Communication Between Erlang and a Binary Component Object Interface [1]

Per Badlund

June 13, 1999

[1]Thanks to: Patrik Nyblom, Björn Lisper, Claes Thornberg

Contents

1 Abstract

This thesis describes how an interface between Erlang and Microsoft's[1] COM[2] object model can be implemented. It also shows how a COM object could be implemented in Erlang.
The Erlang to COM interface consists of two parts. The first is a server-like application written in C that is responsible for receiving commands and parameters from Erlang, convert these parameters to suitable data types and call the run-time interface on the requested COM object. The second part is a set of Erlang modules that encapsulates the call to the server-like program.
The COM to Erlang interface (writing COM objects in Erlang) part describes how it is possible to implement COM objects in Erlang by creating a middle-layer between the calling application and Erlang. This middle-layer serves as the COM objects front-end to the world. The COM objects functionality is implemented in Erlang. A proposal for how the code should look like for making it possible to create all necessary code through compiling an IDL file is presented.

[1]Microsoft, Microsoft Windows NT, Microsoft Windows 95 and Win32 are registered trademarks (TM) by Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, USA 0298. Other product or company names mentioned herein may be the trademarks of their respective owners.

[2]Component Object Model

2 Overview of this report

2.1 Study

Chapter 1 is an abstract of this report

Chapter 2 gives a short summary about this report

Chapter 3 presents the background and goals with this thesis.

Chapter 4 gives a brief introduction to Erlang

Chapter 5 is an introduction to Microsoft's COM object model.

Chapter 6 shows how to do a call to a COM object from C, which is used in my implementation of the Erlang to COM interface.

Chapter 7 presents a way to invoke a method on an object on-the-fly

Chapter 8 describes how COM/DCOM handles security questions

Chapter 9 tries to compare DCOM and CORBA.

2.2 Implementation

Chapter 10 describes the demands on and prerequisites for the Erlang to COM and COM to Erlang interfaces.

Chapter 11 describes the part of the Erlang to COM interface written in Erlang.

Chapter 12 describes how the part of the Erlang to COM interface written in C works.

Chapter 13 presents a way to implement a COM object in Erlang.

Chapter 14 presents a way to implement a COM object in Erlang using C.

Chapter 15 summarises the results of this thesis.

Chapter 16 explains some of the acronyms used in this thesis.

3 Introduction

3.1 Background

Erlang System / OTP is a programming environment for soft real-time systems. Its primary platform has traditionally been Unix, but it (Erlang/OTP) has since a couple of years also been available for Windows NT, Windows 95 and the real-time operating system VxWorks. Since the origin platform was Unix, the system's model for communicating with the world around has been suited for that environment, which is not always suitable (for example in the NT-world).
Since NT's popularity is growing, Erlang/OTP's customers demands for communication through the interfaces that NT provides (for example ODBC and COM/DCOM) are also growing. Today, Erlang has support for communication through CORBA, but in the NT environment, using COM has a lot of benefits. COM is also a requirement for communication with MTS (Microsoft Transaction Server).

3.2 Goals

This master thesis has two goals:

- Examine how this interface can be implemented and develop an experimental interface (Erlang to COM).
- Investigate how a COM to Erlang interface can be implemented.

3.3 Brief overview of implementation

The following chapters will in detail investigate and explain how an Erlang to COM and also how a COM to Erlang interface can be implemented. One can say that all problems are about passing data from one environment to another. In this report it is about passing data from Erlang to a C environment, and later on, in the opposite direction.
On a higher level of abstraction, the Erlang to COM interface actually consists of two parts, one written in Erlang, and one written in C. But on a lower level of abstraction, the model of this interface actually consists of 4 parts. And it is this model that the implementation of the interface, which this report describes, is built on. Figure 1 describes how the interface is split into four parts. Each part is described thoroughly in a separate chapter in this report, but here is a brief overview of these four fundamental building blocks of the Erlang to COM and COM to Erlang interface:
Erlang COM API
Described in chapter 13
This is an Erlang module. It implements the Erlang COM function call.
Erlang in/out module
Described in chapter 11
This part is responsible for sending data from Erlang to the part of the interface written in C.
The in/out part written in C
Described in chapter 12
The COM libraries
These librarys are not developed in this thesis, they are developed by Microsoft and shipped with Windows This part is responsible for receiving data from the Erlang in/out module, calling the COM libraries and writing the result back to the Erlang modules.

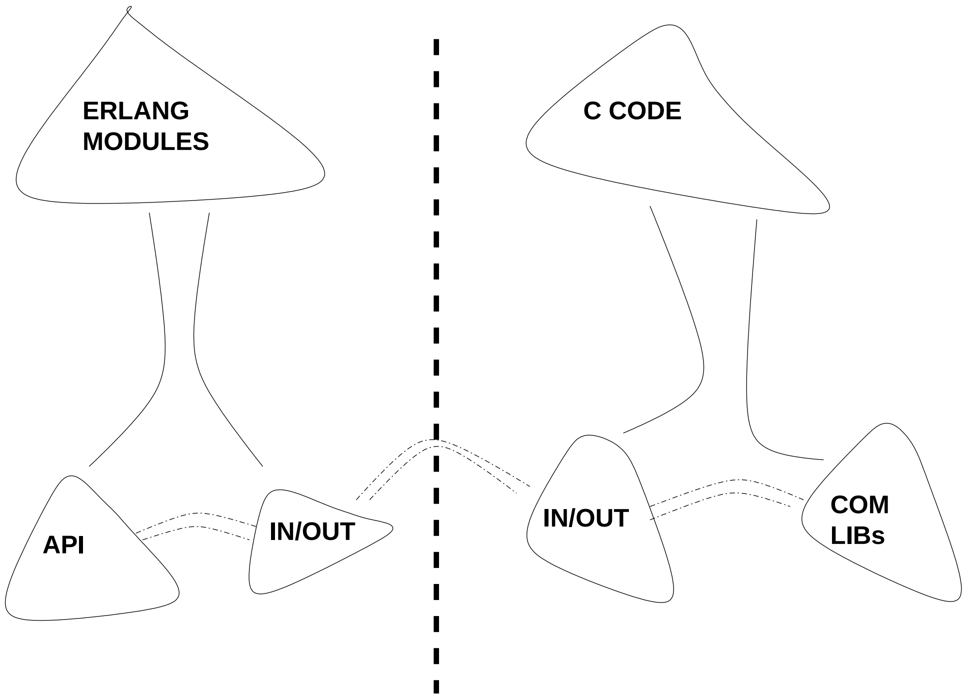

Figure 1: Large overview of interface.

4 Introduction to Erlang

This chapter is by no means a complete introduction to Erlang. It rather gives enough information to understand the Erlang code that is part of thesis theses and shown in later chapters. For a more complete introduction, see [2]
Erlang is a programming language developed at the Ericsson and Ellemtel Computer Science Laboratories. The goal with the development of Erlang was to see if modern declarative programming paradigms could be used in modern industrial telecommunications switching system.
Erlang is suitable for concurrent, real-time, distributed, fault tolerant systems:

4.1 Concurrency

The Erlang concurrency model is based on *processes*. The processes communicate with each other through asynchronous message passing. A process is created by the command `spawn` which returns a process identifier. To send the message `Msg` to this process, the syntax is `Pid ! Msg`, where the variable `Pid` is the process identifier. To receive a message, this syntax is used:

```
receive
   Message1 ->
     ...;
   Message2 ->
     ...;
   ...
end
```

This receive function also is an example of another property of Erlang, the Pattern Matching. The receive message takes the first message which matches one of the message patterns from a queue of messages waiting for the receiving process. Another property of Erlang is the Module system. Since large concurrent programs can be difficult to maintain, the program can be divided into modules. Example:

```
-module(factorialmodule)           %1
-export([factorial/1])             %2

factorial(0) -> 1;                 %3
factorial(N) -> N * factorial(N-1).%4
```

Row 1 declares the module `factorialmodule` (row 1), and the module exports one function `factorial/1` (row 2) which is function that takes one parameter (= /1).

This concurrency mechanism is lightweight, so creating and deleting these processes does not cost very much computational effort.

4.2 Real-time

Erlang is a *soft* real-time system. This means that it is suitable for applications where the response time is allowed to be in order of milliseconds. Since Erlang has a built-in notion of time, it is possible for the programmer to set time-outs, that is, the programmer can instruct the program to take some action if a message that a `receive` statement is waiting for has not arrived within a certain number of milliseconds.

Example:

```
receive
   Message1 ->
     ...;
   Message2 ->
     ...;
   ...
   after 1000 ->
     do_something() % Do something after 1000 ms.
end
```

4.3 Distribution

In distributed Erlang, the concept *node* is important. A node is a running Erlang system that can take part in distributed transactions. The name assigned to a node is globally unique and always consists of two parts separated by a `@`. The Built In Function , BIF, `spawn/3` (the version of the function `spawn` which takes 3 parameters) creates a process and returns a Process Identifier on the local node. Many processes can run on a node. If we want to create a process on another node, the BIF `spawn/4` is used which takes as the first parameter the node where the process should be spawned. The Process identifier returned from the `spawn/4` node can almost always be used as the Pid returned from `spawn/3`

4.4 Fault tolerance

Since Erlang is used for high reability systems, the error handling technique is very important. Basically it is possible to

- monitor the evaluation of an expression
- monitor the behaviour of other processes

To monitor the evaluation of an expression the functions `catch` and `throw` are used. The normal behaviour for a program when the expression `Expression` is evaluated and fails is to exit abnormally. This default behaviour can be changed using `catch`. The statement `catch Expression` returns the value of the expression `Expression` if the evaluation succeeds, otherwise the statement will return the tuple {`'EXIT','reason'`} where 'reason' might be for example `badarith`, `badarg`.
Normally when a function has been evaluated, control is returned to the caller. `throw` is used to bypass this. When using throw within a `catch` statement, then a direct return is made to the `catch`. The function pair `catch` and `throw` can be used for example to guard against bad code.
It is possible to link processes together. If a linked process terminates (normally or abnormally) it sends the message {`'EXIT',Exiting_Process_Id,Reason`} to all processes that the exiting process is linked to. If `'reason'` differs from `'normal'` the default action of the receiving process is to terminate and send `'EXIT'` signals to all processes that it is currently linked to. But this behaviour can be overridden.

4.5 Data Types

Erlang has the following data types:

- Constant data types, as numbers (for example 123,-3,3.1415e2). These numbers are divided into integers and floats. Atoms are also constants, for example `hello` and `'this is an atom'`

- Compound data types. To this group belongs `Tuples` (for example {`1, 2, 3`}, {`a, b, c`}) and Lists (for example [] (the empty list), `[a b c]`,`[a b [c d]]`.

- Components of Tuples and Lists - this makes it possible to create arbitrary complex structures.

4.6 Ports

Sometimes an Erlang program wants to communicate with a program written in another language. For the Erlang programmer it is convenient to see this external process as an Erlang process. To do so, this external process must behave like an Erlang program. To create this illusion, an abstract called a `Port` provides a byte-oriented communication channel between the Erlang program and the external process. The external program communicates with the Erlang process connected to that port through a sequence of bytes. The Erlang process communicates with the external program by sending messages to the Port. For more detailed information about ports, see [2].

5 Introduction to COM - Microsoft's Component Object Model

5.1 What is COM?

COM stands for "Component Object Model". To get the whole picture, we begin with the history behind COM.

The force behind the development that resulted in COM was the need for support for sharing data. In early applications, this was partially solved by the cut-and-paste functions. But this method had a major disadvantage: the information about the application that initially created the data was lost when the data was transferred to another application. An early solution by Microsoft was their Dynamic Data Exchange, DDE.
DDE's disadvantage was that it used a MS-Windows message based mechanism and used "invisible" windows to generate a queue to serialize messages. So DDE is MS-Windows specific and was not ported to other operating systems. Another reason why DDE was not really a good solution was that the DDE protocol was poorly specified, which lead to inter-operability problem between programs.
Layered on top of the DDE were the more known OLE version 1 (OLE1), Object Linking and Embedding. The feature with OLE1 was that it allowed objects to be embedded into compound documents but still had a link back to its creator. When an object needed to be edited, this was done in the application that created the object. OLE1 was a great improvement over DDE, but soon OLE1 was not powerful enough to handle more demanding objects like video and audio. The new developed version of OLE was initially called OLE2, then called just OLE and now this functionality is called COM.
But COM is more than making it possible to insert audio/video into a Microsoft Word document, it is also a way of building complex, distributed applications (Distributed COM) by using COM objects as a fundamental building block.

5.2 Overview of COM

One can compare a COM object with an object in an object-oriented programming language. It consists of encapsulated, runnable code. It has methods to access the object, but one cannot directly access internal variables. A COM object can implement many methods, and these methods are grouped into *interfaces*. From a COM programmer's view, the interfaces are the important elements of an application. The components are just implementation details of the interfaces.

5.3 What is COM and what is it NOT?

A COM component is a piece of executable code, distributed as either a Win32 Dynamic Link Library (DLL) or an EXE file. This means that it can be shipped in a binary form. These components link dynamically. The components are language independent since they are compiled to DLLs or EXEs. To the COM user, the components behave transparently: they can be local or on another machine on the LAN,WAN or Internet (providing that DCOM, the distributed version of COM, is available). This is possible since the COM components announce their existence in a standard way through the Registry.
But COM objects can not be:

- compared with computer languages like C++, Ada, Smalltalk etc. It is meaningless to say that COM is better than C++ since there is this fundamental

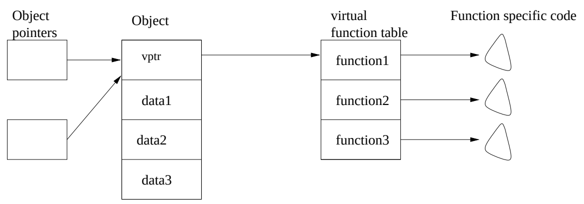

Figure 2: C++ binary representation of an object.

difference: COM is a standard that tells us how to implement components (in for example C++).

- replaced by DLLs or EXEs since COM uses both these formats for exposing themselves to the world.

And above all, COM is not an API like the Win32 API, but the Win32 API do provide some basic functions for managing the components.

5.4 Interfaces

One reason why the component market is growing, is the fact that today's vendors have to release new versions of programs quite often. But changes are often minor, so if programs are split into modules they could be updated by changing a single component/module instead of compiling a whole new version. But the developers also have to remember that the new versions of the code they are releasing must be backward compatible with the previous version (often several versions back). The COM object fulfils all these needs. An object may only be called a COM object if some conditions are fulfilled. One of those is that it is forbidden to remove methods in an interface (or remove the interface), we may only add interfaces to a component. Another condition that must be fulfilled is that we have to implement the *IUnknown* interface. This interface contains the following methods:

- QueryInterface, which is used for querying an object for a certain interface.

- AddRef, which increments the objects internal reference counter. When a new instance of an object is created, this method must be called.

- Release, which is used to decrement the reference counter when the program does not need the instance anymore. When the counter is zero, the instance should be removed.

5.5 Interfaces - a binary representation

According to [3] an interface is a specific memory structure containing an array of function pointers, and each array element contains the address of a function implemented by the component. Notice that an interface says nothing about how a method is implemented, it just points out the address of the runnable code that implements this method. Since COM objects can be compared with objects in object-oriented languages, it is interesting to see how C++ could implement its objects.

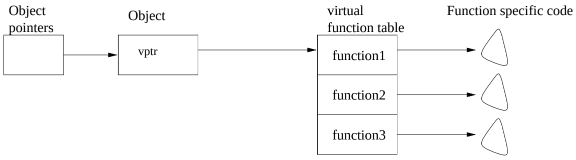

Figure 3: COM representation of an object.

The code that generates the binary memory layout in Figure 2 looks something like this in C++:

```
interface IFace
{
   virtual void __stdcall function1() = 0;
   virtual void __stdcall function2() = 0;
   virtual void __stdcall function3() = 0;
}

class MyClass : public IFace
{
public:
 // here we implement interface IFace !
 virtual void __stdcall function1() { cout << ‘‘fun1’’ << endl;}
 virtual void __stdcall function2() { cout << ‘‘fun2’’ << endl;}
 virtual void __stdcall function3() { cout << ‘‘fun3’’ << endl;}

 // the constructor
 MyClass(int nr)
 : my_data1(nr+1), my_data2(nr+2), my_data3(nr+3)
 {}

 // our instance data
 int my_data1;
 int my_data2;
 int my_data3;
};
```

Not all C++ compilers on all platforms creates memory-layouts as the one in Figure 2. But all C++ compilers on the Windows platform do so, which is why it has become a de-facto standard

We should notice though, that *interface* is not a keyword in C++, but actually defined as a struct. But in C++ classes and structs are the same except for that structs use public inheritance and access by default rather than private [4]. Figure 2 is quite similar to the description of a COM interface that is found in Inside COM [3]. The only difference is that when dealing with a COM object, there is no way to access the instance data directly (as it can be done when using C++). So our picture of a COM interface looks like the one in Figure 3.

The example above is not really a COM object since it does not satisfy the requirements specified in the contract: the IUnknown interface is missing! This can be solved by letting MyClass also inherit from IUnknown (which is declared in the file `objbase.h` which is included in Microsoft's Software Development Kit (SDK). So, if MyClass is allowed to inherit from IUnknown, we end up with a COM object which has the binary layout described in Figure 4.

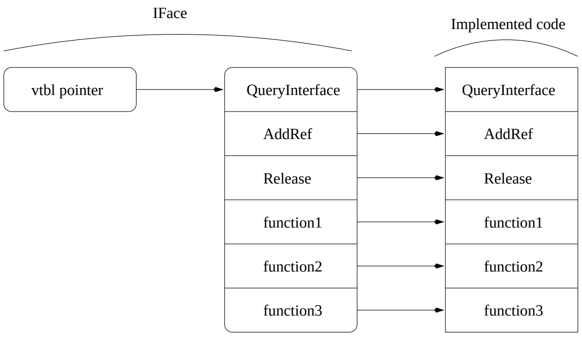

Figure 4: Our object as a COM object.

5.6 When is this information needed?

When using for example C++ it is not necessary to know what a Virtual Function Table (vtbl) looks like. But, how does the client know which position a function has in the vtbl? The client includes a header file containing the interface description in the form of an abstract base class. The compiler reads this header file, and for each member function in the abstract base class it assigns a number. This number is equal to the position the function has in the vtbl. So when we in C++ do the following call:

```
pIMyInterface->MyMethod(msg);
```

then the compiler interprets it as:

```
(*(pIMyInterface->vtbl[IndexOfMyMethod]))(pIMyInterface,msg);
```

But when not using C++, e.g. when using C, then the programmer will have to do the C++ compiler's work by hand. The call would be:

```
(*((*pIMyInterface)+IndexOfMyMethod))(pIMyInterface,msg);
```

5.7 Naming of interfaces and objects

Each class and interface is assigned an ID (CLSID and IID). An ID is actually a type defined as a 128-bit (16-byte) structure called a GUID - *Globally Unique Identifier*. The GUIDs are constructed by an algorithm that uses the computer's network card ID (48 bit), 60-bit for the timestamp which represent the count of 100-nanosecond intervals since 00:00:00.00 15 October 1582. This algorithm will roll-over around AD 3400. So it is safe to assume that every GUID is unique. But how do we use these GUIDs?

5.8 The Registry

The database that holds all the information about programs, system resources in Windows 95 and Windows NT is the Registry. The Registry is a tree structure, with different branches for different applications. Every branch ends up with, perhaps after several sub-branches, some information on how to start a program or find a specific file. (See Figure 5 for a basic view of the registry). The Registry tree starts with four main-branches, of which COM only uses the first main-branch, the HKEY_CLASSES_ROOT. In that branch, all the components' CLSIDs are listed, and each CLSID is represented by a string, which have this hexadecimal representation: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx. But luckily, each CLSID has as its

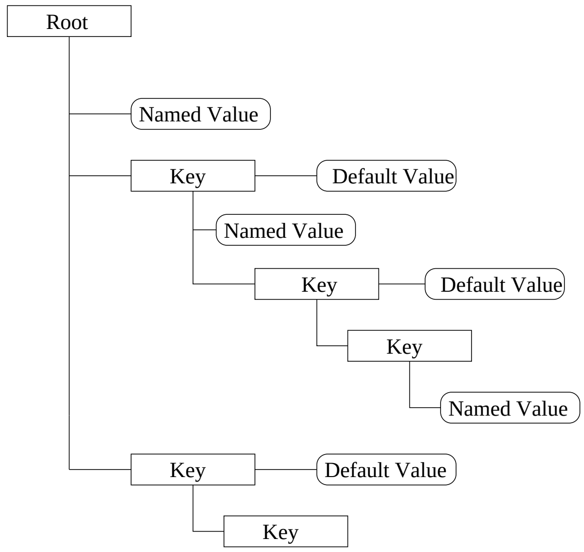

Figure 5: Structure of the Registry.

default value its friendly-name. So in some cases, if we want to, we can replace all our CLSIDs with a friendly name, for example "My New Component". But observe: when using the friendly name, we can no longer assume that the name is globally unique. So when a user requests a pointer to an interface of a component, the user supplies the CLSID, and the IID (or the friendly-names). The COM libraries then look up the registry, find the key that has as its default value the compiled code (.dll or .exe), execute all the necessary steps to get the pointer and return it.

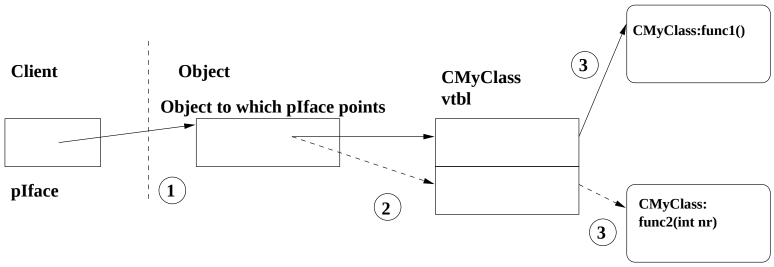

Figure 6: Virtual function call through interface pointer.

6 General COM calls from C

This section describes how a call to a COM object could be made from a C program. Reading it helps the understanding of the part of the implementation of the Erlang to COM interface described in chapter 12 and the C code implementing the functionality described in chapter 12.

First of all, the COM libraries need to be initialised. This is done by doing this call:

```
hr=OleInitialize(NULL);
```

The result from this call is stored in `hr` as an `HRESULT`. One should always examine the result from calls that return a ĤRESULT. As a first control, the `FAILED` macro can be used, for example:

```
if (FAILED(hr))
  do_something(...)
```

To create an instance of an object (and also obtain a pointer to a certain interface), one can use the `CoCreateInstance` call, which have this syntax:

```
CoCreateInstance(REFGUID clsid,
 ?? ??, /* must be NULL */
 ?? context,
    REFGUID iid,
 void** pInterface);
```

If we want to call a method in a certain interface without the run-time method invocation support that is implemented in the COM libraries, we can use the binary structure of the vtbl and write like this (the interface pointer is called pIface):

```
(*((*pIface) +1))(pIface,4711)
```

Step 1: (*pIface) this is the first indirection
Step 2: *((*pIface) +1) is the index into the vtbl
Step 3: call the function with pIface and the argument 4711
This call passes 4711 to the second function in the array. The call is visualised through Figure 6. The numbers in the picture corresponds to the steps above.
When using C++, then this would be much easier. If the object (MyObject) is declared in the file "MyObject.h" ,that is, all the interface descriptions etc are declared, this file is included before compilation of the program. Then a pointer can be declared as:

(assuming that MyObject contains the interface IMyInterface)

```
IMyInterface* pIMyInterface;
CoCreateInstance(CLSID_MyObject,NULL,IID_IMyInterface,(void**)pIMyInterface);
...
pIMyInterface->Func2();
...
```

If the syntax for `CoCreateInstance` is studied, we can see that the type for the CLSID and IID is `REFGUID`. This is a reference to a GUID. If we study C++ code that contains a CoCreateInstance call, it could look something like this:

```
hr=CoCreateInstance(CLSID_MyObject,NULL,IID_IFoo,(void**)myPointer);
```

If we try to use this code in the C program there will be an error. This is because C++ uses passing by reference while C uses passing by value. So we must "force" this reference passing through this call:

```
hr=CoCreateInstance(&CLSID_MyObject,NULL,&IID_IFoo,(void**)myPointer);
```

A "gentle" introduction to COM can also be found in "Dr GUI" on Microsoft's website [4]. It is a good start, almost like "COM for dummies" .

It is important to know that the CLSIDs and IIDs passed to this COM-wrapper as strings are not used in that format in for example the `CoCreateInstance` call. A function has also been created in this COM interface implementation to convert a GUID in string format to a GUID in its correct binary form.For information about this binary form, please see the On-line manual in Visual C++ [5].

7 Automation, or the IDispatch interface

When programming in object-oriented languages, or programming using COM objects, the standard approach is to communicate directly with an object through it's methods. This chapter introduces the concept of Automation, a way to indirectly communicate with an object.

7.1 About Automation

Automation focuses on run-time typechecking. But it does so at the expense of speed and compile-time type checking. Macro writers are grateful for this run-time typechecking, but for C++ programmers this requires much more work. Often Automation replaces code generated by compiler with code written by the programmer. Automation is not separated from COM, but build on top of it. Two concepts are widely used when talking about Automation: *Automation Server* and *Automation Controller*. The Automation Server is a COM object that implements the IDispatch interface. An Automation Controller is a client that communicates with an Automation Object through its IDispatch interface.

7.2 Why Automation?

The section above mentioned that using Automation decreases the performance of the application. The alternative would be to create include-files which describe the layout of every COM component that the programmer wants to use, and then get a pointer to the binary structure described in the include file. And when the programmer gets this pointer, he can easily call functions using it. But one of the conditions of this thesis was that it is not allowed to recompile code or dynamically create new code. So we actually have no idea of what this new interface that we want to use looks like. Without this knowledge, it is impossible to "manually" do any method-calls. But when using automation, the COM libraries solve this problem.

7.3 The IDispatch interface

The IDispatch interface makes it possible to encapsulate a whole COM object and make all the interfaces and methods accessible through one single interface, the IDispatch interface. The methods of the IDispatch interface are the following (appearing in the same order as in the virtual function table) :

- GetTypeInfoCount
- GetTypeInfo
- GetIDsOfNames
- Invoke

When discussing Automation, the most important methods are the GetIDsOfNames and Invoke. GetIDsOfNames takes a name of a function and returns its DISPID (an ID unique only for the specific instance of IDispatch) which identifies the function in the object. Invoke takes this DISPID and executes the function. Figure 7 illustrates the IDispatch interface (note that the IDispatch *must* inherit from the IUnknown interface). This is not the only way to implement a dispinterface, i.e. an IDispatch interface, see [3, pp283]. But for the purpose of understanding this report, understanding Figure 7 is enough.

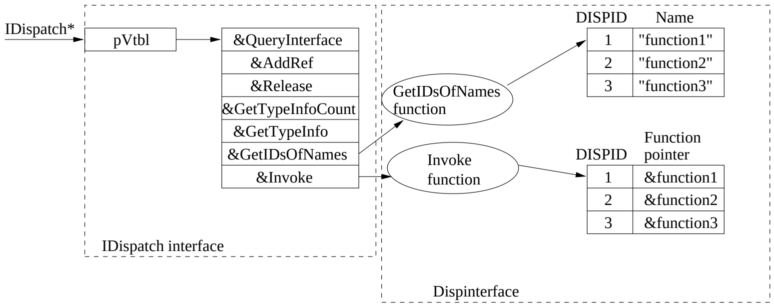

Figure 7: C++ binary representation of an object.

7.4 Digging deeper into Automation

This report focuses quite a lot on Automation, since the Erlang to COM interface described in this thesis totally relies on the IDispatch interface.
The IDispatch interface is defined in the file `oaidl.idl`, and the definition of the IDispatch interface looks like this:

```
[
    object,
    uuid(00020400-0000-0000-C000-000000000046),
    pointer_default(unique)
]

interface IDispatch : IUnknown
{
    typedef [unique] IDispatch * LPDISPATCH;

    HRESULT GetTypeInfoCount(
                [out] UINT * pctinfo
            );

    HRESULT GetTypeInfo(
                [in] UINT iTInfo,
                [in] LCID lcid,
                [out] ITypeInfo ** ppTInfo
            );

    HRESULT GetIDsOfNames(
                [in] REFIID riid,
                [in, size_is(cNames)] LPOLESTR * rgszNames,
                [in] UINT cNames,
                [in] LCID lcid,
                [out, size_is(cNames)] DISPID * rgDispId
            );

    [local]
    HRESULT Invoke(
                [in] DISPID dispIdMember,
                [in] REFIID riid,
                [in] LCID lcid,
                [in] WORD wFlags,
                [in, out] DISPPARAMS * pDispParams,
                [out] VARIANT * pVarResult,
                [out] EXCEPINFO * pExcepInfo,
                [out] UINT * puArgErr
            );
```

8 Distributed COM (DCOM) compatibility

Although the Erlang to COM interface implementation described in this thesis has not got explicitly implemented DCOM support, Erlang users can use the developed application to do DCOM calls. To enable this, the user must make some changes to the Registry to inform the object that the invocation will be on another machine. This is how it is done:
Store the remote server name under a new key in HKEY_CLASSES_ROOT

```
[HKEY_CLASSES_ROOT\APPID\{<appid-guid>}]
``RemoteServerName''=<DNS name>
```

Also, under the class ID entry for the component, add a new named value that points to the APPID:

```
[HKEY_CLASSES_ROOT\CLSID\{<clsid-guid>}]
``AppId''=''<appid-guid>''
```

The APPID essentially represents a process that is shared by multiple CLSIDs. All objects in this process share the same default security settings. The APPID concept can be used to avoid redundant registry keys. For example, CLSIDs that are known to always run on the same machine (perhaps because they are implemented in the same DLL) can all point to the same APPID key and thus share the same RemoteServerName registry key.

But sometimes this won't do. For example, when someone usually creates an instance of an object on the local machine, but then wants to create it on a remote machine. This could be done the hard way, first changing the registry (adding the APPID key), do the call and then restore the Registry. This is not a good idea. But there are some functions in the COM libraries which will take care of this situation: *CoCreateInstanceEx, CoCreateInstanceFromFile, ...* . The problem is that today the developed application described in this thesis uses the CoCreateInstance call to get a pointer to an object. The difference between CoCreateInstance and CoCreateInstanceEx is that the last one takes some parameters for explicitly telling on which machine the object is. A proposal on how to change the developed Erlang to COM interface so that it can be used to do both local and remote invocations:

The last parameter in the call from Erlang is a string which now must have the default value `persist`[3] . If we instead give this string the possibility to take the values "local" or "remote", we can depending on which alternative chosen, either call CoCreateInstance or CoCreateInstanceEx. We must also add parameters to the call from Erlang where we specify on which machine we wishes to create the object. This is no problem, since the code in Erlang that takes care of sending the parameters to the C-server just loops and pushes the parameters to the Erlang Port.

8.1 Security

Connecting to a COM object functionality is implemented in Erlang actually means transparently connecting to a distributed Erlang node. One security check is done

[3]This parameter is completely unnecessary,but it can be used as a reminder: we want a "persistent" connection with an object, several calls to one object instead of several calls to several objects

when connecting to the Erlang node via the Erl_interface library, when one has to supply the distributed node's cookie. But this does not really solve the security problem. This only shows that the COM object is authorised to connect to the node, not that the user of the object is authorised. Different ways for DCOM to authenticate a user is described in the following sections.

8.2 Security Policies

DCOM uses four aspects of security:

- Access Security: check that a user is allowed to call a specific object.
- Launch Security: controls the creation of processes when object instances are created.
- Identity: the system must protect itself from malign objects.
- Connection Policy: eavesdropping on the data must be prevented. The system must also be protected from objects stealing an authenticated object's identity.

8.3 Access Security

One of the most obvious security issues is how to protect objects from unauthorised access. Example: how can we prevent an evil person from using the `TakeDownTheWholeSwitch` object? Currently, DCOM provides access on a per process level, but also gives the programmer the possibility to perform validations on a per object, per method or even per method parameter basis, and also per caller basis.

8.4 Launch Security

To validate if a client may launch a new process with a COM object is not a very good idea, since the launching of the process may be considered a security breach. The launch security check is performed by the COM libraries, and its decision is based on privilege information stored in the Registry, external to the object.

8.5 Identity

It can be difficult to authenticate an object. One way is to let the object inherit the caller's ID, and based on the caller's privileges COM can chose to either grant or deny the request. This works fine as long as an object is only used by one caller, since the identity of an object is set at creation time. When an object is created, the resources the object might need also be configured to the caller's ID. So, if the privileges are too restrictive, many calls will fail. If they are too generous, we've got a potential security hazard. Although we can solve this problem by dividing the callers into groups, an easier way is to give the object a dedicated security identity.

8.6 Connection Policy

Eavesdropping on packets has become a problem (e.g. when sending information about their credit cards). If someone alters the data on a packet, the network should detect this and a retransmission should take place. The problem is how to know that a non-altered packet really originates from a legitimate caller. DCOM supports authentication of the caller, and it is possible to choose how often this should be done (on every packet, on the first packet, etc.). But this also affects the transmission overhead. The current version of DCOM supports two choices of data protection: privacy and integrity. The integrity choice implies that every packet sent between a client and the COM object server contains authentication information. Privacy implies that integrity is chosen *and* that the data is encrypted.

8.7 How DCOM identifies a caller

To protect us from attackers who pretend to be someone else, DCOM must also be able to identify a caller. This check is done by a security provider, for example the NT LAN Manager, NTLM, and the security provider returns a unique token to the client, which then is used during the rest of the session. The default behaviour of DCOM is to require authentication only when the initial connection is established. The token is then cached on the server and used when the server detects a call from the same client. The application can choose whether only the first call of a method invocation should contain authentication information, or if every packet should contain this information.

8.8 How DCOM protects the caller

So far, only the security of the object has been discussed. But we also know that the caller must be protected from malign objects. We can compare this with Java applets and viruses spread with executables. The problem is that DCOM allows objects to inherit the callers ID, so the objects could actually perform operations it would not be able to do alone. The following options are defined:

- Anonymous - which means that the object is *not* allowed to obtain the identity of the caller.
- Identify - the object can only detect the security identity of the caller but is not allowed to inherit this identity
- Impersonate - the object is allowed to impersonate the caller and can do local operations, but is not allowed to call any other objects on behalf of the caller. Note that this alternative is still a security hazard, since the object may perform arbitrary operations on the local machine.
- Delegate - the object may do anything the caller can do. In NT 4.0, no security provider supports this alternative.

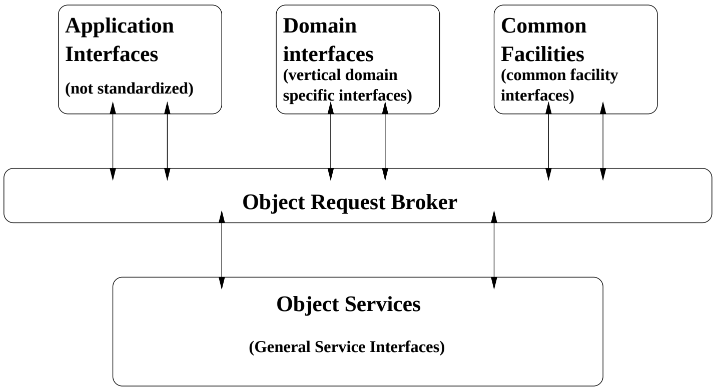

Figure 8: The Object Management Architecture.

9 COM vs. CORBA

COM/DCOM and CORBA[4] are two completely different standards. COM is a binary standard in that sense that it basically defines the binary layout of a COM object (and some rules that apply to interfaces), while the CORBA standard is actually a number of standards for improving object inter-operability between independently developed applications across networks.
The intention with this chapter was to compare the two dominating standards for distributing tasks; CORBA and COM/DCOM. When comparing two things, by nature the compared things should be different. The result, in this case, was that the compared standards actually were *too* different to be compared in a just way. So, instead of comparing COM/DCOM with CORBA, this chapter introduces some standard specific features and, if possible, describe the other standard's corresponding feature.

9.1 The CORBA standards

The COM standard has already been described in this report. This chapter will give a brief overview, but for more information, see the OMG web page http://www.omg.org.

The Object Management Architecture, OMA, reference model splits the OMG problem space into high level components. The OMA can be seen as three segments consisting of five objects:

- Application Oriented. These are solution specific components that are closest to the end user.
- System Oriented. Mainly concerned with the infrastructure of distributed objects. In the OMA, this segment is represented by the ORB and Object Services.
- Vertical Market Oriented. Represented in the OMA model by Domain Interfaces.

The OMA model is shown in Figure 8, and here is a brief overview of the components in that figure:

[4]Common Object Request Broker Architecture

- Object Request Broker (ORB). The ORB is CORBA's "engine". The ORB is responsible for (among other things) delivering the client's request (call) to the implementation of an object. COM's equivalence is the COM libraries. The ORB is defined by its interfaces. Any ORB that provides these interfaces is approved. What happens beyond those interfaces is not specified. So, there are several different ORBs (Orbix, VisiBroker, etc.) with different characteristics and performance.
- Object Services. These components provide the life-cycle management of the objects. COM's equivalence is the internal reference counter of each object. They are also used for keeping track of relocated objects and relationships between objects.
- Common Facilities. Also known as CORBAFacilities. Provides a set of generic application functions that can be configured in order to suit a particular implementation. Examples of facilities are: printing, electronic mail etc. The purpose of this standardisation is to give better options for end users for configuring their working environments.
- Domain Interfaces. Vertical areas that provide functionality that are of interest for users within a certain industry.
- Application Objects. This is not really an OMG standardisation activity. These objects can be compared with COM objects.

The goal with the OMG Object Model is to define the externally visible characteristics of objects in a standard and implementation-independent way. The OMG object model is based on the following concepts:

- Objects
- Operations
- Types and Subtypes

An object in the CORBA world is like an object in any object-oriented language, and the object's behaviour is characterised by its methods (operations). COM objects are characterised by their interfaces.

A relationship between CORBA objects can be established, since CORBA objects can be viewed as instances of Types, and Types in turn can be derived into sub-types. So there could exist a relationship between for example sailing-boat objects and submarine objects since both could be derived from the Type Boat. There are no such explicit relationships in COM. The COM libraries cannot handle relationships between objects in the same way as CORBA can. This might be a result of the fact that objects in the COM world are less important then the interfaces! In the COM world, one would rather say that two methods could be related since they are both implemented in the same interface.

In COM, an object is represented by a CLSID, but to the programmer, the object is represented by a pointer to one of the object's interfaces. This pointer is created by calling a function in the COM libraries with the CLSID and IID as parameters.

In CORBA, an object within an ORB is represented by an *object reference*. An object reference will identify the same object each time the reference is used in a request, and an object can be denoted by multiple, distinct object references. A difference between object references and COM's pointers to interfaces on an object is that one cannot explicitly create or destroy objects in CORBA. Objects are created and destroyed as an outcome of issuing requests. The outcome of object creation is revealed to the client in the form of an object reference that denotes the new object. In COM, we can explicitly create an interface pointer (COM's counterpart to CORBA's object reference) by doing a CoCreateInstance(...,pointer,...).

This permits a program written in a particular language to access object references independent of the particular ORB. The language mapping includes language-specific data types and procedure interfaces to access objects through the ORB. And it also includes the structure of the different interfaces that an ORB uses, the client stub interface, dynamic invocation interface, skeletons on the object side, object adapters, and of course, the ORB interface. A big difference between CORBA and COM is that the object reference is only valid as long as the client is alive. The COM representation of the object is valid until AD 3400 (approximately).

We mentioned earlier that the client stubs and object skeletons are needed to access the object. These are created by compiling an OMG IDL (OMG's extension to Interface Description Language) file. In COM, the IDL file is used to create the type-library that COM objects use. But as in COM, the ORB can manage without IDL files. If there is an interface description in the runtime interface repository, this will also do. In COM, including a header file with the object description is enough, since COM is a binary standard.

The stubs are mainly used when the client is programming in a none object-oriented language. The stubs present the means of accessing an object, or rather, the object's methods. The stubs make the calls to the ORB. The stubs are usually optimised for a specific ORB. There is also a possibility for the client to dynamically invoke operations on objects. They are made through the Dynamic Invocation Interface. The client specifies the object, the method and parameter and their types. Then the call is made. Microsoft's interface that provides the equal service is the IDispatch interface, which is used in the developed implementation of the COM interface. CORBA clients can get the necessary information about parameter types etc, for example from the Interface Repository; in COM this information can be fetched from the type-library. The skeletons are so called "up-call" interfaces, which means that the calling direction is upwards (see Figure 9). These interfaces map to the methods on the object. These interfaces are used by the ORB to call the methods. As it is possible for the client to use the Dynamic Invocation Interfaces, there are corresponding skeleton interfaces for dynamic invocation. The implementation skeletons are often up-call interfaces. The object communicates with the ORB through the Object Adapter (see Figure 9) . The services provided by this Object Adapter are for instance: generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations. Figure 9 shows how static and dynamic requests move from the client to the object implementation.

COM uses the Registry for retrieving information about objects and interfaces at runtime (that is, COM uses the Registry for locating the type library). CORBA has two *repositories* for this purpose: the Interface Repository and the Implementation Repository. The Interface Repository stores IDL information about interfaces as objects. These can be retrieved at run-time. It is also possible to store other information about objects, for example debugging information. The Implementation Repository provides all information that the ORB needs to locate and activate implementations of objects. Additional information about ORB objects can also be stored here.

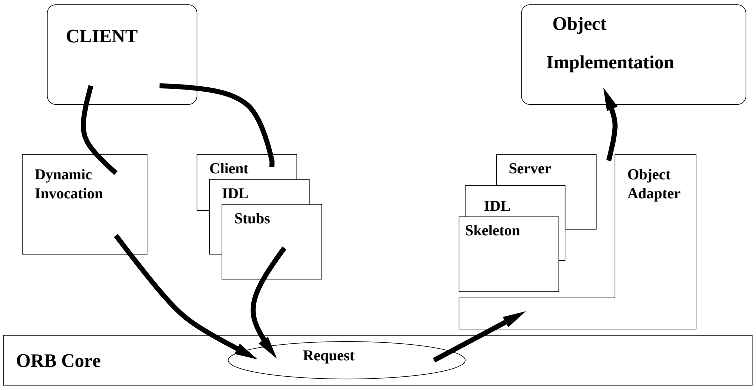

Figure 9: Static or Dynamic CORBA Requests.

9.2 Remote invocation of objects on a protocol level

The DCOM protocol is an application level protocol for object oriented remote procedure calls. The protocol is a set of extensions, layered on top of the (DCE) RPC specification. This protocol is called ORPC, “Object RPC”. In an ORPC, the object ID field in the invocation header contains an ”IPID”. This IPID is a 128-bit identifier that identifies a particular interface on a particular object in a particular server. Its static type is UUID, but it is not usually generated using this algorithm. So instead of being scoped globally, they are scoped only relative to the server process that originally allocated them. The objects IID is passed in the interface ID field. The arguments are found in the body. The IPID might contain enough information to point out a particular interface on a particular object on a particular server. But it does not include enough information to actually carry out an invocation. The protocol solves this problem by the “Object Exporter Identifier” (OXID). The OXID can be thought of as the implementation scope of the object (could be a process, a whole machine or even a thread in a process). On each machine there is a unique OXID resolver service. This service keeps track of all exported and imported IPIDs. The OXID is responsible for determining the RPC string bindings that allow calls to reach their target IPID. With every OXID object there is a COM object termed an ”OXID object”, which at least implements the IRemUnknown interface, a COM interface through which remote management of reference counts and requests for interfaces are returned.

9.3 COM vs CORBA, differences in implementation

The main difference between the developed implementation of COM interface and the CORBA implementation is that CORBA’s ORB is implemented as a part of Erlang. I have to use a wrapper to communicate with the COM libraries. However, this difference will disappear when the COM functionality is implemented in Erlang. Then the COM libraries can be compared with the ORB implemented in Erlang.

10 Design considerations

10.1 The part written in C

When using COM from C/C++ (or any other language), it is assumed that the user is able to include object specific headers, create new code dynamically and do a new compilation whenever the user wants to. When these requirements are fulfilled, using COM is no longer Rocket Science. But these requirements differ quite a lot from those that effected the work on developing the Erlang to COM interface. The requirements that the implementation had to adjust to were:

- No assumptions about compilers allowed. We cannot assume that there is a compiler available, and if there is a compiler available, we cannot know what compiler it is. That means that we can neither generate new code and compile it, nor include new headers depending on the object the Erlang user wants to work on and recompile the code..
- We cannot use C++, although this language is the perfect language for working with COM, C must be used, since the rest of the Erlang Runtime System, ERTS, is written in C (otherwise it might not be possible to insert the COM functionality into the run-time system).

10.2 No compilers or C++

The fact that no code generation is allowed is not really a very big problem. Unless we are not trying to dynamically create a new COM object, we can do without any new code. But the fact that no compilers are allowed makes it a bit tricky when we want to call a method on a COM object, since we cannot “import” knowledge about the object through the included header file. Often there are special header-files for each object that describes the structure of the COM object and also gives the C-programmer some very useful macros for accessing methods. Luckily, there is a method for examining the properties of a COM object without including any files: through the Win32 API examine type-libraries (a binary description of a COM interface, created by compiling a IDL[5] file with MIDL, Microsoft's IDL compiler).

This method had to be chosen, although this effects the performance and makes the implementation dependent of type-libraries. However, since using C this should probably have to be done anyway, since the implementation needs to “simulate” COM's (and Microsoft's C++) object model, and to be able to do that, the program must have knowledge about the COM objects structure.

Since ERTS is written mainly in C, the goal of this thesis was to examine if it was possible to implement the COM functionality in ERTS (and therefore in C). If the implementation was just targeted for some specific COM objects, programming this in C would not be a major disadvantage. But the functionality implemented had to be very generic, no assumptions whatsoever. The problems connected to this generic implementation are:

- Many possible combinations of parameters to take care of.
- A lot of code to examine type-libraries and interpret the data retrieved from those.

[5]Interface Description Language, see Chapter 13 for an example

- Those objects that we want to use must implement the IDispatch interface. Although most objects do implement this interface, it is still a limitation. The objects must implement the IDispatch interface since the Erlang to COM interface is using the Automation method Invoke in the COM-wrapper. The Invoke method is used for run-time invocation of methods on a COM object. This could not be done in any different way, due to the demands of a generic COM wrapper.

10.3 The Erlang to COM interface

When designing the interface, basically two alternatives exist for communication between the Erlang application and the COM "wrapper" written in C:

- Using the Erl_interface library
- Using port-programs.

The chosen alternative was to use a (Erlang) Port program, basically because this was at the moment the simplest way to go, but the Erl_interface might be given another thought. For more information about Ports, see the introduction to Erlang chapter in the beginning of this thesis. Since the Erlang to COM interface communicates through an Erlang Port, all arguments/values are sent as strings. This implies that a lot of marshalling is done. If the Erl_interface alternative had been chosen, we would still have to do a lot of marshalling, although COM only accepts a limited number of parameter types, but this marshalling would probably be a bit easier. The problems with using a port-program is that `stdin` and `stdout` are used for communication (according to [2] the file descriptors 3 and 4 can be used instead, but due to problems when using them, they were not used). This makes debugging very hard. The main reason why the application used port-programs was that the implementation had to be generic. If the Erl_interface alternative had been chosen instead, a lot more code would probably have to be written, since we cannot in advance know what the next parameter type would be. With the Erl_interface solution, at every step in the COM-wrapper, the server needs to be prepared to accept any data type. When passing strings, the receiving COM-wrapper does not care what parameter type it is receiving, it just receives and stores the string. Later when marshalling, the string needs to be compared with hard-coded data-types. Another reason why it is better to use port-programs and strings is that less memory is consumed, since we do not need to allocate memory for all data types. And finally, if the Erl_interface alternative had been chosen, the COM-wrapper would have to be started at the same time as the system starts, since the Erl_interface connection can only be established *from* the C-program to Erlang. So even if we don't do any COM calls, a resource consuming COM-wrapper would run anyway.

10.4 Design of the Erlang-COM interface

The Erlang COM API doesn't resemble the syntax used in most (object-oriented) languages that support COM. In for example C++ we could call a method simply by writing

```
CoCreateInstance(CLSID,IID,...,PointerToMyInterface);
PointerToMyInterface->MyMethod_1(``Hello World'',4711);
PointerToMyInterface->MyMethod_2(``Good Bye World'');
```

Implementing this syntax in Erlang would be very difficult. But when studying the two calls above on a higher level of abstraction, we can see that we need a representation of an object, which we then use to call two methods. This idea must also be implemented in Erlang. The solution described here is, for each COM call, supply the CLSID and the IID, and let these two arguments represent the idea of having a pointer to an object. The obvious setback with this solution is that we only can have one instance of an object at a time, since the instance is represented by concatenating the CLSID and IID (both are the same for all instances). It would not be difficult though, to implement the functionality of having many instances of the same object at the same time, we can for example enumerate the instances and append the instance's number to the concatenated string.

11 Interfacing COM from Erlang

11.1 Tools needed for interfacing COM from Erlang

To be able to use the COM interface, we need the two created Erlang modules described in this chapter. But we also need information about the COM object that we want to use: the Class ID (CLSID), the Interface ID (IID), the method name, the parameter types (including if it is an in or out[6] type) and the parameter values. We can get all this information from for example:

- Microsoft's OLE/COM object viewer
- The COM object's documentation

The Erlang part of the Erlang to COM interface consists of two Erlang modules: `com_server` and `new_server`, which are both described in the rest of this chapter. For a better understanding of the following two sections, please study Figure 10 in parallel.

11.2 The `com_server` module

This module exports the function needed for the Erlang programmer to make a call to a COM object, the `call_com` function. Before the user can make a COM call, this module must be "initiated", this is done by calling the function `com_server:start_link()`, which is also exported by this module. The `start_link` function basically spawns a process (which function is exported from the `new_server` module) which takes care of transferring the information from Erlang to the COM libraries. For a brief overview how the com_server module interacts with the new_server module and what states that they can be in, take a look at Figure 10.
Syntax for the `com_server:call_com` function:

```
call_com(string Clsid, string Iid, string MethodName, tuple ParamTypeTuple
         tuple ParamTuple, string type)
```

Note: the last parameter is unused but must have the value "persist". This will probably change if a DCOM version of this code is made.[7]

Parameters

Name	Type	comment
Clsid	string	The component's class id.
Iid	string	The component's interface id
MethodName	string	The name of the method you want to call
ParamTuple	tuple of parameter values	A tuple with parameter values as strings
ParamTypeTuple	tuple of parameter types	A tuple with parameter types as strings

The CLSIDs and IIDs are strings that identify a globally unique Class ID and Interface ID. They usually have the form:

`{DB034511-047C-11D2-80FA-00AA00336A3E}`

[6] a parameter of out type means that the result from the method invocation should be returned in one of the parameters

[7] The reason why this parameter was added was the need for making several calls to one object, through a persistent connection

but in the call_com function call,it is a bit more flexible:

- lower-case and upper-case letters can be mixed, and used as wished.
- no - , { or } signs needed

The ParameterTypeTuple is a simple Erlang tuple which describes the types of the parameters (described in string format). The types are mainly those that are supported by Microsoft's VARIANT structure. One thing should be noted though, when a method returns a value in a parameter, the type of this returned parameter should be described in ParameterTypeTuple as "out" and the type of the out-parameter should be described in the ParameterTuple as for example "long*" (returning a pointer to a long). The parameter tuple is a simple Erlang tuple which consists of the parameter values as Erlang strings, i.e. "4711", "http://www.erlang.se/".
The "variant" parameter type should be handled a bit different. It should be represented in the parameter type tuple as "variant", and should be represented in the value tuple as a tuple like:

```
{``variant-type'',''value''}
```

The following code launches Microsoft's Internet Explorer and navigates to the Erlang site:

```
com_server:call_com(
"{0002DF01-0000-0000-C000-000000000046}",
"{0002DF05-0000-0000-C000-000000000046}",
"Navigate",
{"bstr"},
{"http://www.erlang.se/"},
"persist"),
com_server:call_com (
"{0002DF01-0000-0000-C000-000000000046}",
"{0002DF05-0000-0000-C000-000000000046}",
"Visible",
{"VARIANT_BOOL"},
{"TRUE"},
"persist").
```

When the `com_server:call_com` function is called, the parameters are automatically passed to a call_back function called `handle_call` (since this Erlang module implements an in the Erlang language pre-defined behaviour called `gen_server`). When this function is called, it sends the message

```
{self(),Clsid, Iid,Methodname,ParamList,TypeList,Type}
```

to the registered[8] Pid `com_service` (described in the sections about the new_server module). It then goes into a receiving state and waits on a message of the form { *success, Msg* } . A timeout is set (hard coded) to 100000 ms, but a good idea would be to overload this function with a function also containing a parameter for the time-out time. When receiving a message, handle_call returns a tuple which has the following form: { reply, Msg, State } to the caller.

[8] It is possible to create a new Erlang process and assign that new Pid a more friendly name by register it with the new name

11.3 The new_server module

The `new_server` module is a FSM which is responsible for sending all the information like Clsid, Iid, Method Name etc to the server (written in C) which makes the actual calls to the COM libraries. This FSM is started when the `com_server:start_link()` is run. The `new_server:init()` module is then run. This `init` function spawns a new process (see Chapter 4) and registers this process' Pid as `com_service`. It also opens a Port (see Chapter 4) to the server responsible for the COM calls (described in Chapter 12). The FSM then goes to a state where it waits for a message of type:

```
{Parent,Clsid,Iid,MethodName,ParamTuple,ParamTypeTuple,Type}
```

This message is sent from the com_server:handle_call/3 function to the registered Pid `com_service`. The FSM's next state is a function called

```
do_the_rest(Port,{Clsid,Iid,MethodName,ParamTuple,ParamTypeTuple,Type})
```

which sends the Clsid, Iid and Method name to the server through the Erlang Port. The next state is the function do_the_loop /3 which takes as arguments the two tuples with parameters and parameter types and converts them to lists (it also takes the Port as an argument). Then this function loops and sends all parameter and types (as pairs) to the server. When everything is sent, the FSM goes into a state where it waits for a reply from the server. Depending on the reply, it marshals the result into a suitable Erlang type and returns a tuple

```
{success, <suitable_erlang_type>}
```

if the call was successful. If an error occurs, this is caught by a timeout alternative and the message

```
{failure, ''timeout''}
```

is sent. The result is caught and send back to the Erlang process which made the original COM call.

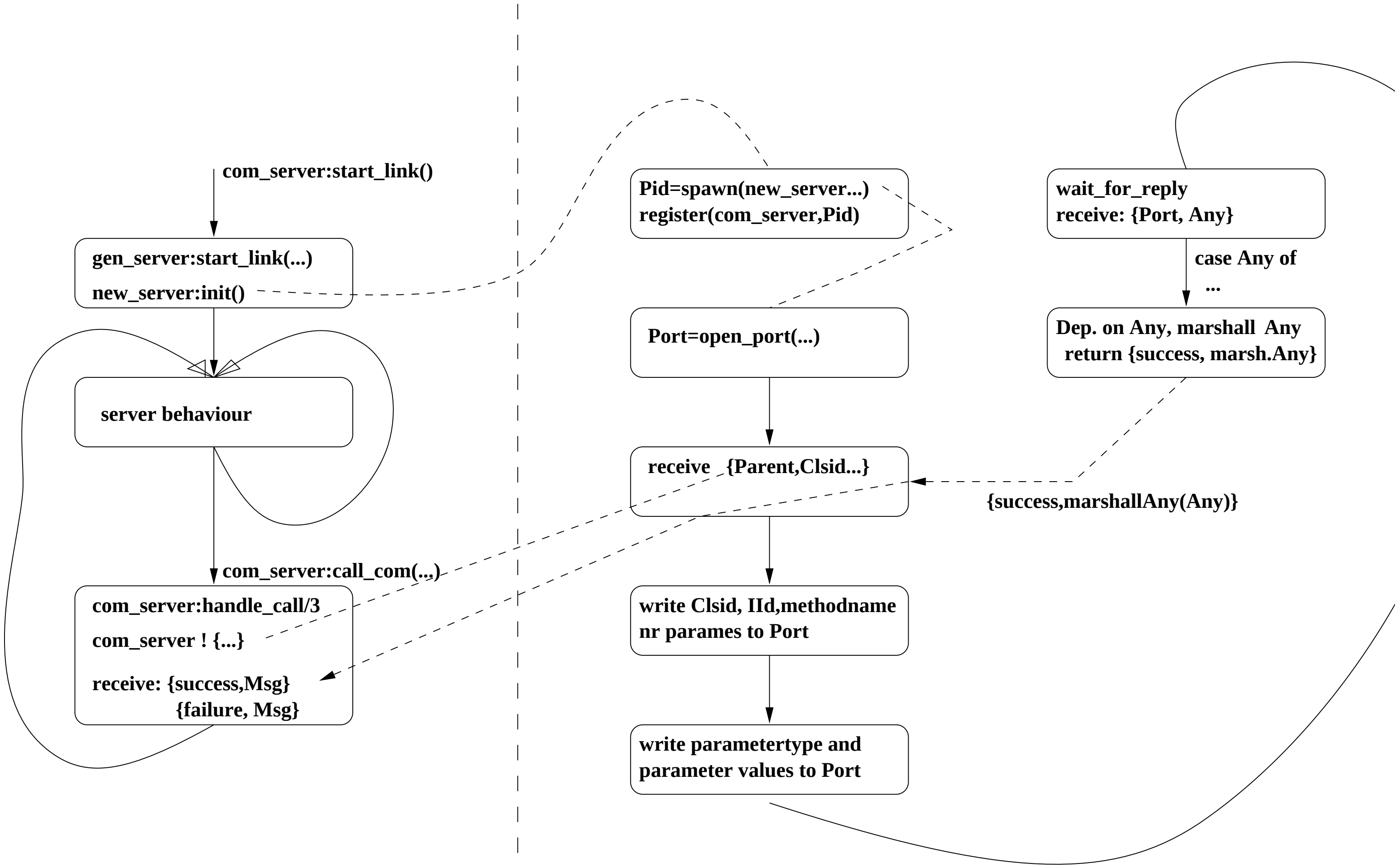

Figure 10: The Erlang part of the Erlang to COM interface

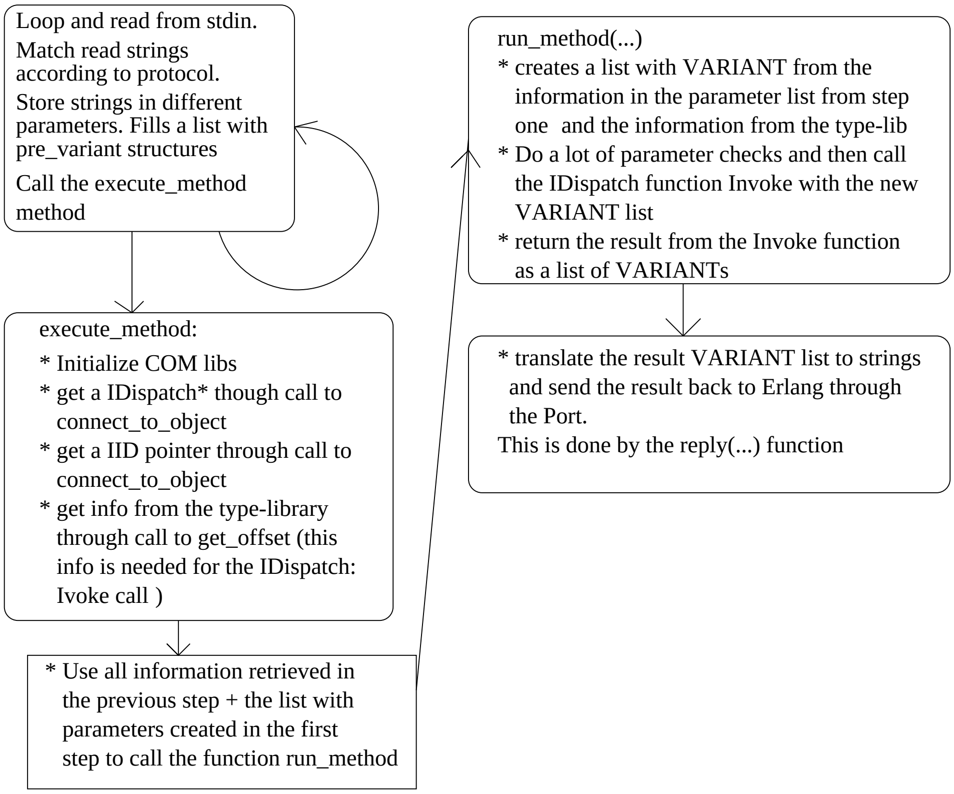

Figure 11: Overview of the server program

12 Server for communication with the COM libraries

12.1 Introduction

The purpose with this server-like program is to take care of all communication with the COM libraries and therefore serving as a middle layer between the Erlang system and the COM libraries.
As mentioned in Chapter 11, the Erlang part of this Erlang to COM interface uses a Port for communication. So the data this server-like program receives is a stream of bytes through `stdin`. Actually, the data sent are strings that this program must interpret and convert to the correct data type. After this program has got all necessary information, it make the call to the COM libraries and the result is send back to the Erlang part of the Erlang to COM interface through the Port.
This chapter describes the code for the current implementation of the part of the Erlang to COM interface that is written in C. Figure 11 gives a short description how the server program works and helps the understanding of the more detailed description in the following sections.

12.2 Description of the current implementation

Mentioned earlier, this server receives the CLSID, IID, parameter values (and types) as strings. But this information needs to be transfered in an organised way, from the Erlang module to this server. A very simple protocol was designed for doing this. The Erlang module must send according to this scheme:

```
begin_clsid
<the_clsid>
end_clsid

begin_iid
<the_iid>
end_iid

begin_methodname
<the_methodname>
end_methodname

begin_nr_parameters
<the_nr_of_parameters>
end_nr_parameters

for nr_parameters do
begin_parameter_type
  case variant type:
    variant
    begin_value_vt
    <the variant type> // VT_I4, VT_ERROR etc.
    begin_value_data
    <the value>

  case bstr type:
    bstr
    begin_par_val // parameter value
    <the value>

  case IDispatch* type:
    IDispatch*
    begin_par_val
    <the value> // the address of the IDispatch*
                // is passed as a string (returned
                // from a previous COM call in Erlang

  // NOTE: Only IDispatch* is taken care of. All other
  // pointer types must also be added. Take a look in
  // the source for more information

  case out type:
    out
    out_type
    <the out type> // long*, IDispatch** etc

end
```

In a previous chapter it was mentioned that the method used for run-time calling a method on a COM object was to use the run-time invocation method `invoke` in the `IDispatch` interface. This function must be (and is) very generic since all kinds of functions could be invoked through this function. So a generic parameter type is used in the `invoke` method call, the `VARIANT` parameter type (Microsoft defined type). It is actually a large struct with different members, one for each supported data type. This program gets its data in string format, so the program must translate the strings to the correct values. When the value is translated, it is temporarily stored in a structure similar to a `VARIANT` structure called `pre_variant` (defined by me). All `pre_variants` are stored in the list called `param_list` for later use. The `pre_variant` structure looks like:

```
typedef struct{
  char* variant_type;
  char* variant_data;
  char* data;
  int* ref_addr_int;
  long* ref_addr_long;
  double* ref_addr_double;
  byte* ref_addr_byte;
  short* ref_addr_short;
  float* ref_addr_float;
  VARIANT_BOOL* ref_addr_variant_bool;
  SCODE* ref_addr_scode;
  CY* ref_addr_cy;
  DATE* ref_addr_date;
  BSTR* ref_addr_bstr;
  VARIANT* ref_addr_variant;
  IDispatch** ref_addr_idisp;
  IUnknown** ref_addr_iunknown;
  SAFEARRAY** ref_addr_safearray;
  BOOL is_outparameter;
} pre_variant;
```

The definition of the pre_variant struct above is the one used in the code. But a much better solution would be to group this struct into three unions and a couple of variables instead. The struct could for example look like this:

```
typedef struct{\\
  char* variant_type;\\
  char* variant_data;\\
  char* data;\\
  union{
     int* ref_addr_int;\\
     long* ref_addr_long;\\
     double* ref_addr_double;\\
     byte* ref_addr_byte;\\
     short* ref_addr_short;\\
     float* ref_addr_float;\\
     VARIANT_BOOL* ref_addr_variant_bool;\\
     SCODE* ref_addr_scode;\\
     CY* ref_addr_cy;\\
     DATE* ref_addr_date;\\
     BSTR* ref_addr_bstr;\\
     VARIANT* ref_addr_variant;\\
    }ref_addr;
```

```
  union{
    IDispatch** ref_addr_idisp;\\
    IUnknown** ref_addr_iunknown;\\
    SAFEARRAY** ref_addr_safearray;\\
    }ref_ref_addr;
}
```

With this solution, we would save a lot of memory (the exact amount is system dependent). The reason why it wasn't used was lack of time.

So, depending on the parameter types, different fields are filled in. Note that usually, the values are represented as strings except for when dealing with an out-parameter, for example the `IDispatch**`. Then a reference to an allocated space is filled in. After the CLSID and IID has been received, the server checks if this combination of Class ID and Interface ID has been called before, that is, many operations on the same COM object. If true, a flag is set indicating this. This flag is used when all parameters have been received and the server is ready to continue the invocation of the call. If the flag has NOT been set, this happens:

- The `execute_method(clsid,iid,method_name,nr_of_parameters,param_list)` method is called. This function in turn:
  - Connects to the object's `IDispatch` interface (using `connect_to_object(...)`).
  - Connects to the requested interface on the object (using `connect_to_object(...)`).
  - Calls the `get_offset` function which returns a struct filled with all the information about the class and interface (this information was read from the type-library).
  - Calls the `run_method` function and supplies the structure returned from `get_offset` as one of the parameters. The `run_method` function does the actual call to the COM libraries via the `IDispatch:Invoke` call. A structure which contains VARIANTs is returned, containing the result of the `IDispatch:Invoke` call.
  - The above structure is sent to the `reply` function which is responsible for writing the result back to the Erlang program

If the flag HAS been set, the same calls as above are made, except for the call to get a pointer to the object's `IDispatch` interface (the function which does this call is named `p_execute_method(...)`). Instead, the "IDispatch*" database is searched for the correct IDispatch*.

12.3 Misc. helper functions

`connect_to_object`

This function first checks if the IID is equal to that of the IDispatch interface. In case of equality:

- First check if there is one in the object-method specific IDispatch-array, nothing will be found if this CLSID-IID pair has not been called before.
- Then check if there is one in the general object specific IDispatch-array. Reason: if we want to call another interface on an object already in use, we must use the current object's IDispatch interface. This is useful when for example different interfaces can change the state of the object. Otherwise the changes would take place on several different instances and we would not get the expected result.

- If the above steps failed, try CoCreateInstance(...,IID_IDispatch).

In case we did not want the IDispatch interface, the connect_to_object:

- First checks if there is a pointer to the requested interface in the object-method specific array (will not find anything if this clsid-iid pair has been called before.)
- Then checks if there is an IDispatch pointer in the array with IDispatch*. We look for the IDispatch pointer that belongs to the pair CLSID-IID. Reason: if we want to call another interface on an object already in use, we must use the current object's IDispatch interface. This is useful when for example different interfaces can change the state of the object. Otherwise the changes would take place on several different instances and we would not get the expected result.
- If an IDispatch* was found, call the IDispatch:QueryInterface where we supply the requested interface ID as a parameter.
- If an IDispatch* was not found with the CLSID-IID search criteria, then look for an IDispatch* with the CLSID criteria (an IDispatch* which belongs to the object).
- Should, depending on the result above, return the most suitable pointer (in the first hand, the pointer that was found with the CLSID-IID criteria, otherwise the CLSID criteria).
- Should, if none was found, do the CoCreateInstance call with the requested CLSID and IID as parameters.
- Should, if that did not work, try to do a CoCreateInstance call and supply the IUnknown IID instead of requested IID. Then try to do the IUnknown:QueryInterface on the requested IID. If the above steps fail, this will probably fail too. But at least we have tried!
- Returns the result from the step above (NULL if failed).

`get_offset`

This method is responsible for retrieving all important information from the object's type-library, as the number of parameters, parameter types etc. The function first fetches the type-library's IID from the Registry (through the `parse_registry` function found in fixclsid.c). Depending on if the IID was found, the get_offset function takes the necessary steps to get a pointer to the ITypeLib through a call to the LoadTypeLib function, a win32 API function. For every type description in the type-library, the ITypeLib:GetTypeInfo call is done, which results in an ITypeInfo pointer. With this pointer, it is possible to check every method name in all the interfaces. When a method name is found that matches that name supplied to this function, the number of arguments are compared. If these also match, one can assume that the correct interface and method has been found. Obviously, this search method is not very satisfying, since it is possible that the method has been overloaded by an interface described later on in the type-library. The next step is to check the method's invocation-type, like PROPERTYGET, PROPERTYPUT and PROPERTYPUTREF (this information is used when the IDispatch:Invoke call is made in the `run_method` call). The type of each parameter is also checked (if it is an `[in]`, `[out]`, `[in, out]`, `[retval]` or `[opt]`. Observe that these flags can be mixed. This is done by setting some bits in an array, one position per parameter, which will later be returned by this function. The get_offset function finally loops

through all the parameters and checks their types. The function must "manually" match the parameter names against some hard-coded names, since the one cannot use the type-library information about parameter types when we have types like "short*", "IDispatch**" etc. The API for reading type-libraries does not seem to work too good when dealing with parameters that have the VT_BYREF flag set. Finally, the return struct is filled with all this information and returned.
The reason why the parameter types are also passed in the `call_com(...)` call is that the COM wrapper needs this information to create the correct parameters to the actual COM call through the COM libraries. Actually, the solution presented in this thesis introduces some redundancy: the parameter type information passed in the call from Erlang to the COM wrapper can also be retrieved when parsing the type library. So what could be done is:

- Change the transfer protocol, that is, remove the parameter type information.
- Rewrite the `get_offset` function so that it can be called earlier in the server-loop.

`run_method`

This function is responsible for the actual method call, the IDispatch:Invoke call. A parameter to the Invoke call is an array of VARIANTs (it is actually more, it is an DISPPARAMS struct, which contains an array of variants but also information about how many parameters there is in this array). Every parameter is represented by a VARIANT. So this array needs to be constructed. For this purpose the information returned from the get_offset call is used. So all parameters are looped through, and for each parameter, a VARIANT is created and at the end of the loop. This VARIANT is inserted into the array belonging to the DISPPARAMS struct. In the cases where the VT_BYREF flag is set, the address retrieved in the main server loop must be used. When the new VARIANT array is filled, we have to change the order of the parameter-VARIANTs so that the last parameter is the first in the array (probably because this is the order in which the arguments are pushed to the stack, but why cannot this be done automatically ?). Then it is time for the actual IDispatch:Invoke call. Without arguments, an "empty" VARIANT array needs to be created. Depending on the invocation type (that was found when scanning the type-library, different flags are set in the call to IDispatch:Invoke.

`reply`

This function is responsible for marshalling of the result and writing the result back to Erlang. It loops through all the parameters. Depending on the type, different version of the `writecmd` function is called. Currently, there are three versions: one for ints, one for floats and one a bit more generic. What is common for all of the versions is that they write the result to erlang via stdout. But observe, when the port to this server from Erlang is opened, the program instructs that the data passing back and forth should have a header of two bytes, which tells the COM_wrapper how large the incoming data is (in bytes). So the `writecmd` functions first of all check how large the outgoing data is and then writes a two byte header which informs Erlang of the data size. Then the actual data is written.

13 Writing COM objects in Erlang

13.1 Introduction

As mentioned earlier in the design considerations chapter, the implementation of the COM object is divided into 2 parts:

- The obligatory parts of a COM object, as the IUnknown, IDispatch interfaces. Class factories, all generic functions who are the same for all objects. These are implemented in a .DLL or .EXE.
- The COM object's special functionality. All methods who are not part of the obligatory interfaces are implemented in the Erlang module.

13.2 Design of an Erlang COM server

A COM server is basically a COM object. A COM object must export certain interfaces and be ready to return pointers (to the object) to a client. Erlang is not very suited for returning pointers. In this chapter, a solution will be presented on how to solve the pointer problem by moving the functions returning pointers from Erlang to a middle layer between the client using the COM object and the COM object implemented in Erlang.

There are several reasons why we would like to write COM objects in Erlang. One reason is to make it possible to communicate with an application written in Erlang from a generic client. Another reason might be a situation where some functionality must be implemented and used in one program (a non-Erlang program), but it is much easier to implement this functionality in Erlang. A reasonable approach would be to put all the implementation of the object specific methods in Erlang, and the rest in a .DLL or .EXE and register the .DLL or .EXE in the Windows Registry as the COM object that we actually are writing in Erlang. So the .DLL or .EXE serves as a wrapper, proxy or whatever the programmer wants to call it. Mentioned earlier was Erlang's difficulties with returning pointers - with the solution provided in this report there is actually no need to do so! All the required COM interfaces (like the IUnknown and IDispatch interfaces) can be implemented in C++, VB, or any other language and compiled into a DLL serving as a middle layer between the client and the Erlang runtime system. An interesting question is how the communication with Erlang should be established. Here, the Erl_interface is used instead of port programs (which were used when the Erlang to COM interface was designed). However, since the .DLL (or perhaps .EXE) file probably needs to be created from a .IDL[9] file, the implementation of the methods in Erlang needs to follow some rules (to make the work for the IDL compiler as easy as possible). The next section describes how the Erlang part of the COM object should be designed.

13.3 Structure of the Erlang part

To make it easier to implement the IDL compiler, or rather extend the current IDL compiler, the Erlang part of a COM object implementation must be as generic as possible. The proposed structure is:

[9]Interface Description Language

```
-module(the_class_name).
-export([start/4,wait_for_input/0,method_1/2,method_2/2,first_time/4]).


start(From,Objectname,Methodname,Paramlist)->
    register(Objectname,self()),
    first_time(From,Objectname,Methodname,Paramlist).


wait_for_input()->
    receive
         {From,Methodname,Paramlist}->
            case Methodname of
% Here you take care of all methods in all interfaces
                method_1 ->
                    method_1(From,Paramlist);
                method_2 ->
                    method_2(From,Paramlist);
                _ ->
                    From ! 'unknown_method',
                    wait_for_input()
                end;
            _ ->
            wait_for_input()
    end.


%
% The first time the object is referenced, this function is called.
% It does the same thing as the wait_for_input() fcn, but is still
% needed.
%

first_time(From,Objectname,Methodname,Paramlist)->
    case Methodname of
        method_1 ->
            method_1(From,Paramlist);
        Other ->
            From ! 'unknown_method',
            wait_for_input()
        end.

%
% Here you implement the methods of all the interfaces
% (except for the IUnknown, IDispatch etc which are taken
% care of in the DLL (or EXE).
%

method_1(From,Paramlist) ->
    %
    % Here you write the method specific code for method_1
    %
    From ! the_result,
```

```
    receive
ack ->
    wait_for_input()
    after 10000 ->
    method_1(From,Paramlist)
    end.

method_2(From,Paramlist) ->
    %
    % Here you write the method specific code for method_2
    %
    From ! the_result,
    receive
ack ->
    wait_for_input()
    after 10000 ->
    method_2(From,Paramlist)
    end.
```

This proposal is a bit clumsy. If we study the code previously shown, we can see that there are a function, `first_time/4` that is only run once. The reason why this new function was invented was due to clumsy coding in the code for the process that registers itself as `comserver`. This process basically does two things when a call to a COM object arrives: if this COM object has already been called (that is, the process implementing this COM object has already been spawned), this object's PID is already stored in a list. Then this PID is fetched and a message is sent to that PID. If this object has not been called before, the `comserver` process spawns a new process for this new object and inserts the new PID into the list. But since we at the same time also want to call a method name in the object, we must also create the function that is only run once (at the creation of the object).
So instead of calling this `first_time/4` function, we could replace the call to this function with a call to the generic `wait_for_input` and let the `comserver` process also send a message to the PID that just have been inserted into the list.

13.4 The communication between the .DLL and Erlang

All COM functionality in Erlang should be put on a distributed Erlang node. This node can be assigned only for this purpose, or COM can share this node with other applications. On this node, the module com-server should be loaded and run. When running comserver:start() a new process with the server part of this module is spawned, and its Pid is registered as comserver. The server that now is running on the distributed Erlang node is waiting for a message which has the form:

```
{From,Objectname,Methodname,Paramtuple}
```

This tuple is sent from the DLL through the Erl_interface connection. Depending on the Object name and Method name, different functions are invoked.
The communication is based on the Erl_interface library. This means that the implementation of the .DLL needs some information about the system, like:

- The computers IP address
- The computers friendly name

- The name of the C-node that is to be generated
- The name of the distributed Erlang node

There are two solutions on how to get hold of this information:

- Add information to the IDL file and let the IDL compiler put this into the code for the .DLL
- Add the data to the Registry on startup of the system and retrieve the data from the Registry when needed.

The best alternative is the last one. Getting this information from the IDL file can be done, but if this is done , all the IDL files also need to be rewritten when for example the name of the distributed Erlang node changes. If this information is stored in the registry, one can easily manipulate this information when needed. I have not extended the available Erlang IDL compiler, that is, made it possible to generate the C++ code needed for generating the DLL by compiling the IDL file. Instead an example on how the DLL code in C++ should be written is provided. The conclusion is that that it should probably be quite simple to generate new code from the IDL compiler if the target code is C++. The reason is obvious if we study the part of the code that implements the .DLL:

```
...
#include ''Iface.h'' // Description of my interfaces
#include ''component.h'' // Description of the object (virtual ... =0)

...
HRESULT __stdcall MyClass::method_1(...)
{
  //Code to call the Erlang part of the COM object on
  //the distributed node via Erl_interface
}

HRESULT __stdcall MyClass::method_1(...)
{
  //As above
}

MyClass::MyClass(...)
{
  //Implementation of constructor.
}

MyClass::~MyClass(...)
{
  //Implementation of destructor.
}

... other standard methods...
```

As the above code shows, only the object specific methods need to be constructed, the rest is generic (except for the class names).
The problem is how to create the object specific methods from the IDL file. A

part of the problem was solved when the decision was taken on what the structure of the Erlang module that represents the Erlang part of the COM object should look like. The following code shows how to call a method on a generic object. The function for connecting to the node is `erl_connect_xinit(host, alive, node, addr, cookie, createion)` where:

- host is the name of the host on which the node is running
- alive is the alive name of the host
- node is the name of the node. The nodename should be of the form: alive-name@hostname
- addr is the 32 bits IP address of host
- cookie is the authorization string required for access to the remote node

For more information, see

```
addr.S_un.S_addr=inet_addr("150.236.20.198"); //my computers IP adress
erl_init(NULL,0);
if(!erl_connect_xinit("chivas",
                      "madonna",
                      "madonna@chivas.du.etx.ericsson.se",
                       &addr,
                       "the_magic_cookie",
                       0))
  {

      cout << ‘‘erl_connnect_xinit failed’’ << endl;

  }
  else
    {
      connection=erl_connect("distr_node_name@its_computer");
      if (connection < 0)
         cout << ‘‘erl_connect failed’’ << endl
      arr[0]=SELF(connection);
      arr[1] = erl_mk_atom("the_class_name");
      arr[2] = erl_mk_atom("the_method_name");
      arr[3] = erl_mk_empty_list(); // if no parameters, otherwise
    // fill with parameters here
      emsg = erl_mk_tuple(arr,4);
      send_result=erl_reg_send(connection,"comserver",emsg);

       if (send_result < 1)
         close_result=erl_close_connection(connection);
     }

ErlMessage rec_msg;
if((rc = erl_receive_msg(connection,bufp,10,&rec_msg)) != ERL_ERROR)
  {
    if (rc != ERL_TICK) {

      emsg = erl_mk_atom("ack");
```

```
        send_result=erl_reg_send(connection,"the_class_name",emsg);
        if (send_result < 1)
            cout << ‘‘erl_reg_send failed‘‘ << endl;
        close_result=erl_close_connection(connection);
    }
}
```

Most of the variables can be fetched from the Registry. The only thing that must be taken from the IDL file is the parameter type and generate code depending on the types. In the beginning of this report, it was said said that since one can't use any compiler (not possible to include any object specific header-files), one has to use the type-libraries. It is easy to create a type library, but first an IDL file is needed, and that's a bit harder. This report will not go into any details about the IDL files, there is quit a lot of information available for free. Remember though that since this program is developed using Microsoft, Microsoft's extensions to the "standard" IDL must also be studied. The IDL file written for the test component is:

```
//
// Server.idl - IDL source for Server.dll
//
// This file will be processed by the MIDL compiler to
// produce the type library (Server.tlb) and marshalling code.
//

// Interface IStatus
[
object,
uuid(477DF344-5459-11d2-8FBE-0060B08699E0),
helpstring("IStatus Interface"),
pointer_default(unique),
dual,
oleautomation
]
interface IStatus : IDispatch
{
import "oaidl.idl" ;

HRESULT GetStatus([out] BSTR* myResult) ;

} ;

//
// Component and type library descriptions
//
[
uuid(477DF345-5459-11d2-8FBE-0060B08699E0),
version(1.0),
helpstring("Type Library for my component")
]
library ServerLib
{
importlib("stdole32.tlb") ;
```

```
// Component
[
uuid(477DF346-5459-11d2-8FBE-0060B08699E0),
helpstring("Component Class")
]
coclass Component
{
[default] interface IStatus ;
} ;
} ;
```

and it is possible to compile it with Microsoft's IDL compiler, MIDL. And since the result should be a type library, the /tlb switch is used:

```
c:\>midl server.idl /tlb server.tlb
```

13.5 The com-server module

In one of the previous sections, we mentioned that the COM-server on the distributed Erlang node should be started with the call comserver:start(). Here is a brief overview of the comserver module:
The comserver:start() call spawns a new process that begins in a receive statement, waiting for a message which has the form:

```
{From,Objectname,Methodname,Paramtuple}
```

The function that this process enters has two in-parameters: `Namelist` and `Pidlist`. When the above message comes, the namelist is scanned for a member 'Objectname'. If this object name is found, the corresponding Pid is fetched from the Pidlist and the message

```
{From,Methodname,Paramtuple}
```

is sent to this Pid. If the Namelist does not contain the 'Objectname', then a recursive call takes place and adds the 'Objectname' to the Namelist and spawns a process which represents this new COM object and adds this Pid to the Pidlist. And this is the reason why the Erlang module should have the same name as the object name, otherwise one would have to supply also the module name to the comserver module.

14 Implementing a COM object in Erlang (using pure C)

The previous solution on how to implement a COM object in Erlang was based on the use of C++ and its object model to represent a COM object. In that solution, the basic idea was to create a DLL that exported the methods of the class. Each implementation of these methods in the DLL just called the Erlang node through the Erl_interface library. This solution works, but to be able to use the existing IDL compiler (with a few modifications) the DLL must be implemented in C instead. Writing a DLL in C is no problem. The problem is how to simulate the C++'s and COM's binary object structure. These object structures are based on the *Virtual table*. So how is this structure represented in C? With a little help from the include file objbase.h we can see that the solution is to represent the vtbl with structs. The following example describes how the C++'s object binary structure is created with structs:

```
typedef struct IMyInterface
{
  const struct IMyInterfaceVtbl FAR* pIMyInterfaceVtbl;
} IMyInterface;

struct IMyInterfaceVtbl
{
  HRESULT (STDMETHODCALLTYPE * QueryInterface) (IID FAR* riid,
LPVOID FAR* ppvObj) ;
  HRESULT (STDMETHODCALLTYPE * AddRef) () ;
  HRESULT (STDMETHODCALLTYPE * Release) () ;

};
typedef struct IMyInterfaceVtbl IMyInterfaceVtbl;
```

Actually, the code above does not really define the virtual function table, it defines the whole interface (and that is what we need). The vtbl is defined in the IMyInterfaceVtbl struct. Note that this struct's members are function pointers! So from now on, we can assume that the code implementing the DLL knows these structures.

14.1 What functions does this DLL need to export?

In order to work properly as a COM object, the DLL needs to export the following "administrative" functions:

- DllRegisterServer - this function makes it possible to use the regsvr32 program to register the DLL in the Registry with all needed sub-keys and the corresponding values
- DllUnregisterServer - is needed to successfully unregister the DLL in the Registry
- DllCanUnloadNow - is used to decide if the DLL can safely be unloaded from the memory (the object is no longer referenced)
- DllGetClassObject - is used by CoGetClassObject to get an interface pointer to the COM object the DLL implements

14.2 Where should we implement the methods?

There are two ways to solve the problem where to put the code that implements the object's methods: either put all code that belongs to the object in one DLL or let the code that implements the methods in a separate DLL (which is loaded from the "administrative" DLL, that is, the DLL which implements the DllRegisterServer,DllUnregisterServer...). In this example the alternative to put all the code in one single DLL is chosen. This leads to a problem which will be described later. So, we have described the vtbl as a struct with function pointers. How do we set these function pointers to point to the implemented methods? First of all, let us study the code that do this for the AddRef method:

```
/* the simulated vtbl... */
IMyInterfaceVtbl theVtbl;
/* to ensure type safety... */
typedef HRESULT (CALLBACK* LPFNDLLAddRef)();
...
/* the function pointer... */
LPFNDLLAddRef lpfnDllAddRef;
...
lpfnDllAddRef = (LPFNDLLAddRef)GetProcAddress(hDLL,
      "AddRef");
theVtbl.AddRef=lpfnDllAddRef;
```

The GetProcAddress function takes as the first parameter a handle to a DLL module. This handle is achieved by using the function LoadLibrary. The problem mentioned above occurs here: the handle needed to use the GetProcAddress is created *outside* of the DLL code! This means that it is also needed to write and export a function that introduces this parameter inside the DLL code!

15 Summary

The goal with this master's thesis was not to develop an application/functionality that without changes can be integrated in Erlang/OTP. Instead, the goal with this thesis was to examine *how* this Erlang-COM interface could be implemented and then create an experimental implementation of this functionality. This thesis actually consisted of two goals: the Erlang to COM interface and the functionality of creating COM objects in Erlang. In this report, I have presented solutions to these two goals.
The code that I have produced is not by any means suitable to be inserted into the Erlang Runtime System. The reason is that this code was experimentally developed, functionality was added when needed. This has made the code pretty large with some redundant parts. This development process has also made the program pretty slow. But again, this was just an experiment. I think that the existing code can easily be re-written to suit the RTS better, since we now know what functionality needs to be implemented. And I think that the bottlenecks and quite a lot of code can be removed when this code is rewritten to suit the RTS better. For example, drivers can be used instead of the simple "protocol" used when transferring data through an Erlang Port to the COM-wrapper.

15.1 Summary - Goal 1

The Erlang to COM interface works as far as I have been able to test it. I have tested it against

- Microsoft Explorer - it is possible to from Erlang control the browser and "surf" from Erlang. The code for the test program is in test_explorer.erl.
- Microsoft Agents - it is possible to control the "Agents" and make them appear on the screen and say something (some extra software needs to be installed on the computer though). The test code is in test_agents.erl.
- A mail COM object - it is possible to send and receive mail in Erlang via this COM object. Code for testing this is also available.

But please observe that the code for the Erlang to COM interface has not been thoroughly tested yet. If something does not work, it is probably because all parameter types might not have been taken care of in the C code for the interface. The reason why I did not put that much effort in fixing bugs like these is mainly because:

- I didn't have enough time to check for all possible combinations of parameter types.
- Since this code never was meant to be inserted directly into the ERTS, most of the code must be rewritten anyway.

But the changes needed to fix the bugs are probably quite easy to find.

15.2 Summary - Goal 2

Although not very much time was spent coding on this part of the COM interface (compared to Goal 1), in my opinion this part works as well as the interface in Goal 1. I've tested this functionality and in my opinion it works quite satisfactory. The big part left on this goal is to rewrite the Erlang IDL compiler to make it generate some code for constructing the DLLs. In this report I have given some proposals on how to standardise these DLLs so that it will be quite simple to modify the IDL compiler.

So how can this report be used? First of all, I think that it gives an understanding about COM, at least as much understanding that is needed to use COM objects. And besides giving proposals on how one can implement the different functions for communicating with COM libraries from Erlang, it also gives some hints about which parts that is important to really think about. In my opinion, especially the code that I have written that scans the type libraries for important information is worth studying, since the documentation for the type library API, the on-line documentation, was not in my opinion really sufficient. All code that I have produced has been handed over to my supervisor on Ericsson Telecom AB, Patrik Nyblom on 1998-10-28.

16 Glossary

API
Application Programming Interface.

CLSID
A GUID that is a class' ID.

COM
Component Object Model

DCOM
The distributed version of COM

DLL
Dynamic Link Library.

ERTS
Erlang Runtime System

Erlang Port
Used by Erlang to communicate with external processes.

Erl_interface
A C library used for communication with an external Erlang process. Developed by Ericsson.

GUID
Globally Unique Identifier. A 128 bits struct that is created through an algorithm that uses a time stamp and the computer's network card id as inputs and generates a globally unique id.

IDL
Interface Description Language. A language for describing interfaces. For example, what methods that an interface contains

IDispatch
A COM interface used for run-time invokation of methods in an interface.

IID
A GUID that is an interface's ID.

IUnknown
A COM interface that implements functions for returning pointers to a specific interface on that object. It also implements methods for reference counting.

Type library
A binary file which is the result of compiling an IDL file with the switch `tlb`. Example: `c:>midl test.idl /tlb test.tlb`

Marshalling
Translating from one type to another.

Port
A communication channel, see Chapter 4

Variant
Microsoft defined data type, encapsulates the supported data for the `IDispatch::Invoke` method. See Chapter 12.

vtbl
Virtual Function table, see Figure 3.

References

[1] Sigbjorn Finne Daan Leijen Erik Meijer Simon Peyton Jones. H/direct: A binary foreign language interface for haskell. http://www.haskell.org.

[2] Damien Watkins. Com:an architecture for developing reusable components. *C++ Report*, 1998.

[3] Microsoft. The component object model: Technical overview. Paper adapted from an article appearing in Dr. Dobbs Journal, December 1994.

[4] P.E. Chung Y. Huang S.Yajnik D.Liang J.Shih C.-Y. Wang Y.-M. Wang. Dcom and corba. side by side step by step and layer by layer. *C++ Report*, 1998.

[5] Microsoft. *DCOM Architecture, white paper.*

[6] Kurt Bergler. Distributed objects. corba versus com: A comparison of technologies. Master's thesis, Technische Universität Wien, February 1996.

[7] Daan Leijen. Functional components: Using com components in haskell. Master's thesis, -, 1998.

[8] Thomas Nordin Simon Peyton Jones Alastair Reid Malcolm Wallace. Green card: a foreign-language interface for haskell.

[9] Niclas Eklund. Transaction service for distributed control systems. Master's thesis, Uppsala University, 1998.

[10] Joe Armstrong Robert Virding Claes Wikström Mike Williams. *Concurrent Programming in ERLANG.* Prentice Hall, 1996.

[11] Dale Rogerson. *Inside COM - Microsoft's Component Object Model.* Microsoft Press, 1997.

[12] Dr Gui. Dr gui web page. http://www.microsoft.com/msdn/news/drgui/.

[13] Microsoft. *Visual C++ 5.0 Online manual.*