP server multi-threaded at a later time. The XML is temporary data that arrive through a network connection – a socket, in our proof-of-concept implementation. The data thus received have no validity after the call has been processed, and it is therefore safe to destroy it while processing it. Since this processing involves no other data areas than the XML buffer itself and stack-allocated local variables, concurrent access to data will not happen during processing. Server internal data structures must be protected, however. We have not delved further into this issue, but it should be resolved in production code.

Data structures

All lists are kept as simply forward-only linked lists. Not very efficient, but simple to implement. Some more advanced data structure would be a good idea, especially when dealing with many objects. All objects are kept in a single linked list, and it would most definitely speed up processing to use a more advanced data structure.
The nameserver list is kept as a simple ASCII text file, and is parsed upon starting the XOIP server. No feature exists to re-read the nameserver list after the server has been started, or to dynamically adding nameserver entries. Removing them dynamically would be a very unsafe action to perform, since no guarantee can be made that nameserver data is not needed during processing of already-instantiated objects of the class being removed. In fact, such access **is** needed in our proof-of-concept implementation.

Schema handling and type verification algorithm

Compiled schemas are kept as tree structures, which will be detailed shortly. The schemas themselves are very simple, and are in fact more accurately described as templates. For instance, a class with the following IDL-declaration

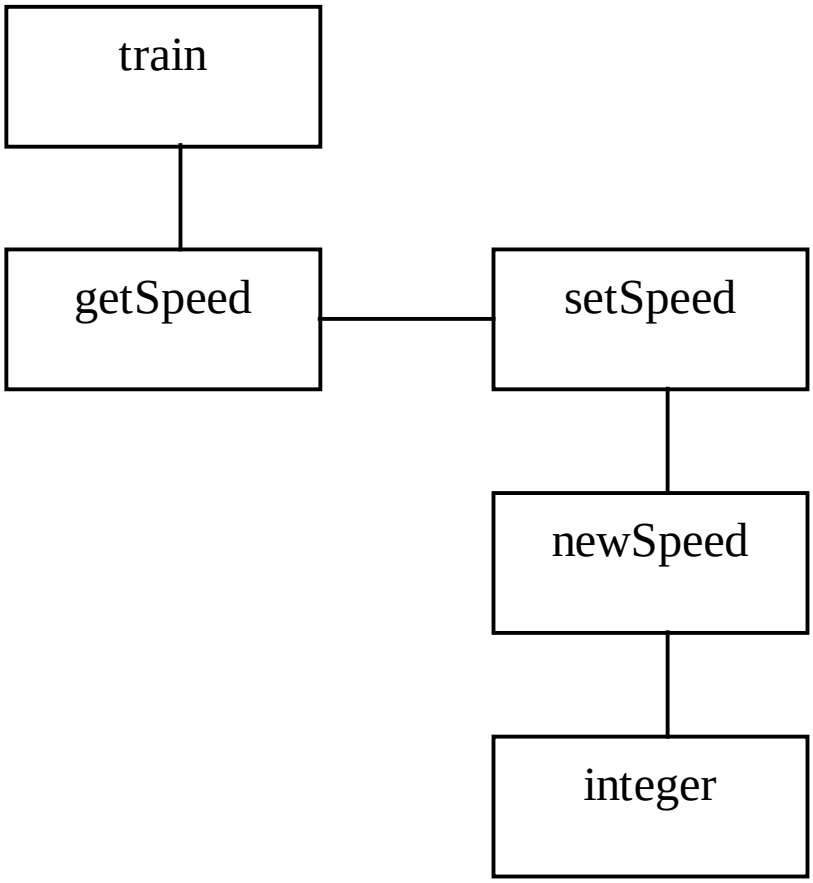
```
interface train {
     short getSpeed();
     short setSpeed(in short newSpeed);
};
```

will have a schema looking like this

```
<train>
     <getSpeed/>
     <setSpeed>
          <newSpeed type=”integer”/>
     </setSpeed>
</train>
```

Notice the absence of return types – this reflects the fact that for our proof-of-concept implementation the schemas were designed exclusively to handle parameter type checking, and for that purpose knowing the return type is not needed. When it is to be used for allowing the client to make a call type-safe, this should be remedied. This could be done easily enough by augmenting the `<getSpeed/>` and `<setSpeed>` entities with types, but for our proof-of-concept implementation, return types were not needed.

Internally, the following structure will be kept:



Internal representation of a schema

The type verification works like this. Whenever a method invocation is received, the <CALL> entity is parsed to decide what class of object is being invoked. If the class already has a compiled schema in store, the compiled schema is used. If not, one is requested from the object system adapter and compiled.
The train entry into the compiled schema structure is generally not used, since the knowledge that this is indeed an object of class train is known through the parsing of the <CALL> entity. Going directly to the getSpeed node, all nodes on that level represents methods (they are internally called siblings). The level is searched for a method matching the name of the method given in the invocation. When it is found, the first parameter is directly below the node and if more than one parameter is to be given, these will be siblings of the first parameter. The list of parameters is traversed and for each parameter, the type of the actual parameter is given as a child node. A check is then made for type safety. For the simple types, this involves just checking whether the string data provided in the XML has a certain format. For an object, things are handled differently. First of all, it is checked that the object reference given actually references a live object. If this is the case, it is checked that the invoked object

and the parameter object can “see” one another. In other words, it is decided whether they will suffer from visibility problems as described in the “Object Visibility” chapter.
While this is happening, the call is also being marshaled into a secondary buffer. This is only necessary in case the invocation actually contains an object reference, since these needs to be marshaled to the object reference used in its native object architecture before the call is delivered to the object system adapter (for module orthogonality reasons). All parameters in all invocations are always marshaled in the current implementation. This makes for simpler code, but it is also a consequence of using a destructive XML parser, since the buffer is lost after parsing and can therefore not be used to transmit the invocation to the object system adapter.

Notice that the compiled schema is very reminiscent of an XML DOM representation of the schema XML.

Architecture and language bindings can be automated

The XML Schema for an XOIP class contains enough information to perform type check. This implies that the client’s compiler should be able to do this, provided that the Schema can be presented in a form that the compiler can handle. Since all the necessary information is readily available through XOIP, this presentation can be automated. Both in terms of language binding (we have hand-coded the template for an automatic XOIP-to-C++ translator, which can be downloaded alongside the source for XOIP), and in terms of architecture. The latter is achieved by making tie objects (discussed in chapter 8) in DCOM, which delegates to the XOIP server for implementation and by having the XOIP server (or possibly some other server) act as CORBA server through the use of DSI (Dynamic Skeleton Interface).

14. Related work

Interoperability has been the topic of much practical interest. The advent of the web and the way applications could take advantage of this has sparked a renewed interest in specifying formal definitions of topics involved. Among these different implementations, we have focused on the ones treated in the following sub-sections.

XML-RPC

This technology was originally developed to be used in a content management application, Frontier. By using XML on top of a HTTP based RPC mechanism, XML-RPC was able to attract a number of developers, because its simplicity.

The message format in XML-RPC is rather simple. Appendix B covers some more details on XML-RPC.

SOAP

Simple Object Access Protocol was originally developed by a couple of developers from DevelopMentor, and Microsoft and IBM soon joined them. When the specification had stabilized, it was submitted to the World Wide Web Consortium.

The SOAP specification describes a message format, and a specific binding for XML based RPC. While the O in SOAP do represent object, it is to some extent a misnomer, since the specification does not deal with object references and invocation on them. We consider this a grave omission from the standard, if SOAP is to be used for interoperability.

A more detailed treatment of SOAP can be found in Appendix C.

W3C/IETF XML protocol

The XML Protocol Working Group was formed after SOAP was submitted to the W3C[41]. Experiences gained from XML-RPC, SOAP, WebBroker etc, suggest that XML based remote procedure call and messaging systems build on top of existing protocols like [HTTP, SMTP] can effectively meet the requirements for intercommunication among web applications. Since the web is heterogeneous by nature, some general requirements must be met. Specifically:

- The envelope and serialization should not be restricted to a certain programming model and must not rely on a particular method of communication
- Simplicity and modularity should be in focus

Based on these general requirements, the working group has identified four components that must be provided by the XML protocol:

[41] The Working group was formed in September 2000 and the draft requirements where published in November 2000.

- An extensible envelope used to encapsulate XML data. Individual applications should be able introduce features and extensions
- A convention for representing the content of RPC messages
- A mechanism for marshaling data such as object graphs an directed graphs, based on the XML Schema specification
- A mechanism for using HTTP as transport, since HTTP is widely used today

The requirements defined by the XML Protocol Working Group have been defined in parallel with the design and implementation of XOIP. Although similar in ideas, there is one central issue on which we differ: in our view object references are central to object systems, and we believe that it is an important omission to leave out[42] in favor of RPC-like mechanisms. Nevertheless, it appears that the continued efforts of this working group will provide a solid foundation with industry-wide acceptance.

CORBA/SOAP Interworking

XML has been the focus of OMG for some time (see [CORBAXML]). In response to the growing number of request for interoperability between CORBA and non-CORBA systems, work has begun on defining an interoperability layer based on XML. SOAP is gaining widespread support as the middle format in many business-to-business (B2B) scenarios. It is important that CORBA based systems are able to seamlessly take part in such scenarios, i.e. to allow calling CORBA objects via SOAP.

This has lead to a request for proposals defining the extent of the intended system: Defining a protocol (marshaling format and message exchange) and mapping a limited set of objects (as implied by the requirements). The Mandatory requirements for proposals are (taken from [CORBASOAP]):

- Support for the full set of IDL types defined in CORBA 2.4
- Support for the semantics of CORBA invocations, including service contexts.
- Use the SOAP extensibility framework (without changing the SOAP protocol) and track ongoing W3C work.
- Define an IOR profile for SOAP
- Provide an interoperability solution that permits native SOAP clients to make invocations that are processed by CORBA servers; that is, present a SOAP view, (as defined in CORBA 2.4 section 17.2.3) of a CORBA service to a CORBA unaware SOAP client.

Proposals shall NOT propose changes and/or extensions to the CORBA object model or to GIOP except as follows:
- definition of new profile tag(s)
- description of any new component tag(s)

The adoption of SOAP as the underlying message format lends some credibility to the SOAP specification as a general message format and RPC-mechanism. It is still too early[43] in the

[42] Support for object references can be built on top of a RPC like mechanism
[43] Submission of proposals is due February 5, 2001.

process to make any conclusive remarks about this undertaking. It appears to be backed by a large percentage of CORBA vendors, which should ensure rapid adoption

CORBA/COM interoperability

The OMG specifies a way to transparently bridge DCOM and CORBA. This allows application architects to design solutions comprised of the best systems for a particular task. Typically scenarios are COM/DCOM used for a Windows based presentation tier, while CORBA is used in the middle-tier and back-end.

To make the integration between DCOM and CORBA transparent, some kind of bridging mechanism is required to translate data types and object references. A bi-directional bridge requires that we must be able to [COMCORBA]:

- Transparently access objects in one system from the other.
- Treat data types from one system as if they were native types in the other.
- Maintain identity and integrity of types passing among the system.
  If a CORBA reference is passed as a parameter into a DCOM and later returned to CORBA, its object reference should remain valid.

The “OMG COM/CORBA Interworking Specification, Part A + B.” describes what bridging between CORBA and COM/DCOM means. It does however not specify how to implement the bridging mechanism. Static bridging requires statically generated marshalling code, i.e. proxies and stubs, for each exposed interface. Dynamic bridging generates marshaling dynamically.

XOIP shares some of the characteristics of DCOM/CORBA bridging, and would likely benefit from further studies on deploying applications based on DCOM/CORBA. XOIP attempts to provide a general interoperability mechanism, whereas DCOM/CORBA bridging are focussed solely on DCOM and CORBA.

Generalized dispatching

In [GD], some amount of cross-architecture is achieved. This has been dealt with in chapter 10, and we shall merely conclude that their system attempts to be a “Grand Unified Theory” of programming paradigms and architectures, but that it would probably work if programmers were to stay within the confines of their DOM language.

XIOP

XML Inter-Orb Protocol (XIOP) is an attempt to merge the flexibility and extensibility of XML with the facilities of CORBA. It is a CORBA Environment-Specific Inter-Orb Protocol (ESIOP) compliant with GIOP mappings. XIOP uses HTTP 1.1 as the transport layer and XML 1.0 for content encoding. Both XML and CDR (the Common Data Representation used by IIOP) encoding can be send across the transport layer.

XIOP was a promising OpenSource project, but since the Object Management Group began to engage in integration of XML and CORBA, interest has declined. The goal of XIOP, to make CORBA objects easily accessible on the web, is markedly different from the interoperability goals we pursue.

Conclusion

We set out to explore how embedded systems and distributed object systems could be integrated. We have demonstrated that the approach described in this thesis can be used to satisfy the demands for increasing integration of heterogeneous distributed object systems. We have done so by devising an interoperability mechanism based on XML as the message format. This provides a high level of flexibility and extensibility; features such as security, transactions and micro payment can be accommodated without significant changes to our approach. In fact the actual message format itself is of minor importance, e.g. the message format of SOAP could easily be used instead, or some variation that is less verbose.

Advocating XML as the underlying message format does not impose a certain programming model on applications using this approach. Since messages composed of XML represent a hierarchical data structure or the state of an object hierarchy, building a suitable abstraction layer on top of the message format and transport is simple. This frees the application programmer from worrying about the technical implementation details, allowing him to concentrate on the problem domain.
For most programming environments, such abstractions could even be automatically generated.

In embedded systems size does matter. Our proof of concept implementation demonstrates that the approach is viable on space and processing constrained architectures. In some situations the frequency of distributed messages or very limited processing power warrants a solution with less overhead. In our view the trade-off between flexibility/extensibility and a slight overhead is worth the expense.

Exposing objects for interoperability requires no interventions in the implementation of the exposed object. Similarly enabling client application to access remote objects from different distributed object architectures does not impose severe demands on the client. All that is needed is access to a generic object adapter. In our view the barriers of entry for participating systems are low.

A number of outstanding issues deserve further investigation.
How to better solve the object visibility problem and what a desirable level of distribution and architectural transparency should be in this case. We have demonstrated that it should be possible to achieve a great deal of architectural transparency at little or no cost.
If the cross-architecture inheritance system we describe turns out to be a feasible approach in practice, it would be very interesting to explore the boundaries of this.

In summary, we have devised a mechanism to allow embedded systems to interoperate with distributed object systems. Through implementation of this mechanism we have shown this to be a feasible approach. The mechanism is sufficiently flexible to allow extensions as needed.

Appendix A: Listing of considered RTOS'es.

Maruti

A research project from the University of Maryland, Maruti has a lot of features going for it. It supports hard real-time applications in a distributed environment, and has a very small (25k) kernel. Unfortunately, it supports no embedded hardware as of yet. More info at http://www.cs.umd.edu/projects/maruti/.

ALBATROSS

A research project from the University of Kaiserslautern, ALBATROSS was designed to allow mobile robots to respond in real-time to external stimuli. Unfortunately, it supports only a narrow selection of hardware (requires a specific bus, among others). More info at http://ag-vp-www.informatik.uni-kl.de/Projekte/ALBATROSS/.

Chimera

A research project from the Carnegie-Mellon University, Chimera has a feature set like most commercial real-time OSes. It supports some types of embedded hardware, but not many. It has good specs, and is a quite fast implementation in terms of kernel execution time. It is one of the few RTOSes to be a real multiprocessor OS, but it is unclear what advantage that offers on today's embedded hardware. It has good compiler/cross-compiler support, and an extensive collection of tools. An interesting product, with more information available at http://www.cs.cmu.edu/afs/cs.cmu.edu/project/chimera/www/chimera.html.

NUCLEUS

A commercial product, NUCLEUS has one of the broadest ranges of target hardware. It supports almost any incarnation of current embedded hardware, and offers a wide variety of tools. It is a quite expensive product, but since no royalties are to be paid it is not overly so for high-volume products. It exists in different versions, where an interesting one allows all kernel-objects to be accessed directly as C++ objects. Since this was compiled into the kernel, it incurs none of the usual performance penalties that OO wrappers usually do. More info at http://www.atinucleus.com/intro.htm.

VxWorks

A commercial product, VxWorks is one of the most widely used real-time operating systems. It runs on almost all current embedded hardware, and has one of the broadest selection of tools. An interesting thing about this product is the implementation of real-time DCOM, the only such implementation to our knowledge. As with all similar commercial products, it is quite expensive. It does, however, offer one of the most complete sets of capabilities in the market. Read more at http://www.vxworks.com/.

RTEMS (Real Time Executive for Military Systems)

The only product we've found that is both freeware, has some support, and runs on a reasonable range of processors. It has a full feature set and as the name implies, it was originally developed for military purposes. Some time ago it was released to the general public, and has been well received. The full source code is included, and it is the product that has been recommended to us by our industry partners. Furthermore, CORBA has been ported

to RTEMS. Read about the CORBA-port at
http://www.connecttel.com/corba/rtems_omni.html and about RTEMS at
http://www.oarcorp.com.

Appendix B: XML-RPC

Originally conceived as a mechanism for exchanging messages between instances of the content management tool Frontier, XML-RPC [XMLRPC] has since evolved into a more generic specification for making simple remote procedure calls. The strength of XML-RPC is the simplicity[44] - it was designed with three goals in mind

- Firewall friendly, i.e. no new features should be required beyond the CGI interface
- Simplicity, someone with markup language knowledge should be able to understand and use XML-RPC
- Easy to implement

In essence XML-RPC specifies a format constructing RPC messages and how to exchange these using the request/response semantics provided by the HTTP protocol.

The message (payload) format, consists of a method name and the corresponding parameters. XML-RPC supports simple scalar types (e.g. strings and integers) as well as structures and arrays. Constructing a Document Type Definition (DTD) to represent syntactically correct messages is trivial, but omitted from the specification. The message is delivered to the server using the POST method of the HTTP protocol. This implies that the object instance receiving the message can be identified by an URI.

Listing 1 shows an example of a message representing an invocation of:

```
email.getNumberNewMessages("myUserName", "myPassword");
```

Notice that establishing a connection to the server is beyond the responsibilities of XML-RPC.

```
POST /xmlrpcInterface HTTP/1.0
User-Agent: Yoyodyne XML-RPC Client 1.0
Host: xmlrpc.emailservice.com
Content-type: text/xml
Content-length: 195

<?xml version="1.0"?>
<methodCall>

<methodName>email.getNumberNewMessages</methodName>
  <params>

<param><value><string>myUserName</string></value></
param>

<param><value><string>myPassword</string></value></
param>
  </params>
</methodCall>
```

[44] *"Does distributed computing have to be any harder than this? I don't think so."* -- Jon Udell, Byte.

Listing 1: XML-RPC request

There are two possibilities for the response

1. The request was accepted and a single result is returned (see Listing 2)
2. An error occurred and a fault element consisting of a structure value with two members - faultCode and faultString is returned.

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 148
content-Type: text/xml
Date: Wed, Jul 28 1999 15:59:04 GMT
Server: Yoyodyne XML-RPC Server 1.0

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><int>10</int></value>
    </param>
  </params>
</methodResponse>
```

Listing 2: XML-RPC response

While simplicity is the biggest asset of XML-RPC, it is obvious that the restrictions introduced hereby constrain widespread adoption of XML-RPC as a general RPC mechanism. Primary shortcomings are

- No support for all but basic XML features, e.g. DTDs or schemas
- Poor type system support (everything is a type-labeled string)
- Transport protocol limited to HTTP
- No object concept in the usual sense of the word

Appendix C: SOAP

SOAP (see [SOAP]) is a natural extension of XML-RPC, addressing most of the shortcomings of the latter. The intent is to distance SOAP from any programming model or implementation specific semantics, instead focusing on facilitating simple lightweight exchange of structured and typed information. To achieve this, SOAP relies on XML and related technologies such as XML Schemas and namespaces.

Ensuring simplicity and extensibility was foremost among the design goals for SOAP. This precluded features commonly found in traditional messaging systems as well as popular distributed object architectures such as CORBA, Java RMI and DCOM. Features specifically excluded are

- Object references
- Distributed garbage collection
- Object activation

SOAP consists of three functionally orthogonal parts: SOAP envelope, encoding, and RPC representation. The envelope represents a message in SOAP. Envelopes can contain an optional header element used to add features, such as security and transactions, to messages. A body element is required in every envelope, and is responsible for describing the message itself. The only body element described in the specification is the fault element used to represent errors. The standard SOAP encoding is based on XML Schemas, but other models are not excluded.

A SOAP body is used for both request and response messages for RPC method calls. A request body is modeled as a structure with the name of the method. The structure contains member elements corresponding to the *in* and *in/out* parameters of the method signature, named and typed accordingly. Likewise the response body is represented by a structure, which is commonly named after the method name appended with the string "Response". The response structure contains members for the return value and possible *out* and *in/out* parameters. Errors are handled by the standard fault element.

Listing 3 and 4 illustrates a SOAP based request/response corresponding to a remote invocation of

```
StockQuote.GetLastTradePrice("DEF")
```

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
    xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle
        ="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Header>
        <t:Transaction xmlns:t="some-URI"
           SOAP-ENV:mustUnderstand="1">
           5
        </t:Transaction>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DEF</symbol>
       </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 3: SOAP request with mandatory header field

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
    xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle

="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Header>
        <t:Transaction xmlns:t="some-URI"
            xsi:type="xsd:int" mustUnderstand="1">
            5
        </t:Transaction>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <m:GetLastTradePriceResponse xmlns:m="Some-
URI">
            <Price>34.5</Price>
        </m:GetLastTradePriceResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 4: SOAP response

SOAP strikes a good balance between simplicity, flexibility and extensibility. The lack of support for object references makes implementing facilities such as naming and trading services harder. Adapting existing distributed systems to SOAP also suffers from the same limitation.

One of the most interesting SOAP implementation is SOAP4J from IBM. SOAP4J demonstrates the modularity of SOAP by providing encoding rules based on XML Metadata Interchange (XMI) as well as the standard XML Schema based encoding rules. In addition to this SOAP4J can be used with the Simple Mail Transfer Protocol (SMTP). Another interesting element of SOAP4J is XIDL [XIDL] which provides a way of describing interfaces to SOAP enabled objects using XML.

Appendix D: XML/Value

In pre CORBA-2.3 representing complex data types such as trees, graphs and recursive structures was cumbersome. The types supported by CORBA at that time were simple scalar types, structs, unions etc. as well as the object reference. When using XML with CORBA one was forced to serialize and manipulate a flat string representation of and XML instance. The value type introduced in CORBA 2.3 made it possible to represent arbitrary graph structures, i.e. essentially the same as XML. This prompted OMG put out a request for proposals [XMLVALUERFP] to define a standard way for representing XML document instances in CORBA.

The response [XMLVALUES], addressed the requirements in the RFP while attempting to ensure that the extensive work on CORBA is leveraged and reused. The RFP stated a set of goals

- Provide CORBA centric transmission and manipulation of XML document instances
- Leverage the CORBA IDL language mappings
- Build upon existing XML technologies and specifications
- Make sure the proposal does not interfere with other CORBA/XML activities (notably XML Metadata Interchange, XMI)

The proposal is based on the features provided by the Document Object Model (DOM) as defined by the W3C [DOM1, DOM2], such as tree manipulation, navigation and traversal. OMG IDL was used in the specifications of the DOM to provide language neutrality, but the interfaces are not using any CORBA specific infrastructure or language mapping. The Majority of the proposal is consequently a reformulation of the DOM IDL using CORBA value types.

Appendix E: XML Schemas and DTDs

The way to enforce validity in XML documents without manually coding the program to do so, has so far been the use of XML DTDs (Document Type Definitions). This allows checking whether a specific document abides by rules that are roughly equivalent to a context-free grammar. This checking has turned out to be insufficient for many purposes, fueling the need for a different form of validity check. Note that it is not necessarily a stricter check, only a different one. Schemas, and the schema language, allow for checking in a far more expressive manner, where for instance data types and value constraints can readily be described. It further allows data types to be inherited, giving it substantial power of data-type expression. Note that the XML Schema specification is, at the time of this writing, in final draft, and is expected to undergo no significant changes until completion. For this reason, this text will treat the draft as a final specification. For more information on how the XML Schema language looks and functions, see [XMLSCHEMA0], [XMLSCHEMA1] and [XMLSCHEMA2].

Purpose of XML Schemas

The following section describes the purpose of XML Schema and is taken from the XML Schema working committee at W3C[45]:

The purpose of the XML schema language is to provide an inventory of XML markup constructs with which to write schemas.
The purpose of a schema is to define and describe a class of XML documents by using these constructs to constrain and document the meaning, usage and relationships of their constituent parts: datatypes, elements and their content, attributes and their values, entities and their contents and notations. Schema constructs may also provide for the specification of implicit information such as default values. Schemas document their own meaning, usage, and function.
Thus, the XML schema language can be used to define, describe and catalogue XML vocabularies for classes of XML documents.
Any application of XML can use the Schema formalism to express syntactic, structural and value constraints applicable to its document instances. For applications that require other, arbitrary or complicated constraints, the application must perform its own additional validations.

XML Schema: A sample

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<xsd:element name="Memo" type="memoType"/>

<xsd:complexType name="memoType">
     <xsd:element name="from" type="string"/>
     <xsd:element name="to" type="string"/>
     <xsd:element name="sent" type="date"/>
```

[45] taken from http://www.w3.org/TR/NOTE-xml-schema-req

```
     <xsd:element name="subject" type="string"/>
     <xsd:element name="text" type="string"/>
     <xsd:elementname="comments" type="commentType"
     minOccurs="0"
          maxOccurs="*"/>
</xsd:complexType>

<xsd:complexType name="commentType">
     <xsd:element name="by" type="string"/>
     <xsd:element name="onDate" type="date"/>
     <xsd:element name="comment" type="string"/>
</xsd:complexType>
```

This specifies that a conforming XML-document must have a Memo-element as it's root, and the Memo-element must have a from, to, sent, subject, text and any number of comments-elements, each having a by, onDate and comment element. It is the responsibility of the XML interpreter to enforce the validity of the document - a document claiming conformance could look like this (assuming the schema was saved as memo.xsd):

```
<?xml version="1.0"?>

<Memo xmlns="x-schema:memo.xsd">
     <from>BillG@microsoft.com</from>
     <to>BillC@whitehouse.gov</to>
     <sent>2000.05.14 00:13</sent>
     <subject>The Bart Simpson defense</subject>
     <text>I didn't do it, nobody saw me do it, you can't
     prove anything!</text>
</Memo>
```

Appendix F: Interface Definition Language (IDL)

The IDL[46] standard was designed by the OMG (Object Management Group) to allow language-neutral specifications of the interfaces offered by CORBA objects, and is often referred to as OMG IDL to set it apart from other IDLs[47]. Dating back to the early 1990s, it has a rather Java-like syntax and structure. Standardized mappings from IDL to a number of languages (C++, Java, Ada and Visual Basic, for instance) exists to provide CORBA developers with a common data definition language that is usable across platforms and programming languages.
OMG IDL does NOT specify data elements – it only specifies what methods exists in what interfaces, what their parameters are and what type of value (if any) they return[48].

A quick example

```
interface Vehicle {
     short maxSpeed();
     short seats();

     void setSpeed(in short toWhat);
     short currentSpeed();
};

interface Car: Vehicle {
     short numberOfWheels();
     bool manualShift(out short numberOfGears);

     string brand();
};
```

This specifies a Vehicle interface with the methods maxSpeed, seats, setSpeed and currentSpeed, and a Car interface, derived from the Vehicle interface, with the additional methods numberOfWheels, manualShift and brand. Note that it says nothing about whether the implementation of the Car interface actually inherits anything from the implementation of the Vehicle interface. It is often useful to consider interfaces as a kind of contract – it specifies what a given CORBA object is offering to do, but not how it is actually done.

Basic datatypes

OMG IDL's selection of basic datatypes shows its close ties to C++. The basic datatypes are short, long, unsigned short, unsigned long, float, double, char, string, boolean, octet and any. String, octet and any are the only exceptions, and while string is obvious, the octet and any types work as follows:

- Octet
  An 8-bit value with no interpretation. Usually used to transfer binary data while avoiding marshalling[49] - image-data is an example of when this could be desirable.

[46] A wealth of information on OMG IDL and CORBA is available through [CORBA231].
[47] For instance, Microsoft designed their own, incompatible, IDL for COM.
[48] An attribute specifier does exists, but it merely maps to two functions for setting and getting the attribute.

- Any
  In some cases, it is desirable to transfer variables of any type to a method, and for these cases the any-type is the solution. It allows any type to be transferred, and furthermore must provide run-time type information to allow the called method to infer the actual type of parameter.

Complex datatypes

OMG IDL specifies a number of ways to make complex types from basic ones. Among these are enumerations, structures, unions, arrays and sequences. Enumerations works almost like in C++, except that you can't specify the ordinal value of the enumerators. Structures works just like in C++, while unions differ somewhat in the way you specify under what circumstances to use any given union-structure. Arrays work much like their C++ counterparts, but open array bound are not allowed. Sequences are variable-length vectors of some type, and can be bounded or unbounded.
In many instances, OMG IDL does not allow anonymous types where C++ would. This means that you need to use typedef (which works just like in C++) more often than in C++. Recursive datatypes can be specified (somewhat clumsily) through structures and unions, by using a sequence[50].
Since the interface-construct in IDL is a type constructor, it follows that object parameters (and return values) can be achieved by using the interface name as a type-name.

Parameters

OMG IDL specifies that parameters can be either *in, out or inout*. *In*-parameters are call-by-value, *inout*-parameters are call-by-reference and out-parameters are call-by-reference with the restriction that the value at call-time is not available to the method called.

[49] Marshalling is the process of transforming parameters between caller and receiver, to ensure that differences in data representation due to differences in machine architecture (among other things) does not disturb the meaning of data. CORBA automatically handles marshalling.
[50] See, for instance, section 4.7.8 in [ADVCORB] for an example.

Appendix G: XML

XML has its roots in SGML derived markup languages. By giving up some of the flexibility of SGML, XML gains simplicity. The SGML recommendation is approximately 550 pages, while the XML recommendation is a mere 26 pages. XML is a result of the 80/20 rule, i.e. with XML we get 80% of the functionality with 20% of the complexity.

When referring to XML, people often think about XML and the related specifications, such as XSL (EXtensible Stylesheet Language), XLink, XPointer, as one technology. The most important base features of XML 1.0 are syntactic rules, simple rules for defining hierarchical data (DTD, Document.Type Definition), and entities

XML documents that are syntactically correct are called well-formed. If in addition a document complies with a given structure as described by the associated DTD, then it is called valid.

Namespaces

Once the XML specification was finished and people actually began to use XML several issues surfaced. One of the most often raised issues was reuse and namespace pollution. The problem was deemed sufficiently important, that an add-on specification was created specifically to handle these problems.

What is needed is a convention that will allow us to treat two elements as being different even though they share the same name. What does an element named "class" contain? We must be able to distinguish between a class from OO and a class in a school.

```
<xdoc xmlns:sch="http://www.mudville-schools.org">
     <sch:class-list>
          <sch:class>Advanced Basket Weaving</sch:class>
          <sch:class>3D Art</sch:class>
          <sch:class>Remedial Reading</sch:class>
     </sch:class-list>
</xdoc>
```

In the above sample the attribute xmlns:sch defines a namespace for elements, by prefixing the element names with "sch". This can sometimes become overly verbose. To remedy this a default namespace declaration is possible, i.e. we do not have to write the prefix to the elements.

```
<xdoc xmlns="http://www.mudville-schools.org">
     <class-list>
          <class>Advanced Basket Weaving</class>
          <class>3D Art</class>
          <class>Remedial Reading</class>
     <class-list>
</xdoc>
```

References

[ADVCORB] *Advanced CORBA programming with C++*, Henning & Vinoski
Addison Wesley Longman Inc., 1999
ISBN 0-201-37927-9

[ADVDGC] *Recent Advances in Distributed Garbage Collection*, Marc Shapiro and Le Fessant, Fabrice and Paulo Ferreira,
Recent Advances in Distributed Systems, 104-126, Springer-Verlag , February 2000.
ftp://ftp.inria.fr/INRIA/Projects/SOR/papers/2000/RAIDGC_lncs1752.ps.gz

[AST] *Computer Networks*, second edition, Andrew S. Tanenbaum
Prentice-Hall International
ISBN 0-13-166836-6

[APP] *Adaptive Parameter Passing*, Cristina Videira Lopes, presented at ISOTAS '96
Northeastern University, Boston
ftp://ftp.ccs.neu.edu/pub/research/demeter/documents/papers/Lop96-adapt-param.ps

[AVDO] *An architectural view of distributed object and components in CORBA, Java RMI and COM/DCOM*, Frantisek Plásil & Michael Stal
Concepts and Tools, 19(1), 1998, p14-28

[CAN] *CAN specifications 2.0*
Robert Bosch GmbH
http://www.can.bosch.com/docu/can2spec.pdf

[COMCORBA] *COM-CORBA Interoperability*, Ronan Geraghty, Sam Joyce, Tom Moriarty, Gary Noone, Sean Joyce
Prentice Hall, 1998.

[CORBA231] *CORBA Specification, 2.3.1*
OMG, October 1999, formal
http://www.omg.org/

[CORBAIND] *CORBA - An industrial approach to open distributed computing*, N. Edwards
In "Open Distributed Systems", Edited by J. Crowford, UCL Press, 1995.

[CORBASOAP] *CORBA/SOAP Interworking, Request for proposal*
ftp://ftp.omg.org/pub/docs/orbos/00-09-07.pdf

[CORBAXML] *CORBA and XML Conflict or Cooperation?* Andrew Watson
OMG, 1999.

[CHLP] *CAN higher layer protocols*
K Etschberger
http://www.stzp.de/papers/icc97/icc97_e.html

[DCEDEV] *OSF DCE 1.0 Application Development Guide*.
Technical report, Open Software Foundation, December 1991.

[DCOM] *Distributed Component Object Model Protocol—DCOM/1.0*
(MSDN Library, Specifications)
http://msdn.microsoft.com/library/specs/distributedcomponentobjectmodelprotocoldcom10.htm

[DCOMARCH] *DCOM Architecture*, Markus Horstmann and Mary Kirtland, 1997
http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomarch.htm

[DISTBETA] *Object-oriented distributed programming in BETA*, Søren Brandt and Ole Lehrmann Madsen.
ECOOP 1994.

[DOMLEV1] *DOM Specification, Level 1*
W3C, 01-10-1998
http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001

[DOMLEV2] *DOM Specification, Level 2*
W3C, 23-09-1999
http://www.w3.org/TR/1999/WD-DOM-Level-2-19990923

[EG3] *Embedded hardware/software community*
http://www.eg3.com

[EMERALD] *Emerald - a general purpose programming language*, Eric Jul, Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, and Norman C. Hutchinson
Software Practice and Experience, 1991, vol. 21, January number 1, pages 91-118

[ETH] *IEEE 802.3: Carrier Sense Multiple Access with Collision Detection*, New York 1985a
http://www.ieee.org

[GCINTER] *Garbage collecting the internet: a survey of distributed garbage collection*, Saleh E. Abdullahi and Graem A. Ringwood
ACM Computing Surveys, 30(3):330-373, September 1998.
http://www.acm.org/pubs/articles/journals/surveys/1998-30-3/ p330-abdullahi/p330-abdullahi.pdf.

[GD] *Generalizing Dispatching in a Distributed Object System*, Ben Hurwitz et al
ECOOP ’94, p. 450 ff
http://www.ifs.uni-linz.ac.at/~ecoop/cd/papers/0821/08210450.pdf

[ICE] *Introduction to CORBA for Embedded systems*, Niall Murphy
embedded.com
http://www.embedded.com/98/9810fe2.htm

[IIOP] *IIOP Complete*, William Ruh, Thomas Herron, and Paul Klinker
Addison Wesley
ISBN 0-201-37925-2

[INCOM] *Inside COM*, Dale Rogerson
Microsoft Press 1997.
ISBN 1-57231-349-8

[INDCOM] *Inside Distributed COM*, Guy Eddon and Henry Eddon
Microsoft Press 1998
ISBN 1-57231-849-X

[JAVARMI] *Java™ Remote Method Invocation Specification (revision 1.50)*
Sun Microsystems, October 1998
ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf

[JINI] *Jini Architecture Specification,Version 1.1*
Sun Microsystems, 2000.
http://www.sun.com/jini/specs/jini1_1.pdf

[JSCR] *JavaScript reference*
ECMA-262 (European Computer Manufacturers Association) ECMAscript reference
http://www.ecma.ch/ecma1/stand/ecma-262.htm

[MOD3NO] *Network Objects*. Birrell et al
Digital Equipment Corporation Systems Research Center, Technical Report 115 (1994).

[MOD3NOGC] *Distributed Garbage Collection for Network Objects*. Birrell et al
Digital Equipment Corporation Systems Research Center, Technical Report 116 (1993).

[MPR] *Motorola press release*
http://mot-sps.com/news_center/press_releases/PR990928C.html

[MSDN] *Microsoft Developer Network*
http://msdn.microsoft.com

[NDC] *A Note on Distributed Computing*, Jim Waldo et. al.
Technical Report TR-94-29, Sun Microsystems Laboratories, 1994
http://www.sun.com/research/technical-reports/1994/smli_tr-94-29.ps

[OOSHVAR] *Lightweight Object-Oriented Shared Variables for Distributed Applications on the Internet*. J. Harris and V. Sarkar
Proceedings of OOPSLA'98. p.296-309

[PATTERNS] *Design Patterns: Elements of Reusable Object-Oriented Software*
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Addison Wesley. October 1994
ISBN 0201633612

[PLAINFOSS] *A Survey of Distributed Garbage Collection Techniques*, D. Plainfoss & Marc Shapir

[PSX] *POSIX.4 Programming for the real world*, Bill O. Gallmeister
O'Reilly & Associates, Inc., 1995
ISBN 1-56692-074-0

[RMA] *Rate Monotonic Analysis*
Educational Materials CMU/SEI-94-EM-11
Carnegie Mellon Software Engineering Institute
http://www.sei.cmu.edu/activities/str/descriptions/rma_body.html

[SBS] *DCOM & CORBA, side by side, step by step, layer by layer*, P. E. Chung et. al.
Bell Labs, Lucent Technologies, Academia Sinica, AT&T Labs
http://www.cs.wustl.edu/~schmidt/submit/Paper.html

[SELF] *Self – the power of simplicity*, David Ungar & Randall B. Smith
OOPSLA '87 Conference Proceedings, pp. 227-241
http://www.sun.com/research/self/papers/self-power.html

[SOAP] *Simple Object Access Protocol 1.1* recommendation
World Wide Web Consortium
http://www.w3.org/TR/SOAP/

[SMT] *Smalltalk-80: The Language,* Adele Goldberg and David Robson
Addison-Wesley Publishing Company, Massachusetts, 1989
ISBN: 0201136880

[SRTOS] *Selecting a Real-Time operating systems*, Greg Hawley
http://www.embedded.com/1999/9903/9903sr.htm

[STROUSTRUP] *The C++ Programming language*, Bjarne Stroustrup
Addison Wesley Longman, Inc., 1997
ISBN 0-201-88954-4

[UCLINUX] *Micro-controller Linux*
http://www.uclinux.org

[UCSIMM] *uCSimm hardware project documentation*
Lineo Inc.
http://www.lineo.com/products/ucsimm/index.html

[UNDCOMP] *Understanding COM+*, David S. Platt
Microsoft Press, 1999
ISBN 0-7356-0666-8

[VNU] *If it's an appliance, network it*
http://newsletter.vnunet.com/News/105174

[WOLLRATH95] *Simple Activation for Distributed Objects*, Ann Wollrath, Geoff Wyant, and Jim Waldo
Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS), Monterey, California, June 1995.

[WOLLRATH96] *A Distributed Object Model for Java System*, Wollrath, Riggs, Waldo
USENIX COOTS 1996

[XIDL] *XIDL, an Interface Definition Language for SOAP-RPC*, Curbera et al.
IBM T. J. Watson Research Center, 01-05-2000.
http://apps.alphaworks.ibm.com/technologies/soap4j/xidl.html

[XIOP] *XML InterOrb Protocol*, an Open Source Project
http://xiop.sourceforge.net/

[XLINK] *XML Linking Language (XLink) Version 1.0*
WorldWide Web Consortium
http://www.w3.org/TR/xlink/

[XPTR] *XML Pointer*
WorldWide Web Consortium
http://www.w3.org/XML/Linking

[XMLRPC] *XML-RPC Specification*, Dave Winer, 16-10-1999
WorldWide Web Consortium
http://www.xmlrpc.com/spec

[XMLSCHEMA0] *XML Schema part 0: A primer*, W3C
WorldWide Web Consortium
http://www.w3.org/TR/xmlschema-0

[XMLSCHEMA1] *XML Schema part 1: Structures*, W3C
WorldWide Web Consortium
http://www.w3.org/TR/xmlschema-1

[XMLSCHEMA2] *XML Schema part 2: Data types*, W3C
WorldWide Web Consortium
http://www.w3.org/TR/xmlschema-2

[XMLVALUERFP] *XML Value RFP*, OMG, 20-12-1999.
WorldWide Web Consortium
ftp://ftp.omg.org/pub/docs/orbos/99-08-20.pdf

[XMLVALUES] *XML/Value Types*, OMG, 20-12-1999.
WorldWide Web Consortium
ftp://ftp.omg.org/pub/docs/orbos/99-12-05.pdf

[XPACT]
http://www.w3.org/2000/xp/

[XPDRAFT] *XML Protocol Working Group Draft Requirements*
World Wide Web Consortium, November 2000
http://www.w3.org/2000/xp/Group/xp-reqs-02

[XPMATRIX] *XML Protocol comparisons*
http://www.w3.org/2000/03/29-XML-protocol-matrix

[XSLT] *XML StyleSheet Transformations*, W3C recommendations
WorldWide Web Consortium
http://www.w3.org/TR/xslt.html