Examensarbete

Evaluation of interprocess communication methods in a component based environment

av

Fredrik Örvill och Magnus Therning

LiTH-IDA-Ex-00/15

2000-02-09

Linköpings universitet
Institutionen för datavetenskap

Final report

Evaluation of interprocess communication methods in a component based environment

By

Fredrik Örvill and Magnus Therning

LiTH-IDA-Ex-00/15

2000-02-09

Supervisor: Patrik Schnell (Nokia), Peter Aronsson (LiTH)
Examiner: Peter Fritzson

Avdelning, institution
Division, department

Institutionen för datavetenskap

Department of Computer
and Information Science

Datum
Date

2000-02-09

Språk
Language

○ Svenska/Swedish
◉ Engelska/English
○ _____

Rapporttyp
Report category

○ Licentiatavhandling
◉ Examensarbete
○ C-uppsats
○ D-uppsats7
○ Övrig rapport
○ _____

ISBN

ISRN

Serietitel och serienummer
Title of series, numbering

ISSN

LiTH-IDA-Ex-00/15

URL för elektronisk version
URL for electronic version

http://www.lysator.liu.se/~gargamel/exjobb/

Titel
Title

Evaluation of interprocess communication methods in a component based environment

Författare
Authors

Fredrik Örvill and Magnus Therning

Sammanfattning
Abstract

Modern application development is often built around a component system where the application programmers never have to bother about interprocess communication. If components reside in different processes the component system handles the communication between them. This thesis evaluates several of the available techniques for interprocess communications (IPC) in a prototype system for distributed components using the Cross-Platform Component Object Model (XPCOM) as the basis. It also takes into consideration marshalling, the transfer of complex data structures, it further discusses security, robustness and correctness.

Our work consists of studying the available IPC methods, designing and implementing a prototype system. The system is implemented in C++ and the evaluation is performed in the Linux operation system.

Nyckelord
Keywords

COM, DCOM, XPCOM, Marshalling, marshal, IPC, Doors, Shared memory, RPC, Pipe, XDR, ASN.1, BER, PER, DER, CER, Linux, Nokia

Abstract

Modern application development is often built around a component system where the application programmers never have to bother about interprocess communication. If components reside in different processes the component system handles the communication between them. This thesis evaluates several of the available techniques for interprocess communications (IPC) in a prototype system for distributed components using the Cross-Platform Component Object Model (XPCOM) as the basis. It also takes into consideration marshalling, the transfer of complex data structures, it further discusses security, robustness and correctness.

Our work consists of studying the available IPC methods, designing and implementing a prototype system. The system is implemented in C++ and the evaluation is performed in the Linux operation system.

Dedication

To our girlfriends, we love you.

Table of Contents

Table of Figures

"It's like nothing we've dealt with before."
- Unknown, TOS

1. Introduction

This chapter serves as an introduction to this master's thesis. It gives the background and presents the problem. The limitations to the problem are stated. Then each of the chapters is introduced as a guide to the reader. It is concluded with the prerequisites placed on the reader and a mandatory section of acknowledgements.

1.1 Objective

This thesis was produced with co-operation from Nokia Svenska AB. Nokia is currently developing a set top box for digital TV where the work in this thesis would be of major interest.

The purpose of this master's thesis can be summarised in the following two goals:

- *To study and evaluate several of the available techniques for communicating between separate processes.*
- *To develop a framework based on distributed components in which the evaluation is performed.*

To accomplish this, the thesis contains an overview of several of the techniques available. Then a system design is presented together with a prototype. Finally the results and our conclusions from the results are given.

1.2 Disposition

Chapter 2 Detailed description of the problem.

Chapter 3 The different technologies used in the thesis are presented. A short introduction to some of the COM[1] implementations is given. Several methods for marshalling and IPC[2] are also presented.

Chapter 4 This chapter contains an analysis of what the requirements are on the system. A presentation of the system from a functional view is given.

[1] Component Object Model
[2] Interprocess communication

Chapter 5 Here the design of the system is presented. The different design decisions are also discussed.

Chapter 6 The implementation is described on a somewhat lower level than in chapter 5. Our evaluation programs and the criteria for the evaluation are introduced.

Chapter 7 In this chapter the results of the evaluations are presented.

Chapter 8 The conclusions from the results in chapter 7 are drawn in this chapter.

Chapter 9 All interesting aspects of this work were not done because the available time was limited. In this chapter some suggestions for future enhancements are presented.

Chapter 10 References to work that is of interest can be found in this chapter.

Appendix A List of abbreviations.

Appendix B Specifications of the various interfaces.

Appendix C Detailed information about the classes implementing the interfaces described in Appendix B.

Appendix D Protocols from the evaluations.

1.3 Prerequisites

To comfortably read this thesis prior knowledge and understanding of COM is useful. A general orientation in the problems faced when separate processes communicate is helpful.

Understanding the design is made easier by knowing the basics in object oriented design.

While the first parts of the thesis try to be as general and as independent of the used operating system as possible, the later part must inevitably take the target platform into account. Everything up to the section covering the details of the implementation should be quite free from references to Linux. From section 6 and onwards a familiarity with the Linux operating system might prove helpful.

1.4 Notation

Throughout the report we have followed the following formats for discussing technical issues.

Names of classes start with the letter C (as in “class”) and are written in italic text, e.g. *CSimpleBuffer*

Names of interfaces start with the letter I (as in “interface”) and are written in italic text, e.g. *IBuffer*

Names of system and function calls as well as programs are written in italic text, and in the case of Unix functions and programs the section for the manual page is put in parenthesis directly after the name, e.g. *midl, ftruncate(2).*

When explaining the design diagrams have been used extensively. An example of a diagram describing a class with interfaces is shown below. To visualise an interface the Microsoft standard of visualising interfaces has been used. An example can be found in Figure 1 below.

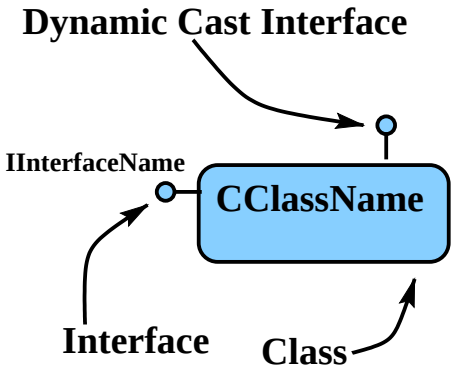

Figure 1: Example of a class with interfaces

1.5 Acknowledgements

In order to finish this work we have depended on the people around us. A big thanks goes out to all the people who helped us keep on track, and even to the ones who frequently interrupted us with questions not even remotely connected to this work.

Without the bountiful help from Nokia in Linköping this thesis would never have seen the day of light. Especially we are indebted to our supervisor Patrik Schnell.

A special thanks goes to Henrik Linde and Petter Axling for providing extensive help with the report.

We also need to thank the examiner and supervisor provided to us from school.

"Our species can only survive if we have obstacles to overcome, without them to strengthen us we will weaken and die."
- Captain Kirk, TOS

2. The Problem

This chapter will introduce the problem in more detail and also provide some necessary background knowledge.

2.1 Problem Definition

As stated in section 1.1 the objective of this work is to accomplish the following:

- *To study and evaluate several of the available techniques for communicating between separate processes.*
- *To develop a framework based on distributed components in which the evaluation is performed.*

As was also mentioned Nokia is currently developing a set top box for digital TV where distributed components will be used as a way of communicating between different processes. These boxes will be running an embedded version of the Linux operating system and the Mozilla web browser will be used to provide an easy-to-configure user interface. Nokia is interested in obtaining fast interprocess communication in their set top boxes since consumer electronics call for a usable product at a reasonable price. This means that even a low gain in time consumption for communication between processes can be worth a lot in the end. At the same time development costs for software should be kept low. A framework that permits this and also makes it easy to maintain the end product is essential.

2.2 Limitations

Because of the current nature of operations for Nokia's set top boxes the thesis is limited to considering interprocess communication involving processes executing on the same CPU[1]. Involving "true" distributed components, distributed over several computers, would be too large a task for a thesis of this size. This limitation is imposed upon the project from Nokia.

[1] Central Processing Unit

Since the environment where the system is to be used is an electronic device for home entertainment with reduced functionality compared to a regular computer, we can assume minute control over which processes will be running. This is ensured from the fact that Nokia designs the box and also decides what programs to install, the user will not have complete control over the device, and he will not be allowed to install and run other programs. Only programs approved by Nokia can be installed and run on the set top box.

This minute control allows us to neglect issues such as starting servers and assuring that they continue to run. But it should be noted that in order to make the system truly useful it has to be taken into consideration.

Issues regarding security, and especially authentication of both servers and clients, can also be neglected because of the rigorous control over the system. Communication will only be done between processes executing on the same computer and attacks on the system from malicious clients are not likely to happen.

2.3 Environment

The work has been performed using regular PC's[1] running the Linux OS. RedHat 6.0 was chosen as the distribution because it was widely used within Nokia in Linköping when this project started, and even though a newer version was released during the project we stayed with the initial version. During the work Nokia also released an internal Linux distribution which is at the moment used for developing the low-level software for the set top boxes.

The intended target platform is a system based on an Intel Pentium processor running Linux. The version of the operating system that eventually will be used is not decided yet.

For the tests we have used the computers we were provided with. The details of these can be found in Table 1. To ensure as accurate results as possible the tests were performed with as few processes running as possible[2].

[1] Personal Computer
[2] Single user mode with only necessary daemons running

CPU	Pentium III, 450MHz
Memory	128MB
Operating system Version	Linux 2.2.12, RedHat 6.0

Table 1: Test Environment

2.4 Distributed Components

In almost all of the bigger systems nowadays there is some level of distribution in the sense that the system is divided into smaller parts responsible for separate tasks. These smaller parts are often running as separate processes in order to make the system more robust. But this also introduces the problem of how these processes will communicate with each other. It is only by co-operation that the system solves the problem, each process on its own will only provide a small part of the solution.

The way the computer industry has been moving the last couple of years, client-server architecture has become more and more dominant when designing systems. In this architecture there exists a server which provides services to clients. Even in complex systems this kind of architecture can be used, maybe requiring many servers and a client to one process might act as the server for another.

There has also been a shift in the software industry during the later part of the ‘80s and the ‘90s to look at problems in an object-oriented way. This has shown to provide a way of thinking that can produce software that is relatively easy to develop and maintain.

What distributed components do is to connect the client-server architecture with object-orientation. A server provides services in the form of components that a client can obtain and use through some kind of interface. Using distributed components also provides a way to force separation of interface and implementation, something which is generally desirable in object-orientation.

2.4.1 Distributed Component Object Model

COM[1] was first developed to provide object linking and embedding. The technology was adopted by Microsoft and then extended to incorporate distribution of components as well (DCOM[2]). Microsoft has extended DCOM

[1] Component Object Model
[2] Distributed Component Object Model

by building a number of technologies on top of it, most notable OLE2[1] and ActiveX.

The technology has proved to be very useful, but its use is mainly concentrated on the Windows platform.

2.4.2 CORBA

OMG[2] defined the first version the CORBA[3] standard in 1991. OMG is a non-profit consortium founded in 1989 with a world-wide presence and consisting of more than 500 members.

While DCOM is an extension to COM, CORBA was designed as a distributed objects system from the beginning. The central part is the so-called ORB[4]. All requests for new distributed objects as well as invocations of methods on distributed objects go via the ORB.

CORBA has wide spread use and the standard is still a work in progress. ORBs are available to a wide range of architectures, including several UNIX variants, Macintosh, Windows and even DOS[5] (DCOM is not available for DOS).

2.5 Mozilla

Netscape (pronounced Mozilla) is one of the most successful web browsers available on the market. In 1998 Netscape Corporation released the source code of its browser and started an effort to continuing development using an open-source model.

In order to make object linking and embedding available to Mozilla it employs a cross-platform version of COM called XPCOM[6], which runs, amongst others, Windows, Linux and Macintosh. The current version of XPCOM, included in Mozilla M11 (milestone 11), does not provide support for distributed components.

2.6 Design Patterns

Patterns is the term used to denote a solution to a general problem which has been proven good from experience in object-oriented programming. Many problems that appear in software architecture are related and what patterns try to do is to abstract the problem up to a level where a general solution can be given. This general solution can then be applied to a

[1] Object Linking and Embedding 2
[2] Object Management Group
[3] Common Object Request Broker Architecture
[4] Object Request Broker
[5] Disk Operating System
[6] Cross Platform Component Object Model

specific problem. The effect of this is that developers can discuss problems and their solutions on a more abstract level by using patterns.

The first catalogue of well-described design patterns for object-oriented programs was [7], which appeared in 1994. Since then the number of patterns has expanded dramatically. The concept of design patterns comes from architecture of buildings and was first described by Christopher Alexander, (see [11] p xii)

2.7 Summary

Our work consists of the following steps:

1. Study the available component techniques.
2. Study the problems surrounding transferring of arguments (marshalling).
3. Study and evaluate the usefulness of different interprocess communication methods.
4. Design a framework that satisfies the requirements.
5. Implement a prototype of the framework.
6. Evaluate the framework and the different interprocess communication methods.
7. Draw conclusions and present suggestions for future extensions and enhancements.

"We're not going anywhere, until we have some information!"
- Captain Kirk, TOS

3. Theoretical Background

In today's operating systems processes are shielded from each other. A process that needs to communicate with another process can not do so directly, but has to use some form of interprocess communication method provided by the operating system.

COM[1] encapsulates this communication in a completely transparent fashion: it intercepts calls to a distributed component made by a client and forwards them to the component in the server process.

This chapter introduces COM and XPCOM[2]. It also explains the ideas behind COM and the different parts of COM that are of interest in this thesis.

3.1 Component Object Model

This section presents the technique and two different implementations. First Microsoft's COM for Windows and then XPCOM are presented. Microsoft's implementation is presented together with the technology since their implementation was the first, and still is the de facto standard.

3.1.1 The Basics

COM is a widely used technique to implement programs in small independent objects. It separates the interface from the implementation in a very useful way. This separation means that a programmer using a COM object only needs to know what the interface looks like and does not have to bother about how the implementation is done.

C++ programmers may think that this is what they have been doing for a decade. The interesting thing with COM is that a COM object may be replaced with another COM object that implements the same interface and possibly others extending it. Programs or COM objects using it do not need to know about the change, nor do they need to be recompiled.

COM is a client server technology where the server serves the client with COM components. There is nothing that prohibits the client to be a COM component itself, but it does not have to. There are two kinds of servers, in-

[1] Component Object Model
[2] Cross Platform Component Object Model

process servers, where the server is actually loaded into the address space of the client. The other kind, out-of-process servers, is implemented as a separate process.

To assist the programmer in her task, an IDL[1] compiler tool creates the basic framework for the COM server. This is explained in more detail later on.

The use of COM is largely centred around the Windows platform. Few implementations are available for other platforms, but there is an ongoing effort to make the technique widely available. The technique has been proven useful for providing modularity in object oriented designs.

3.1.2 In-process Servers

What was said earlier about loading the server into the address space of the client may sound just like a description of a DLL[2]. While this is true, it is not the whole truth. An in-process server in COM is more than a DLL. It is a precise way of implementing a DLL so that the interfaces are interchangeable and extendible.

The way to acquire a COM object from an in-process server is shown in the upper part of Figure 2 below. The client process starts by asking for a specific object (identified with a unique ID). The SCM[3] locates the server for the object. The server is then loaded into the client's own memory space and a handle for the object is returned. Subsequent calls for getting interfaces is made through this handle.

Calls for an object's services are accomplished like regular function calls. This is possible since a server that resides in the same memory space as the client implements the object. This makes method calls on in-process server objects really fast, about one microseconds overhead.

Passing an interface of a component in an in-process-server to another component running in-process is simple since they are both in the same memory space.

3.1.3 Out-of-process Servers

COM objects do not have to be implemented as DLLs, they can be ordinary EXE-files[4] as well. The EXEs must be accompanied with a DLL. However, this DLL can be automatically generated with Microsoft's IDL compiler tool. An out-of-process server does not have to run on the same machine as the client. When an out-of-process server is used, an automatically generated

[1] Interface Definition Language
[2] Dynamic Link Library
[3] Service Control Manager
[4] Executable files

proxy DLL is loaded into the client. This proxy DLL provides communication transparently with the server.

When client and component reside on different machines, COM simply replaces the local interprocess communication with a network protocol. Neither the client nor the component is aware that the wire that connects them has just become a little longer.

The automatically generated proxy acts as the component itself, so that the client is unaware of whether an in-process or out-of-process server is used. The proxy only forwards all requests to the real component. These kinds of method calls take much longer time than calls to an in-process server.

Acquiring a COM component offered by an out-of-process server puts a bit more stress on the COM system. The procedure is illustrated in Figure 2 below.

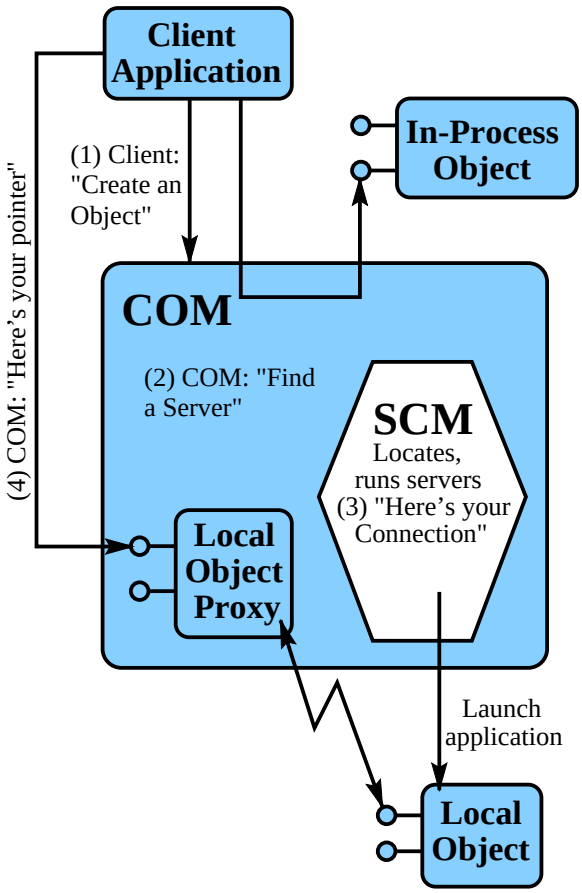

Figure 2: Acquiring a COM component

When the client asks for an object the SCM realises that the object needs an out-of-process server to be running. It then launches the server or contacts the SCM of the machine where the server resides. The connection is then handled by a local object proxy, which in turn is implemented in a

DLL. This DLL is loaded into the client's memory space and the client is given a handle to the object proxy. Subsequent calls to the object are then made through the object proxy.

Figure 3 depicts a memory layout for a client using both an in-process server and an out-of-process server for its objects.

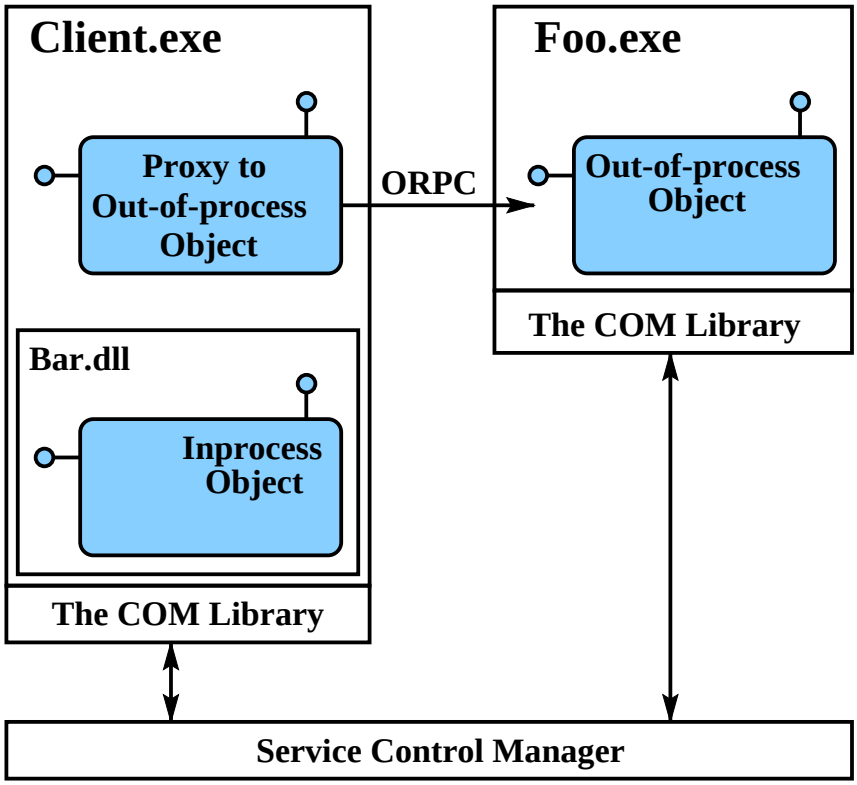

Figure 3: COM example

Calls for an object's services needs some extra handling too. The procedure is shown in Figure 4. As can be seen object proxies are needed on both sides of the process boundary in order to make it transparent to both the client and the server.

Passing an interface of one component to a component running in a remote server is a problem, since the server needs to load proxy code and somehow obtain a connection to the actual component that was passed. To achieve this the interface is marshalled, not the component itself. On the client side the unmarshalling of the marshalled interface results in a proxy component.

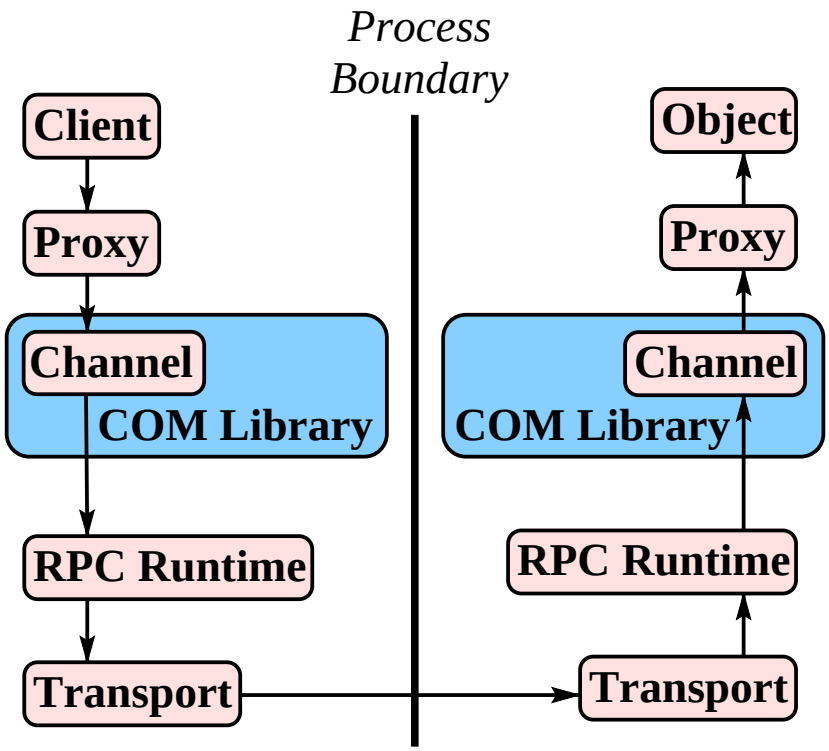

Figure 4: Calling an out-of-process component

3.1.4 Cross-platform COM

XPCOM is another flavour of COM, and is mainly used in the Mozilla web browser project. XPCOM is aiming at being a platform independent implementation of a subset of COM. Since XPCOM is such a limited implementation the number of prerequisites for a platform to be supported is minimal, and as a result the number of supported platforms contains, amongst others, such different platforms as Linux, MacOS and WindowsNT.

The architecture for XPCOM is very similar to Microsoft's implementation, but the names have been changed to better suit the Mozilla project. Another limitation imposed on the programmer is that only the use of C++ is supported at the moment.

3.2 Marshalling

Marshalling is the process of converting data into a format suitable for transmission over a communication channel. In the context of this thesis we are concerned with marshalling of arguments to functions. The converted data is then transported to the other object/program and marshalled back to the local representation of the data. Also, pointers must be followed and not just copied.

To be able to marshal a structure, the structure of the data must be known. This information is needed both when encoding and decoding. Often, the structure of the data is described in some language.

If data is to be transported over a network, all data in the structure must be converted to some intermediate form, since many things can be different in the target architecture. Both length of data types and byte order can be different. If the receiver is on the same CPU[1], or set of CPUs, the data does not need to be converted at all, aside from pointers. The pointers must be followed and the data pointed to must be copied. On the receiving side the pointers must be updated again to point to the correct data.

To write the code for encoding and decoding the data that is transferred is a tedious and erroneous task. In order to help the programmer, special programs have been developed to assist the process. Figure 5 shows the parts involved in automatically acquiring the files used in the program being developed. The specification file is written in a special language, IDL[2]. The IDL is independent from the implementation language. Then an IDL compiler tool processes the specification file and generates two things. First a header file describing the structures involved in the protocol in a way understandable by the implementation language. Secondly stubs are created, these are used for the actual encoding and decoding.

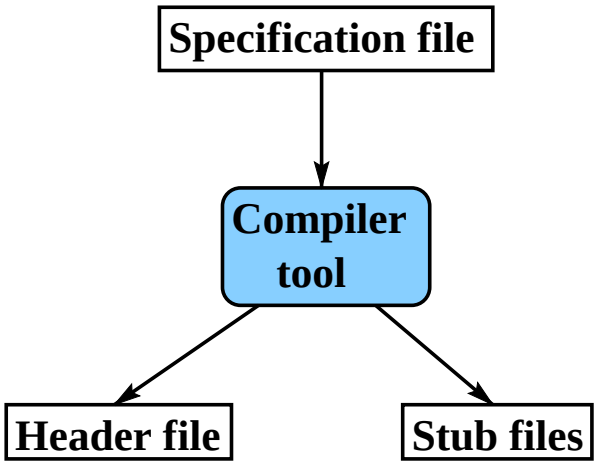

Figure 5: IDL compiler tool process

There are several standardised ways of doing marshalling. They mostly involve packing normal data types into a transport representation and back to the local representation. They also include ways of defining own complex data structures and data types.

[1] Central Processing Unit
[2] Interface Definition Language

For an IDL to be considered useable it has to support the following two things:

- Structures
- Pointers

All the IDLs presented here supports both.

3.2.1 Encoding Rules

Together with the ASN.1[1], a standard for describing data formats in an implementation independent way, a set of encoding rules have emerged. ASN.1 is a very powerful syntax for describing data formats and in many people's opinion it is too powerful to be of practical use. But later years' advances in encoding of data have made it possible to improve on the encoding rules. ASN.1 is emerging as the preferred way of describing data formats.

The power of ASN.1 has as result that the encoding rules have to be complex and quite verbose. It also results in that when encoding a piece of data the encoder always has to choose between two or more ways to do the encoding. As can easily be understood this makes the encoding hard to do by hand for a programmer. By now there exists several good tools to assist in this.

3.2.1.1 Basic Encoding Rules

BER[2] was the first and most basic of the encoding rules that surfaced together with the ASN.1 standard. It uses a concept called TLV[3] in which a parameter's value is proceeded by its type and length. BER is a very verbose format of passing information, and it was long thought that in order to ensure extendibility of a format this type of verbose format was needed.

3.2.1.2 Packed Encoding Rules

PER[4] was developed to address the verbosity of BER. PER is both more compact and more efficient in encoding. A computer actually uses less CPU cycles to produce a PER encoding compared to a BER encoding [3]. This is mostly an effect stemming from usage of restrictions on the values being transmitted. This imposes a need on both sides of the communication line to know more about the information being transmitted, but through careful engineering extendibility was achieved.

[1] Abstract Syntax Notation One
[2] Basic Encoding Rules
[3] Type, Length, Value
[4] Packed Encoding Rules

3.2.1.3 Distinguished and Canonical Encoding Rules

In BER there is always an option on how to do an encoding in ASN.1. Whenever there is an option in what way to encode, DER[1] resolves it in one direction and CER[2] in the other. This makes DER and CER roughly equivalent. For the application considered in this work they are similar enough to be treated as one.

According to [3], CER is technically superior to DER. Despite this DER has become the de facto distinguished/canonical encoding for BER.

3.2.2 External Data Representation

XDR[3] [1] is a standard for both description and representation of data. It is useful for transferring data between different computer architectures. The major difference between ASN.1 and XDR is that XDR uses implicit typing, while ASN.1 uses explicit typing. That is, in XDR the sender and receiver must know both types and ordering of the data.

XDR uses a language to describe data formats. Besides the standard data types, own structures and types can be defined in this language. The language supports use of recursive structures as well as pointers.

3.3 Interprocess Communication

The operating system must provide some way for processes to share data and communicate. IPC[4] can be done in many different ways. Therefore the operating system may provide several ways to share data and to communicate with other processes.

IPC can be achieved by letting processes share memory or by giving the processes some form of data channel to communicate through.

3.3.1 Remote Procedure Call

RPC[5] [4] is one of the most used way for achieving communication between two processes executing on separate computers. RPC was originally developed by SUN and it is used in their NFS[6].

It uses XDR for representing the data being transferred. The IDL for XDR is extended to also accept declarations of procedures. By doing this it is

[1] Distinguished Encoding Rules
[2] Canonical Encoding Rules
[3] External Data Representation
[4] Interprocess Communication
[5] Remote Procedure Call
[6] Network File System

possible to get the compiler tool (*rpcgen*) to produce stubs for both the server and the client.

The process for developing a program using RPC is shown in Figure 6.

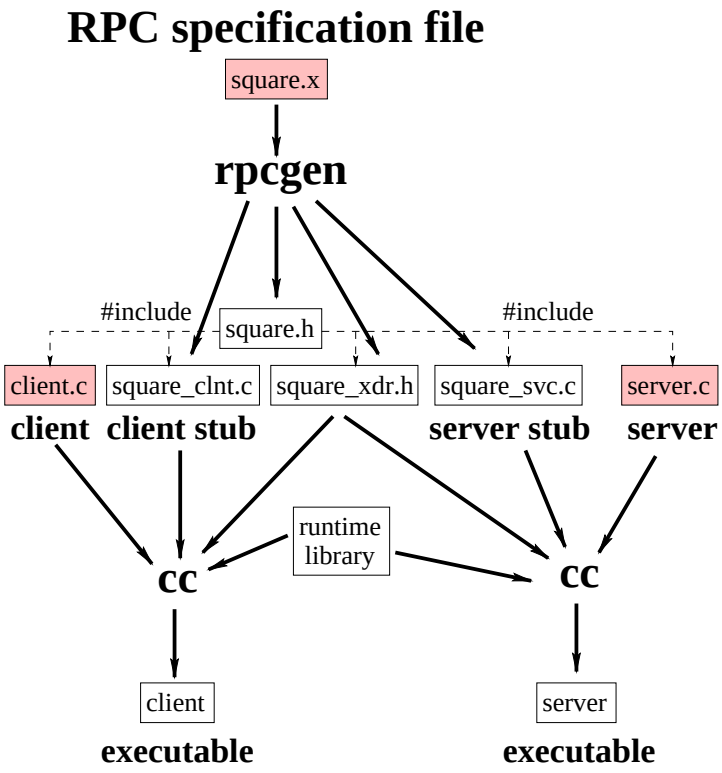

Figure 6: RPC compiler tool process

The architecture for RPC can be seen in Figure 7, the similarities to Figure 4 are easy to see. The only thing COM has added is two new layers, the object proxy and the actual COM library.

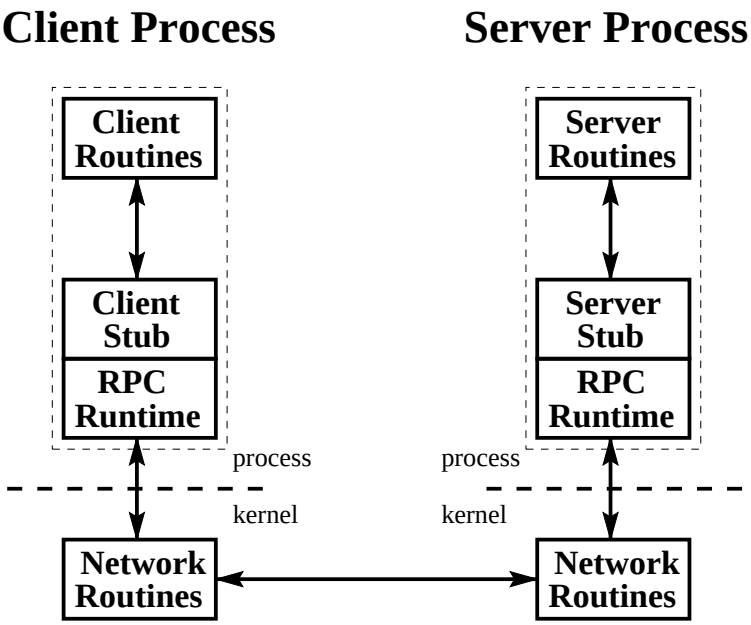

Figure 7: Calling a remote process via RPC

COM uses a derivative of RPC called ORPC[1] to achieve distribution of objects. ORPC is implemented as a layer on top of standard RPC.

3.3.2 Unix Pipes

Using pipes and sockets is the oldest way of achieving IPC in Unix. The original Unix pipes were unnamed and therefore hard to use as IPC between unrelated processes. In System III of Unix (1982) a named pipe (FIFO[2]) was introduced to remedy this problem. Both pipes and FIFOs are half-duplex, however, there are some flavours of Unix that provide full-duplex pipes. A pipe behaves like a single communication channel and all processes that connect to it get access to everything that is written to it.

Pipes are created using the *pipe(2)* function call. Usually this call returns two file descriptors, one for reading and one for writing.

FIFOs are sometimes referred to as *named pipes*, since the only difference between a pipe and a FIFO is that each FIFO has a pathname associated with it. To create a FIFO the system call *mkfifo(3)* is used. After creation the FIFO can be open using the *open(2)* system call. Since a FIFO is half-duplex it can only be opened in either read- or write mode, not in both.

3.3.3 Unix Domain Sockets

What is usually referred to as sockets is actually an API to several protocol families. One of the most used, TCP/IP[3], allows connections between

[1] Object Remote Procedure Call
[2] First In, First Out
[3] Transmission Control Protocol/Internet Protocol

computers via a network, but there are others available and the most interesting in connection with this work is a protocol family called Unix domain sockets. It only allows connections to sockets on the same machine and hence they are a lot like named pipes, but there are some important differences. All sockets are bi-directional and they are connection-oriented; each connection to a socket results in a new communication channel.

A Unix domain socket is created by providing *bind(2)* with a filename. For a client to connect to an existing socket is done through the *connect(2)* call.

But because of the extra functionality sockets can be expected to be somewhat slower than pipes. This was also confirmed by looking at the source code of Linux; it contained a lot more code to be executed than the implementation of pipes did. Because of this we decided early on to not include Unix domain sockets in the evaluation.

3.3.4 Shared Memory

The fastest way for two processes to communicate would obviously be if they shared a portion of their memory with each other. There are several ways to achieve this in a Unix environment. Two major systems exist, POSIX[1] and System V shared memory. They are very similar to each other in concept. They both provide shared memory by memory-mapped files and shared memory objects. They also provide ways of synchronising access to the shared memory. A third way of sharing memory is to map a file directly using *mmap(2)*.

Mapping a file into the memory of a process is accomplished by using the *mmap(2)* system call. First, a file is opened (possibly creating it at the same time), then the file descriptor is used as an argument to *mmap(2)*. Subsequent reads and writes to the memory space mapped to the file will be translated into operations against the file in question, thus the need to use *read(2)*, *write(2)* and *lseek(2)* is eliminated.

Mapping in the way described above requires that a file is created before mapping. This is sometimes unnecessary, e.g. when sharing memory between related processes. By mapping the file */dev/zero* (in SVR4[2]) or by giving *mmap(2)* a special flag (in 4.4BSD[3]) one obtains a piece of mapped memory that can be shared across a *fork(2)*.

The POSIX way of doing shared memory is a little different from the use of *mmap(2)*. POSIX offers the use of a *memory object* which can be shared. Creation of the object is done through *shm_open(2)*. This object is then mapped into memory using *mmap(2)*. Memory-mapping files is described above.

[1] Portable Operating System Interface, with an X thrown in to make it sound cooler [18]
[2] Unix System V Revision 4
[3] Berkley System Distribution version 4.4

The concept in System V is almost the same as in POSIX, but the names of the function calls differ. To obtain a memory object which can be shared the *shmget(2)* function is used. To map the object into memory *shmat(2)* has to be called. Another difference between the two presented ways of achieving shared memory objects is that POSIX allows the size of the object to be changed (through a call to *ftruncate(2)*) while the size is fixed from creation in System V.

3.3.5 Message Queues

Message queues can be looked upon as linked lists of messages. Messages can be put into the queue removed from it. Each message is a record with a priority. Message queues have kernel persistence in that they will keep on existing even though the creator of it has terminated, it will even keep its messages for reading by another process later. As with shared memory there exist two major systems, POSIX and System V.

The System V way of using message queues involves opening or creating a queue using *msgget(2)*. To put a message onto the queue *msgsnd(2)* is used and to receive a message one uses *msgrcv(2)*. POSIX provides similar calls, *mq_open(2)* to open or create a queue, *mq_send(2)* to put a message on the queue and *mq_receive(2)* to receive a message.

The main differences between these two systems are the limits imposed on the programmer. System V has system wide limits regarding both maximum number of queues and the maximum number of messages. POSIX on the other hand only puts limits on the maximum number of queues a single process can have open at once.

This method to achieve IPC was not included since Linux only provide System V message queues. The limits would make the system utterly useless.

3.3.6 Doors/Linux

Doors/Linux is an implementation of the Solaris Doors API[1]. It is essentially an RPC mechanism. Doors are made visible to the applications programmer as standard UNIX file descriptors. To make a door visible to other applications, a process may attach an existing door descriptor to an existing file. For another process to use the door, it opens it and then calls *door_call(3x)* using the door descriptor.

[1] Application programming interface

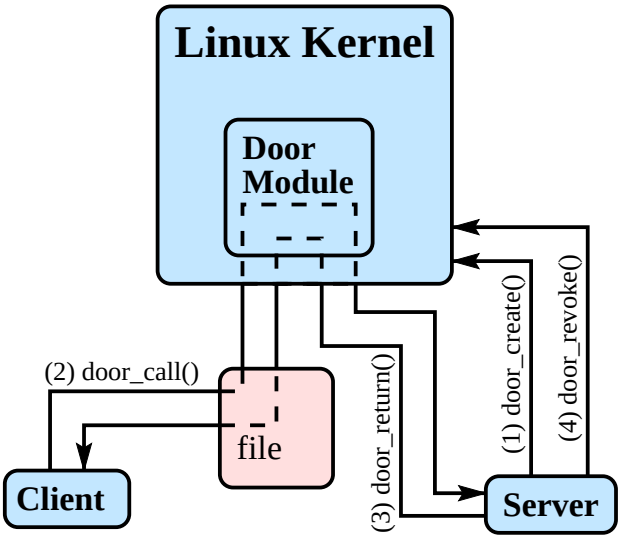

Figure 8: Doors/Linux example

The Doors API was engineered with performance optimisations in mind. The result is a fast way to do interprocess communication. This is achieved by letting the kernel decide if the arguments should be copied or mapped directly into the target process. If they are mapped into the target process's memory space, the pages are only copied if a process attempts to modify it. Also, server threads will be created in the calling process in proportion to the load on the server (at most one thread per concurrent request). The server has control over how many threads are created via *door_server_create(3x)*.

Doors is implemented by use of shared memory and is therefore faster than for example pipes, where the kernel have to copy the data between the processes.

An example of how a connection is established and how a call is made is illustrated in Figure 8. The client locates the door through the normal UNIX file system. Here is a normal life of a door:

1. The server creates the door. The client opens (open(2)) the door by locating it in the file system
2. door_call by the client to the server
3. door_return back to the client
4. door_revoke by the server. The door is now closed and is no longer valid.

Points 2 and 3 are normally repeated many times before the door is revoked.

A complete definition of the Solaris Doors API and a more detailed description of the Doors/Linux implementation can be found at [12].

3.4 Authentication and Encryption

Before a program sends sensitive data it must assure itself that the receiver is in fact the one intended and not someone else. In order to assure this a client might ask a server to authenticate itself when the connection is established. It might also be in the interest of a server to force its clients to authenticate themselves, perhaps in order to limit the services offered to clients of lesser security clearing.

There are several strategies and algorithms usable for authentication. They share the common trait to be quite complex.

Together with authentication comes the subject of whether the transmission itself is secure. Authentication of both sides involved in the communication ensures both sides are in fact whom they say, but it does not prevent someone to tap in on the transmission and eavesdrop. To prevent eavesdropping, the data needs to be encrypted in one way or another.

Encryption algorithms are publicly available in abundance. Implementations are available commercially as well as freely.

The subject of authentication, as well as that of encryption of data will not be considered in this work as was mentioned in section 2.2. It was deemed as too big a task to include these two areas in the research. A well-designed system should however not necessitate large changes in order to add it, this is left as an exercise for the reader.

“I hope you will look beyond what powers moves my hands and look at what thought inspires them to move.”
- Captain Jared, DS9

4. Problem Analysis

This chapter starts with an analysis of the problem and discusses roughly what parts are needed in order to make the system work. Then what can be considered an almost perfect distributed component system is described. Finally a description of the functions put into the system resulting from this research are described.

4.1 Component Object Model

As stated in section 3.1 multiple implementations of COM[1] exists today. The goal of this research is not to implement it yet one more time, but instead to examine different methods of IPC[2] in a COM environment. This leads us to assume that there will be a fully functional implementation of COM available and all that is done is extensions to it.

The changes to the used COM implementation are kept to a minimum. This is mainly done out of convenience. We do not want to depend on the availability of the source code for the implementation we decide to use. The implementation we use might also be in a stage of development during the time this research is done, and by just extending it we can change to newer, and preferably, better versions without too many hassles.

A full implementation of COM is not needed since no communication with other computers is needed. Even though the extension we make will make it possible to use out-of-process servers we do not follow any Microsoft standards since these standards only cover the usage of ORPC[3] for IPC and the source code for Microsoft COM is not available.

Looking at Figure 2 one can see that something corresponding to SCM[4] is needed. SCM works as a database where both in-process servers and out-of-process servers register themselves with the IDs of the components they offer. In the COM implementation we use, there will most certainly already be some kind of database available for registering servers. Whether it can handle both in-process and out-of-process servers depend on how much of

[1] Component Object Model
[2] Interprocess communication
[3] Object Remote Procedure Call
[4] Service Control Manager

the COM standard it implements. No matter what the case is we will extend the system with one more database. This is in order to let running servers register themselves together with the ID of their component and the preferred method of IPC. The IPC method is an extra field that is not present in whatever database that is used with the used COM implementation. Adding the field would mean a major change to the database and hence changes to the source code are needed. Adding a second database makes the whole system larger but instead changes to the underlying COM implementation are not needed.

The system must also offer an interface that simplifies accesses to the database.

The system will cater servers and clients. It must be simple for a server to use the system and offer its component to clients. It must also be simple for a client to obtain a component offered by a server. Two separate interfaces, one for servers and one for clients are needed to satisfy the needs of them both.

4.2 Feasible Features

The making of a system of distributed objects is a complex task. Many aspects have to be weighed in. Security versus efficiency is one of the more difficult ones to resolve. It is however an easy task to list a set of features that are desirable from a user's (programmer's) point of view. In this section these features are listed and described in detail.

The most important feature is that of transparency. The programmer should never have to take into consideration whether the object's server is located in a remote computer, locally or even loaded into the client's memory space. This is accomplished by using one single method of acquiring an object. The system has to take care of all of the details regarding locating a server and then establishing a connection with it. It is also important to provide the programmer of the server with a tool that relieves her (or him) of the tedious task of writing code for object proxies if that is necessary.

In the available implementations of COM it is possible for the programmer of the client to control what kind of server he or she wants. The different servers are:

in-process server - dynamically loaded library

in-process handler - dynamically loaded library with an object that handles some of the functionality of the object, the rest is handled by a remote server

local server – a server running in a separate process on the same machine as the client

remote server – a server running on a different machine from the client

These four types of servers cover all the possible scenarios regarding the server's location. There are however some extra features that might be desirable to have some control of. It should be noted that controlling things regarding the server's location and implementation should be optional. A programmer should not have to consider it if he or she does not want to.

Other features that a client process might want to control are the use of encryption of arguments. Sending sensitive data from the client to the server process on a remote machine for processing in clear text is not a realistic option in many applications. Encryption of arguments can be done using a number of different algorithms. These algorithms often differ in level of security and speed, where a higher level of security often results in lower speed. It is not hard to conceive a scenario where the client process might want to choose the encryption algorithm to use when communicating with a certain server.

On the server side it is desirable to have the means to limit the services offered to clients who are trying to establish a connection. It is not hard to imagine a scenario where a server is keen to offer services to clients located on the same local area network as itself, while clients connecting from other parts of the Internet are considered to be insecure and therefore not offered the same services. Making it impossible for computers located outside the local area network to connect the machine running the server might not be a realistic solution to this problem. Having the clients authenticate themselves in some way could solve the problem. There exists a plethora of authentication schemes and it should be possible for the server and client to agree on what scheme should be used before the connection is usable.

When connecting to a remote server the way of communicating is somewhat limited. The COM standard says that a superset of RPC[1] (called ORPC[2]) should be used. There might however be other ways to send data between the client and the server. When it comes to local servers, RPC is definitely not the most efficient way to transmit arguments. Other schemes

[1] Remote Procedure Call
[2] Object Remote Procedure Call

to achieve call to local servers are abundant and include pipes and shared memory among other. A client who explicitly asks for a local or remote server also should have the option to ask for a specific communication method. The reason for this is mainly that a client knowing that it will use the object in a sporadic way might not care about the communication method. The client who will make excessive use of the object might on the other hand want to make sure that only the fastest communication method is used.

One thing that is absolutely needed in order to make a distributed system of objects usable is the possibility of passing objects between clients. Extra care must be put into this in order to make it safe in regard to authentication and encryption of arguments.

4.3 System Functions

In this section the system functions implemented in the prototype are presented. Only a subset of the features presented in the previous section are implemented as the time available was limited, and some of the features were not needed in order to accomplish the goal of the research.

The system only offers four different functions:

Registering a server of a specific component.

Unregistering a server.

Creation of a remote object.

Method calls on a remote object.

Each presentation is divided into three parts. First a short description of the function is presented. It gives some background to the problem addressed. Then the conditions that must be met before the function can be used are given. Some functions depend on the execution of other functions before they can be used. Lastly a more in-depth presentation of the function is given. It further specifies the problems and peculiarities of each desired function.

There are some preconditions that are common to all the desired functions. They are presented in the following section.

4.3.1 Preconditions For All Functions

The system is built on several different parts. Some of these play a very central role in all the functions presented below and are therefore explained in a little more detail here. Parts and components specific to a certain function is described when it is first mentioned in the text.

Some sort of database is needed where running servers can register themselves and inform the system of what components they supply (see section 3.1.3). How this database is designed or how the servers access it, is not decided. For now we just assume there is some way for the servers to access the database.

4.3.2 Register A Server

This function describes the problems surrounding registration of a server of a component.

Preconditions

None, except the ones mentioned in section 4.3.1.

Function Description

A server that offers components must somehow tell the system that its components are available. The way to do this is to register it with the system.

When a server starts it is supposed to register itself with the system. This is done in order to make the components it offers available to clients.

Each server implements at least on method of IPC, this too is registered in the system.

A server must also have some sort of connection point where it can be contacted. Together with the IPC method this provide enough information to make it possible for clients to contact the server.

4.3.3 Unregister A Server

This function handles the process surrounding the unregistering of a server.

Preconditions

The server must of course be running and be registered (see section 4.3.2).

Function Description

A server that stops executing has to tell the system that its component is no longer available to clients. A server should not stop executing as long as it has clients who are connected to it, unless a non-recoverable error occurs.

As the server prepares to exit it has to remove all its entries in the database.

4.3.4 Creating A Remote Object

The first obstacle stumbled upon is the case of how to create an object on the server side.

It might be desirable to be able to specify the method of IPC to be used.

Preconditions

The component is in fact not implemented in an in-process server, but by a separate process that is running. The DLL[1] implementing the object proxy is registered with COM under the component ID previously mentioned.

There must also be a sender component registered with COM that implement the IPC method used by the server.

Function Description

Given a certain component's ID the program would like to obtain an interface implemented by this component.

On the server side an object must be created and then an ID for this object must be transmitted to the client. The ID is needed for future references to the object.

4.3.5 Method Call On A Remote Object

The natural way to extend the function presented in section 4.3.4 is by calling a method of the interface.

Preconditions

The client needs to have obtained an object proxy in the manner described in section 4.3.4.

Function Description

This is probably the most central function. Without the ability to call a method on a component the whole system is meaningless.

By using the ID obtained through the execution of the function presented in section 4.3.4 the correct object is referenced on the server side. Arguments are passed through a communication channel and then given to the actual object. The results are passed back in the opposite direction.

[1] Dynamic Link Library

O'Brien: "Do we make a run for it?"
Sisko: "We make a run for it all right. Run right at it."
O'Brien: "Ahh. Pattern suicide."
- from DS9

5. Design

This chapter presents the design and our design choices. The presentation has been divided into two parts. First the overall design, where the behaviour and design ideas shared between several of the different components is presented. Secondly, the more detailed design issues for the separate components are given.

5.1 Overall Design

One of the overall design decisions that was taken in an early stage was to make it as easy as possible to change between different solutions (see section 4.2 for a discussion about why this is important). This proposed using a highly modularised design. The different areas where work needed to be done was:

- Marshalling.
- Interprocess communication.

Both of these areas are covered in detail in the following sections.

Another decision that was made was to try to use design patterns as much as possible. This will ensure that the design is both sound and robust, it also helps the understanding of the design from a reader's point of view.

5.1.1 Component Object Model

The many different flavours of COM[1] available make it inevitable to have to choose one. In this decision several things has to be weighed in. The things that have to be considered are:

[1] Component Object Model

- Closeness to the target environment.
- Availability to source code.
- Availability of an IDL-compiler supporting distributed components.
- Support for distributed components.

Among these four the first two were deemed most important. The closeness to the target environment (Linux) is a must to get reliable data from the tests. The source code is an important asset since the design of our implementation is very dependent on the design of the core system. The other two criteria are of minor interest since these parts can be implemented by hand for the evaluation in this thesis. For the system to be complete at least an IDL[1] compiler must be written, or extended, to generate code with support for distributed components.

COM for Windows is out of the question because of the difficulties with obtaining the source code. The implementation that is left after this is XPCOM[2], not an ideal choice since it does not fulfil the last two criteria. But the availability of the source code and the fact that it runs fine on Linux makes it a candidate.

The proposed design for extending XPCOM in order to accomplish the needed tests can be viewed in Figure 9. An effort is made to keep the changes in XPCOM minimal and to instead extend it. This design uses the Forwarder-Receiver ([11], 307 pp.) design pattern in combination with the Client-Dispatcher-Server ([11], 323 pp.) design pattern.

The system consists of several parts. As can be seen in Figure 9 below there are mainly three parties involved, the client, the server and the Server Management System (SMS). The SMS itself is made up of two parts, first the Server Manager (SM) and then a set of interfaces used to access the SM.

[1] Interface Definition Language
[2] Cross Platform Component Object Model

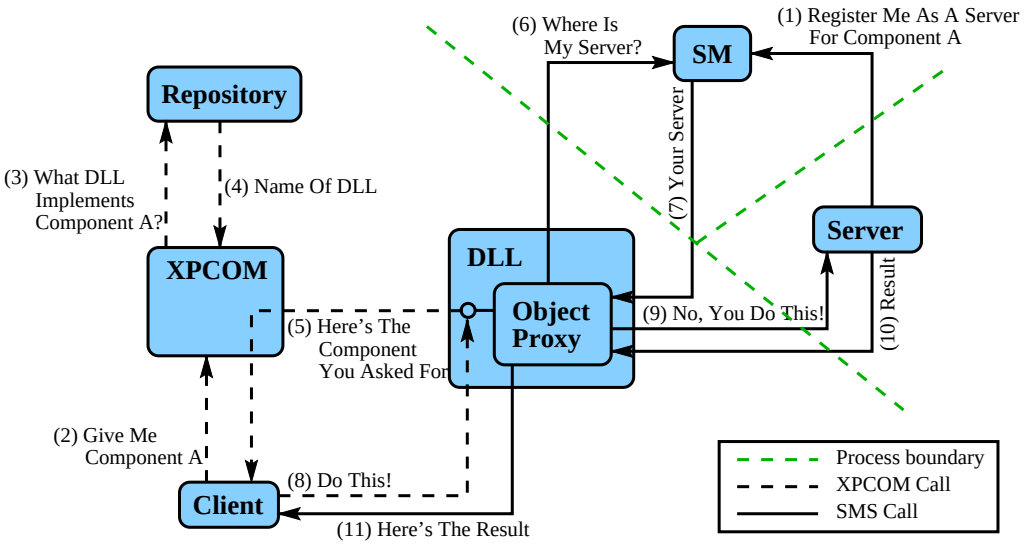

Figure 9: The proposed architecture of our system

The steps involved are the following: First the server registers itself with the SM (1). When the client is started and a component asked for (2) XPCOM looks into its repository to find the DLL[1] implementing the component (actually it implements the object proxy) (3). Once the name of the DLL is found (4) it is loaded into the memory of the client (5). Once loaded the object proxy contacts the SM in order to acquire the server port (6), once it is returned (7) it is used for all the future connections between the server and this object proxy. In order to execute a method call (8) the object proxy relays the call to the server (9) via the connection established earlier. The result is returned first to the object proxy (10) and then to the client (11).

In the figure above (Figure 9) the filled arrows denote parts needed to be added, while the dashed arrows are already present in XPCOM.

In this design it is not possible for the server to be started if it is not already running. The reason for this is that adding this functionality is deemed as a minor feat but it also requires a change in the representation of the repository. That would be a major change to the core system of XPCOM and hence is not desirable.

It should be noted that on this level nothing is said about several servers for a single component, a situation that might be desirable. The question about what to do if the exactly same server is started twice is also dodged gracefully.

5.1.2 Marshalling

Marshalling is, as was described above, concerned with representation of data. Agreeing on the representation of arguments and data to be

[1] Dynamic Link Library

transferred is crucial if the data is to be transferred between two different architectures. This research is not concerned with communication between different architectures since the processes that will communicate will be executing on the same computer. This made it easy to come to the conclusion that marshalling into a network representation of data is not necessary.

The use of an IDL compiler tool is tightly coupled with marshalling, and this is one thing we do need. To require that the programmers themselves make the code for packing the data would only result in errors or that nobody uses the system. The use of an IDL and the accompanying compiler tool is therefore necessary.

Changing representation of data is not necessary, because of the reasons previously stated. There is however one exception. In order to successfully pass interfaces between processes there is a need to marshal them. An interface is simply a pointer to an object in C++. This pointer can not, of course, be transferred as it is between two processes. The way to marshal an interface is described in [5] (206 pp.).

In conclusion can be said that marshalling is only needed for passing interfaces. However, the benefits of using an IDL compiler tool will be utilised.

Unfortunately the available IDL compiler tool for use with XPCOM (*xpidl*) is not capable to generate source code for stubs and object proxies. This makes it necessary to put more work than earlier expected into coding the components. In the future, an IDL compiler tool capable of generating everything needed for distributed components must be written. In order to perform the tests at hand it is acceptable to use *xpidl* and write the rest by hand.

Since this thesis is more geared towards comparing IPC methods than implementing a component system, marshalling of interfaces will not be considered in our prototype, but there is nothing inherent in the design that prevents adding it later on.

5.1.3 System

As can be seen in Figure 9 above, an object proxy on the client's side will handle the communications. This proxy will start by acquiring an initial connection to the server. If this connection is used for all communications with the object or if a new one is established per object is up to the implementation. If the same connection is used for all objects, some sort of identification must be sent with all requests, to uniquely identify the object that is actually called. Potential problems regarding synchronisation issues have to be handled by the server.

The runtime system, consisting of SM and the object proxy, has to be changed slightly for each method of IPC[1]. The common denominator between all the methods to be evaluated is the need to set up a connection between the client and the server. After that what remains is but to send requests and return values over the connection.

To achieve a piece of shared memory to pass the arguments in it is necessary to acquire the name for the object to be shared. When pipes are used the path to the servers pipe is needed. In order to use Doors/Linux the name of the door has to somehow be transmitted. So, the runtime system will still be passing around objects, but different kinds of objects depending on the mechanism used for IPC.

Once the connection is established the server has to take care of incoming requests and handle them appropriately. The return value is then transferred back to the client.

5.2 Detailed Design

In this section the design is presented in more details.

5.2.1 Cross-platform COM

The necessary extensions involve making it possible to, from within a client, locate a running server's access point, whether it be a path to a Door, a named pipe or the port it is listening to. As has been explained above the way chosen to do this is by having an SMS where each running server must register itself and which clients then can use to locate the desired server.

Two different ideas for the SM will be presented here. The first one makes use of a process, which must always be running, to manage the registration of new servers. The second one utilises an external database to achieve the desired result.

Implementing the SM as a separate process makes it necessary for the client to first connect to the SM in order to register itself. The client would, in order to acquire a connection to a server, have to first connect to the SM to get the location of the server and then it can continue to establish the actual connection. When the server stops running it will have to unregister itself by once again connecting to the SM. One is facing several problems in this scenario. First of all the SM has to be robust in the sense that it can reliably handle many connections in a short amount of time. The registration process also leads it to manage a list of available servers with their respective paths of access. Managing this list while there is several connections coming in imposes the need for synchronisation. When taking

[1] Interprocess communication

all of this into account it makes the idea of making the SM into a process of its own quite a cumbersome way to take.

The second idea for implementing the SM emphasises on the fact that the role of the SM as described above is in fact the same as the role of a database. Basing the implementation of the SM system on a database relieves the programmers to a large extent. It also makes it possible to choose a suiting environment for implementation, where things like robustness and reliability can be weighed in. All the objections to the idea with making the SM a separate process are vanished when it is replaced by a database. As a matter of fact it introduces some interesting features to be used. Among these is the use of transactions, which make it possible to backtrack if for some reason a server dies while registering.

The set of XPCOM interfaces providing access to the SM would look almost the same no matter which of the designs chosen for the SM. An overview of the design is presented in Figure 10. And as can be seen in Figure 11 the component will implement two interfaces.

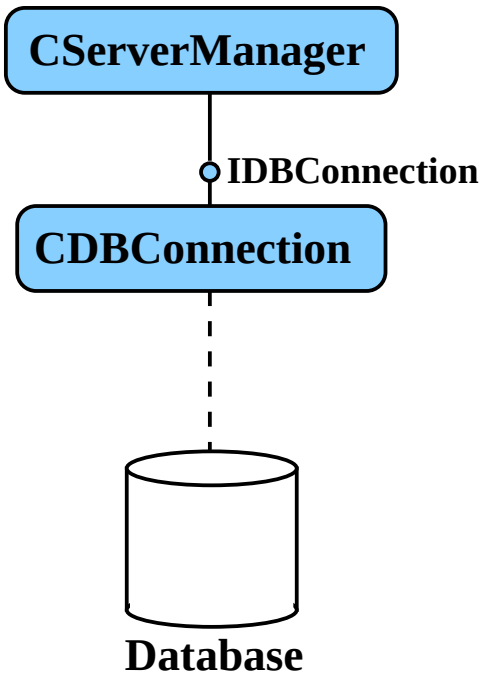

Figure 10: The ServerManager's connection to the database

This design, seen in Figure 10, where *CServerManager* knows of an *IDBConnection* makes it possible to change the SM without having to make big changes to the component. Changing the SM will demand a change in the *CDBConnection*, but this will have to be done no matter what the design looks like. This design is simple and adequate to handle the functions presented in section 4.3.

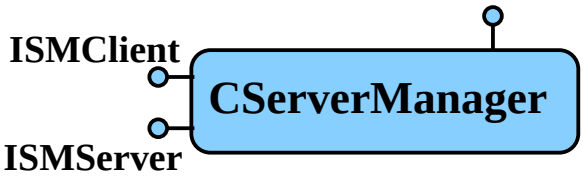

Figure 11: IServerManager

The *CServerManager* will, conceptually, work in two different modes. For the server it will be a component where one registers oneself. For the client, on the other hand, it is the component that helps in locating the desired server. This makes it only logical to offer two different interfaces to the component.

The time limit imposed on conducting the research for this master's thesis made it easy to choose the design where the SM is implemented using a database. This is not only the best choice in the aspect of difficulty of implementation; it is also the most general implementation and allows further development.

5.2.2 Marshalling

The system is designed to be as modular as possible. This makes adding marshalling later on easy. This easiness is desirable since the system might some day be extended in ways where marshalling is actually needed.

By implementing the marshalling as a COM interface the desired modularity is easily achieved. The sending of the data will also be done through an interface. There has to be some sort of connection between these two interfaces since they both will work on the same data, the marshaller (*IArgMarshal*) will prepare it before the sender (*ISendAndReceive)* sends it away. When data comes back *ISendAndReceive* will receive it and then the unmarshaller (*IArgUnmarshal)* will be used to extract the actual results. The obvious approach to this problem is to let the two interfaces somehow share a buffer that they work upon. But there are several ways to achieve this.

The naïve approach would be to let *IArgMarshal* and *IArgUnmarshal* work on a simple buffer that is not connected in any way to the *ISendAndReceive* that will be used. The result of the marshalling of a variable would simply be written into this buffer and later the buffer is turned over to the *ISendAndReceive* for sending it. As can easily seen by the astute reader this will result in unnecessary writes in some cases.

To get around the problem with unnecessary writes into the buffer we let *ISendAndReceive* offer a factory method (see Factory Method Pattern [7] 87 pp.) for the buffer. In this way the buffer will be tailored to suit the IPC used. *IArgMarshal* (and *IArgUnmarshal*) will use this factory method in order

to retrieve the interface to the correct buffer. The design is described in Figure 12.

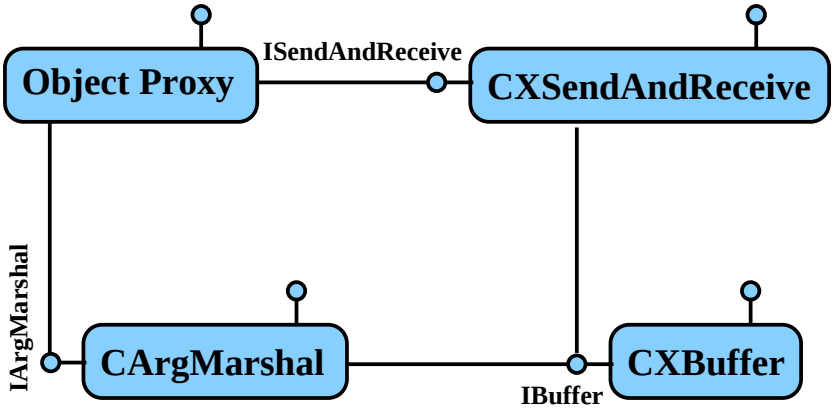

Figure 12: IBuffer

Performing marshalling would then be a simple matter of obtaining an *IArgMarshal* and an *ISendAndReceive*, then giving *IArgMarshal* the *IBuffer* supplied by *ISendAndReceive*. Then *IArgMarshal* does the actual marshalling. When done, one just tells *ISendAndReceive* to send whatever is in the buffer. Unmarshalling is done in an analogous way.

5.2.3 System

Object proxies on both sides will always handle the interprocess communication between a server and its client. This is done in order to make it easy for the programmer and relieve her or him from the tedious task of implementing the communication. In the future the task of producing this code will be fully automated through the use of an IDL compiler tool. The lack of an IDL compiler tool capable of this for XPCOM necessitates writing this code by hand for now.

In this section the design is presented together with an explanation of the intended use. In the following sections the peculiarities for the different techniques for IPC is presented.

5.2.3.1 Object Proxy

In order to make it easy to change the IPC mechanism the design is made as modular as possible. An overview of the idea behind the design is presented in Figure 13.

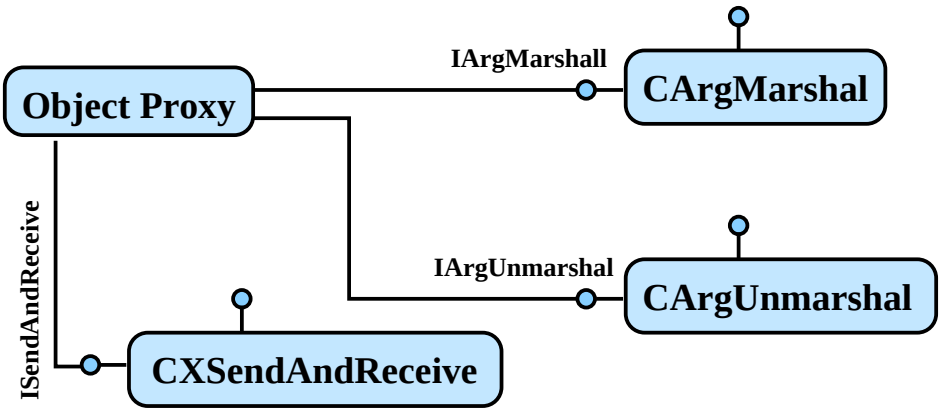

Figure 13: The role of the object proxy

The proposed design makes it easy to change the behaviour of a single part. By exchanging the components implementing *IArgMarshal* and *IArgUnmarshal* a different scheme for marshalling is deployed. In a similar way exchanging the *ISendAndReceive* easily switches the mechanism for IPC. As can be seen the component will not rely on the assumption that both marshalling and unmarshalling is done by the same component.

5.2.3.2 Server

A server can be divided into two parts. The first part handles the request on an initial stage and unmarshalls the arguments before passing them on to the other part, the object itself. The opposite way is taken before returning the results to the object proxy.

The schematics of a server are shown in Figure 14 below.

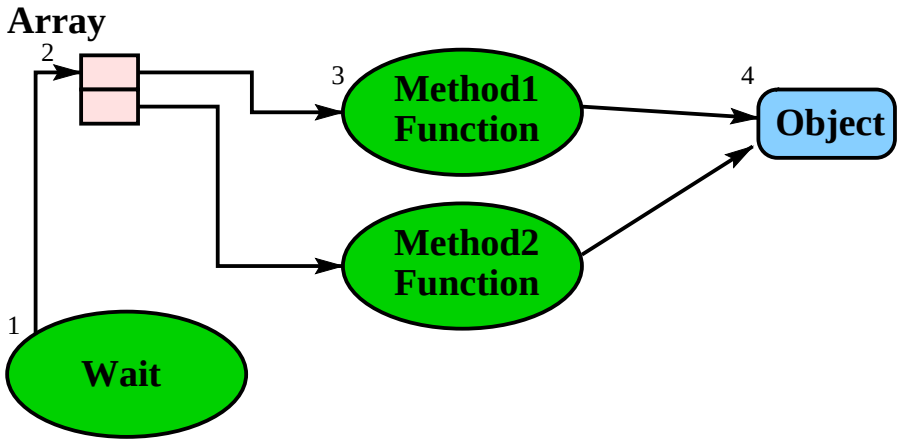

Figure 14: Server schematics

A request is sent to the listening procedure (1). A service identifier is extracted and then used to index into the array of procedures (2). The indexed procedure is called (3). In this procedure the arguments for the

method are extracted and the method is invoked on the correct object (4). The results are then returned back to the listening procedure. The results are then packaged appropriately before sending them to the client.

5.2.3.3 Locating A Server

When it comes to looking up the server there are at least three different ways to accomplish it. The first scheme achieves flexibility through object oriented design, the second is slightly faster but less general. Finally, the third one is a hybrid between the two first. Figure 15, Figure 16 and Figure 17 depicts the three different designs proposed. In some figures a component named *CXSendAndReceive* appears. The X is exchanged for the different IPC methods e.g. *CRPCSendAndReceive*, *CDoorsSendAndReceive*, etc.

The first alternative uses dynamic type inference. By letting the database include information about the type of the server that implements the desired component it is possible to develop one general component which implements *ISendAndReceive*. Utilising the *ISMClient* interface the *CXSendAndReceive* can decide which kind of connection it needs.

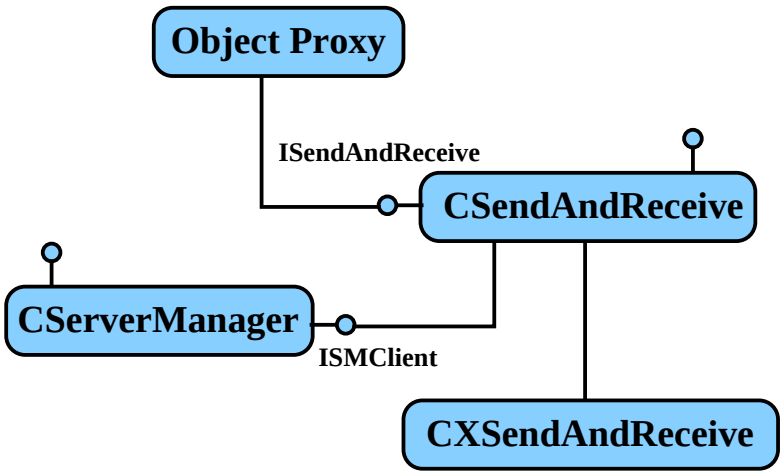

Figure 15: The dynamic approach to SendAndReceive

The second alternative is compile-time static. By deciding at compile time what server to use for a given object proxy. Instead of implementing one general *CSendAndReceive* component, several specialised ones are implemented, one for each type of IPC.

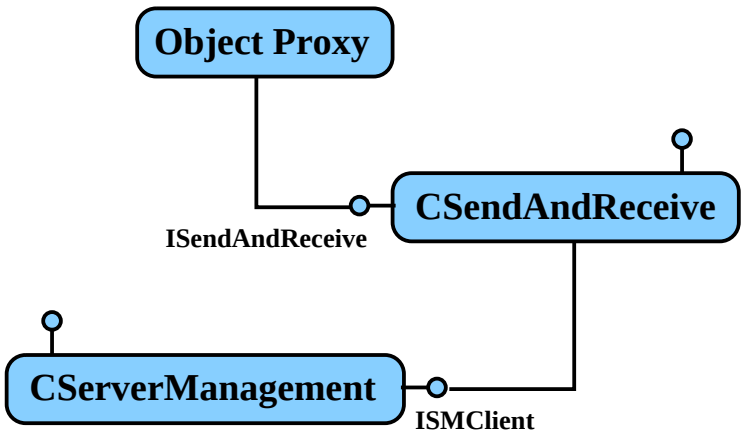

Figure 16: The static approach to SendAndReceive

A third alternative would be to let *ISMClient* offer a factory method for the sender (see Factory Method Pattern 87 pp. in [7]). The factory method in *CServerManager* would look at the registered servers for this object proxy and chose a suiting one. Depending on the IPC method supported by the server the correct *CXSendAndReceive* is created and its *ISendAndReceive* is then returned to the object proxy.

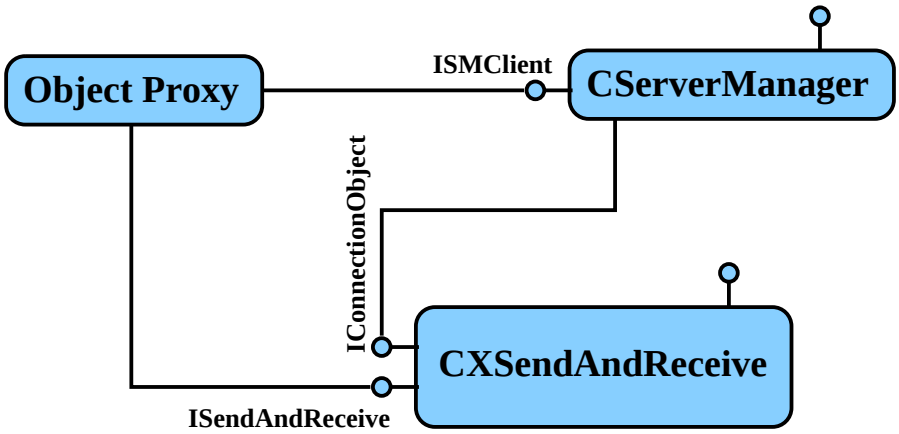

Figure 17: The improved dynamic approach

Alternative one is as general as can be while alternative two is static after compiling. The generality offered by the first design is desirable but since the third alternative offers the same amount of generality without relying on object orientation it gives a general feel of “COM closeness” and because of this it was chosen above the others.

5.2.3.4 Omissions

Some issues that were neglected earlier demand attention at this point. Allowing several servers for the exact same object and using the same IPC might be desirable in order to achieve a general distribution of objects. It

also makes it possible to have to make a choice of which server to use. This would make the system too big for this thesis, but the design makes it feasible to extend the system in the future with this feature.

While servers are expected to keep on running forever they sometimes do not quite live up to the expectations. Detecting the sudden demise of a server requires some sort of regular checking. To make the system reliable and truly usable a mechanism for detecting the death of a server and, if needed, restarting it is necessary. This part of the system will however not be considered in this thesis.

5.2.4 Interprocess Communication

All the different methods of IPC are implemented through separate components implementing the *IConnectionObject* interface. Efforts have been made to make these components isolated from the object proxies. Peculiarities with the different designs are discussed in the following sections.

The general design of the different components for IPC is shown in Figure 18 below.

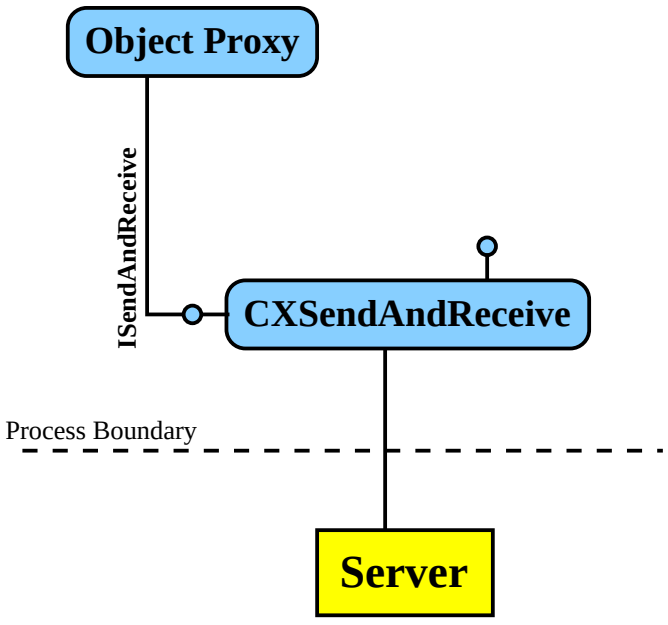

Figure 18: SendAndReceive design

5.2.4.1 Standard Remote Procedure Call

RPC[1] is the only available method of interprocess communication in the implementation made by Microsoft. Through their extension to RPC, called

[1] Remote Procedure Call

ORPC[1], they achieve full distribution with just one implementation. It is full in the sense that both local and remote servers are using the same means of IPC.

Usage of RPC for IPC is achieved through the component called *CRPCSendAndReceive*. As the name suggests it implements *ISendAndReceive* and also *IConnectionObject* (see Figure 19 below).

Figure 19: CRPRSendAndReceive

Including RPC is done to get something to compare the other methods with. Since RPC is the original method one gets, in this way, a reasonably good index to compare the others to is achieved.

5.2.4.2 Unix Pipes

Named pipes are used when the component *CPipeSendAndReceive* provides the *IConnectionObject* and *ISendAndReceive* interfaces (see Figure 20 below).

Three pipes are involved in the process. One pipe is created by the server and is used to establish connections with clients. The other two is created per client by the clients and is used for all communication with the server. The pipe created by the server is only used to transfer the names of the other two named pipes.

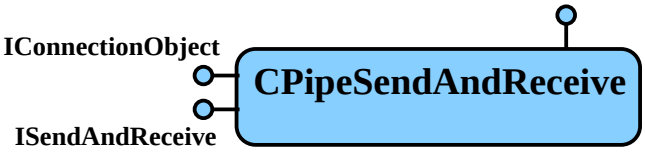

Figure 20: CPipeSendAndReceive

5.2.4.3 Doors/Linux

Usage of Doors as IPC is achieved through the component called *CDoorsSendAndReceive*. The component offers the two interfaces that are needed in order to function as an IPC component, *IConnectionObject* and *ISendAndReceive* (see Figure 21).

[1] Object Remote Procedure Call

Designing and implementing IPC using Doors is very straightforward. The server creates a door, attaches it to a file and registers the file name through the *ISMServer* interface. The client fetches the file name through the *ISMClient* interface, opens the file and then calls the door using the door descriptor. As door argument, a buffer with marshalled data is supplied. The returned data is a buffer with the marshalled result.

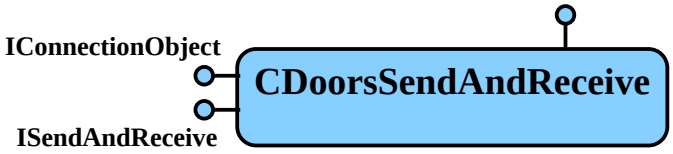

Figure 21: CDoorsSendAndReceive

5.2.4.4 Shared Memory

Sharing memory between processes can, as was discussed earlier, be achieved in three different ways. The implementation uses one of these and the component that supplies this IPC method is called *CShmSendAndReceive*. The component offers the two interfaces that are needed in order to function as an IPC component, *IConnectionObject* and *ISendAndReceive*. Figure 22 shows a conceptual view of the component

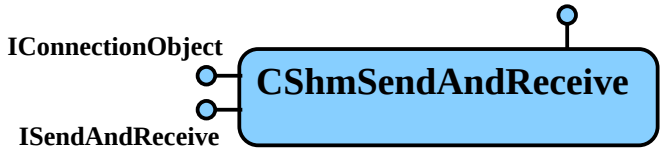

Figure 22: CShmSendAndReceive

5.3 System Functions

The design is supposed to address the problems encountered in the description of the system functions presented in section 4.3. That it in fact does so is shown in the following sections. For each of the system functions we have provided an interaction diagram to clarify the function design, Figure 23 trough Figure 26. The interaction diagrams follows the UML standard notation [17].

5.3.1 Register A Server

Please refer to section 4.3.2 for a detailed description of this function.

When starting, a server has to register itself with the SM in order to make its components available to clients. By obtaining the *ISMServer* interface from a *CServerManager* all the needed information can be passed to the system.

The data entered into the database has to differ slightly between different kinds of IPC methods. The offered components' ID and the type of IPC used by the server are entered together with a string (path) that tells future clients how to obtain contact with the server. This string is different from method to method. The exact interpretation of this string is presented together with the details for each specific IPC-component presented in section 6.4.

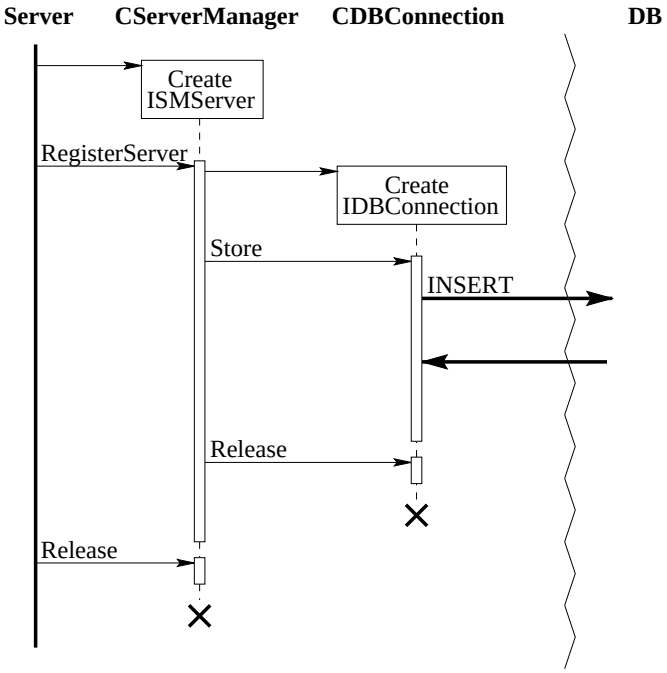

Figure 23: Interaction diagram for registering a server

5.3.2 Unregister A Server

Please refer to section 4.3.3 for a detailed description of this function.

As was the case above the server gains access to all the needed system functions by obtaining the *ISMServer* interface from *CServerManager*.

Identifying its own entry in the database is simply a matter of matching the component ID and the IPC method against an entry in the database.

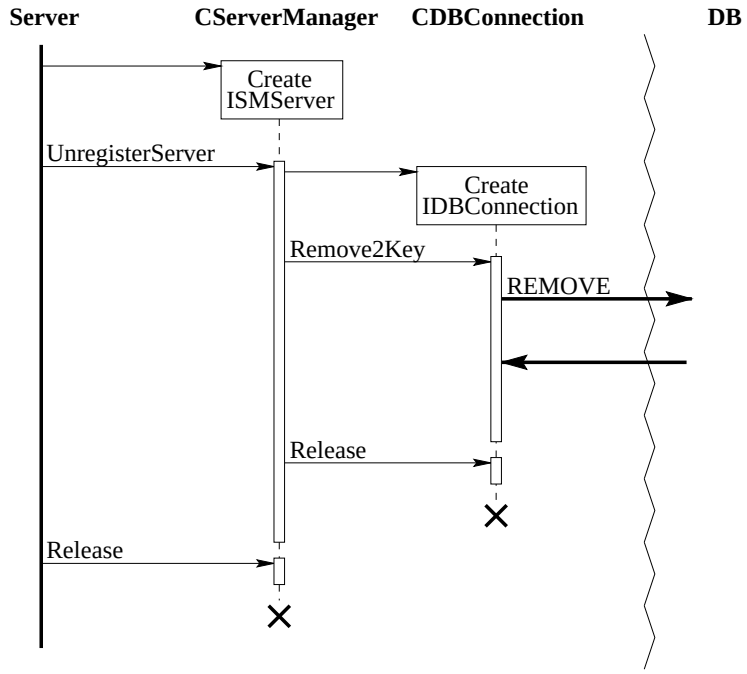

Figure 24: Interaction diagram for unregistering a server

5.3.3 Creating A Remote Object

Please refer to section 4.3.4 for a detailed description of this function.

Creating a remote object is a quite involved process. The first thing that should be noted is that it is actually the object proxy that is registered under the desired component's ID. Which in turn causes COM to create an instance of the object proxy instead of the real component.

Upon creation the proxy obtains the *ISMClient* interface of *CServerManager*. Using this it can then obtain the *SendAndReceive* of the correct *CXSendAndReceive* component depending on the IPC method used by the server. But before the *ISendAndReceive* interface is returned to the proxy an object is created on the server side. *CServerManager* initiates the creation by using the *IConnectionObject* interface of the *CXSendAndReceive* component. The ID of the remote object is kept within the *CXSendAndReceive* component.

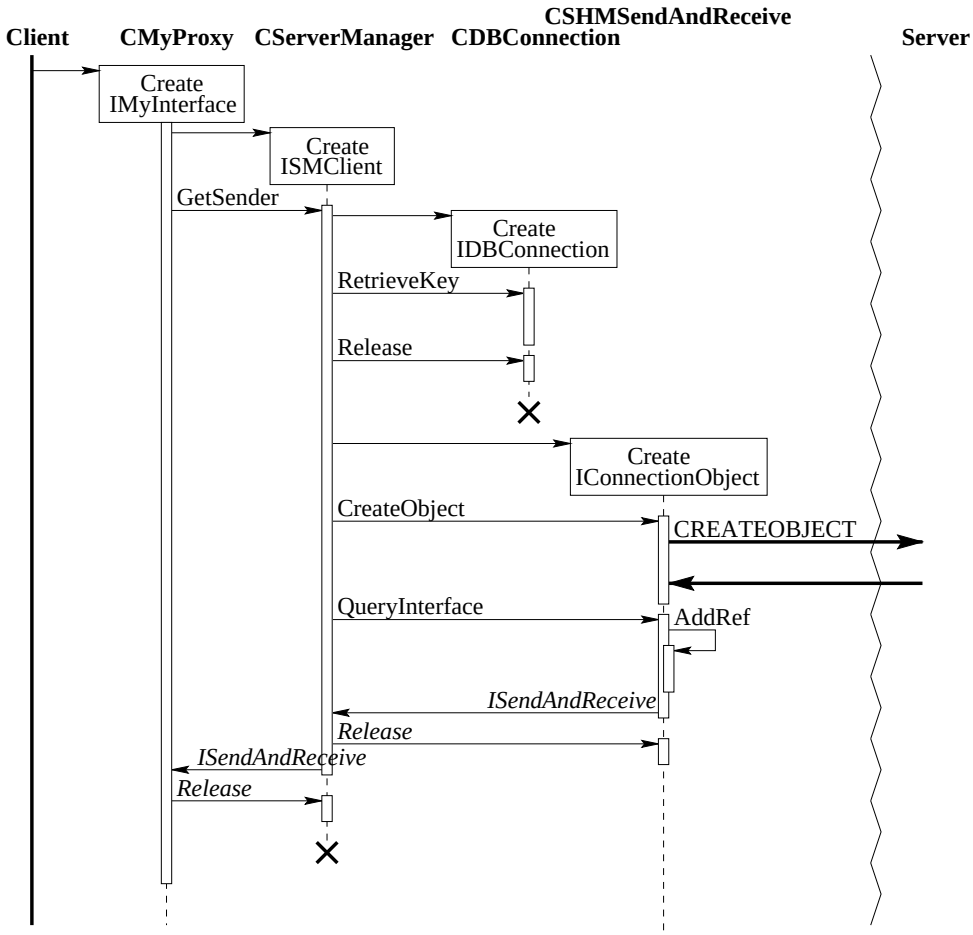

Figure 25: Interaction diagram for creating a remote object

5.3.4 Method Call On A Remote Object

Please refer to section 4.3.5 for a detailed description of this function.

The called method is entered on the object proxy. It obtains an *IBuffer* interface to use from the *ISendAndReceive* interface it kept from its time of creation (see section 4.3.4).

It then obtains the *IArgMarshal* interface from *CArgMarshal* and tells it to use the *IBuffer* that it just got. An ID of the method is marshalled first and then the arguments are put into the buffer. When done the proxy tells its *ISendAndReceive* to send the buffer and wait for a reply.

The details regarding how the buffer is transmitted to the server differs depending on the IPC method used and this is presented in detail in section 6.4. When the data arrives it is handed over to a *CArgUnmarshal* component through use of *IArgUnmarshal*. This interface is passed on to a server stub, which handles the unmarshalling of the arguments, and calls the method on the actual component.

The return path is similar, after receiving the results from the methods they are marshalled into a buffer that is sent back to the client. They are then unmarshalled and used as the result of the method call on the object proxy.

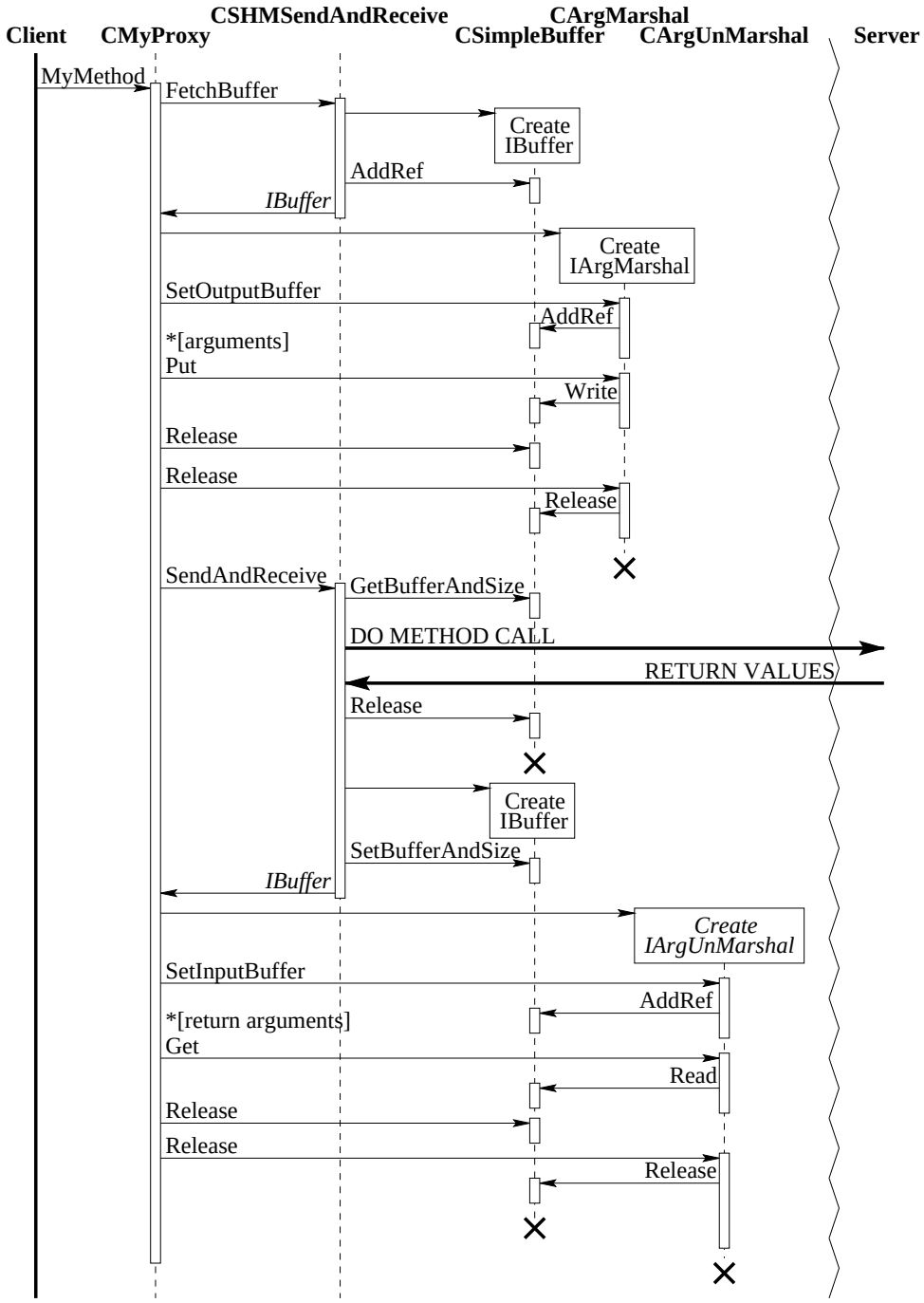

Figure 26: Interaction diagram for method call on a remote object

5.4 Evaluation Suite

Evaluating the implementation properly is important in order to acquire the desired data. The nature of this study is to compare different ways of

communication between processes, this implies that measuring time is of the essence.

There are however several other aspects that are of interest, one of them is usability from a programmers view.

These aspects, time, usability etc. will be discussed in the following subsections. Because of the central role played by the time aspect and the usability aspect they have been honoured with a subsection each, while the others are discussed briefly in the last subsection.

5.4.1 Evaluating Time

Two different time intervals are of interest. First it is the time it takes for a client to get an exclusive connection to a server, called time to connect from here on. The second interesting interval is the time it takes for the client to make a method call on a component in the server, from here on this interval is called time for method call.

5.4.1.1 Time To Connect

Time to connect is interesting in the initial phases of an object proxy's life span. Once the exclusive connection from the object proxy to the server is established it has never to be done again, unless this connection is mysteriously lost.

Measurement of the time to connect is conducted by making a large number of connections to one server by constructing objects.

5.4.1.2 Time For Method Call

The time for method call presumes that there is a working connection to the server implementing the component. This is of crucial interest since this time has a high impact when using distributed components. There are actually two different aspects of a method call that is of interest. First a simple method call with none or very little data that has to be transferred. The second is the case were marshalling has influence on the speed.

Timing a method call is done similarly in the two cases described above: by performing a large number of method calls. In the first case the evaluation program used will perform a large number of calls on a method taking a small amount of data as its arguments, While, in the second case, a larger amount of data will have to be transferred between the client and the server.

5.4.2 Evaluating Usability

One might object the described type of testing is too geared towards special uses of components. In order to show the usability of the design an

evaluation program showing a possible future use could be made. The future use might be modelled by letting two programs being each other's servers, and thence they are also each other's clients.

It should be noted that the usability of the system is not effected by the choice of IPC method for a server.

5.4.3 Evaluating Other Aspects

There are aspects of a system of this kind that are a lot harder to evaluate by simply making a program and run it. Some of these aspects are listed in the following list:

- Robustness
- Correctness
- Security issues
- Memory consumption

In the following paragraphs these aspects will be dealt with briefly. The reason for not dealing with them more thoroughly is that, even though they are important, they do not have a significant impact on the work described in this thesis. They are, however, important enough to be mentioned, and in the making of a production system built on the work in this thesis they would be crucial.

5.4.3.1 Availability

Robustness deals with the matter of reliability of the system. This is something that has an impact on both the server and the client. Servers expect that the database where they registered stays up and available to clients. They also rely on the clients for releasing components properly before exiting, otherwise the server will experience memory leaks and might be rendered dysfunctional.

Clients rely on the database for finding servers, without the database the clients will not be able to perform its task satisfactorily. They also rely on the fact that once a server has been contacted and a remote object has been created it will be available for as long as the client might need it. As can be easily seen the component proxy is the glue that connects the client and the remote component, without the existence of a proxy the client is practically helpless.

The system has no problems with robustness built into its design. By choosing a reliable database both servers and clients can be sure of its availability. Since the clients will be programmed by the future users of the system, it is hard to assure that clients will always be well behaved and

release all its remote components. A way around this would be for a server to release unused components after a while. This approach is however not applicable, since we cannot prophesise about the future clients of the system. While one might use a remote component very frequently another might make calls on its remote component scarcely.

The other things mentioned above are dealing with automatically generated code. The use of an IDL compiler tool will produce the code for the server as well as code for the component proxy. Making this code robust is quite easily handled since it is not left to the individual programmer to make it.

5.4.3.2 Correctness

The issue of correctness is very important. It deals with whether the system actually does what it is supposed to in every step. Both the client and the server have to be correctly implemented and behaving in a manner acceptable to the system. The future use of an IDL compiler tool will assure that this is the case.

5.4.3.3 Integrity

One might think that since the system is designed to be used on a single machine there is no need to pay attention to the security of the system. There are however several possible scenarios where malign programmes perform attacks on the system. Either in an attempt to bring the system into a state where it is useless or in an attempt to read and possibly alter data transferred between the client and the server.

Bringing down the system can be done in a number of ways. The most obvious way is to make an attack on the database in order to make it impossible for servers to register and for clients to find its servers. But this type of attack is something that we expect the database to handle appropriately.

A less obvious way of attacking the system is to register evil components in COM with the same UUID[1] as the components in the system (e.g. *CServerManager*) which would result in clients and servers using this component instead. Making sure that attackers can not register wicked components is something that we expect the COM-system in use to handle (in our case XPCOM) in an orderly fashion.

Performing attacks on a single server is quite easy, but the way to go about it differs a bit from server to server. The servers that depend on the existence of files will be helpless if the files are deleted or altered. In the same way other objects shared between the server and the client can be attacked. These issues should be dealt with if the system is to be used in an

[1] Universal Unique Identifier

environment where malign processes are expected to run. It is however something that is a joint issue between the system and the underlying operating system.

It is also a simple thing to replace a running server with another by altering the database to direct clients in another direction. But once again this is something that should be handled by the database.

In conclusion can be said that the issues of security is not a trivial one, and solving it includes taking the underlying file system, database and operating system into consideration.

5.4.3.4 Memory Consumption

There is really no inherent memory hog in the design. Since the design only is concerned with interfaces the choice of the implementations of these interfaces are free to employ whatever algorithms they choose. The design has, however, been done with memory consumption in mind and therefore allows the implementation of interfaces to utilise peculiarities regarding connection management and transfer of data within each IPC method. It should therefore not be anything in the design that prevents a memory efficient implementation.

“I want facts, not poetry.”
- Unknown, TOS

6. Implementation Details

Here are different decisions for the implementation presented. This chapter might be a bit technical in its content, and a familiarity with UNIX, particularly Linux might be helpful. In the last section the evaluation suite is described.

6.1 Cross-platform COM

When designing the system one of our goals was to keep the changes of XPCOM[1] to a minimum. The design fulfils this goal; in fact there were no changes at all to XPCOM.

In order to fully incorporate the system in XPCOM a number of changes will be needed. First of all the repository of XPCOM needs to be extended to handle out-of-processes. It is also necessary to change the interface for component creation present in XPCOM. The extension needed is to let the programmer choose what type of server is acceptable and, in the case of out-of-process servers, what type of IPC[2] to use. It is also needed to add the capability of starting servers to XPCOM, since it is not a reasonable constraint that the server has to be running before a client is started.

In order to relieve the programmers an IDL[3] compiler tool capable of generating both object proxies and stubs for the server is needed. Ideally the *xpidl* program would be extended in this way.

6.2 Marshalling

Marshalling into some sort of network representation is not done. However one might argue that the packaging of data that is done via the *IArgMarshal* and *IArgUnmarshal* is in fact marshalling into a network representation that just happens to be the same as the internal representation of the computers where the prototype was implemented. There are however a few more details that makes it correct to say that we actually do marshalling.

The *CArgMarshal* component implemented in the prototype does put in some extra information into the buffer. This is done in the marshalling of arrays, the size of the elements in the array is known by both client and server from compile time, but the number of elements is not. This

[1] Cross Platform Component Object Model
[2] Interprocess Communication
[3] Interface Definition Language

necessitates putting the number of elements into the buffer before the contents of the array.

One thing that the prototype does not implement is marshalling of interface pointers. In order to obtain a useful system this has to be added together with mechanisms for passing them between clients properly.

6.3 Server Management System

The Server Management System (SMS) acts as a database and works in two modes. The first handles the registration and unregistration of servers. The other handles the clients' requests to locate servers. These two interfaces, *ISMServer* and *ISMClient*, are implemented in the component *CServerManager*.

CServerManager does not contain the database itself, just a connection to it. The actual database is implemented through the interface *IDBConnection*. Our implementation of *CDBConnection* uses an SQL[1][13] database called mySQL [14]. The choice of a SQL server might seem a bit strange, but it is just for our prototype and not intended for the final system (see chapter 9).

The SQL database gives some major advantages. Firstly, it makes synchronisation unnecessary since the mySQL server handles this automatically. Secondly, we do not have to care about how the data is stored. Thirdly it was really easy for us to use it through a library called mySQL++ [15], a C++ abstraction for the mySQL database.

Thanks to mySQL, our Server Manager (SM) will be fairly robust.

6.4 Interprocess Communication

The following sections discuss the separate implementations of the components used for IPC. They also bring up issues regarding the servers where it is needed.

Some parts of these discussions are common to all the different implementations and therefore should be identified as such. Common problems are the following:

- How to identify the referred object?
- How to identify which service is requested?

The problem with identifying the particular object which this request is referring to can be solved in one way. By letting each object be connected to a unique identity and then letting the proxy supply this identity every time

[1] Structured Query Language

a request is made. There exists several ways of implementing this, the two ways chosen in this work are the following:

- Unique identity number
- Unique port

Letting a unique number identify an object is a simple and straightforward way. This scheme demands that this identifier is sent together with the arguments for every method call. This can be worked around in the cases where the IPC method permits creating new ports of entry into a server dynamically. By letting a port be connected with a certain object one can avoid sending the identification data back and forth. The identifier is preferably kept in the component implementing the *ISendAndReceive* interface, i.e. not in the object proxy.

Numbering each method in the interface and then transmitting this number with the arguments easily solves the second problem. It is possible to make a solution similar to the one above to avoid sending this number, but this would result in the creation of too many ports even for a medium sized interface and a few objects. These numbers are preferably defined in the source code for the object proxy since this is different for each new component.

6.4.1 Standard Remote Procedure Call

In the implementation effort was put into making the changes to the automatically generated code as small as possible. Obviously some changes were needed in order to make the design a little bit more flexible. One should however keep in mind that all of the source code produced for the proxies and the servers should be automatically generated.

In the file fed to *rpcgen* three functions were declared. One used for creating a new object (*create_proc()*), one for destroying objects (*destroy_proc()*) and one for performing a method call in an object (*call_proc()*). When creating a new object an identifier is returned, this identifier is then used in subsequent calls on this object.

There is also a need to be able to identify the correct server. In this study the implementations of the servers all have the same program number and different version numbers. If this approach would be taken in a production system there would be a need to dynamically allocate new version numbers. For the prototype it is acceptable to make this a compile time decision.

In RPC[1] it is reasonably much work to dynamically create new ports hence the implementation uses unique numbers to identify objects on the server

[1] Remote Procedure Call

side. These unique numbers are in fact the addresses of the individual objects in the server memory space.

RPC is used as the index in the comparison of the different IPC methods. This is since COM[1] is using ORPC[2] that in turn is based upon RPC. An important question is then whether our implementation is close enough to the one based upon ORPC. The fact that COM is built into Windows makes it very hard to monitor its behaviour. And since the source code for Microsoft's implementation is not available makes it impossible to analyse it in detail. Some important observations were made through simple test programs. Firstly, the registration of servers is similar, but a bit more complicated, since it is in fact the factory for the object that is registered in Microsoft's implementation. This fact makes it inevitable to perform marshal of interfaces. This does however speak in our favour, our implementation seems less complicated and hence it might actually be faster. Secondly, a server created using Microsoft's IDL compiler tool is not multithreaded by default. Since our server is not multithreaded either we still consider it appropriate to use the RPC implementation as index for the comparison. Another observation that can be done is the fact that creating new threads with each request on an object is quite expensive.

6.4.2 Unix Pipes

As mentioned before, the pipes used here are named pipes (created with *mkfifo(3)*).

Because there does not exist a tool to generate code (like in the RPC case), everything was written from scratch. But since the code is meant to be automatically generated this was kept in mind when it was written. A resemblance to the structure of the RPC implementation is obvious and intended.

The server creates a named pipe somewhere in the filesystem, registers the complete name and path to it in the SM through the *ISMServer* interface. It then waits for a connection on the pipe. When the client is instantiated it creates two named pipes (one for each direction of communication), fetches the path to the server pipe and opens it. The client then writes the names of the two previous created named pipes to the server pipe and then closes it. For the rest of the client's life all communication with the server is done on the two the pipes created. This design makes it easy to make the server multithreaded.

When the communication channel is established the client writes its call type, object ID, possibly a function number and marshalled data to the server on the in pipe and reads the result from the out pipe.

[1] Component Object Model
[2] Object Remote Procedure Call

The call type can be:

- *create*
- *call*
- *destroy*

When the call type is *create* the object ID is ignored. In this case a new object is created and an ID is returned to the client proxy. This ID is then sent along with all calls and also when the object is destructed.

After the server has read all the data sent by the client proxy, the server does what the client asked for. That is, it creates an object, calls the right function or destroys the object. After that, a result is always returned, even if it is zero bytes long.

6.4.3 Doors/Linux

As for Pipes, there is no tool to generate code, so everything is written from scratch with automation in mind.

When the Doors server is started, it first creates a door and a file. The server then connects the door to this file and registers it. Acquiring an *ISMServer* interface and calling the *RegisterServer* function with the file name as identifier does the registration. This door descriptor (which is acquired by opening the file) is used for all connections to the server.

When a connection is made to the server door, the first thing that is done is determining the type of call. The call type can be:

- *create*
- *call*
- *destroy*

If no data is sent with the door call it is considered to be a *create* call. When a *create* type of call is received a new object is created. Then a unique identifier is returned to identify the object on subsequent calls. This unique identifier is actually the address to the object in the server address space. This is a security risk and must be solved in a real project. Here this risk is not considered.

When the door call has data attached to it, this identifier is extracted from the call data and converted to a pointer to the object. Then the call type is extracted and the correct method in the object is called, or if it was a *destroy* call, the object is destructed.

The client proxy locates the door by looking up the ID for the object in the database (through the *ISMClient* interface) and opening the file.

Doors/Linux is still alpha software and does not behave as the Solaris equivalent. To make the doors client server solution work, we had to do some none optimised code tricks. The documentation states that if the result buffer is too small to hold the result, a new buffer will be allocated by the system. This does not work unless the result is larger than 64kB, for some unknown reason (remember that Doors/Linux is alpha quality software). Therefore, a result buffer of 64kB is always allocated before a door call where a result is expected.

6.4.4 Shared Memory

There are a few factors to consider regarding the implementation of a shared memory IPC component and server. The first and most important one is what type of shared memory should be used. System V shared memory was used mainly because of two reasons. First, it is easier to use than using *mmap(2)*, second, POSIX[1] shared memory is not available in Linux yet. Since System V shared memory was used it was natural to also use System V semaphores for synchronisation.

Performing IPC by use of shared memory offers no help tools in form of IDL compiler tools or the like. This made it necessary to implement code for contacting the server and handing over the results by hand.

Communication with the server is done via a shared memory object that persists through the lifetime of the server. Passing of arguments is then done in a shared memory object that the client creates when it is started. Access of the persistent memory object and contact with the server is accomplished through the use of three semaphores.

The synchronisation scheme is made a bit complicated by the fact that the size of System V shared memory objects can not be changed. This forces a scheme where chunks of data is transferred one at a time in case the data has a size exceeding that of the shared memory object. The scheme for making a method call is described in the following clauses.

When started, the server creates a shared memory object that persists during the lifetime of the server. It also creates the three semaphores needed for synchronisation, the semaphores are called G (Global), C (Client) and S (Server). The server then locks semaphore C once and S twice in order for it to suspend itself.

[1] Portable Operating System Interface, with an X thrown in to make it sound cooler [18]

When a client starts it creates a shared memory object of some fixed size. This is the memory that will be used for transferring arguments to the server in future method calls.

Whenever the client wants to call a method on a remote object it starts with locking semaphore G. When this lock is obtained it can be sure that it is alone in communicating with this server. It also means that no one else can communicate with the server until the semaphore is unlocked.

The memory object created by the server is then filled out with information regarding the method call. The object's identity and the position of the clients shared memory object. The size of the marshalled data is also supplied.

Because the size of the arguments can change while the size of the shared memory object owned by the client is fixed there might be a need to split the arguments into chunks that are passed one at a time. The synchronisation is achieved by using semaphores C and S. Since the server is locked and waiting for semaphore S the client can wake the server by unlocking it. If the client then locks semaphore C it will be suspended until the server is done with its processing of this chunk and it unlocks semaphore C. The server then suspends itself by locking semaphore S. This is repeated until all of the arguments are transferred from the client to the server.

When the server has received all the arguments it can unmarshal them and perform the method call on the object that was requested by the client. The results from the call is then marshalled and put into the clients shared memory object. In case the results does not fit it will be split into smaller pieces just like was the case with the arguments. Passing of control is done in the same way as described above.

When the last piece of the results are passed over to the client the server puts itself into the same position as before the client woke it up, i.e. it will suspend itself by trying to lock semaphore S after unlocking semaphore C.

The client takes care of the results and then it unlocks semaphore G in order to let other clients to contact the server.

Identifying the correct server boils down to the matter of locating the correct persistent shared memory object for the server together with its semaphores. Since each System V IPC object (both shared memory and semaphores) is connected to a filename the natural thing is to put the filename in the SM. There is nothing that prevents different types of objects to be connected to the same filename and hence only one filename is needed for both the shared memory and the semaphore.

6.5 Evaluation Suite

As an evaluation of our system, an evaluation suite is assembled. This suite is composed to test different aspects of the system. As stated earlier (see section 5.4), time is our primary evaluation criteria.

6.5.1 Evaluation Program One

The first program evaluates the connection overhead and the component creation/destruction overhead imposed by the different IPC methods. The program creates and destroys one component without performing any other operations. The time is measured and entered in our evaluation protocol in Appendix D

6.5.2 Evaluation Program Two

This program measures the overhead of a simple method call with very little data as argument. The program creates one component and then calls it with an integer as argument. The component then returns the square of this integer.

6.5.3 Evaluation Program Three

This program measures the overhead of a large method call with a very large data structure as argument. The program is run three different times with 60kB, 1MB and 16MB as arguments respectively. The argument is an array of integers. This evaluates the overhead imposed by the marshalling done by the system and also the overhead of transferring large data structures using the selected IPC method.

“They say time is the fire in which we burn.”
- Dr. Soran, Generations

7. Results

This chapter presents the results obtained. The presentation has been divided into two parts, first a discussion about using different types of marshalling methods. Then comes a comparison of the different methods for transferring the data.

7.1 Marshalling

Because of the nature of our system, it is only providing communication between clients and servers located on the same computer, marshalling is not done in the sense of changing bit order of data. On the other hand all data that is to be transferred from the client to the server as arguments and the data transferred back from the server to the client has to be packaged in a transferable format. Some data types also need extra information to be transferred.

Examples of data types that need extra information to be transferred are arrays and strings (which is actually an array of characters). These need their length to be transferred in order for the receiver to be able to decode them.

The design of the system makes it easy to add marshalling where representation is changed into some network standard.

7.2 Interprocess Communication

The evaluation suite consists of three programs (see section 6.5). To evaluate the different IPC[1] methods, these programs are executed ten times each for every IPC method. The mean result and the standard deviation from the ten runs is calculated and entered in Appendix D.

Below is a short description of the outcome of the three evaluation programs. For a precise display of the results see Appendix D.

7.2.1 Evaluation Program One

Evaluation program one measures the time to connect to the component server, create one component and destroy it. The absolute time it takes to create and destroy one component is about 0.8 to 1.6 ms.

[1] Interprocess Communication

Doors is the fastest and the close runner up is shared memory which only takes 1.04 times as long time. Slowest is RPC[1], which takes 2.02 times as long time as with Doors. Pipes take 1.24 times as long as Doors. See Figure 27.

The difference in time corresponds to the different operations needed to get a connection to the server. The actual time to create the component in the server is constant in the different IPC methods, since it is the same code that executes in the server.

The time difference is not that significant between the different method except for RPC. If a program using this system often creates and destroys objects, this might have an impact on the program's performance if the RPC method is chosen.

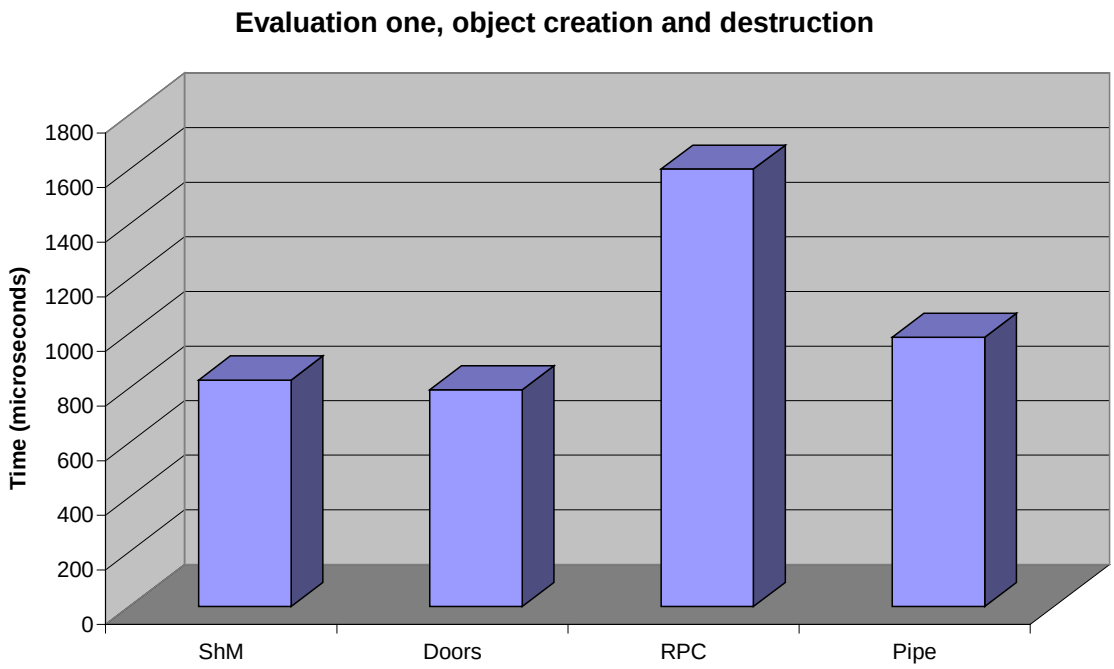

Figure 27: Evaluation Program one results

7.2.2 Evaluation Program Two

Evaluation program two measures the time to make a method call on a component with very small arguments. In this case the argument was only one integer. The absolute time it takes to do a method call is about 160-300 microseconds.

Doors is again the fastest and the first runner up is shared memory, which takes 1.52 times as long as Doors. Pipes and RPC is a bit further behind,

[1] Remote Procedure Call

1.82 and 1.85 times as slow, but the difference between pipes and RPC is not significant. See Figure 28. The “Rest” time is the same in all methods and is about 50 microseconds. “Rest” time is the set-up time for the IPC method.

Here, as in evaluation program one, the performance impact on a program using this system can be huge. Since method calls are the fundamentals of a program and normally is performed very often, this time differences may have a big impact, especially if the method computes in short time and is called often. Normal method calls on an in-process server is much faster than this, about 150 times faster, a normal method call takes about one microsecond. The overhead of the method call can in this case, from a programmer's point of view, be substantial; i.e. a fast method can take over 100 times as long time as the programmer expected. See sections 3.1.2 and 3.1.3 for an explanation of these differences.

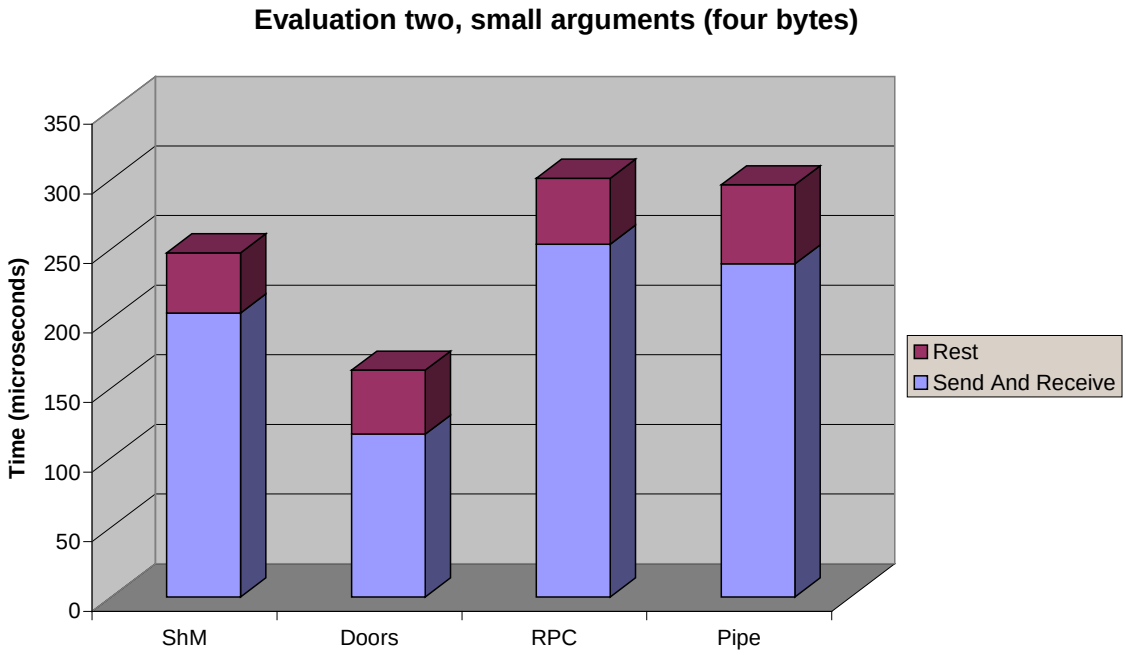

Figure 28: Evaluation Program two results

7.2.3 Evaluation Program Three

Evaluation program three measures the time to make a method call on a component with large arguments. Large in this context is anything from a couple of kilobytes to several megabytes.

Evaluation program three is run with three different sizes of arguments: 60kB (3A), 1MB (3B) and 16MB (3C).

Evaluation program three sends a large array of integers to the server. The server just returns one integer as the result. This implies that it is not only the time to transfer the array that is measures but also the time to marshal (on the client side) and unmarshal it (on the server side). The marshal/unmarshal time is not as small as expected but in fact rather large and is therefore marked in the diagrams. Although it might seem strange to measure this time and not just the transfer time, this is more like a real situation and hopefully says more about how the IPC method will behave. The results from these evaluation runs can be found in Figure 29 (3A), Figure 30 (3B) and Figure 31 (3C). Detailed numbers are found in Appendix D.

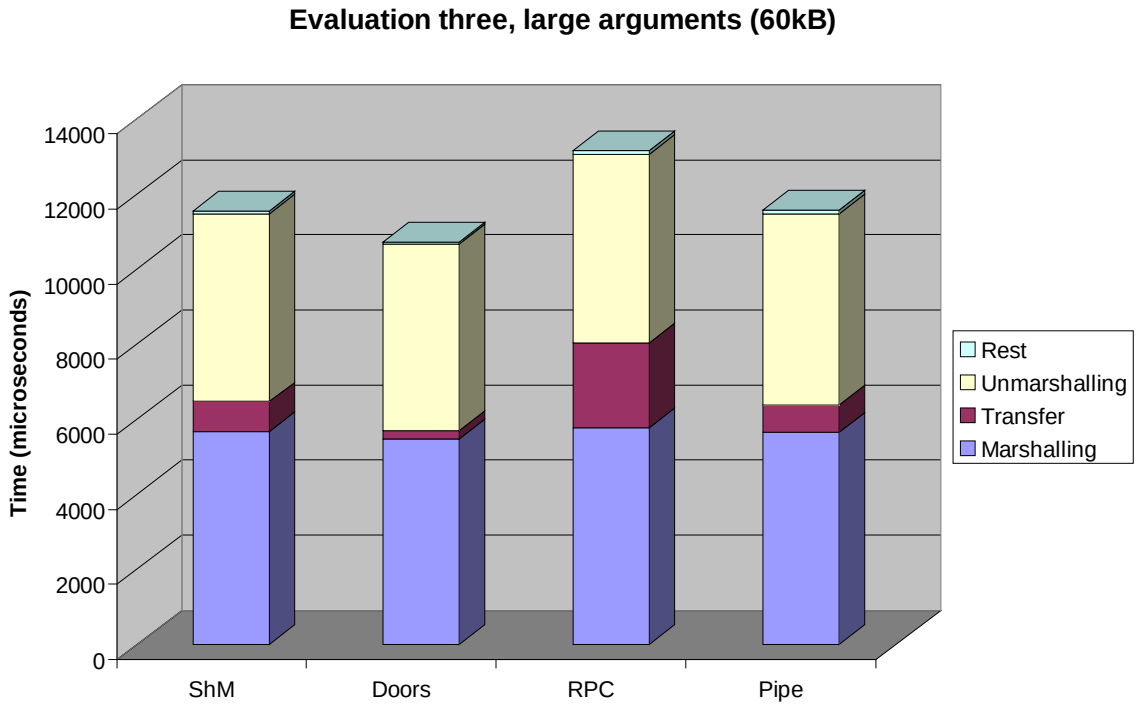

Figure 29: Evaluation Program 3A results, large arguments (60kB)

Doors is fastest in all evaluation runs. In evaluation 3A, with moderately large (60kB) arguments, the difference between RPC is 1.23 times as slow as Doors. This difference stays about the same when the argument size is increased, which can be seen in 3B and 3C. The difference between Doors, Pipes and shared memory is almost the same in all evaluations. Shared memory and Pipes are 1.06 to 1.11 times slower then Doors.

It should be noted that evaluation three not is how a component system normally is used. 16MB as arguments is not a normal situation and if such large data structures have to be transferred there are better ways than through the component system. The earlier evaluations, evaluations one and two, are more close to real usage.

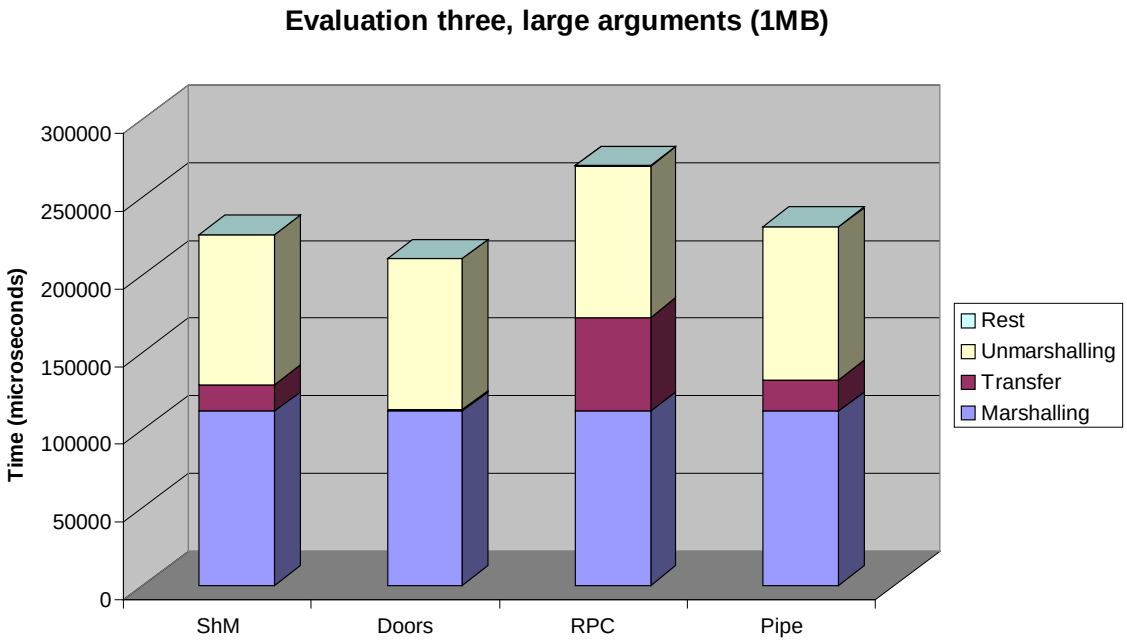

Figure 30: Evaluation Program 3B results, large arguments (1MB)

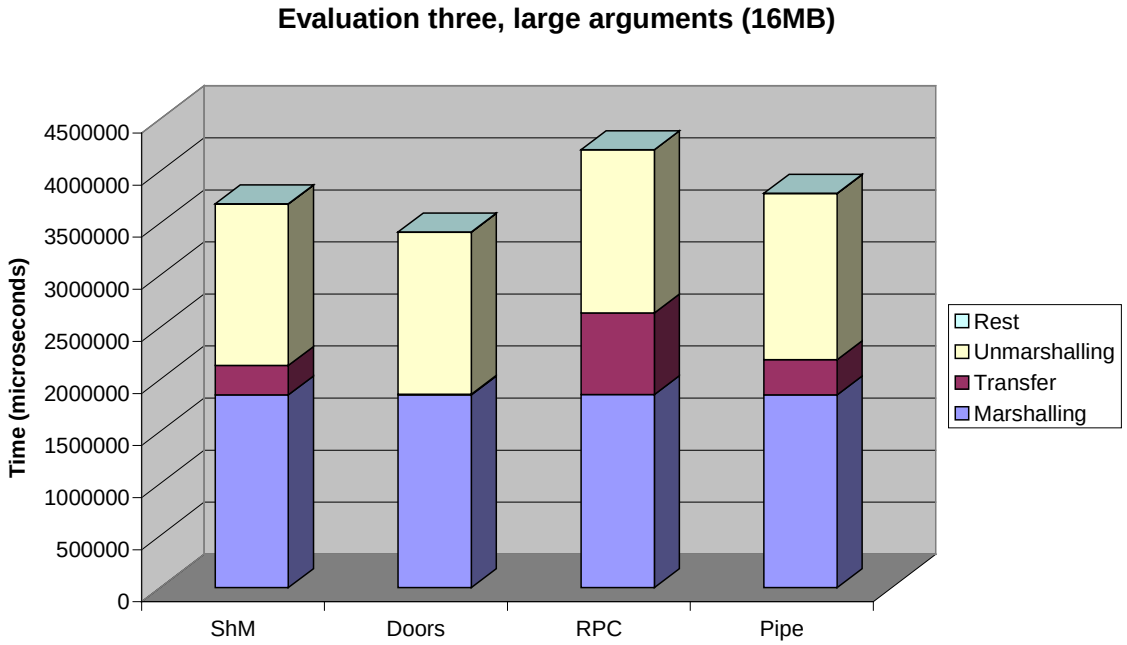

Figure 31: Evaluation Program 3C results, large arguments (16MB)

If just the actual transfer times are compared, Doors is absolutely fastest, e.g. in 3A the transfer time in Doors is just 10.5% of the transfer time in RPC. The difference is less than 1% in 3C, but because of the large times

to marshal/unmarshal, this huge difference is hidden. It is however obvious in the diagrams.

To give an even better feel for how the system behaves Figure 32 and Figure 33 are included here. Figure 32 shows the total time and the actual transfer time for evaluation program three as a function of the argument size. Figure 33 shows just the actual transfer times as a function of the argument size.

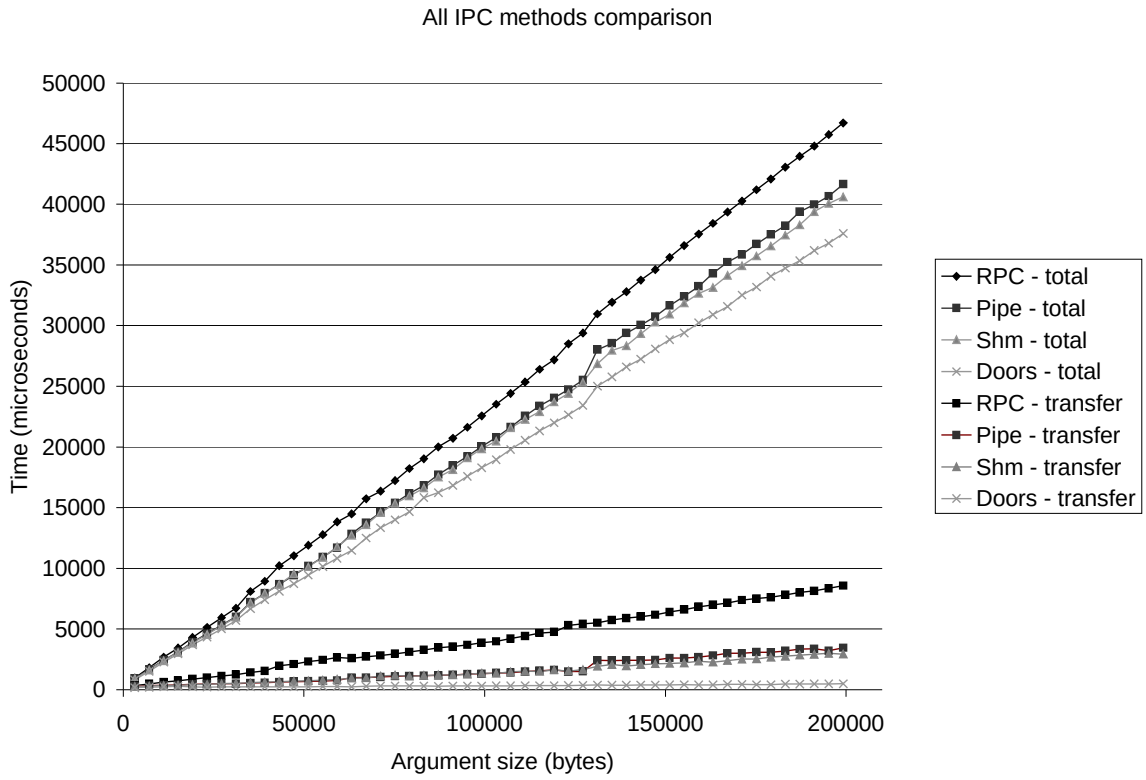

Figure 32: All methods total and transfer time

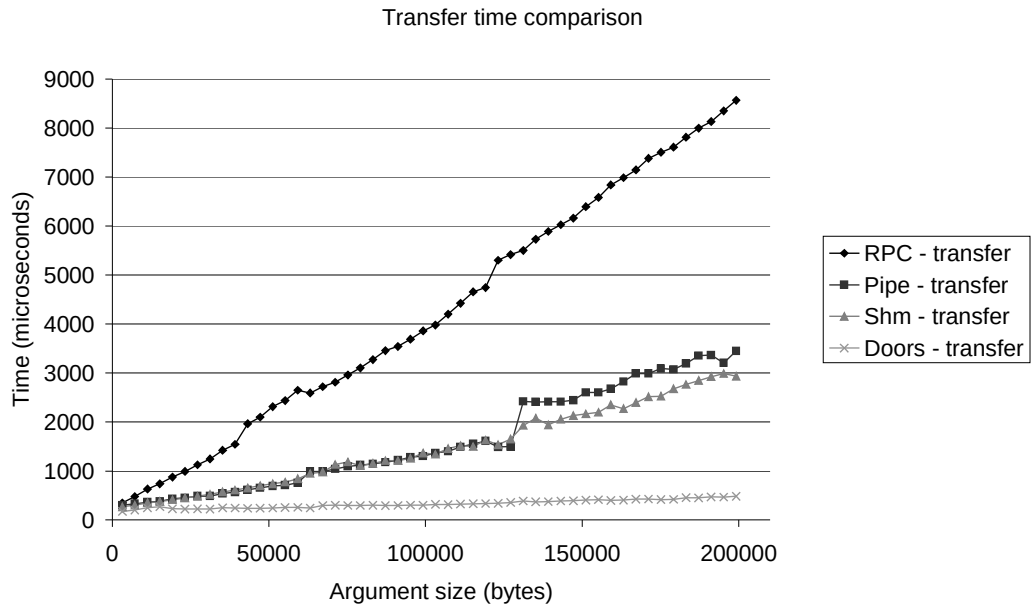

Figure 33: All methods transfer time

Finally, Figure 34 together with Figure 32 show that the standard deviation for our evaluations are low, so that our mean values really show the behaviour of our system. In Figure 34 the standard deviation is calculated from ten measurements made on the total time for each argument size.

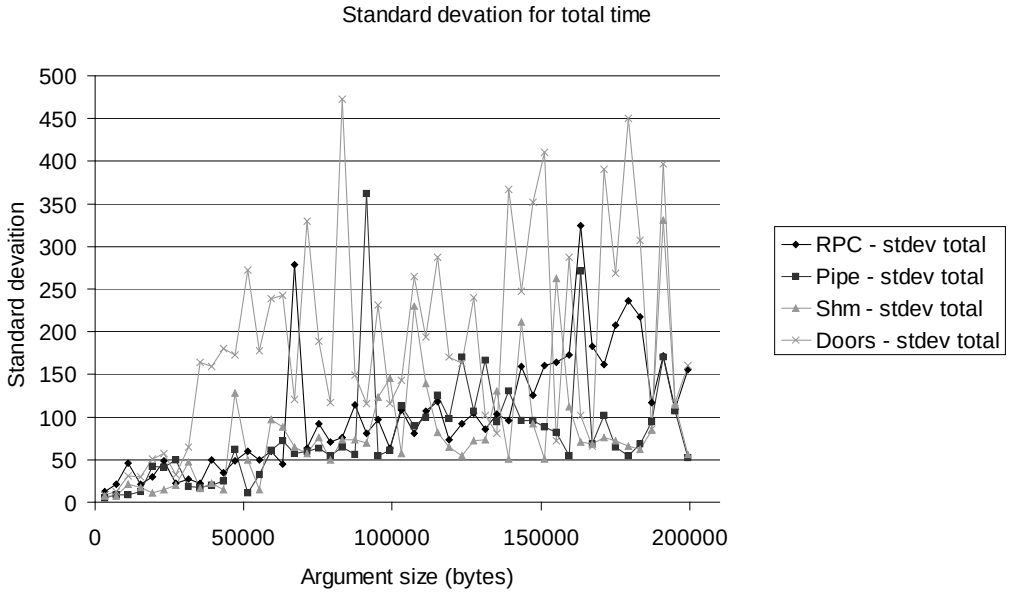

Figure 34: Standard deviation for total time in evaluation three

O'Brien: "We won!"
Sisko: "You sound surprised."
O'Brien: "Surprised? I'm astonished! Not that I mind."
– from DS9

8. Conclusions

From the results presented in the previous chapter a number of conclusions can be drawn. A discussion of these conclusions is given here.

8.1 Marshalling

Since all communication in our system is done between components on the same computer, no marshalling is needed. The only thing that is done is packing complex data types to make it possible to transfer them.

Since no marshalling is done, this surely must be faster than anything else. Although no marshalling is done, evaluation program three surely shows that packing our complex data structure before sending it takes a lot of time.

If the time to marshal/unmarshal data can be reduced, the relative differences in our evaluations will increase to Doors' advantage. This fact is clearly visible in the evaluation protocols where marshalling/unmarshalling is not the operation that takes the longest time, i.e. in evaluation one and two. In numbers, if the marshalling/unmarshalling time can be reduced to half the time, the relative difference will be increased to about 1.4 times instead of 1.23 in evaluation 3A (Doors compared to RPC).

8.2 Interprocess Communication

In all evaluation programs Doors is the fastest. The runner up in all evaluations is shared memory and it is close behind in them all but evaluation two. Also, Pipes is not far behind Shared memory in all evaluations. RPC[1] is definitely slowest in all cases and should probably be avoided if performance is an issue.

The choice therefore is between Doors/Linux, Pipes and Shared memory. All of these have their benefits and downsides. As mentioned before, Doors/Linux is alpha software and is not reliable although very fast when it works. Shared memory is a limited resource, there can only exist 128 shared memory segments system-wide at a time, which limits the number of object servers that can exist simultaneously. This limitation is no hindrance in a smart implementation. Even for pipes and RPC such limitations exist.

[1] Remote Procedure Call

For pipes the maximum number of file descriptors per process (1024 in Linux) limits the maximum number of objects that can be created. This limitation is just an implementation detail and can easily be fixed. For RPC the limitation is the number of possible TCP[1] ports in the RPC system. This limit is harder to do something about and is a big disadvantage for RPC. Doors has also a limit because for each door that a client needs to keep track of it has to have a door descriptor, and it is like any descriptor a limited resource. In Linux the maximum number of descriptors is 1024 per process.

8.3 Summary

With all of this taken into consideration the best choice of IPC[2] method must be shared memory. RPC and pipes is too slow and Doors/Linux is still alpha quality software.

The limit of 128 object servers per system is quite small, but is probably nothing to worry about. One server often serves more than one type of object, and if more than 128 object servers are expected in the system, another implementation than the one in our prototype is needed. This is not expected and therefore not considered in this thesis.

In a system where interprocess communication is planned from the beginning, when the system is designed, it is possible to optimise the performance by constraining the number of IPC transactions and size of such transactions. However, in this system the process boundary is invisible to the application programmer and it is therefore important that the system does not add too much overhead.

Obviously it is best if the programmer knows of and plans for the extra time added by the COM[3] system.

As noted in sections 3.1.3 and 7.2.2, the time difference between a normal method call and a method call on an out-of-process server is substantial. A method call on an out-of-process server can take as much as 100-150 times longer than on an in-process server.

[1] Transmission Control Protocol
[2] Interprocess Communication
[3] Component Object Model

"I know when to quit."
- Chakotay, Voyager

9. Further Studies

Here we present some subjects that has to be further investigated in order to make our system ready to be a part of a product such as Nokia's set top boxes.

9.1 Database

The SQL database in our prototype is really just thrown in to get the system up and running without too much work. The database needs to be replaced with a more adequate solution. In the Mozilla browser project, there exists a database that might be suitable. Changing the system database is easy; it is just a matter of providing a different implementation of the IDBConnection interface.

9.2 Server Management System

In our prototype system, servers register themselves when they are started. In a real system it is better if servers are registered when they are installed and just started as necessary. The Server Management System (SMS) or some other part of the system can handle this.

9.3 Marshalling

As our evaluation revealed, marshalling and unmarshalling is what takes most time when sending large data. This is something that probably can be greatly improved and in this way one can gain a lot in overall performance.

To make the system truly useful marshalling of interfaces must be handled appropriately.

9.4 IDL compiler tool

From a programmer's point of view, this system is close to useless without an IDL[1] compiler tool to aid her in the development. This is probably one of the first things that have to be added to the system. The IDL compiler tool has to generate both a proxy DLL and server code from the IDL specifications file.

[1] Interface Definition Language

9.5 Servers

Our prototype shows that our system design works and that our implementations of the servers work. In the future, code for the servers shall be automatically generated (see section 9.4 above), but to get a fast implementation, perhaps threaded, some work has to be done.

"It's a good crew, they deserve to know."
- Unknown, TOS

10. References

[1] XDR, External data representation, RFC 1832, http://www.ietf.org/rfc/rfc1832.txt

[2] Doors/Linux homepage http://www.rampant.org/doors/

[3] ASN.1 Complete, Prof. John Larmouth, http://www.nokalva.com/asn1/larmouth.html

[4] RPC, Remote Procedure Call, RFC 1831, http://www.ietf.org/rfc/rfc1831.txt

[5] Essential COM, Don Box, Addison-Wesley, 1998,ISBN 0201634465

[6] RPC man pages, http://www.openbsd.org/cgi-bin/man.cgi

[7] Design Patterns, Erich Gamma et al., Addison Wesley, 1998, ISBN 0201633612

[8] The C++ Programming Language 2nd Edition, Bjarne Stroustrup, Addison Wesly, 1995, ISBN 020153992

[9] C++ Primer 2nd Edition, Stanley B. Lippman, Addison Wesley, 1995, ISBN 0201548488

[10] Inside COM, Dale Rogerson, 1997, Microsoft Press, ISBN 1572313498

[11] A System of Patterns, Frank Buschmann et al. Wiley 1998, ISBN 0471958697

[12] UNIX Network programming Volume 2 – Interprocess communications, W. Richard Stevens, Prentice-Hall PTR, 1999, ISBN 0130810819

[13] Database system concepts, Abraham Silberschatz et al., McGraw-Hill, 1998 3rd Edition, ISBN 0070310866.

[14] mySQL, a free SQL database, http://www.mysql.com/

[15] mySQL++, a free C++ API to mySQL, http://www.mysql.com/download_mysql++.html

[16] Linux Application Development, Michael K. Johnsson et al, Addison-Wesley, 1998, ISBN 0201308215

[17] UML Distilled: a brief guide to the standard object modeling language 2nd edition, Martin Fowler, Addison-Wesley, 1999, ISBN 020165783X

[18] http://ftp.sunet.se/foldoc/foldoc.cgi?query=POSIX

[19] Software Requirements, Karl E. Wiegers, Microsoft Press, 1999, ISBN 0735606315

(All links to web pages and ftp sites were confirmed to work and contain the contents intended on 2000-02-03.)

Appendix A Abbreviations

Abbreviation	Comment
4.4BSD	Berkeley System Distribution version 4.4 A branch on the UNIX tree developed at the university of Berkeley.
API	Application Programming Interface.
ASN.1	Abstract Syntax Notation One A standard for representing data.
BER	Basic Encoding Rules One of the encodings of data connected to ASN.1.
CER	Canonical Encoding Rules One of the encodings of data connected to ASN.1.
COM	Component Object Model A component model used heavily within Microsoft's products.
CORBA	Component Object Request Broker Architecture. A system for distributed object, standardised by OMG.
CPU	Central Processing Unit The brain of a computer.
DCOM	Distributed Component Object Model The distributed version of COM, it allows components to be implemented in external servers.
DER	Distinguished Encoding Rules One of the encodings of data connected to ASN.1
DLL	Dynamic Link Library.
DOS	Disk Operating System.
FIFO	First In, First Out.

IDL	Interface Definition Language.
IPC	Inter Process Communication.
IP	Internet Protocol
NFS	Network File System.
OMG	Object Management Group.
ORB	Object Request Broker.
ORPC	Object Remote Procedure Call The protocol used within DCOM in order to transfer arguments and results between the client and the server.
OS	Operating System.
PC	Personal Computer.
PER	Packed Encoding Rules One of the encoding rules of data connected to ASN.1.
POSIX	Portable Operating System Interface, with an X thrown in to make it sound cooler [18] A standard for UNIX systems.
RPC	Remote Procedure Call A protocol that lets a client call procedures implemented in a remote server transparently.
SCM	Service Control Manager.
ShM	Shared Memory A non-standard abbreviation.
SM	Server Manager The manager of servers and their location in the work presented in this thesis.
SMS	Server Management System The full system for managing servers that is presented in this thesis.

SQL	Structured Query Language A query language for databases.
SVR4	Unix System V Revision 4 AT&T's version of the UNIX operating system.
TCP	Transmission Control Protocol.
TLV	Type, Length, Value.
UUID	Universally Unique Identifier.
XDR	External Data Representation The data representation used in RPC to allow servers and clients to run on different architectures.
XPCOM	Cross Platform Component Object Model The version of COM that is used within the Mozilla web browser.

Appendix B Interface Specification

This is a somewhat more organised description of the interfaces designed for the system described in this report. Ideally this appendix is all that is needed to write programs using the services offered by the system.

IArgMarshal

Used within the object proxy to marshal arguments before sending or returning them via some IPC method.

Interface ID

1c195754-855a-11d3-974b-008c7599b88

Extends

nsISupports

Methods

nsresult SetOutPutBuffer(buffer)
const void *buffer

Sets the output buffer for marshalling. The component implementing IArgMarshal handles reference counting for the IBuffer.

nsresult PutChar(cData)
const signed char cData

Marshals the character (8 bits) provided in cData.

nsresult PutUChar(ucData)
const unsigned char ucData

Marshals the unsigned character (8 bits) provided in ucData.

nsresult PutShort(sData)
const short sData

Marshals the short integer (16 bits) provided in sData.

nsresult PutUShort(usData)
const unsigned short usData

Marshals the unsigned short integer (16 bits) provided in usData.

nsresult PutInt(iData)
const int iData

Marshals the integer (32 bits) provided in iData.

nsresult PutUInt(uiData)
const unsigned int uiData

Marshals the unsigned integer (32 bits) provided in uiData.

nsresult PutLong(lData)
const long long lData

Marshals the long integer (64 bits) provided in lData.

nsresult PutULong(ulData)
const unsigned long long ulData

Marshals the unsigned long integer (64 bits) provided in ulData.

nsresult PutFloat(fData)
const float fData

Marshals the float (32 bits) provided by fData.

nsresult PutDouble(dData)
const double dData

Marshals the double (64 bits) provided by dData.

nsresult PutArray(pcData, uiElementSize, uiLength, fMarshFunc)
const char *pcData
const unsigned int uiElementSize
const unsigned int uiLength
const void *pfMarshFunc

Marshals an array of elements. pcData points to the data. uiElementSize tells how large each array member is and uiLength tells the number of elements in the array. For each element the function pointed to by pfMarchFunc is called to perform the actual marshalling.

nsresult PutPointer(pcData, pfMarshFunc)
const char *pcData
const void *pfMarshFunc

Marshals a pointer by calling the function pointed to by pfMarshFunc to perform the marshalling.

IArgUnMarshal

Used within the object proxy to unmarshal arguments before sending or returning them via some IPC method.

Interface ID

1c195753-855a-11d3-974b-008c7599b88

Extends

nsISupports

Methods

nsresult SetInputBuffer(buffer)
void *buffer

Sets the input buffer for unmarshalling. The called handles reference counting.

nsresult GetChar(cData)
signed char *cData

Unmarshals a character (8 bits) into the memory pointed to by cData.

nsresult GetUChar(ucData)
unsigned char *ucData

Unmarshals an unsigned character (8 bits) into the memory pointed to by ucData.

nsresult GetShort(sData)
short *sData

Unmarshals a short integer (8 bits) into the memory pointed to by sData.

nsresult GetUShort(usData)
unsigned short *usData

Unmarshals an unsigned short integer (8 bits) into the memory pointed to by sData.

nsresult GetInt(iData)
int *iData

Unmarshals an integer (32 bits) into the memory pointed to by iData.

nsresult GetUInt(uiData)
unsigned int *uiData

Unmarshals an unsigned integer (32 bits) into the memory pointed to by uiData.

nsresult GetLong(lData)
long long *lData

Unmarshals a long integer (64 bits) into the memory pointed to by iData.

nsresult GetULong(ulData)
unsigned long long *ulData

Unmarshals an unsigned long integer (64 bits) into the memory pointed to by ulData.

nsresult GetFloat(fData)
float *fData

Unmarshals a float (32 bits) into the memory pointed to by fData.

nsresult GetDouble(dData)
double *dData

Unmarshals a double (64 bits) into the memory pointed to by dData.

nsresult GetArray(pcData, uiElementSize, uiLength, pfUnmarshFunc)
char **pcData
const unsigned int uiElementSize
unsigned int *uiLength
const void *pfUnmarshFunc

Unmarshals an array of elements. uiElementSize provides the size of each element, pfUnmarshFunc unmarshals each element. The number of elements in the array is returned in uiLength, and the array itself is returned in pcData.

nsresult GetPointer(pcData, uiElementSize, pfUnmarshFunc)
char **pcData
const unsigned int uiElementSize
const void *pfUnmarshFunc

Unmarshals a pointer by calling pfUnmarshFunc. The element pointed to is returned in pcData. uiElementSize provides the size of the element pointed to.

IBuffer

An interface used to access a buffer that is both readable and writable.

Interface ID

1c195759-855a-11d3-974b-008c7599b88

Extends

nsISupports

Methods

nsresult SetBufferAndSize(pcBuffer, iDataSize, iBufferSize)
char *pcBuffer
int iDataSize
int iBufferSize

Sets the buffer and its size. It also sets the size of the data already written in the buffer.

nsresult GetBufferAndSize(pcBuffer, iDataSize)
char **pcBuffer
int *iDataSize

Returns a pointer to the buffer and the size of its data.

nsresult Write(pcData, iSize)
char *pcData
int iSize

Writes iSize bytes from the memory pointed to by pcData into the buffer.

nsresult Read(pcData, iSize)
char **pcData
int iSize

Returns iSize bytes in a buffer pointed to by pcData. The needed memory is allocated and needs to be freed by the caller.

IConnectionObject

An interface closely related to the ISendAndReceive interface presented below. It allows the creation of objects.

Interface ID

3d592c6e-395c-4f4b-989c-1cda96eeae28

Extends

ISendAndReceive

Methods

nsresult CreateObject(sPath)
const char *sPath

Used to ask the server to create an object. The sPath argument is used to contact the correct server.

IDBConnection

An interface used to access a database that accepts SQL queries. Used by ISMServer and ISMClient.

Interface ID

1c195751-855a-11d3-974b-0008c7599b88

Extends

nsISupports

Methods

nsresult Store(iType, strPath, strCID)
int iType
char *strPath
char *strCID

Stores the arguments in the database. srtCID together with iType is the primary key in the database.

nsresult Retrieve1Key(iType, strPath, strCID)
int *iType,
char **strPath,
char *strCID

Retrieves the type and path for the component corresponding to the CID from the database.

nsresult Retrieve2Key (iType, strType, strCID)
int iType

char **strType
char *strCID

Retrieves the path for the component corresponding to the CID and type from the database.

nsresult Remove1Key(strCID)
char *strCID

Removes all components with this CID from the database.

nsresult Remove2Key(strCID, iType)
char *strCID
int iType

Removes the component with this CID and type from the database.

ISendAndReceive

Used by the object proxy to do the actual method-call.

Interface ID

1c195852-855a-11d3-974b-0008c7599b88

Extends

nsISupports

Methods

nsresult FetchBuffer(intBuffer)
void **intBuffer

Gets the buffer from this component that will be sent. Use this buffer together with IArgMarshal to marshal the arguments. No reference counting will be needed on the caller's part.

SendAndReceive(intBuffer)
void **intBuffer

Send the contents of the buffer and then retrieve the return values. The return values are delivered in the intBuffer argument. No reference counting will be needed on the caller's part.

ISMClient

The interface used by a client to locate a server for a component.

Interface ID

1c195755-855a-11d3-974b-0008c7599b88

Extends

nsISupports

Methods

GetSender(ciidCIID, iType, pISAR)
nsCID &ciidCIID
int iType
void **pISAR

Returns the correct type of implementation of an ISendAndReceive interface depending on what the type is. E.g., if the type is RPC, a CRPCSendAndReceive component will be instantiated and returned. The ciidCIID is looked up through the IDBConnection interface. If the type is set to ANY_TYPE, it is up to the implementation to choose one.

ISMServer

The interface used by a server for a component for registering and unregistering itself.

Interface ID

1c195750-855a-11d3-974b-0008c7599b88

Extends

nsISupports

Methods

RegisterServer(ciidCIID, strPath, iType)
nsCID &ciidCIID
char *strPath
int iType

Registers a component's CID, server type and location (Path).

UnregisterServer(ciidCIID, iType)
nsCID &ciidCIID
int iType

Unregisters a component. The type must be specified. After this is done, a component proxy can no longer locate a server. Must be called when a component server ends its execution.

Appendix C Interface Implementation

Design of the classes implementing the interfaces.

CArgMarshal

The class implementing the IArgMarshal interface. The marshalling is very simple, since it only copies things byte by byte.

Component ID

1c195757-855a-11d3-974b-008c7599b88

Program ID

sms.cargmarshal.1

Interfaces

IArgMarshal

Attributes

IBuffer *m_pBuffer

The interface for the buffer where everything will be marshalled into.

Methods

CArgMarshal()

Simple constructor.

~CArgMarshal()

Simple destructor. It releases m_Buffer if it is set.

nsresult QueryInterface(iid, ppv)
const nsIID &iid
void **ppv

Part of nsISupports.

nsresult AddRef()

Part of nsISupports.

nsresult Release()

Part of nsISupports.

nsresult SetOutputBuffer(buffer)
const void *buffer

Part of IArgMarshal. Sets m_Buffer to the value provided in buffer if not NULL. The old m_Buffer is released properly first. AddRef() is called upon the new buffer.

nsresult PutChar(cData)
const signed char cData

Part of IArgMarshal.

nsresult PutUChar(ucData)
const unsigned char ucData

Part of IArgMarshal.

nsresult PutShort(sData)
const short sData

Part of IArgMarshal.

nsresult PutUShort(usData)
const unsigned short usData

Part of IArgMarshal.

nsresult PutInt(iData)
const int iData

Part of IArgMarshal.

nsresult PutUInt(uiData)
const unsigned int uiData

Part of IArgMarshal.

nsresult PutLong(lData)
const long long lData

Part of IArgMarshal.

nsresult PutULong(ulData)
const unsigned long long ulData

Part of IArgMarshal.

nsresult PutFloat(fData)
const float fData

Part of IArgMarshal.

nsresult PutDouble(dData)
const double dData

Part of IArgMarshal.

nsresult PutArray(pcData, uiElementSize, uiLength, pfMarshFunc)
const char *pcData
const unsigned int uiElementSize
const unsigned int uiLength
const void *pfMarshFunc

Part of IArgMarshal.

nsresult PutPointer(pcData, pfMarshFunc)
const char *pcData
const void *pfMarshFunc

Part of IArgMarshal.

CArgUnMarshal

The class implementing the IArgUnmarshal interface.

Component ID

1C19575A-855A-11d3-974B-0008C7599B88

Program ID

sms.cargunmarshal.1

Interfaces

IArgUnmarshal

Attributes

IBuffer *m_Buffer

The interface for the buffer where the marshalled data is.

Methods

CArgUnmarshal()

Simple constructor.

~CArgUnmarshal()

Simple destructor.

nsresult QueryInterface(iid, ppv)
const nsIID &iid
void **ppv

Part of nsISupports.

nsresult AddRef()

Part of nsISupports.

nsresult Release()

Part of nsISupports.

nsresult SetInputBuffer(void *buffer)

Part of IArgUnmarshal. Sets m_Buffer to the value provided in buffer if not NULL. The old m_Buffer is released properly first. AddRef() is called upon the new buffer.

nsresult GetChar(cData)
signed char *cData

Part of IArgUnmarshal. cData needs to point to an already allocated memory area.

nsresult GetUChar(ucData)
unsigned char *ucData

Part of IArgUnmarshal. ucData needs to point to an already allocated memory area.

nsresult GetShort(sData)
short *sData

Part of IArgUnmarshal. sData needs to point to an already allocated memory area.

nsresult GetUShort(usData)
unsigned short *usData

Part of IArgUnmarshal. usData needs to point to an already allocated memory area.

nsresult GetInt(iData)
int *iData

Part of IArgUnmarshal. iData needs to point to an already allocated memory area.

nsresult GetUInt(uiData)
unsigned int *uiData

Part of IArgUnmarshal. uiData needs to point to an already allocated memory area.

nsresult GetLong(lData)
long long *lData

Part of IArgUnmarshal. lData needs to point to an already allocated memory area.

nsresult GetULong(ulData)
unsigned long long *ulData

Part of IArgUnmarshal. ulData needs to point to an already allocated memory area.

nsresult GetFloat(fData)
float *fData

Part of IArgUnmarshal. fData needs to point to an already allocated memory area.

nsresult GetDouble(dData)
double *dData

Part of IArgUnmarshal. dData needs to point to an already allocated memory area.

nsresult GetArray(pcData, uiElementSize, uiLength, pfUnmarshFunc)
char **pcData
const unsigned int uiElementSize
unsigned int *uiLength
const void *pfUnmarshFunc

Part of IArgUnmarshal. The memory needed to hold the array will be allocated, and needs to be released by the caller.

nsresult GetPointer(pcData, uiElementSize, pfUnmarshFunc)
char **pcData
const unsigned int uiElementSize
const void *pfUnmarshFunc

Part of IArgUnmarshal. The memory needed to hold the data pointed to will be allocated, and needs to be freed by the caller.

CDBConnection

Class implementing the IDBConnection interface.

Component ID

1c195758-855a-11d3-974b-0008c7599b88

Program ID

sms.dbconnection.1

Interfaces

IDBConnection

Attributes

Connection *m_pCon

The connection to the SQL database.

Query *m_pQuery

The query object used to send queries via the connection object to the SQL database.

Methods

nsresult Store(iType, strPath, strCID)
int iType
char *strPath
char *strCID

Part of IDBConnection.

nsresult Retrieve1Key(iType, strPath, strCID)
int *iType,
char **strPath,
char *strCID

Part of IDBConnection.

nsresult Retrieve2Key (iType, strType, strCID)
int iType
char **strType
char *strCID

Part of IDBConnection.

nsresult Remove1Key(strCID)
char *strCID

Part of IDBConnection.

nsresult Remove2Key(strCID, iType)
char *strCID
int iType

Part of IDBConnection.

CDoorsSendAndReceive

Class implementing the IConnectionObject by use of Doors.

Component ID

d36c8ca2-92d4-46a4-bf44-dd2b2cf50155

Program ID

sms.doorssendandreceive.1

Interfaces

IConnectionObject

Attributes

Int m_iDoor

The Door descriptor for this object. This descriptor uniquely identifies the object created on the server.

IBuffer* m_pBuffer

The buffer that is sent when SendAndReceive() is called. It is a CSimpleBuffer.

Methods

CDoorsSendAndReceive()

Simple constructor. The attributes are initialised to zero.

~CDoorsSendAndReceive()

Closes down the connection to the server and also releases the buffer.

nsresult QueryInterface(iid, ppv)
const nsIID &iid
void **ppv

Part of nsISupports.

nsresult AddRef()

Part of nsISupports.

nsresult Release()

Part of nsISupports.

nsresult FetchBuffer(intBuffer)
void **intBuffer

Part of ISendAndReceive. A new CSimpleBuffer will always be created, witch means this method is destructive. m_pBuffer will be set to the last one created.

SendAndReceive(intBuffer)
void **intBuffer

Part of ISendAndReceive. The contents of the m_pBuffer will be sent to the server and the results are provided in the returned IBuffer (intBuffer).

nsresult CreateObject(sPath)
const char *sPath

Part of ISendAndReceive. Contacts the server at sPath. The server creates an object and returns a door descriptor for this object. This descriptor is stored in m_iDoor.

CRPCSendAndReceive

Class implementing the IConnectionObject by use of RPC.

Component ID

55cbc269-96ef4128b2f3e0603219be22

Program ID

sms.crpcsendandreceive.1

Interfaces

IConnectionObject

Attributes

CLIENT *m_pClient

The pointer for a CLIENT structure that is needed for the RPC connection with the server.

int m_iID

An identifier that is sent every time a call is made to the server to identify the object.

IBuffer *m_pBuffer

The buffer that is sent when SendAndReceive() is called. It is a CSimpleBuffer.

Methods

CRPCSendAndReceive()

Simple constructor. The attributes are initialised to zero.

~CRPCSendAndReceive()

Closes down the connection to the server and also releases the buffer.

nsresult QueryInterface(iid, ppv)
const nsIID &iid
void **ppv

Part of nsISupports.

nsresult AddRef()

Part of nsISupports.

nsresult Release()

Part of nsISupports.

nsresult FetchBuffer(intBuffer)
void **intBuffer

Part of ISendAndReceive. A new CSimpleBuffer will always be created, witch means this method is destructive. m_pBuffer will be set to the last one created.

SendAndReceive(intBuffer)
void **intBuffer

Part of ISendAndReceive. The contents of the m_pBuffer will be sent to the server and the results are provided in the returned IBuffer (intBuffer).

nsresult CreateObject(sPath)
const char *sPath

Part of IConnectionObject. Initialises m_pClient to something that makes sense and also contacts the server to ask it to create an object which ID is then returned and stored in m_iID.

CServerManager

This class is responsible for registering and unregistering object servers and for locating servers for client proxies. It is divided into two parts, one for the servers and one for the clients. The client part contains a factory method for locating and creating the correct ISendAndReceive interface (see section 5.2.3.3). The server part contains methods for registering

Component ID

1C195756-855A-11d3-974B-0008C7599B88

Program ID

sms.servermanager.1

Interfaces

ISMServer

ISMClient

Attributes

None.

Methods

nsresult QueryInterface(iid, ppv)
const nsIID &iid
void **ppv

Part of nsISupports.

nsresult AddRef()

Part of nsISupports.

nsresult Release()

Part of nsISupports.

GetSender(ciidCIID, iType, pISAR)
nsCID &ciidCIID
int iType
void **pISAR

Part of ISMClient. Returns the correct type of implementation of an ISendAndReceive interface depending on what the type is. E.g., if the type is RPC, a CRPCSendAndReceive component will be instantiated and returned. The ciidCIID is looked up through the IDBConnection interface. If the type is set to ANY_TYPE, it is up to the implementation to choose one from the database.

RegisterServer(ciidCIID, strPath, iType)
nsCID &ciidCIID
char *strPath
int iType

Part of ISMServer. Registers a component's CID, server type and location (Path).

UnregisterServer(ciidCIID, iType)
nsCID &ciidCIID
int iType

Part of ISMServer. Unregisters a component. The type must be specified. After this is done, a component proxy can no longer locate a server.
Must be called when a component server ends its execution.

CSimpleBuffer

Class implementing IBuffer in a simple way. Notice that once the caller turns over an allocated area of memory to CSimpleBuffer he has to yield sovereignty of the buffer.

Component ID

1C19575A-855A-11d3-974B-0008C7599B88

Program ID

sms.csimplebuffer.1

Interfaces

IBuffer

Attributes

char *m_pcBuffer

A pointer to the buffer holding the data

char *m_pcBufReadPos

Marks the position for reading from the buffer

char *m_pcBufWritePos

Marks the position for writing into the buffer.

int m_iBufSize

The size of the buffer.

int m_iBufDataSize

The size of the data in the buffer.

Methods

nsresult QueryInterface(iid, ppv)
const nsIID &iid
void **ppv

Part of nsISupports.

nsresult AddRef()

Part of nsISupports.

nsresult Release()

Part of nsISupports.

nsresult SetBufferAndSize(pcBuffer, iDataSize, iBufferSize)
char *pcBuffer
int iDataSize
int iBufferSize

Part of IBuffer. Sets m_pcBuffer to pcBuffer, the old one is freed first. The attributes are set according to the suggestions from the arguments.

nsresult GetBufferAndSize(pcBuffer, iDataSize)
char **pcBuffer
int *iDataSize

Part of IBuffer. Simply sets pcData and iDataSize to m_pcBuffer and m_iBufDataSize respectively. CSimpleBuffer still keeps the memory pointed to by m_pcBuffer.

nsresult Write(pcData, iSize)
char *pcData
int iSize

Part of IBuffer. Copies the number of bytes told by iSize from the memory pointed to by pcData into m_pcBuffer. The buffer will be enlarged if needed.

nsresult Read(pcData, iSize)
char **pcData
int iSize

Part of IBuffer. Reads iSize bytes of data into an allocated area of memory, then sets pcData to point to this area. The caller is responsible for freeing the memory.

bool EnlargeBuffer()

Enlarges the buffer kept by this object. The old buffer is copied to the new one and then freed.

int LeftToRead()

Calculates the number of bytes left to read in the buffer. Used when reading data from the buffer.

int SizeLeft()

Calculates the number of bytes left in the buffer. Used when writing data to the buffer.

Appendix D Evaluation Protocols

D.1 Evaluation One

Conducted by: Fredrik Örvill

Test Program: Evaluation one: Creation of one component

Results:

Server	RPC	Doors	Shared Memory	Pipes
Total (μs)	1602.00	793.90	829.60	985.90
Standard deviation for total time	15.20	17.27	13.20	5.65
Fastest		X		
Slowest	X			
Relative to fastest	2.02	1.00	1.04	1.24
Relative to slowest	1.00	0.50	0.52	0.62

D.2 Evaluation Two

Conducted by: Fredrik Örvill

Test Program: Evaluation two: Method call, small arguments (four bytes)

Results:

Server	RPC	Doors	Shared Memory	Pipes
Total (μs)	300.90	163.00	247.20	296.10
Send&Recevie time (μs)	253.40	117.00	204.10	239.40
Rest (μs)	47.50	46.00	43.10	56.70
Standard deviation for total time	7.10	0.44	0.75	0.88
Fastest		X		
Slowest	X			
Relative to fastest	1.85	1.00	1.52	1.82
Relative to slowest	1.00	0.54	0.82	0.98

D.3 Evaluation Three

Conducted by: Fredrik Örvill

Test Program: Evaluation three (3A): transfer of large arguments, 60kB

Results:

Server	RPC	Doors	Shared Memory	Pipes
Total (μs)	13157.20	10738.60	11560.80	11587.00
Unmarshalling (μs)	5036.40	4961.00	5008.60	5099.10
Transfer (μs)	2255.10	237.60	784.30	741.00
Marshalling (μs)	5777.30	5473.50	568834.00	5652.70
Rest (μs)	88.40	66.50	79.50	94.20
Standard deviation for total time	90.30	200.75	50.10	45.46
Fastest		X		
Slowest	X			
Relative to fastest	1.23	1.00	1.08	1.08
Relative to slowest	1.00	0.82	0.88	0.88

Conducted by: Fredrik Örvill

Test Program: Evaluation three (3B): Transfer of large arguments, 1MB

Results:

Server	RPC	Doors	Shared Memory	Pipes
Total (μs)	269950.60	209991.50	225603.80	230799.70
Unmarshalling (μs)	97482.40	97027.70	96102.40	98419.90
Transfer (μs)	60295.70	831.70	17174.80	20182.00
Marshalling (μs)	111958.80	111955.80	112109.80	111958.30
Rest (μs)	213.70	176.30	216.80	239.50
Standard deviation for total time	912.57	2240.70	567.10	57.80
Fastest		X		
Slowest	X			
Relative to fastest	1.29	1.00	1.07	1.10
Relative to slowest	1.00	0.78	0.84	0.85

Conducted by: Fredrik Örvill

Test Program: Evaluation three (3C): Transfer of large arguments, 16 MB

Results:

Server	RPC	Doors	Shared Memory	Pipes
Total (μs)	4205658.00	3414282.00	3684858.00	3786666.00
Unmarshalling (μs)	1565511.00	1556104.00	1549083.00	1596356.00
Transfer (μs)	784819.30	6056.40	282856.60	339354.80
Marshalling (μs)	1852150.00	1850468.00	1850706.00	1848760.00
Rest (μs)	2177.00	1653.30	2212.10	2194.70
Standard deviation for total time	9780.34	26374.81	3771.45	5174.57
Fastest		X		
Slowest	X			
Relative to fastest	1.23	1.00	1.08	1.11

This page intentionally left blank.